

# Introduction

Learning to control agents directly from high-dimensional sensory inputs like vision and speech is one of the long-standing challenges of reinforcement learning (RL). Most successful RL applications that operate on these domains have relied on hand-crafted features combined with linear value functions or policy representations. Clearly, the performance of such systems heavily relies on the quality of the feature representation. Recent advances in deep learning have made it possible to extract high-level features from raw sensory data, leading to breakthroughs in computer vision [6, 12, 9] and speech recognition [3, 4]. These methods utilize a range of neural network architectures, including convolutional networks, multilayer perceptrons, restricted Boltzmann machines, and recurrent neural networks, and have exploited both supervised and unsupervised learning. It seems natural to ask whether similar techniques could also be beneficial for RL with sensory data.

This project demonstrates that a convolutional neural network can learn successful control policies from raw video data in the Pong RL environment. The network is trained with a variant of the Q-learning [14] algorithm, with stochastic gradient descent to update the weights. The goal is to create a single neural network agent that is able to successfully learn to play pong. The network was not provided with any game-specific information or hand-designed visual features, and was not privy to the internal state of the emulator; it learned from nothing but the video input, the reward and terminal signals, and the set of possible actions—just as a human player would. In this project, Deep learning model is built to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards.

## Objectives

The objectives of this project are as follows:

1. To build the deep learning model that successfully learns control policies directly from high-dimensional sensory input using reinforcement learning in the RL Pong environment.
2. To train the network with a variant of the online Q-learning algorithm that combines stochastic gradient descent mini-batch updates with experience replay memory to ease the training of deep networks for RL
3. To learn Deep Q-learning and implement it in the RL Pong environment using TensorFlow.

## Agent Description

**Pong** is a [table tennis](#) sports game featuring simple two-dimensional graphics, manufactured by [Atari](#) and originally released in 1972.

## Utility-based agents

Partial observability and stochasticity are ubiquitous in the real world, and so, therefore, is decision making under uncertainty. Technically speaking, a rational utility-based agent chooses the action that maximizes the **expected utility** of the action outcomes—that is, the utility the agent expects to derive, on average, given the probabilities and utilities of each outcome. The utility-based agent structure appears in Fig. 1.

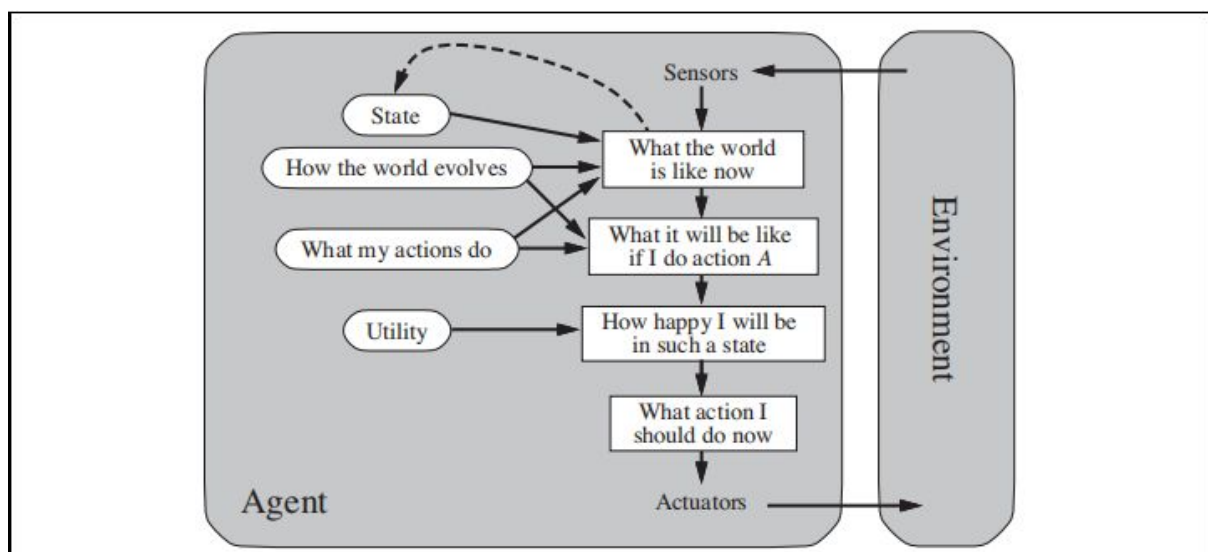


Fig. 1. A model-based, utility-based agent

In this project, the agent interacts with the environment through a sequence of observations, actions, and rewards. The goal of the agent is to select actions in a fashion that maximizes cumulative future reward. More formally, we use a deep convolutional neural network to approximate the optimal action-value function;

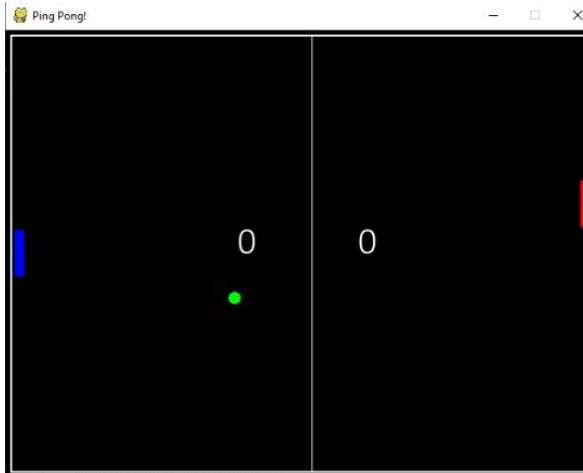
$$Q^*(s,a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t = s, a_t = a, \pi],$$

which is the maximum sum of rewards  $r_t$  discounted by  $\gamma$  at each time step  $t$  achievable by a behavior policy  $\pi = P(a|s)$  after making an observation( $s$ ) and taking action ( $a$ ).

## PEAS

The PEAS is shown in TABLE I.

TABLE I  
AGENT DESCRIPTION

Performance measure	Game Score
Environment	<p><math>640 \times 480</math> Board containing two in-game paddles and a ball as shown in figure.</p> 
Actuators	Do nothing, move upward and move downward within the boundary.
Sensors	$80 \times 80 \times 4$ images.

# Agent Environment

The agent environment is described in TABLE II.

TABLE II  
AGENT ENVIRONMENT

Environment types	Description
Fully Observable	An agent's sensors give it access to the complete state of the environment at each point in time.
Episodic	The agent's experience is divided into atomic "episodes" (each episode consists of the agent perceiving and then performing a single action), and the choice of action in each episode depends only on the episode itself.
Strategic	The environment is deterministic except for the actions of other agents.
Competitive Multi-agent	The opponent is trying to maximize its performance measure, which by the rules of Pong, minimizes the agent's performance measure.
Static	The environment is unchanged while an agent is deliberating.
Discrete	A limited number of distinct, clearly defined percepts and actions.

## Problem Specification

Pong is a [two-dimensional sports game](#) that simulates [table tennis](#). The player controls an in-game paddle by moving it vertically across the left or right side of the screen. They can compete against another player controlling a second paddle on the opposing side. Players use

the paddles to hit a ball back and forth. The goal is for each player to reach twenty points before the opponent; points are earned when one fails to return the ball to the other.

### Assumptions:

- ❖ Board size:  $640 \times 480$
- ❖ The left player is a DQN-trained Player and the right player is explicitly programmed to play pong(hard-coded AI). The game ends when one of the players achieves 20 points before the opponent.

### Initial State:

- ❖ Score:  $0 - 0$
- ❖ Left Paddle Position: (10, 215)
- ❖ Right Paddle Position: (620, 215)
- ❖ Ball position: (307.5, 232.5)

Initial State is shown in Fig. 2.



Fig. 2. Initial State

### Goal State:

$\max(\text{Left Player Score}, \text{Right Player Score}) \geq 20$

Goal State is shown in Fig. 3.

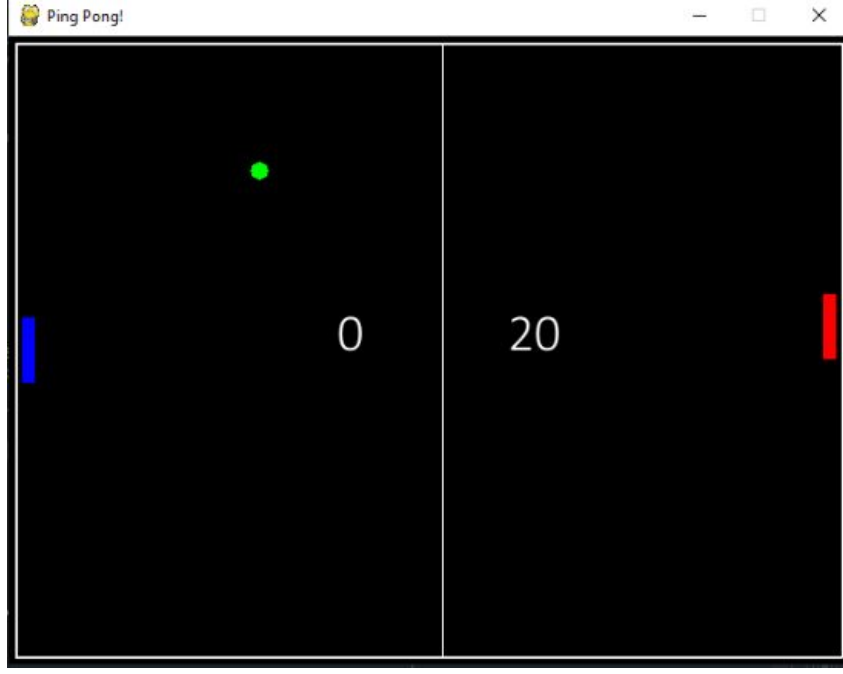


Fig. 3. Goal State

### Constraints:

- ❖ In-game paddles cannot be moved horizontally and can only be moved vertically.
- ❖ Left paddle movement:  $(10, 10) - (10, 420)$ .
- ❖ Right paddle movement:  $(620, 10) - (620, 420)$ .

## State-Space Representation

The agent interacts with an environment through a sequence of observations, actions, and rewards. The goal of the agent is to select actions in a fashion that maximizes cumulative future reward. More formally, we use a deep convolutional neural network to approximate the optimal action-value function;

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

which is the maximum sum of rewards  $r_t$  discounted by  $\gamma$  at each time step  $t$  achievable by a behavior policy  $\pi = P(a|s)$  after making an observation( $s$ ) and taking action ( $a$ ). The schematic illustration of convolutional neural network is shown in Fig. 4.

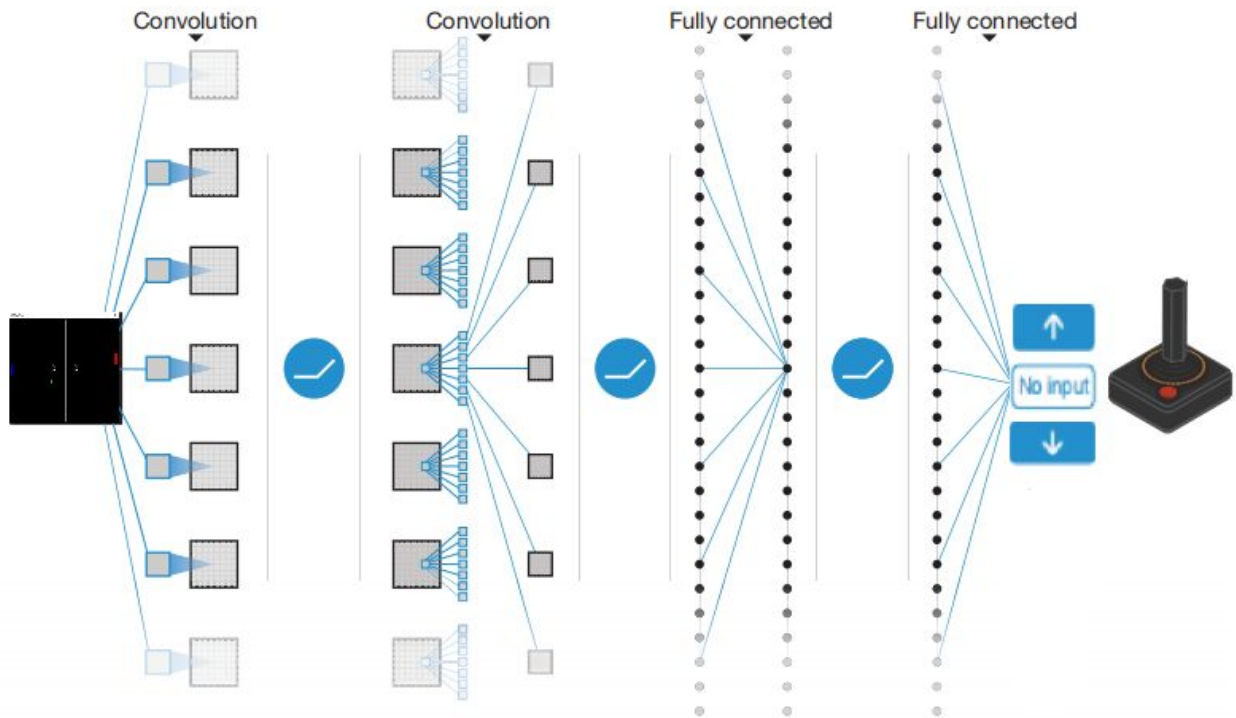


Fig. 4. Schematic illustration of the convolutional neural network

Different Possible states the agent can attain is shown in Fig. 5.

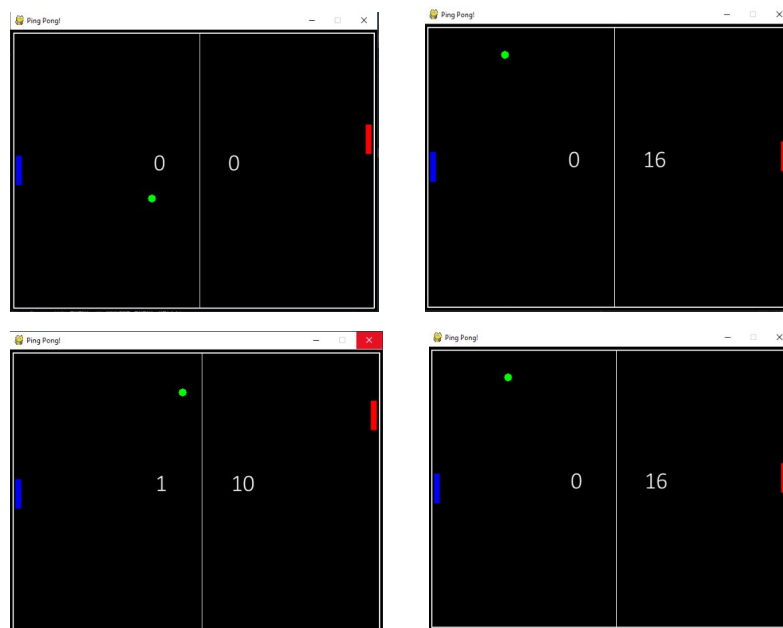


Fig. 5. Different possible states the agent can attain

# Algorithm used –Technical description

## Algorithm

We consider tasks in which an agent interacts with an environment  $\mathcal{E}$ , in this case, the RL Pong Environment, in a sequence of actions, observations, and rewards. At each time-step, the agent selects an action  $a_t$  from the set of legal game actions,  $A \in \{1, \dots, K\}$ . The action is passed to the emulator and modifies its internal state and the game score. In general,  $\mathcal{E}$  maybe stochastic. The emulator's internal state is not observed by the agent; instead, it observes an image  $x_t \in \mathbb{R}^d$  from the emulator, which is a vector of raw pixel values representing the current screen. In addition, it receives a reward  $r_t$  representing the change in game score. Note that in general, the game score may depend on the whole prior sequence of actions and observations; feedback about an action may only be received after many thousands of time-steps have elapsed.

Since the agent only observes images of the current screen, the task is partially observed and many emulator states are perceptually aliased, i.e. it is impossible to fully understand the current situation from only the current screen  $x_t$ . We, therefore, consider sequences of actions and observations,  $s_t = x_1, a_1, x_2, a_2, \dots, x_t, a_t$  and learn game strategies that depend upon these sequences. All sequences in the emulator are assumed to terminate in a finite number of time steps. This formalism gives rise to a large but finite Markov decision process (MDP) in which each sequence is a distinct state. As a result, we can apply standard reinforcement learning methods for MDPs, simply by using the complete sequence  $s_t$  as the state representation at time  $t$ .

The goal of the agent is to interact with the emulator by selecting actions in a way that maximizes future rewards. We make the standard assumption that future rewards are discounted by a factor of  $\gamma$  per time-step, and define the future discounted *return* at time  $t$  as

$$R_t = \sum_{t'=t}^T \gamma^{t'-t} r_{t'}$$



where  $T$  is the time-step at which the game terminates. We define the optimal action-value function  $Q^*(s, a)$  as the maximum expected return achievable by following any strategy, after seeing some sequence  $s$  and then taking some action  $a$ ,

$$Q^*(s, a) = \max_{\pi} \mathbb{E}[R_t | s_t = s, a_t = a, \pi],$$

where  $\pi$  is a policy mapping sequences to actions (or distributions over actions). The optimal action-value function obeys an important identity known as the *Bellman equation*. This is based on the following intuition: if the optimal value  $Q^*(s', a')$  of the sequence  $s'$  at the next time-step was known for all possible actions  $a'$  then the optimal strategy is to select the action maximizing the expected value of  $r + \gamma Q^*(s', a')$ ,

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[ r + \gamma \max_{a'} Q^*(s', a') \middle| s, a \right]$$

The basic idea behind many reinforcement learning algorithms is to estimate the action-value function, by using the Bellman equation as an iterative update

$$Q_{i+1}(s, a) = \mathbb{E}[r + \gamma \max_{a'} Q_i(s', a') | s, a].$$

Such *value iteration* algorithms converge to the optimal action-value function,  $Q_i \rightarrow Q^*$  as  $i \rightarrow \infty$  [13]. In practice, this basic approach is totally impractical, because the action-value function is estimated separately for each sequence, without any generalization. Instead, it is common to use a function approximator to estimate the action-value function,

$Q(s, a; \theta) \approx Q^*(s, a)$ . In the reinforcement learning community, this is typically a linear function approximator, but sometimes a nonlinear function approximator is used instead, such as a neural network. We refer to a neural network function approximator with weights  $\theta$  as a Q-network. A Q-network can be trained by minimizing a sequence of loss functions  $L_i(\theta_i)$  that changes at each iteration  $i$ ,

$$L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} \left[ (y_i - Q(s, a; \theta_i))^2 \right]$$

where  $y_i = \mathbb{E}_{s' \sim \mathcal{E}} [r + \gamma \max_{a'} Q_i(s', a'; \theta_{i-1}) | s, a]$  is the target for iteration  $i$  and  $\rho(s, a)$  is a probability distribution over sequences  $s$  and actions  $a$  that we refer to as the *behavior distribution*. The parameters from the previous iteration  $\theta_{i-1}$  are held fixed when optimizing

the loss function  $L_i(\theta_i)$ . Note that the targets depend on the network weights; this is in contrast with the targets used for supervised learning, which is fixed before learning begins. Differentiating the loss function with respect to the weights, we arrive at the following gradient,

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[ \left( r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i) \right]$$

Rather than computing the full expectations in the above gradient, it is often computationally expedient to optimize the loss function by stochastic gradient descent. If the weights are updated after every time-step and the expectations are replaced by single samples from the behavior distribution  $\rho$  and the emulator  $\mathcal{E}$  respectively, then we arrive at the familiar *Q-learning* algorithm [14]. This algorithm is *model-free*: it solves the reinforcement learning task directly using samples from the emulator  $\mathcal{E}$  without explicitly constructing an estimate of  $\mathcal{E}$ . It is also *off-policy*: it learns about the greedy strategy  $a = \max_a Q(s, a; \theta)$  while following a behavior distribution that ensures adequate exploration of the state space. In practice, the behavior distribution is often selected by an  $\epsilon$ -greedy strategy that follows the greedy strategy with probability  $1 - \epsilon$  and selects a random action with probability  $\epsilon$ .

## Training algorithm for Deep Q-Networks

The full algorithm for training deep Q-networks is presented in Algorithm 1 [2]. The agent selects and executes actions according to an  $\epsilon$ -greedy policy based on  $Q$ . Because using histories of arbitrary length as inputs to a neural network can be difficult, our  $Q$ -function instead works on a fixed-length representation of histories produced by the function  $\phi$  described above. The algorithm modifies standard online  $Q$ -learning in two ways to make it suitable for training large neural networks without diverging.

First, we use a technique known as experience replay [8] where we store the agent's experiences at each time-step,  $e_t = (s_t, a_t, r_t, s_{t+1})$  in a data-set  $D = e_1, \dots, e_N$  pooled over many episodes into a *replay memory*. During the inner loop of the algorithm, we apply  $Q$ -learning updates, or mini batch updates, to samples of experience,  $e \sim D$  drawn at random from the pool of stored samples. After performing experience replay, the agent selects and executes an action according to an  $\epsilon$ -greedy policy. 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. In practice, This algorithm only stores the last  $N$  experience tuples in the replay memory and samples uniformly at random from  $D$  when performing updates.

---

**Algorithm 1** Deep Q-learning with Experience Replay

---

```

Initialize replay memory  $\mathcal{D}$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights
for episode = 1,  $M$  do
    Initialise sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$ 
    for  $t = 1, T$  do
        With probability  $\epsilon$  select a random action  $a_t$ 
        otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ 
        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathcal{D}$ 
        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$ 
        Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$ 
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to equation 3
    end for
end for

```

---

## Preprocessing and Model Architecture

### Preprocessing

Working directly with raw RL Pong frames, which is  $640 \times 480$  pixel images with a 128-color palette can be demanding in terms of computation and memory requirements. So a basic preprocessing step is applied to reduce the input dimensionality. The raw frames are preprocessed by first converting their RGB representation to gray-scale and down-sampling it to  $80 \times 80$  pixel images that roughly capture the playing area. For the experiment in this project, the function  $\phi$  from algorithm 1 applies this preprocessing to the last 4 frames of history and stacks them to produce the input to the  $Q$ -function.

### Model Architecture

There are several possible ways of parameterizing  $Q$  using a neural network. Since  $Q$  maps, history action pairs to scalar estimates of their Q-value, the history and the action have been

used as inputs to the neural network by some previous approaches [11], [7]. The main drawback of this type of architecture is that a separate forward pass is required to compute the Q-value of each action, resulting in a cost that scales linearly with the number of actions. Instead, an architecture in which there is a separate output unit for each possible action is used, and only the state representation is an input to the neural network. The outputs correspond to the predicted Q-values of the individual action for the input state. The main advantage of this type of architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network.

The exact architecture, shown schematically in Fig. 6, is as follows. The input to the neural network consists of an  $80 \times 80 \times 4$  image produced by the preprocessing map  $\phi$ . The first hidden layer convolves 32 filters of  $8 \times 8$  with stride 4 with the input image and applies a rectifier nonlinearity [5], [10]. The second hidden layer convolves 64 filters of  $4 \times 4$  with strides 2 again followed by a rectifier nonlinearity. This is followed by a third convolutional layer that convolves 64 filters of  $3 \times 3$  with strides 1 followed by a rectifier. 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. The number of valid actions in Pong is 3.

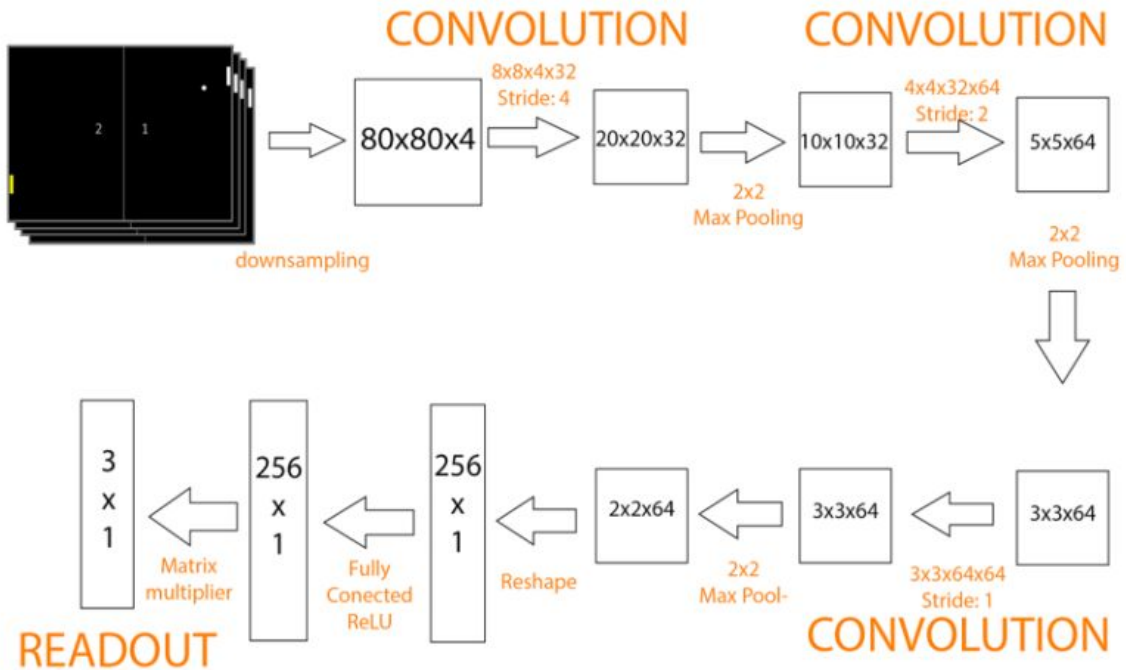


Fig. 6. Deep Q Network architecture diagram

## List of Hyperparameters and their values

The values of all the hyperparameters were selected by performing an informal search on the games Pong, Breakout, Seaquest, Space Invaders, and Beam Rider [1].

TABLE III  
LIST OF HYPERPARAMETERS AND THEIR VALUES

Hyperparameter	Value	Description
Minibatch size	32	The number of training cases over which each stochastic gradient descent update is computed.
Replay memory size	590,000	SGD updates are sampled from this number of most recent frames.
Agent history length	4	The number of most recent frames experienced by the agent that is given as input to the Q- Network.
Discount factor	0.99	Discount factor gamma used in Q-Learning update.
Learning rate	1e-6	The learning rate used by adam optimizer.
Initial exploration	1.00	The initial value of $\epsilon$ in $\epsilon$ -greedy exploration.
Final exploration	0.05	The final value of $\epsilon$ in $\epsilon$ -greedy exploration.
Final exploration frame	500	The number of frames over which the initial value of $\epsilon$ is linearly annealed to its final value.

Replay start size	500	Uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory.
-------------------	-----	---

## Outcome analysis

Better results were achieved after approximately 1.38 million-time steps, which corresponds to about 25 hours of game time. Qualitatively, the network played at the level of an experienced human player, usually beating the game with a score of  $20 - 2$ . Fig. 7 shows the maximum Q value for the first  $\sim 20,000$  training steps. As can be seen, the maximum Q value increases over time. This indicates improvement as it means that the network is expecting to receive a greater reward per game as it trains for longer. In theory, as we continue to train the network, the value of the maximum Q value should plateau as the network reaches an optimal state.

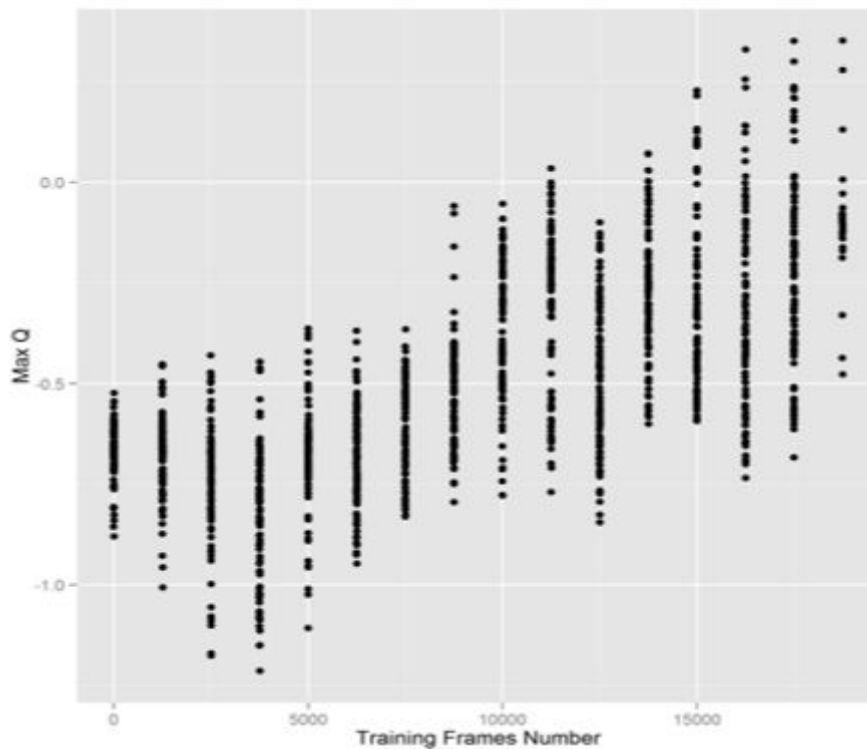


Fig. 7. Q-value plotted for 20,000 training steps

# Conclusion

This project introduced a deep learning model for reinforcement learning and demonstrated its ability to master control policies for the RL Pong environment, using only raw pixels as input. It incorporates ‘end-to-end’ reinforcement learning that uses reward to continuously shape representations within the convolutional network towards salient features of the environment that facilitate value estimation. A variant of online Q-learning is presented that combines stochastic mini batch updates with experience replay memory to ease the training of deep networks for RL.

# References

- [1] Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.
- [2] Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Graves, Alex, Antonoglou, Ioannis, Wierstra, Daan, and Riedmiller, Martin. Playing atari with deep reinforcement learning. *arXiv preprint [arXiv:1312.5602](https://arxiv.org/abs/1312.5602)*, 2013.
- [3] George E. Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(1):30–42, January 2012.
- [4] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. In *Proc. ICASSP*, 2013.
- [5] Kevin Jarrett, Koray Kavukcuoglu, MarcAurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR 2009)*, pages 2146–2153. IEEE, 2009.

- [6] Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1106–1114, 2012.
- [7] Sascha Lange and Martin Riedmiller. Deep auto-encoder neural networks in reinforcement learning. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–8. IEEE, 2010.
- [8] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993.
- [9] Volodymyr Mnih. *Machine Learning for Aerial Image Labeling*. PhD thesis, University of Toronto, 2013.
- [10] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML 2010)*, pages 807–814, 2010.
- [11] Martin Riedmiller. Neural fitted q iteration—first experiences with a data-efficient neural reinforcement learning method. In *Machine Learning: ECML 2005*, pages 317–328. Springer, 2005.
- [12] Pierre Sermanet, Koray Kavukcuoglu, Soumith Chintala, and Yann LeCun. Pedestrian detection with unsupervised multi-stage feature learning. In *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR 2013)*. IEEE, 2013.
- [13] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
- [14] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.