Paul Steenkiste
Jaak Joonas Uudmäe
December 15, 2016

CS221 Final Report: Doom AI

Introduction

The goal of our final project was to build an agent that would master challenges inspired by the 1993 computer game Doom. We trained it to do such tasks as finding enemies, shooting them, and dodging incoming fireballs. We used environments designed for the OpenAI Gym to train and test our agent. The OpenAI framework provides the screen buffer for the current moment in the game, and allows us to input an action (say, 'ATTACK' or 'MOVE_LEFT'). The system then performs that action in the game environment and gives us the next moment's screen buffer. We were able to train our agent to perform very well on three of these environments.

Literature Review and Comparison

There has been a lot of work done on building AI agents specifically to master Doom. Every year a group called ViZDoom, aiming to advance Reinforcement Learning research, holds a competition in which agents play against each other in a "deathmatch" Doom tournament. This competition was most recently won by teams from Facebook and Intel this past September.

Like in many areas of Artificial Intelligence, almost all of the submissions to this competition rely on deep neural networks with many thousands of hidden layers to learn optimal strategies from the raw pixel arrays. As we will detail in the following sections, we approached our three tasks quite differently, opting instead to utilize more classic RL techniques, such as Value Iteration and Q-Learning. There are a few reasons for this choice. First, we simply did not have the resources or time required to train such massive networks. Second, we wanted to see if the more fundamental and intuitive algorithms we have seen in this course could be successfully applied in a domain that has been dominated by large corporations wielding behemoth black-box neural nets.

For Level 3, we actually did decide to implement a neural network, but even here our approach diverged from any of the submissions to the ViZDoom competition. Our neural network had only a single hidden layer which we trained only for a few days (as opposed to the many hundreds of hours required to train the other Doom agents). Because of our method of preprocessing the pixel array before sending it to the neural network, though, our agent was still able to perform very well.

That said, there are of course limitations to our more classic approach. First, by their very nature, our techniques are not wholly generalizable. We used our knowledge of what needed to be accomplished (e.g. "find the monster and shoot it") to formulate the state space, instead of just using the raw pixel array. While our techniques (using cross correlation to find relevant objects, reducing the state space through pixel manipulation, etc.) can be applied to a wide variety of games, they will need to be tailored for each specific task at hand. A corollary limitation is that our agents have only mastered the levels they were designed for and trained on. The goal of the ViZDoom competition is to have an agent that can interact with a full environment of the Doom

universe that it has never seen before, filled with every kind of enemy and weapon. While the policies learned by our agents on their individual levels can perhaps be used to build a full Doom player, it is not obvious how to do so.

Level 1: Basic

1.1 The Challenge

To build our intuition of the challenges of the project and what methods are and are not successful in overcoming those challenges, we began with a simple level, in which there is a single monster in the room and the goal is simply to shoot it as fast as possible. In each state our agent can take one of the three actions: ATTACK, MOVE_LEFT, or MOVE_RIGHT. OpenAI defines the rewards as follows: -1 every 0.028sec for waiting around, -5 for missing a shot, +100 for killing the monster. They also defined the threshold for "beating" the level as averaging 10 points per episode every 100 episodes, which corresponds to killing the monster in 3 seconds with a single shot.

1.2 Model

Before implementing any algorithms, we had to decide what features to extract from the given screen buffer, as using each unique screen buffer without doing any feature extraction would have given us much too large of a state space, making state exploration impossible. We chose to extract the horizontal distance (in pixels) from the monster to the center of the screen. Modeling our feature extractor after the identityFeatureExtractor from the Blackjack assignment, instead of using the raw 'distance' to the monster as our feature, we decided to have the features be every possible pair of (distance, action), and the value be the indicator function: '1' if our current distance and action matched.

To calculate the distance to the monster for a given pixel array, we used cross-correlation, which compares a template against a larger image by "sliding" it across the image and computing a similarity score at each position. Then, it returns the "most similar" position,

indicating that the object in the template is most likely to be at that position in the image (see Figure 1).

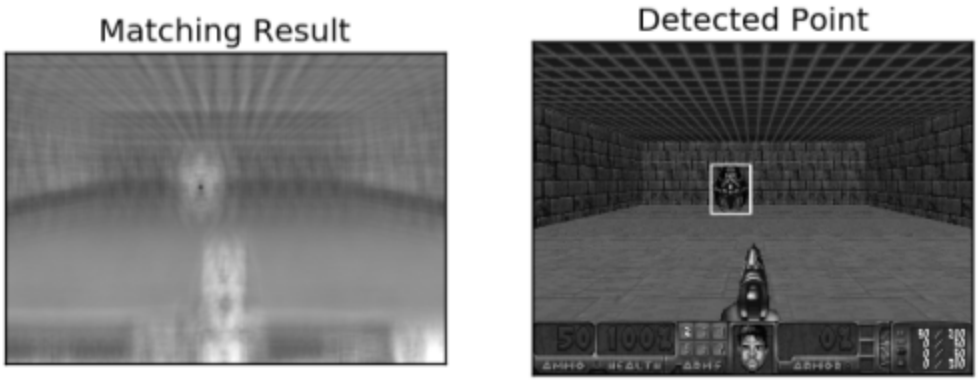

Figure 1: Cross-correlation measures of the screen on the right. The darker the point, the more "similar" the pixels.

Figure 2: Templates used for template matching

In our case, there were three possible orientations of the monster (see Figure 2), and so we performed cross-correlation using all three templates and took the global maximum. To speed up this process, instead of searching for the monster over the full screen buffer, we only searched over the middle third of the screen, skipping the areas where the monster could never be. We also downsampled the screen to make the search space smaller.

Doing this, we were able to correctly find the monster every time. Thus, our state space was reduced to { $dist_{min}$, … , -1 , 0 , 1 , … , $dist_{max}$ }. However, we did not know what the minimum and maximum distances to the monster were, nor precisely how these distances correspond to the game world (e.g. is a distance of -7 close enough that performing ATTACK will kill the monster and yield the +100 reward?). This brings us to our first algorithm.

1.3 Algorithm 1: Model-Based Monte Carlo and Value Iteration

To learn the optimal policy, we have to know how our actions change our state, and the rewards we receive as a result. In order to learn the transition probabilities and rewards of our new state space, we performed Model-Based Monte Carlo simulation. The data learned through this simulation then defined a Markov Decision Process.

To solve the resulting MDP, we implemented Value Iteration, modifying the code we wrote in the Blackjack assignment to suit our needs.

1.4 Analysis of Results, Algorithm 1

Before starting, we implemented two "baseline" agents to compare our results against. The first is simply a random actor: at each state it choosing from the 3 possible actions uniformly. This yielded an average reward of -163. The second baseline agent alternated between moving right and shooting - that way, it is sure to shoot the monster if it spawns on the right side of the room. This actually got an average reward of -529, much worse than the random actor. We realized that this is because roughly half the time the monster spawns on the left side of the room, and so the agent shoots all its ammo and time runs out, yielding a score of -1250. Because the random agent performed better, we used it as our baseline.

We also established an "oracle" before working on the problem. Because humans do not play these challenge levels, we had to look online to other submissions to see the best of what has been done before. The maximum average reward achieved in the past was 80, and so we set this as our oracle.

Our first algorithm proved to be highly successful: our Monte Carlo simulation yielded very accurate predictions of the transitions and rewards, and then Value Iteration came up with the intuitive strategy: shoot if the monster is in front of you, and move in the monster's direction if not. After Value Iteration, our agent received an average reward of 79.4, well above the baseline as well as the threshold for "beating" the level set by OpenAI.

1.5 Error Analysis, Algorithm 1

Our performance, however, was not quite as high as those of past submissions. We attribute this to the fact that the Monte Carlo simulation did not yield perfect data: the OpenAI environment is imperfect, sometimes giving the wrong reward (for instance, giving -1 when we killed the monster). Running the simulation for more iterations would probably have improved the data, averaging out the glitches.

1.6 Algorithm 2: Q-Learning

We aimed to surpass this performance using Q-Learning to learn the optimal policy while still exploring the state space. We again modified the Q-Learning code from the Blackjack assignment to work with our model. Given each screen buffer, after extracting the distance to the monster as our feature, we updated the Q-value for the current (state,action) pair, choosing our action using an epsilon-greedy approach, with epsilon slowly decaying as time went on (so that we would explore less and less as we learned how to interact with the environment).

1.7 Analysis of Results, Algorithm 2

After training for a few thousand iterations, the performance of our agent actually surpassed our oracle of the best previous implementation, achieving an average score of 84.02 (see Figure 3). Our agent ostensibly performed flawlessly, always immediately moving towards the monster and shooting at the earliest possible time to kill it.

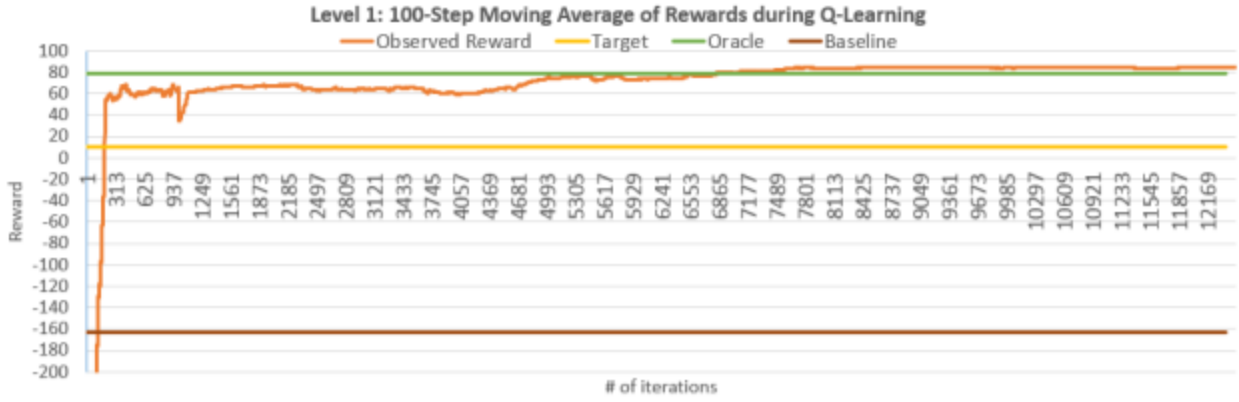

Figure 3: Level 1: Basic Rewards Using Q-Learning

A demonstration of our agent in action is uploaded to YouTube:
https://youtu.be/hlIJUFfRw0c .

Level 2: Defend Center

2.1 The Challenge

For our second challenge, we chose a slightly more difficult level: here, instead of just having to shoot one enemy, our agent must turn around and shoot as many enemies as possible with the provided 26 bullets. The agent is positioned in the center of a circular room with enemies approaching and attacking from all sides. Our agent can take three actions: ATTACK, TURN_LEFT, and TURN_RIGHT. The rewards specified by the OpenAI website are: +1 for killing an enemy, and -1 for dying. We also manually added punishment for missing a shot (-1 points) to speed up learning. OpenAI sets the target for "passing" the level to be averaging 10 points per episode every 100 episodes, which corresponds to killing 11 enemies before dying.

2.2 Model and Algorithm

Again, we decided to harness the Q-Learning algorithm. We needed a different feature set, however, since there could be multiple enemies on the screen at once (or none at all). Therefore, we tried a different computer vision approach for feature extraction. In this level there

were two types of enemies, who could be in any orientation or distance away from the agent. At first we tried to use a scale- and rotation-invariant algorithm to match templates, but soon realized that there were no fast algorithms that could achieve that. We ultimately decided to do the opposite of what we did in the previous level: instead of finding the best match with an enemy template, we performed cross-correlation across the center of the screen using the wall texture as our template. Areas where this value is low indicate that the wall is not there, and thus that a enemy is present. We divided the screen into ranges of "enemy" and "not enemy" ranges (see Figure 4). We did not have to downsample the screen this time, as our search region was already so small.

We chose our state space to be the position of the largest "enemy" range (i.e. the closest enemy) relative to us. Thus our state space was { LEFT , RIGHT , FRONT , NONE }. We did this because intuitively the agent should care more about enemies that are closer (and are therefore more of an imminent threat).

Similarly to the basic level, our features were tuples of (enemy_direction, action), and we searched the space and updated the Q values using an epsilon-greedy approach, with decaying epsilon.

Figure 4: Monster Detection in Level 2: Defend Center

2.3 Analysis of Results

To establish a baseline for this level, we had an agent alternate between turning right and shooting, so that it would kill any enemies to its right. Similarly to our baseline for Level 1, this seems like the best that an agent can do without any learning. This yielded an average reward of 2.75. The baseline approach actually worked relatively well, but the agent wasted a lot of bullets. We wanted our agent ultimately to only shoot when there was an enemy in front (and it ended up learning exactly that). Our oracle was set to 25, which is equivalent to not missing a single shot.

After running the learning algorithm overnight, the agent was able to perform better than both our baseline and OpenAI's specified target, with an average reward of around 16. Our agent took much longer to reach the target on this level than on the previous one because it was a more complex problem, as the agent has to shoot multiple enemies and can die itself.

A demonstration of our agent in action is uploaded to YouTube:
https://youtu.be/llbDPmmqEHw .

Figure 5: Level 2: Defend Center Rewards using Q-Learning

2.4 Error Analysis

Compared to other submissions on the OpenAI website, we did relatively well: some of the submissions exceeded our average, but most of them were lower. Our agent hardly ever died before running out of ammo. The main issue it faced was that it missed some shots, thus wasting bullets and running out of ammo faster. However, it only missed shots when the target was considerably far away - whenever the target was close, it always hit it.

Our agent seems to have reached a ceiling with our current approach and feature detection, stalling at around the 16-point mark. After watching our agent perform, we realized that it often shoots while being slightly misaligned with one type of the enemy. The problem is with our feature extractor: the shape of one of the enemies makes the range in which we 'detect' an enemy bigger than what the true "hit" range of the enemy actually is. This makes our feature extractor think that the enemy is in front when it is slightly to one side, causing the agent to miss. We tried many different thresholds and heuristics for determining that an enemy was in front, but with no success. Thus, a better method of feature extraction, or better parameter tuning, is needed in order for the agent to perform better.

Level 3: Take Cover

3.1 The Challenge

In this level our agent's only goal was to survive. On the opposite side of the room, monsters spawn at random locations at random times and shoot fireballs at the agent. It takes one to three fireballs to kill the agent. This level was the most challenging out of the three since there could be any number of fireballs on screen at any time. Furthermore, the fireballs could be incoming from an area that our agent does not see. For example, when there's an enemy on the far-right and we stand against the far-left wall, we cannot see that enemy. Our agent could take one of two actions: MOVE_LEFT or MOVE_RIGHT. The only reward specified by the OpenAI website was +1 for every 0.028sec survived. The threshold for "beating" this level set by OpenAI is to survive 20 seconds on average. This level caught our attention because there were no submissions on the OpenAI website that had solved it; we aimed to be the first to solve it.

3.2 Model and Algorithm

Trying to model the position of every fireball on the screen would have made the state space far too large for an algorithm such as Q-Learning to be feasible. Thus, we decided to use a neural network to learn the optimal strategy.

The neural net we constructed was a very simple one, with only one hidden layer of 1000 neurons. It therefore would not have been able to make sense of the full raw pixel array, and so we had to do some preprocessing. We noticed that the pixels of the fireballs were the only ones whose red channel ever exceeded a certain threshold. Thus, we eliminated the green and blue channels of the provided pixel array, and then filtered the red channel so that only the pixels of the fireballs remained (see Figure 6). We then set those values to be 1 and the rest to be 0 (to put it into an easy format for the neural net)

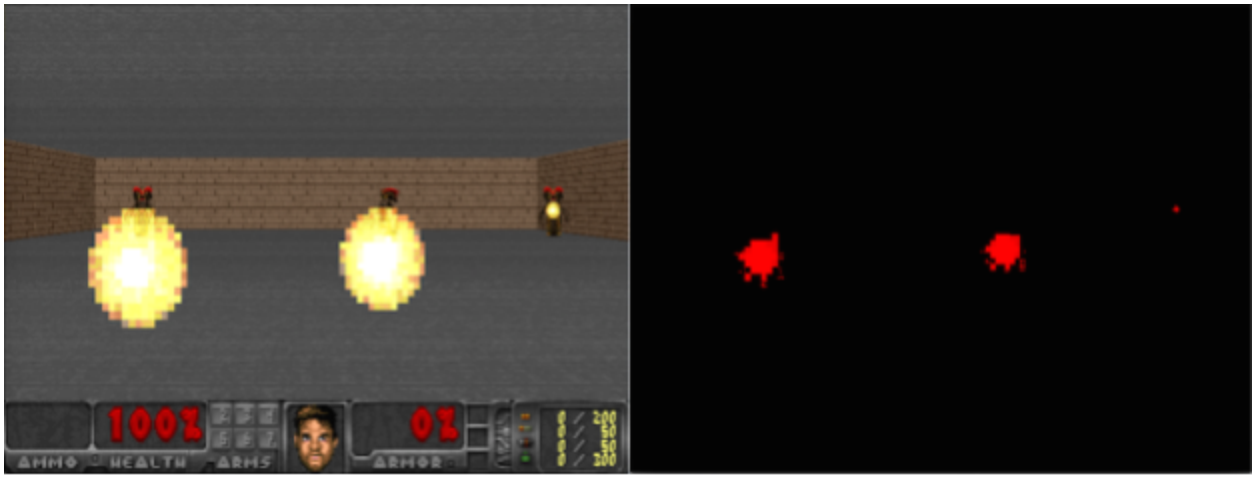

Figure 6: Fireball Detection in Level 3: Take Cover

One final pre-processing detail: in order to capture the motion of the fireballs, what we actually passed to the neural network was the difference between two subsequent frames. Thus, it

would be able to discern the path of the fireball. This is important, as whether the two closest fireballs in the above screenshot are going right or left is pivotal in deciding whether to move right or left. The position of the fireballs at a single instant is not enough information.

Now, instead of implementing a Deep Q-Network like Google's DeepMind group did to learn the Atari games, we decided to implement Policy Gradients using a neural network. The idea, however, is very similar. To determine which action to take, we preprocess the current game screen and send it to the neural network. It returns a single value between 0 and 1 (where we use the sigmoid function to confine the output to that range) which represents the probability that we should take the action MOVE_RIGHT. We then sample an action using that probability. The fact that our action is randomized in this way encourages exploration, much like our epsilon-greedy method in previous levels: it discourages us from getting stuck in local optima.

Training is done as follows: we run through an iteration of the level, and observe the reward at the end. We then retrospectively update the weights in the neural network for every action we took along that path using backpropagation. We take the gradient for each action to be the reward we earned at the end of the run; thus, actions we took during episodes in which we earned a high reward will be more likely to be repeated. The idea is that, on average, good actions lead to high reward while bad actions lead to good reward. While it certainly happens that a good action is taken along the way to a bad reward and thus the good action is unjustly punished, over a long training period these occurrences are mitigated, and the neural net learns.

In order to implement policy gradients with a neural network, we modified the code provided by Andrej Karpathy on his blog post "Deep Reinforcement Learning: Pong from Pixels." We then trained over the course of a few days.

3.3 Analysis of Results

For the baseline, we hard coded a simple agent who moved constantly from the left wall to the right wall. It performed surprisingly well, surviving for an average of 15 seconds. It makes sense that it performed so well, as the monsters are constantly shooting fireballs in the direction of the agent, and so being constantly in motion facilitates evasion. We understand this intuitively as humans - however, it is difficult for the neural net to learn that skill simply from the pixel array. This is probably why no agent has ever solved the level before.

It was difficult to determine an 'oracle' for this level since humans have never played it, and no AI had solved it before. Theoretically, an agent could survive infinitely long if it was skilled enough. Observing that the deterministic agent was actually pretty close to the 'target,' we picked an oracle of 25 seconds.

This algorithm took by far the longest to run, with over 100,000 iterations. Compared to other Neural Networks, however, the time it took to achieve a good result was actually quite short. It managed to reach the required target of surviving 20 seconds at about 90,000 iterations, making us the first team to solve this OpenAI level (see Figure 7).

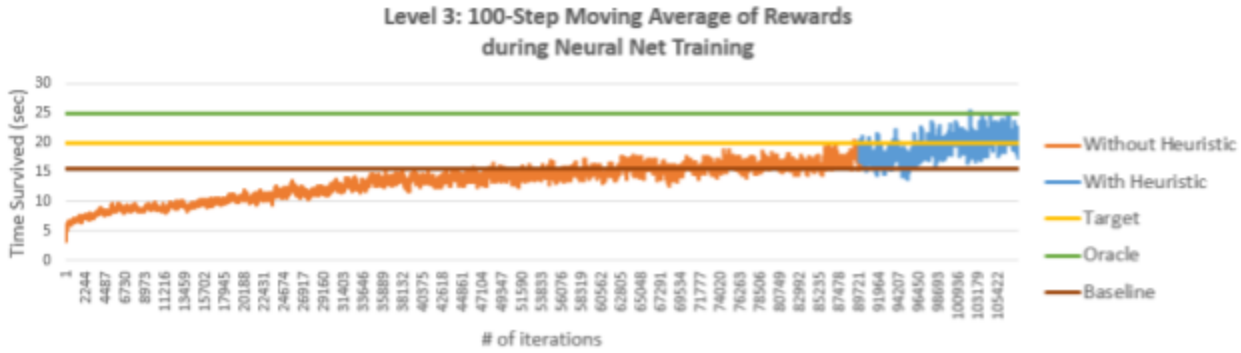

Figure 7: Level 3 Average Time Survived in Seconds

We noticed, however, that its rate of learning seemed to diminish around that time. Watching the agent as it played, we noticed that it would often get "stuck" next to the walls: when it was against one of the walls, it would attempt to flee incoming fireballs in the direction of the wall. However, because it could not go any farther, it would die. We realized that it had no information about where the walls were, and so had no way of learning that moving towards the wall when in right next to it was doomed to fail.

Our first attempt at addressing this problem was to add its location to the state space. First we tried adding the agent's location as a single continuous scalar, and then we tried adding numHorizontalPixels [0,1] variables, where the current position's value is the only 1. After a lot of training on both augmented models, neither of them were able to perform better than our original model.

We finally elected simply to add a heuristic to alter the probability of MOVE_RIGHT when the agent is at or near the wall. When at the right wall, we force the probability to be zero (because we cannot move in that direction), and when near the left wall, we force it to be one. When very close to either wall, we also scaled the probability up or down appropriately. We tuned these parameters to find good values.

This heuristic worked quite well: when it was added at around the 90,000-iteration mark, the rate of learning increased noticeably (see Figure 7).

A demonstration of our agent in action is uploaded to YouTube:
https://youtu.be/xFfNfL7D97k .

3.4 Error Analysis

While the performance of our agent is quite impressive, it still occasionally makes an obviously wrong choice, and still sometimes gets stuck in the corners. This arises from the shallowness of the neural net. While this simplicity served our purposes in striving for intuition (in addition to suiting our limited resources), it clearly creates a ceiling to what the agent can do, as a single layer simply cannot capture all the nonlinearities inherent in such a complex task. Thus, to improve performance more, more hidden layers will need to be added.

Finally, it would have been nice not to have needed a domain-specific heuristic to facilitate learning. While it worked in this instance, it is not generalizable.

Conclusion

Due to the variety of challenges in the three levels, we were forced to rethink our models and approaches every single time. This allowed us to implement several different algorithms: Monte Carlo, Value Iteration, Q-Learning, and Neural Networks. There were times when it was obvious that a certain approach wouldn't work, such as on Level 3 when the state space was much too large for Q-Learning. There were also times when multiple approaches would have worked; for example, there were many possible computer vision approaches to extracting the position of the enemy in Levels 1 and 2. We also saw vividly how different algorithms vary in the length of the learning process (e.g. training an agent for Q-Learning on Level 1 was leaps and bounds faster than training a neural net for Level 3), and how even the same algorithm can be much slower when the environment changes a little bit (e.g. performing Q-Learning on Level 1 was much faster than doing so on Level 2).

All in all, this project has inspired us to immerse ourselves into the AI world more deeply, and has given a new unique point of view and appreciation of all the work already done in the field and the opportunities still unexplored.

Paul Steenkiste
Jaak Joonas Uudmäe
December 15, 2016

Sources

Article about Carnegie Mellon's Doom AI:
https://techcrunch.com/2016/09/21/scientists-teach-machines-to-hunt-and-kill-humans-in-doom-deathmatch-mode/
Articles about Facebook and Intel's Doom AI:
https://www.engadget.com/2016/09/22/facebook-and-intel-reign-supreme-in-doom-ai-deathmatch/
http://www.techworm.net/2016/09/facebook-intel-built-doom-playing-ai-dominated-video-game-competition.html
Challenge (evaluation metric for success):
https://gym.openai.com/envs#doom
CV2 Template Matching:
http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_template_matching/py_template_matching.html
Doom AI competition:
http://vizdoom.cs.put.edu.pl/competition-cig-2016
Doom (1993)
https://en.wikipedia.org/wiki/Doom_(1993_video_game)
Environment for levels:
https://github.com/ppaquette/gym-doom
OpenAI, Doom Level 1:
https://gym.openai.com/envs/DoomBasic-v0
OpenAI, Doom Level 2:
https://gym.openai.com/envs/DoomDefendCenter-v0
OpenAI, Doom Level 3:
https://gym.openai.com/envs/DoomTakeCover-v0
Research into deep learning for Doom:
https://www.cmu.edu/news/stories/archives/2016/september/AI-agent-survives-doom.html

YouTube Level 1:
https://youtu.be/hlIJUFfRw0c
YouTube Level 2:
https://youtu.be/llbDPmmqEHw
YouTube Level 3:
https://youtu.be/xFfNfL7D97k