AI Attack: Learning to Play a Video Game Using Visual Inputs

Jerry Jiayu Luo
Stanford University
jerryluo@stanford.edu

Rajiv Krishnakumar
Stanford University
rajk@stanford.edu

Neha Narwal
Stanford University
nnarwal@stanford.edu

Abstract

In this project we present a purely supervised learning model for a gaming AI. We train a convolutional neural network based on AlexNet [9] on a self-generated dataset of "Tetris Attack" gameplay, which consists of both gameplay footage and player inputs recorded for each frame. We describe in detail the modifications we apply to this convolutional architecture to improve the performance of our model and subsequently the game play by the computer agent. After training the model, we then compare the agent's performance to various forms of random play. After training, we report a classification accuracy of 40%. We observe that we are overfitting our data even with regularization, and there is an imbalance in the classified actions. Despite these issues, our agents perform better than all forms of random play. We carry out exhaustive assessment of our results through a confusion matrix and saliency maps. We finish with a discussion of the current shortcomings in the model and also consider methods of building more complexity into our model for increased robustness.

Figure 1: A screenshot of the endless mode gameplay of Tetris Attack. The white outline around the purple and green block located near the center of the screen denote the cursor. In addition, the tops of the next rows of blocks that are pushing the current blocks upwards can be seen near the bottom of the screen. Each new row takes a few seconds to fully appear on the screen.

1. Introduction

One of the overarching goals of the deep learning community is to create AI that can make optimal decisions based on raw visual and audio input. In 2014, Google DeepMind successfully used convolutional neural networks (CNNs) with deep reinforcement learning to teach a computer to play classic score-based Atari games [12]. We propose to study a complementary problem: teaching a computer to play modern games through mimicry of human actions using gameplay screenshots as input data. Primarily, we propose training a CNN on the Super Nintendo game "Tetris Attack" in the "endless" mode using both gameplay frames and input keys generated from a human player (one of the researchers).

Tetris Attack is a puzzle game which consists of a grid of colored blocks along with a cursor. The player can move the cursor around to swap any 2 adjacent blocks horizontally. Whenever at least 3 blocks of the same color are placed adjacently in a row or column, these blocks disappear and points are awarded to the player. The aim of the game is to obtain as many points as possible by constantly making blocks disappear as they slowly keep refilling the screen from below (Figure 1). If any column of blocks reaches the top of the screen (and touches the wall), the game ends. Extra points are awarded if the player manages to make combos, e.g. make sets of 4 or more blocks disappear at once or create chain reactions in which a falling block causes a set of blocks to disappear after having landed next to 2 similar blocks.

At any point in time the player has a choice of 6 actions: they can press any of the directional buttons (Left, Right, Up or Down) to move the cursor in the corresponding direction, or they can press "A" which swaps the two blocks that the cursor is over, or they can press "R" which accelerates the rate at which the blocks refill from the bottom for a brief period. Although the concept of the game is fairly simple, the

optimal action for the player can vary wildly given a small change, similar to how for a Rubik's cube the action would be very different between two scenarios in which only two colors were swapped.

In this paper we discuss how we train and evaluate an agent that can play Tetris Attack using a CNN whose inputs are raw pixel values from a game screenshot. We first begin with a brief discussion on related works in section 2. Then in section 3 we describe in detail how our convolutional models were trained to predict the next move by employing supervised learning and the modifications we applied to the convolutional architecture to improve the performance of our model and subsequently the gameplay by the computer agent. We then discuss how we obtained gameplay data and measured agent performances in section 4. We then compare these approaches to assess the performance of our model and dive deep into the underlying causes for the current performance in section 5. This includes exhaustive assessment of our results through visualizations such as saliency maps and confusion matrices. Finally in section 6 we draw conclusions from our results and discuss future work that can be done to improve upon them.

2. Related Work

The concept of using CNNs to make real-time decisions based on visual inputs can be applied to a variety of fields. One of the earliest and still most active applications is in the field of self-driving cars [1, 3, 10, 7, 2] which also includes visualizing and interpreting the decisions made by the model [8]. Additionally, new applications have started to emerge in other fields including robot learning manipulation [18, 13] and forest trail detection [5]. The aim of all these models is to try to create an agent that can perform as well as a human in the desired tasks by training it using supervised learning. This is done by training a CNN on a set of images representing what a human saw at a given moment along with the corresponding action taken by the human at the time. Recently, this method is also being used to train agents to play video games. The data for this is gathered by recording both the screen and the buttons pressed while a human expert is playing the game. The CNN can then be trained using only this data. This approach was already shown to be very successful in action games, ranging from simple games where there were only possible 2 actions [11] to more complicated games in which there were 30 possible actions [4]. We propose to implement this method in a puzzle game with 6 possible actions. As already alluded to in section 1, substantial work has been done in creating video game playing agents using Deep Q-learning networks (DQNs) [12, 15, 6, 16]. This work has in fact created agents that can outperform humans. However our aim, similar to the aim of Chen & Seff [4], is to work on improving the agents trained purely by supervised learning methods, which require much less time and resources than methods using DQNs.

3. Methods

3.1. General Approach

We used CNNs for our classification problem, where the input was the raw pixel values of a game screenshot, and the output was the log unnormalized probabilities of the 6 different possible player actions (Equation 2). The entire dataset consisted of about 17,000 screenshot/action pairs, and our validation set consisted of 500 randomly selected pairs from that dataset. This way we were able to use as much of our limited data as possible to train our model. We minimized the softmax loss on the training set (Equation 1), which is the cross-entropy between our estimated distribution of class probabilities and the true distribution of class probabilities (which should have p = 0 everywhere except a p = 1 for the our correct class).

Softmax loss function:

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}\right) \tag{1}$$

Estimated class probabilities:

$$P(y_i|x_i; W) = \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}} \tag{2}$$

3.2. CNN Architecture

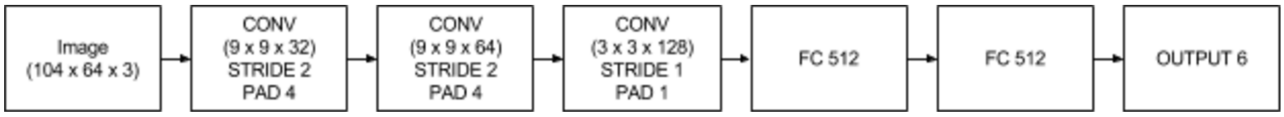
The foundational CNN architecture for our model is inspired by AlexNet [9].

The first model we tried uses two small 3x3 filters joined by 2x2 max pool layers. The convolutional layers are followed by fully connected layers, as shown in Figure 3a. We labeled this as the narrow model, since the total receptive field is only 7x7 after the convolutional layers.

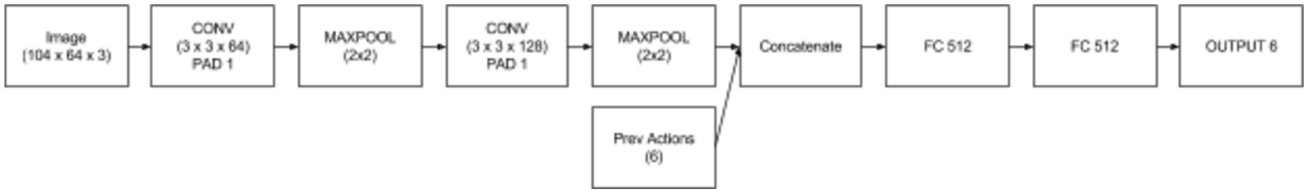
To improve the accuracy of our model, we modified the architecture to capture a larger receptive field. The second model uses 9x9 filters with stride 2 without max pool layers, followed by a 3x3 filter and then fully connected layers, as shown in Figure 3b. We labeled this as the wide model, since the total receptive field is 33x33 after the convolutional layers.

The third model uses both the current screenshot as well as the previous action to predict the next action (Figure 3c). This model tries to capture the intent of the player from the previous action, since without context, it is possible that there could be multiple objectives that the agent can accomplish, where each objective may take multiple steps to achieve. Not having this context could mean the agent will not be able to focus on a single objective. We labeled this as the appended model, since it appends the representation of the image with a one-hotted version of the previous action.

(a) The narrow model



(b) The wide model



(c) The appended model

Figure 2: Depiction of the three different CNN architectures used to train the agent.

3.3. Model training

Our models were trained and tested using the Tensorflow framework with an Nvidia GTX 1060 GPU. We trained using stochastic gradient descent, with mini-batch sizes of 16. We used the Adam optimizer with a constant learning rate, as we found that annealing it didn't improve the results. We optimized both dropout and L2 regularization to maximize the validation accuracy. Our final hyperparameters for all 3 models were :

- Learning rate: 1×10^{-4}
- β_1: 0.9
- β_2: 0.999
- ϵ: 1×10^{-8}
- Dropout keep probability: 0.5
- L2 regularization: 1×10^{-2}

To assess the performance of the models as they trained, we monitored both the loss and the prediction accuracy of the training and validation datasets at every 100 steps (see subsection 5.1 for more details). We ran for a total of 10,000 to 20,000 steps, depending on the architecture.

4. Experimental Setup and Datasets

Our datasets were gameplay frames mapped to key inputs from a human player. We will describe two ways in which we collected the data and discuss their pros and cons.

Initially to obtain this data, we played the game on a computer using an emulator and open an instance of the Keyboard Utility Tool while simultaneously capturing frame and keyboard input data using Bandicam Screen Recorder. A total of 15,415 frames along with their corresponding keypresses were captured. This method allowed us to gather data efficiently (i.e. we would be able to get a lot of data in a reasonable amount of time), however some of the frames would essentially be duplicated. This would happen because occasionally we would hold down a button for multiple frames and hence create several data points (one for each frame) which looked almost identical.

The second way of capturing the data was using screenshots instead to extract frames from videos. Every time a button was pressed, a screenshot of the gameplay would be captured along with the corresponding button press. This would avoid the issue of capturing multiple similar looking frames for each keystroke. At first, 46,875 frames were captured using Bandicam. However the delay time between the button press and the screencapture was varying and so we would not always be able to capture the correct frame for the corresponding button press. Therefore we switched over to a Linux operating system so that we could employ the 'Screencapture' and 'evdev' packages to capture about 17,436 frames. There were still small time delays between the button press and the screen capture (and our data still had some noise) but the noise was minimized and the frame to key inputs were more consistent.

In all methods, the captured frames were cropped and down sampled (using bilinear interpolation [14, p.123-128]) to an image size of (104 x 64 x 3) which was subsequently used to train the model. This image size was a compromise of reducing the number of features (pixels) as much as possible whilst still having the cursor be clearly visible in each picture. The bilinear interpolation was done by interpolating pixel color values and introducing a continuous

transition into the output even where the original material has discrete transitions.

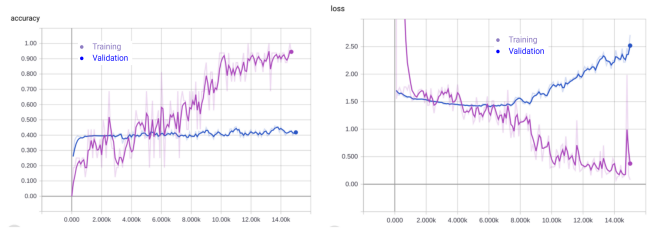
After collecting the data, we trained our models (described in subsection 3.2) and evaluated the accuracy on the validation set for each one (which we discuss in section 5). Then we used the models to create agents and that played the game. To do this, we first take a screenshot of the game (using the 'Screencapture' package similarly to when training the data). We then downsampled the image and used it as an input for a single forward pass through our model. The agent would then implement the action suggested by the model. This whole procedure took approximately 50ms to do, and was repeated until the game ended after which the final score would be recorded. We evaluated each agent's final score 10 times to smooth out random noise. Our models are efficient enough that the limiting factor in the agent speed is the time it takes to capture the screen and downsample it (as opposed to the forward pass through the CNN).

5. Results

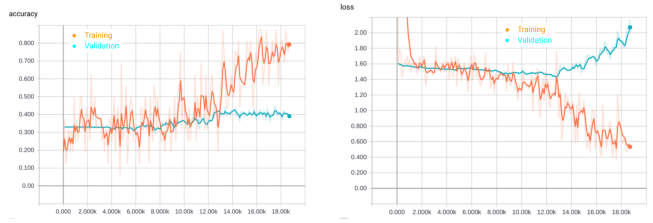
5.1. Training Results

The training results for all three models appear to be overfitting, even after trying various levels of dropout and L2 regularization. A very high level of either dropout or L2 regularization would result in the training accuracy getting worse without any notable improvements in the validation accuracy. For all models the training accuracy improved to around 90%, while the validation accuracy capped at around 40%. Both the narrow and wide models had similar training results, where both the training and validation accuracies improved for the first 10,000 steps or so, but soon afterwards the training accuracy started improving very quickly while the validation accuracy stayed constant. For the appended model, the training accuracy overfitted very easily. Even though the validation loss goes down significantly, the validation accuracy doesn't improve at all.

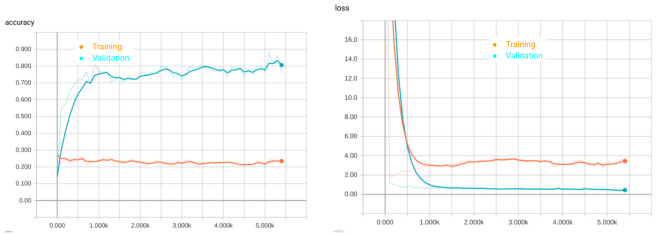
This overfitting suggests that the data isn't rich enough and that we need to gather more data. For both the wide and narrow model, the fact that the validation loss decreases for the first 10,000 steps indicates that at that moment there is still information to be learned from the training set. After 10,000 steps, we've extracted all we can from the training set and further training will only result in overfitting. Having more data will mean that we will have more examples that contain various configurations the model isn't familiar with. One possible explanation for why even 17,000 samples isn't enough is that the game is strategically complex. The game requires multi-step strategic planning that requires looking far in advance, rather than requiring fast and accurate movements while looking at near future moves, such as for games like Breakout or Space Invaders. Even if we had a representation of the blocks themselves, finding the optimal combination of moves is its own AI challenge.



(a) The narrow model results



(b) The wide model results



(c) The appended model results

Figure 3: Plots showing training and validation accuracies and losses for the various models as a function of epoch.

5.2. Confusion Matrix

We calculated the confusion matrix for the wide model in order to observe how well the model was classifying certain actions Figure 4. The model tends to predict action A much more than any other action. For each actual action (ignoring R for now), the highest predicted action was either the true action or A. (There is an exception for Up but it is very close) This suggests that the model is over aggressively classifying actions as A. Further analysis of the precision and recall of each action (in Table 1) shows that A's recall is higher than its precision while every other action's precision is higher than its recall. This corroborates the theory that the model is over-aggressive in classifying actions as A and over-cautious in classifying actions as anything else.

In the data that we collected there were very few times that R was pressed, so there isn't much data for the model to work with when predicting R, which is why the model never predicted it correctly.

Ignoring actions A and R and just looking at the directional actions, we see that the predictions are actually pretty good. The only action that had poor classification perfor-

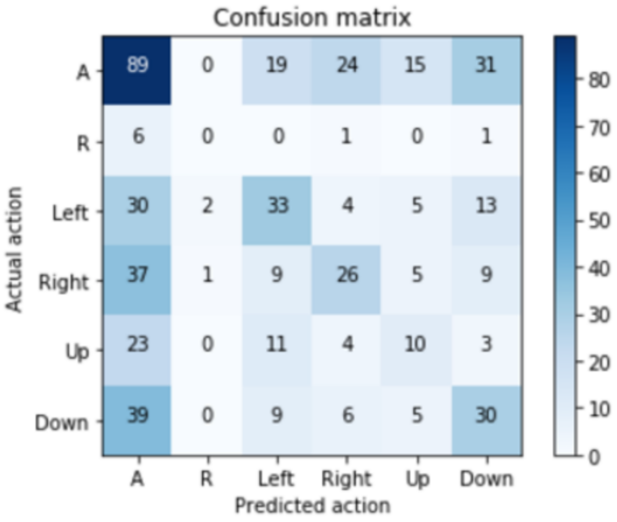


Figure 4: The wide model confusion matrix.

	A	R	left	right	up	down
Precision	0.40	0.00	0.41	0.40	0.25	0.34
Recall	0.50	0.00	0.38	0.30	0.20	0.33

Table 1: Wide model classification model

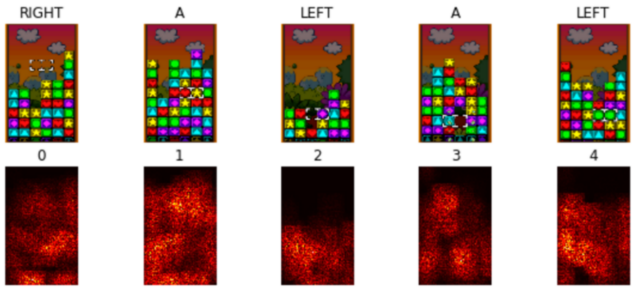


Figure 5: Examples of saliency maps from the wide model.

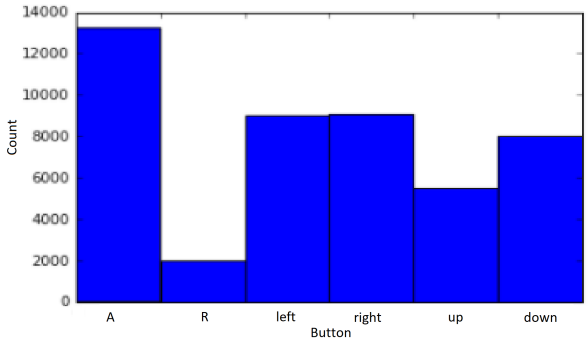


Figure 6: Histogram of training data actions.

mance was Up, whose recall was bad because it was oftentimes classified as Left. A possible explanation for this is Up was the least present directional action in the dataset, so just like R we have fewer data points to learn from.

5.3. Saliency Maps

We computed the saliency map for the wide model in order to observe which pixels matter for classification. This is achieved by taking the gradient of the correct action's score with respect to the pixels of the image, while keeping the weights of the model fixed [17]. By doing this we can observe which pixels in the image correspond to the largest change in the correct class's scores.

In the salience maps in Figure 5, the red colors correspond to where the pixels affect the scores. Brighter red indicates larger gradients. The saliency maps seems to be bright only where the blocks are, which indicates that the model has learned to ignore the background. However, it should also light up extra bright where the cursor is and where potential combos are, but unfortunately that is not the case. It is unclear why the brightest spots are where they are. The best current explanation is that they are due to noise and the lack of data richness. Given enough training examples so that we aren't overfitting, we expect the saliency maps to be brightest at the location of the cursor and potential combos.

5.4. Gameplay

After training, we use the predictive models to actually play the game and compare the agents' performances.

The first agent we test performs no actions. As expected we consistently get 0 points.

We then test an agent with a uniform random policy, where each of the 6 actions is sampled with $\frac{1}{6}$ probability during each step. This also gave very poor performances, with an average of 6 points per game. Pressing R speeds up the game and also gives a nominal amount of points, and this agent presses R more than any others. This causes the agent to die quickly, as well as get those 6 points. This agent is never able to get a real combo.

Next we test an agent with a weighted random policy, where the probability of the 6 actions is modeled by the probability distribution of the training dataset Figure 6. This agent does significantly better than the uniform random agent, mostly because it doesn't press R as often and wouldn't kill itself quickly. It would oftentimes go up into the sky where there are no combos and get stuck there. The variance of the score ended up being very high since sometimes it would randomly get lucky combos.

We then test both our narrow and wide models, which

	Do Nothing	Uniform Random	Weighted Random	Narrow Model	Wide Model	Appended Model
Mean	0	6	132	190	212	15
Standard Deviation	0	0	179	145	197	26

Table 2: Wide model classification model performance

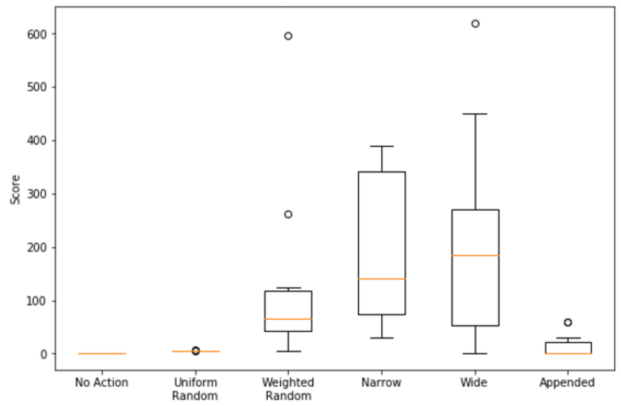


Figure 7: Agent performances.

both seem to have similar performance. Although they also get lucky combos, they subjectively seem to be slightly more consistent and deliberate than the weighted random agent. In addition these agents tend to consistently keep the cursor on top of the blocks and rarely get stuck in the sky doing nothing. As a result, the scores for both the narrow and wide models are slightly higher than the scores for the weighted random agent. Consistent with the analysis done with the confusion matrix, the model tends to press A a lot, and would sometimes press it multiple times in the row, even if there are no combos available in the current location. Sometimes this would go on for 10+ moves. Another interesting observation is that sometimes it would look like the agent is trying to reach for a goal a few steps away, but then all of a sudden change directions to do something else. One possible explanation for this is that since the model has no history of the previous moves, at every step the agent is starting over to look for a combo, which makes it unable to pursue a goal for multiple timesteps.

Lastly we tested the appended model, whose purpose was to mitigate this 'short term memory loss' problem, but its performance ended up being very poor. This agent does seem to keep a more stable movement trajectory by doing multiple consistent directional actions in a row, but would end up getting stuck on the edges a lot (e.g. trying to move down when it's already at the bottom). Getting stuck seems to be the main reason why the performance of the appended model was so bad compared to the others. The full summary of our results can be seen in Table 2 and Figure 7.

6. Conclusions and Future Work

In this project we trained agents to play Tetris Attack using only supervised learning, and then compared the agent's performance to various forms of random play. We used CNNs of various architectures to classify the predicted next action given the current game screenshot. We discovered that we are overfitting our data even with regularization, and there is an imbalance in the classified actions. Despite these issues, our agents perform better than all forms of random play.

There are many ways we can improve the agent's performance. One obvious action that would lead to better results is to gather more data. We used around 17,000 samples to train our models, and the overfitting indicates that we may not have enough.

Since the confusion matrix analysis shows that there are many false positive A predictions, we can change our cost function such that it penalizes false positive A's more. This should reduce the aggressiveness with which the model predicts A, and hopefully reduce the amount of time the agent is stuck pressing multiple A's consecutively.

In our appended model we just concatenated the post-convolutional representation of the image with a one-hotted representation of the action. This resulted in concatenating a 512 dimensional vector with a 6 dimensional vector, which led to an unbalanced amount of representation between the image and the action. Also, just using the one-hotted representation of the action failed to capture any possible features between the actions in a higher dimensional embedding space. One possible solution is to embed the actions into a 512 dimensional embedding space in order to both balance the amount of representation and to capture potential higher level action features.

We would like to thank Joe Chen for his valuable help and insight. All the code used to write this paper is available at `https://gitlab.com/cs231n-spring2017`.

References

[1] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars. *arXiv:1604.07316 [cs]*, Apr. 2016. arXiv: 1604.07316.

[2] M. Bojarski, P. Yeres, A. Choromanska, K. Choromanski, B. Firner, L. Jackel, and U. Muller. Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car. *arXiv:1704.07911 [cs]*, Apr. 2017. arXiv: 1704.07911.

[3] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving. pages 2722–2730, 2015.

[4] Z. Chen and D. Yi. The Game Imitation: Deep Supervised Convolutional Networks for Quick Video Game AI. *arXiv:1702.05663 [cs]*, Feb. 2017. arXiv: 1702.05663.

[5] A. Giusti, J. Guzzi, D. C. Cirean, F. L. He, J. P. Rodrguez, F. Fontana, M. Faessler, C. Forster, J. Schmidhuber, G. D. Caro, D. Scaramuzza, and L. M. Gambardella. A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots. *IEEE Robotics and Automation Letters*, 1(2):661–667, July 2016.

[6] M. Hausknecht and P. Stone. Deep Recurrent Q-Learning for Partially Observable MDPs. *arXiv:1507.06527 [cs]*, July 2015. arXiv: 1507.06527.

[7] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue, F. Mujica, A. Coates, and A. Y. Ng. An Empirical Evaluation of Deep Learning on Highway Driving. *arXiv:1504.01716 [cs]*, Apr. 2015. arXiv: 1504.01716.

[8] J. Kim and J. Canny. Interpretable Learning for Self-Driving Cars by Visualizing Causal Attention. *arXiv:1703.10631 [cs]*, Mar. 2017. arXiv: 1703.10631.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[10] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[11] J. Lewis. Playing Super Hexagon with Convolutional Neural Networks.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv:1312.5602 [cs]*, Dec. 2013. arXiv: 1312.5602.

[13] A. Nair, D. Chen, P. Agrawal, P. Isola, P. Abbeel, J. Malik, and S. Levine. Combining Self-Supervised Learning and Imitation for Vision-Based Rope Manipulation. *arXiv:1703.02018 [cs]*, Mar. 2017. arXiv: 1703.02018.

[14] W. H. Press, editor. *Numerical recipes in C: the art of scientific computing*. Cambridge University Press, Cambridge ; New York, 2nd ed edition, 1992.

[15] K. Ranjan, A. Christensen, and B. Ramos. Recurrent Deep Q-Learning for PAC-MAN.

[16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan. 2016.

[17] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. *arXiv:1312.6034 [cs]*, Dec. 2013. arXiv: 1312.6034.

[18] Y. Yang, Y. Li, C. Fermuller, and Y. Aloimonos. Robot Learning Manipulation Action Plans by "Watching" Unconstrained Videos from the World Wide Web. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 3686–3692, Austin, Texas, 2015. AAAI Press.