Playing Games with Deep Reinforcement Learning

Debidatta Dwibedi
debidatd@andrew.cmu.edu
10701

Anirudh Vemula
avemula1@andrew.cmu.edu
16720

Abstract

Recently, Google Deepmind showcased how Deep learning can be used in conjunction with existing Reinforcement Learning (RL) techniques to play Atari games[10], beat a world-class player [13] in the game of Go and solve complicated riddles [3]. Deep learning has been shown to be successful in extracting useful, nonlinear features from high-dimensional media such as images, text, video and audio [11]. For control tasks like playing games, researchers have traditionally used handcrafted-features which require a lot of human effort. Convolutional neural networks (CNN) can be used to extract useful visual features from the high-dimensional input, such as a video game screen, to learn near-optimal value function in such control tasks. We chose the *Arcade Learning environment* as our testbed to examine how deep learning can be applied to a reinforcement learning setting and analyze performance of various modern algorithms. The learning is performed end-to-end with no game knowledge included in the architecture or during training. We also show how the architecture of the deep network used, can be modified to achieve superior performances and compare performances on the task of playing a complicated game *Seaquest*.

1 Introduction

The Reinforcement learning (RL) problem involves an agent interacting with an unknown environment (whose dynamics may or may not be known), by executing a sequence of actions and learning from the observations and rewards received [14]. The goal of the agent is to maximize cumulative rewards. Unlike a classic planning problem, an RL agent doesn't start with perfect knowledge of the environment, but learns through experience. This is very much in line with the behavioural psychology school of thought which postulates that the behaviour of animals and people can be modulated by providing different rewards and punishments.

Recent deep learning works have shown that deep learning systems are capable of extracting high-level features from raw sensory data in supervised learning tasks. These algorithms are finally reaching human level performance in many areas of artificial intelligence such as computer vision, speech recognition [1] etc. In computer vision especially, deep learning systems are the state-of-the-art algorithms in tasks like image classification [6] and semantic segmentation [4]. This has been made possible due to the re-emergence of artificial neural networks in the form of convolutional neural networks (for images) and recurrent neural networks (for audio). The availability of large annotated datasets, computationally powerful GPUs and mathematical tricks such as ReLU came together to make this re-emergence possible by bringing down the time taken for training such models.

In this report, we examine how deep learning has been applied to the reinforcement learning problem, the challenges faced and how they have been overcome. We then proceed to implement several of the most recent works and analyze their performance on the same testbed. We use *Arcade learning environment* as our testbed for this purpose and compare different algorithms based on the cumulative scores obtained by the respective trained agents. We start with an open-source implementation of Deep Q Learning written in Python. We first attempt to understand how the baseline Deep Q learning framework works and proceed to make modifications to that code to implement an LSTM based Q network and a dueling architecture network. We show how changing architecture can result in superior performances over conventional architectures. We will end it with some results and follow it up with potential future directions in this rapidly advancing field.

2 Related Work

Our work is related to recent research in Reinforcement learning, deep learning in supervised setting and deep learning applied to reinforcement learning (deep RL).

2.1 Reinforcement Learning

Most of the successful cases of reinforcement learning have been in the domain of developing agents for games. One such work is *TD-Gammon*, a backgammon playing program which was able to beat world-class players without any human intervention. The program was learnt entirely using reinforcement learning and self-play. It used a popular model-free reinforcement learning algorithm, known as *Q-learning*, to learn an optimal policy by approximating the value function using a multi-layer perceptron with one hidden layer.

The same program, unfortunately, could not be shown to be as effective in other domains such as the game of Go, resulting in the widespread belief that the program could only work in the special case of backgammon and performs poorly in other domains. Subsequently, research in RL shifted to the simple case of using linear function approximators where the training procedure in Q-learning can be theoretically proven to converge. But most real-life control tasks exhibit non-linearity which cannot be captured effectively using linear function approximators. Hence, these systems have always fallen short of achieving human-level performance in real-life tasks.

2.2 Deep Learning

As discussed before, most of the state-of-the-art systems in computer vision and other related fields of AI use deep learning. Irrespective of the domain it is applied to, deep learning can be thought of as performing a series of non-linear operations on the input data in a cascaded fashion. The major reason why deep learning has been a resounding success is the fact that it acts as a high-quality feature extractor on the data at multiple levels. For example, in case of images the first convolutional layer (in CNNs) is shown to extract low-level features such as edges and corners. The subsequent layer uses the features from the previous layer as input to extract textures in the image. These textures are combined in a non-linear fashion by the next layer to identify parts and objects in the image. These multi-level features are extracted by minimizing a given loss function on the training set using *backpropagation*. Our work deals with convolutional neural networks [8] that are designed specifically for image input. CNNs have been shown to be highly effective in image-related tasks such as image classification [6] and semantic segmentation [4]. CNNs will be described in greater detail in Section 3.2.

2.3 Deep Reinforcement Learning

Recently, Google Deepmind developed *Deep Q-network* (DQN), a deep neural network architecture, that has been shown to be capable of learning human-level control policies on a variety of different Atari 2600 games [10]. DQNs learn to estimate the Q-values (state-action value function) of selecting each action from the current game state. Since, the state-action value function is a sufficient representation of the agent's policy, a game can be played by selecting the action with the maximum Q-value at each timestep. Learning policies in this way from raw screen pixels to actions, these networks have been shown to achieve state-of-the-art performance on several Atari 2600 games. Note that the same network can be used in several tasks without any change and that the learning is end-to-end from raw pixel values to Q-values without any need for handcrafted features or human intervention. The architecture of DQN is described in Section 5.1.

DQNs have been extended in subsequent years to achieve better performance in complicated games. *Recurrent Deep Q-Networks* combine a Long Short Term Memory (LSTM) and deep Q-network [5] which makes the network capable of handling partial observability, and the recurrency inherent in the LSTM confers benefits when the quality of observations change during evaluation time. Recurrent DQN, unlike DQN, doesn't require multiple game frames as input and is capable of taking only a single input frame and integrating information across time using the LSTM layer. We describe the architecture in greater detail in Section 5.2. More recently, Wang et. al. [15] proposed a new architecture called the *Dueling Architecture DQN* which explicitly separates the representation of state values and state-dependent action advantages. It consists of two streams that represent state value function and action-advantage functions, while sharing a common convolutional feature learning module. The two streams are combined using a special aggregating layer to produce an estimate of the Q-value. This architecture is the current state-of-the-art in playing Atari games and achieves better than human-level performance in most games. This architecture is described in Section 5.3.

3 Background

Before describing the above architectures in detail, we lay out the required background for the reader. The background would briefly cover the important concepts in reinforcement learning and deep learning that can help the reader in understanding the later part of the report.

3.1 Reinforcement Learning

Q-learning,[16], is a popular learning algorithm that can be applied to most sequential tasks to learn the state-action value function. It is an off-policy learning algorithm as we don't execute the policy that we try to learn during training, instead we use a behaviour policy which is exploratory in nature. But the target action (or the desired action) is the one predicted by the policy learnt until now and we apply the following update rule iteratively at each time step t:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_t + \gamma Q(S_{t+1}, A') - Q(S_t, A_t))$$

where A_t is the action that the behaviour policy predicts in state S_t whereas A' is the action that our current learnt policy predicts. The above update rule can be derived from the Bellman optimality equation for MDPs, but that discussion is beyond the scope of this report.

We define another important quantity, the *action advantage* function which relates the state-action value function with the state value function in the following way:

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s)$$

Observe that $\mathbb{E}_{a\sim\pi(s)}[A_\pi(s, a)] = 0$. Intuitively, the value function V measures how good it is to be in a particular state s. The Q function, however, measures the value of choosing a particular action when in this state. The action advantage function subtracts the value of the state from the Q function to obtain a relative measure of the importance of each action.

Ideally, for small state-action spaces we can implement the state-action value function $Q(S, A)$ as a lookup table, which can be accessed whenever the agent has to take an action. But, for large state-action spaces a lookup table approach would be memory intensive and infeasible to implement. Most works deal with this by using a linear function approximator for state-action value functions i.e. they model $Q(S, A)$ as a linear function of a fixed set of learnt parameters. For increasingly complex tasks such as games where linear functions aren't good enough to model $Q(S, A)$, we have to resort to non-linear function approximators such as neural networks. In such a scenario, the newly developing field of deep neural networks, such as CNNs, can lend its capability to model highly non-linear relationships.

3.2 Convolutional Neural Networks

Convolutional neural networks[8], a type of deep neural network, are designed specifically for image input. The intuition driving CNNs is that a filter that is useful for a right top patch in the image would also be useful for a patch in the bottom right part in the image. This leads to the idea of weight sharing across the entirety of the image which dramatically reduces the number of parameters in a fully connected neural network which takes all pixels of an image as input. The other commonly used layer is a max pooling layer which is used as a method of sub-sampling to go from higher dimensions in the image to lower dimensions. Pooling transfers the activations from a patch of units in the previous layer to a single unit. If the activations are being averaged then it is called as average pooling. However, in practice, max pooling has been shown to be superior. In max pooling, the maximum activation in a patch is transferred to the unit in the next layer and the rest of the activations in that patch is discarded. In [8], deep neural network tries to learn a function to classify images of handwritten characters. However, the same network can easily be used as a function approximator by simply changing the loss function on the last layer from softmax loss to euclidean loss. In Deep RL, CNNs will be used as function approximators where in they will take an image or a stack of images as input and try to estimate a state-action value function from the visual input.

4 Deep Q Learning

In this section, we describe how deep learning has been applied to reinforcement learning in control tasks, describe the challenges faced and how they were resolved.

4.1 Loss function

The basic idea behind many RL algorithms is to estimate the optimal state-action value function, by using the bellman equation as an iterative update

$$Q_{i+1}(S, A) = \mathbb{E}[R + \gamma Q_i(S', A')|S, A]$$

As noted in section 3.1 for large state-spaces this approach is impractical to estimate state-action value function for each state-action pair independently. Instead, we use a function approximator to estimate the Q-values. In our case, if we use a neural network with weights θ as the nonlinear function approximator (Q-network), we can train the network by minimizing a **sequence** of loss functions $L_i(\theta_i)$ that changes at each iteration i,

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

where $y_i = \mathbb{E}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})|s, a]$ is the target from the iterative update given above, θ_i is the set of parameters at iteration i and $\rho(s, a)$ is the distribution induced by the behaviour policy. Note that the loss function keeps changing from iteration to iteration as the target changes, unlike supervised learning where the target is fixed for each data point and hence, the loss function is fixed before learning begins.

4.2 Challenges

Reinforcement learning presents several fundamental challenges from a deep learning perspective. In contrast to supervised learning setting, where large amounts of hand-annotated data is used to train the deep network, in RL, the algorithm must learn the parameters of the deep network from a scalar reward signal that is frequently sparse, delayed and noisy. The effect of action taken at the current timestep can be observed after thousands of timesteps, which is quite contrasting to what we see in supervised learning where there is a direct association between inputs and targets. Additionally, in supervised learning we assume that the input training data is independent of each other but in the case of RL, the input data is highly sequential and correlated with each other. This breaks the assumption in most supervised training procedures and hence, these procedures converge to a local minima or take too long to converge, in some cases. Also in RL, the incoming samples are not drawn from a stationary distribution, as it is usually assumed in supervised learning. The input data distribution keeps changing during training and the training algorithm has to accommodate the non-stationarity of the distribution. All these challenges are resolved by using *Experience replay* which was introduced in [9].

4.3 Experience Replay

Experience replay has been primarily introduced to break the correlations in the input data thereby making them independent of each other, so that supervised learning procedures can still be applied. This also results in the input distribution to be stationary as now subsequent data points are not sequential (or correlated with each other). At each timestep when the agent takes an action, the tuple $(state, action, next\ state, reward)$ is stored in a database ϵ.

After several thousand steps of random exploratory actions, the agent builds up a large database ϵ, which we refer to as the experience replay database. Subsequently, when the network tries to minimize the loss function to get more refined Q-values, instead of sending the data sequentially, we randomly sample a mini-batch of transitions from the database ϵ. The loss function is then minimized on this mini-batch of transitions. This ensures that the data sent for training (or backpropagation procedure) is not correlated and hence the procedure converges to a better minima. For a fixed number of such transitions, the Q-function is held fixed and is only updated for all the sampled transitions in a mini-batch at one go. This is similar to performing stochastic gradient descent using mini-batches. Hence, the new loss function is given by

$$\begin{aligned} L_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim \epsilon}[(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) \\ - Q(s, a; \theta_i))^2] \end{aligned}$$

5 DQN and its variants

In this section, we briefly describe the architectures of DQN and its variants that have been shown to achieve human-level performance in various control tasks.

5.1 Deep Q-Network

The architecture for deep Q-network introduced in [10], is shown in Figure 1. The network takes in the last 4 frames of the game, each of size 84×84. Note that these frames are sampled from the experience replay database ϵ during training. We use the last 4 frames as it can capture time-related features such as velocity etc. The first hidden layer convolves 16 8×8 filters with stride 4 with the input image and applies a rectified nonlinearity. The second hidden layer convolves 32 4×4 filters with stride 2, again followed by rectified nonlinearity. The final hidden layer is fully-connected and has 256 rectifier units. The output layer is fully-connected linear layer with a single output for each valid action which predicts the Q-value for that action in the current state. This network is trained using the popular backpropagation procedure.

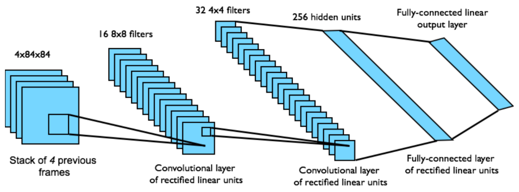

32 4x4 filters
256 hidden units
Fully-connected linear output layer
16 8x8 filters
4x84x84
Stack of 4 previous frames
Convolutional layer of rectified linear units
Convolutional layer of rectified linear units
Fully-connected layer of rectified linear units

Figure 1: Architecture of Deep Q-Network

5.2 Recurrent Deep Q-Network

As shown in Figure 2, recurrent deep Q-networks replace DQN's first fully connected layer with an LSTM layer. Also, note that the input to the recurrent DQN is a single frame, unlike 4 frames as is the case in DQN. The convolutional layers also have a different number of filters and stride length compared to DQN, as observed in the Figure 2.

The main motivation behind recurrent DQN is that in the case of games where the agent needs to remember events more distant than 4 frames in the past, DQN would fail to learn such an agent. In other words, any game that requires a memory of more than four frames will appear non-Markovian to DQN because the future game states (and rewards) depend on more than just DQNs current input. The LSTM layer in recurrent DQN serves to alleviate the problem by integrating information across time and allows distant events to be remembered by the agent.

Also, in the case of partially observed environments where the agent does not receive observations at every timestep, DQN cannot generalize to the unseen states. Recurrent DQN, on the other hand, uses the recurrency within the LSTM layer to better estimate the underlying state when no observation is received and generalizes well. Thus, recurrent DQN can also deal with POMDPs (partially observable markov decision process) unlike DQN.

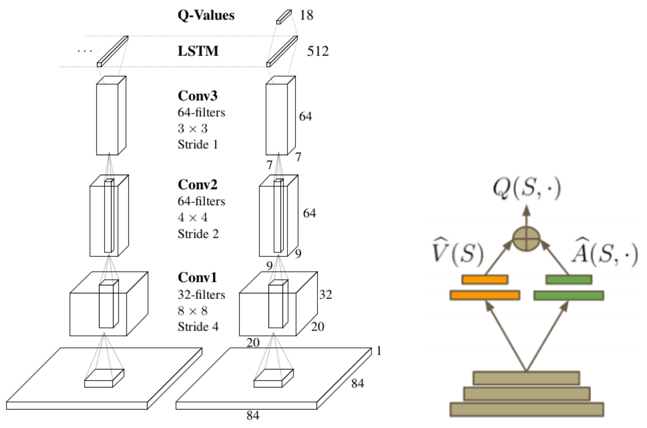


Figure 2: Architecture of Deep Recurrent Q-Network(left) and Dueling Network(right)

5.3 Dueling Architecture DQN

The Dueling architecture DQN is illustrated in Figure 2. The lower layers of the dueling network are convolutional as in DQN. However, instead of following the convolutional layers with a single sequence of fully connected layers, two sequences (or streams) of fully connected layers are used. The streams are constructed such that they have they have the capability of providing separate estimates of the value and advantage functions. Finally, the two streams are combined to produce a single output Q function. As in DQN, the output of the network is a set of Q-values, one for each valid action.

The two streams are aggregated using a special aggregation function given below:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a';\theta,\alpha) \right)$$

where θ denotes the parameters of the convolutional layers, α and β denotes the parameters of the two streams of fully-connected layers, and $\mathcal{A}$ denotes the action set.

The advantage of the dueling architecture lies in its ability to learn the state value function V efficiently. With every update for the Q values in the dueling architecture, the value stream V is updated. This is not the case in DQN, where only the value of one of the actions is updated, the values for all the other actions remain untouched. This more frequent updating of the state value function V allows for a better estimate and is essential when the number of actions is large.

Also, note that the difference of Q-values for a given state is quite small compared to the magnitude of the Q-value as the state value function dominates the magnitude (remember that $Q_\pi(s,a) = V_\pi(s) + A_\pi(s,a)$). This difference in scales can lead to small amounts of noise in the updates can lead to reorderings of the actions, and thus make the policy switch abruptly in a noisy fashion. The dueling architecture with its separate advantage stream is robust to such effects.

6 Experiments and Results

6.1 Setup

We have chosen the *Arcade learning environment*(ALE), [2], as our testbed. ALE consists of a collection of Atari 2600 games and provides an interface to interact with the games through a program

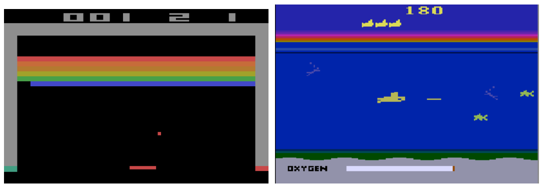


Figure 3: Screenshots of Breakout(left) and Seaquest(right)

instead of a controller. For any game, it provides a list of valid actions available for that game. We can then choose a given action and the ALE executes the action in the game. At every point in the game, we get the visual state of the game in the form of images of dimension 210×160. ALE also provides us a reward at each time which is usually the score in the Atari game.

We have chosen two games *Seaquest* and *Breakout* to test DQN and its variants on and compare their performances. Breakout is a relatively easy game where the player controls a paddle on the bottom and has to hit a ball that keeps moving to break tiles present on the top of the screen. The objective of the game is to break the most tiles without missing the ball during the entire episode. An example screen is shown in Figure 3.

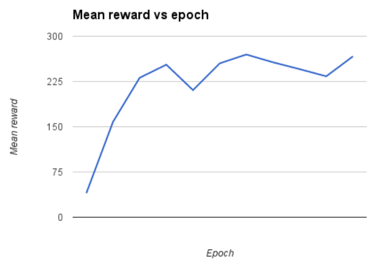
Seaquest, on the other hand, is a difficult game where the agent has to trade-off between killing fishes underwater gaining rewards and resurfacing to fill up oxygen. The objective of the game is to kill as many fishes as possible without running out of oxygen. The inherent trade-off between the actions of killing the fish and filling up oxygen (you cannot kill fish unless you are underwater and you cannot fill oxygen unless you are on the surface) makes the game difficult. An example screen is shown in Figure 3.

We implemented DQN and its variants using *Theano* and *Lasagne*, which are deep learning frameworks in Python. We interfaced our learning algorithms with ALE so that the learning algorithm obtains pixel values and scores from ALE and outputs action values to ALE. We use an ϵ (exploration factor) value of 0.9 at the start of the training and reduce it to 0.1 after a million timesteps. This is to encourage exploration at the start when it has no prior knowledge and to encourage exploitation after it builds some experience.

6.2 Results

For each game and each approach, we trained the agent for 30 epochs where each epoch consists of 1 million game timesteps. Each game took a training time of about 3-5 days on a modern GPU to complete the required number of epochs. As the epoch number increases, the agent gains experience and usually starts to play better than previous epochs.

In the game of Breakout, we observed that all three showed the same performance and there is no advantage in using recurrent DQN and dueling architecture DQN over DQN. Figure 4 shows the mean reward obtained by the agent trained using DQN, in each epoch (over the million timesteps). Note that the training is mostly stable with small dips in the performance across epochs, which mostly resulted due to the random exploratory nature of the agent.



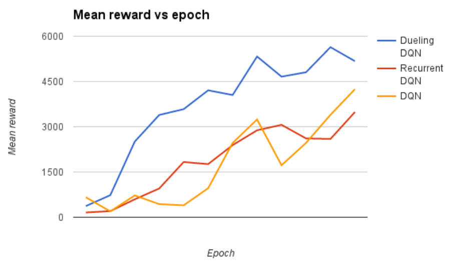


Figure 4: Results of DQN on Breakout(left) and Seaquest(right)

Figure 5 shows some of the filters learnt by DQN in the first convolution layer at the end of 30 epochs. We can see that the filters try to localize the position of the paddle and the ball on the game screen. For example, the second filter has the characteristic of finding a pixel with different value compared to its immediate surroundings. This is true for the ball in the game of Breakout which has a width of a pixel and is mostly against a dark background. Similarly, the other filters try to localize the position of the paddle and the tiles to get a better estimate of the current state of the game.

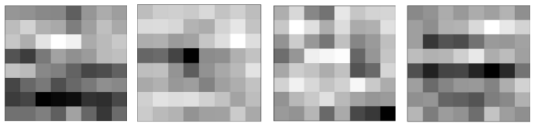


Figure 5: Filters learnt by first convolution layer for Breakout

We present the results of different approaches against each other on the game of Seaquest in Figure 4. Note that the dueling DQN approach completely overshadows all the other approaches in terms of its performance. Also, observe that the training is relatively stable than the other two approaches due to the more frequent updates of the state value function. Among DQN and recurrent DQN, the recurrent DQN is more stable in training as can be observed from the figure. DQN's performance is very erratic and noisy in comparison. Although, DQN ends up with a higher mean reward at the end of 30 epochs, the relatively stable nature of recurrent DQN is preferred as it is consistent. A video of the agents trained using our implementations of DQN and Dueling architecture DQN playing Seaquest is uploaded at `https://www.youtube.com/watch?v=TqIUM7qlDC0`.

6.3 Discussion

From the results, presented in section 6.2, we can observe that the dueling DQN performs better than both the recurrent DQN and DQN approaches. We reason that this is due to the frequent updating of the state value function V in the case of dueling DQN that lets it capture the value of each state more efficiently when compared to the other two. This efficient estimate is essential in the case of complicated games like Seaquest, as the state-space is huge and we might not observe the same state many times. In addition to this, since dueling DQN explicitly estimates the action advantage function, it doesn't run into the difference in scales issue (discussed in section 5.3) unlike DQN and recurrent DQN.

Recurrent DQN, on the other hand, has a more stable training curve than DQN as it integrates information across time more effectively. Hence, it captures long-range features in addition to short-range features, whereas DQN can only capture features that are spread within 4 frames of each other. But since the LSTM layer tries to account for long-range dependencies, the training procedure is slow.

7 Conclusion and Future Work

In this report, we present the recently introduced and rapidly developing field of deep reinforcement learning. We highlight the need for deep learning in reinforcement learning when dealing with real-life high-dimensional control tasks. The challenges in introducing the paradigm of deep learning into RL are detailed and we explain how modern approaches such as DQN tackle these challenges. Further, we have introduced a few variants of DQN such as Recurrent DQN and Dueling architecture DQN which show better performance in complicated games where there are long-range dependencies and huge state-spaces. Finally, we ran several experiments using the games of Breakout and Seaquest to compare the performance of these approaches and reason what happens under-the-hood in these algorithms.

As a future work, we aim to try out more recent variants such as hierarchical deep reinforcement learning [7], which have shown to capture extremely long-range dependencies in games like Montezuma's revenge that demand deep exploration in the initial stages of training. We also want to observe how prioritized experience replay, [12] works and examine its performance in games like Seaquest, when compared to Dueling DQN.

References

[1] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. *arXiv preprint arXiv:1512.02595*, 2015.

[2] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 2012.

[3] Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate to solve riddles with deep distributed recurrent q-networks. Technical Report arXiv:1602.02672, 2016.

[4] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[5] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *arXiv preprint arXiv:1507.06527*, 2015.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[7] Tejas D Kulkarni, Karthik R Narasimhan, Ardavan Saeedi, and Joshua B Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *arXiv preprint arXiv:1604.06057*, 2016.

[8] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[9] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993.

[10] 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.

[11] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. Multimodal deep learning. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 689–696, 2011.

[12] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[13] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[14] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

[15] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[16] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.