Reinforcement Learning on 2048

Ahmed Bouhoula, Matheus Carívs Castro, Matheus Centa

Abstract—**In this project, we aim at training a game-playing agent for the 2048 game. We implement an OpenAI Gym environment to model the game and use the Deep Q-Learning (DQN) algorithm from the Stable Baseline library to train multiple agents varying the states encoding, reward function, network type and structure. Results show that encoding states using one-hot encoding is crucial for better performance. We have also concluded that Convolutional Neural Networks (CNN) are more efficient than Multilayer Perceptrons (MLP) for the purpose of this game.**

Keywords: ***reinforcement learning, 2048, stochastic games, dqn, off-policy, stable-baselines***

I. Introduction

2048 is a one-player stochastic game that went viral in 2014. The game's mathematical aspects and moderate complexity encouraged us to design a Reinforcement Learning (RL) agent that performs well on it. To achieve this, we implemented the game as an OpenAI Gym environment [1]. In our implementation, the action space is discrete and features only four elements, but the state space grows exponentially with the board size. Since the number of states in a 4x4 game is bounded by 10^{19}, it would be unfeasible to visit all the states to maximize the value function in each state. We, therefore, opted for an approximate algorithm: DQN.

II. Background and Related Work

The best AI agent for 2048 we have found was developped by Robert Xiao [1]. It implements an expectimax search in C++ and goes over 10^7 states per second. It has a 94% chance of reaching the 16384 tile and 36% of reaching the 32768 tile. It uses heuristics to make the reward function more representative of the strategy we would want an agent to follow.

In [2], the authors use N-tuple networks (NTNs). They achieve a highest score of 261526 which means that the agent was able to reach the 16384 tile quite often. This agent is outperformed by [1] and uses up to $2 \cdot 10^9$ parameters. We will not further investigate this method.

In [3], the authors use imitation learning to train their agent: A Convolutional Neural Network (CNN) is implemented with the grid state as input and a 4-dimention vector representing the probability for each action as output, the network is trained based on the action chosen by an experimented player. Their agents performs worse than [2]. Still, it reaches the 16384 tile 2% of the time and the 4096 tile 57% of the time.

In [4], the authors implement a Probabilistic Policy Network which they try to improve using Approximate Dynamic Programming and Monte-Carlo Tree Search. However, their agent never reaches the 2048 tile. The maximum tile it reaches is 1024.

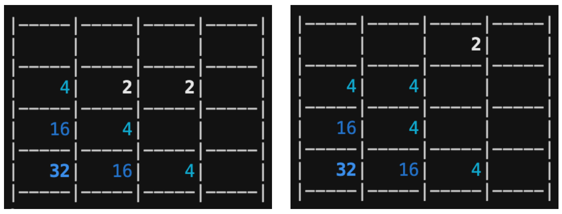
As Reinforcement Learning becomes more and more widely used, more libraries that efficiently implement different RL methods emerge. Stable Baselines [5] is a set of improved implementations of RL algorithms based on OpenAI Baselines that we use in order to create our agent.

III. The Environment

2048 is a fully-observed stochastic game played on a 4x4 grid. Every turn, the player chooses an action $a \in A = \{up, right, down, left\}$. Some cells of the grid contain tiles whose values are some power of 2. When the player chooses an action, all the tiles of the grid move along the chosen direction. If two tiles are adjacent and share the same value, they are merged into one single tile. The value of the new tile is the sum of values of the merged tiles. Each newly generated tile cannot merge with another tile in the same turn. After every turn, if the chosen action is valid (i.e. changes the configuration of the grid), a new tile of value 2 or 4 appears randomly in one of the remaining cells.

The game ends if none of the remaining actions are valid. The goal of the player is to maximize his score (i.e. reward function). Initially the score is set to 0 and the grid contains only one tile of value 2 or 4. After every merge operation, the score is incremented by the value of the newly generated tile.

One of the main challenges of this game is the exponential number of board configurations (i.e. states) which is bounded by 10^{19}, which makes it impossible to use standard techniques that require visiting all the states.



```
|-----|-----|-----|-----|    |-----|-----|-----|-----|
|     |     |     |     |    |     |     |    2|     |
|-----|-----|-----|-----|    |-----|-----|-----|-----|
|    4|    2|    2|     |    |    4|    4|     |     |
|-----|-----|-----|-----|    |-----|-----|-----|-----|
|   16|    4|     |     |    |   16|    4|     |     |
|-----|-----|-----|-----|    |-----|-----|-----|-----|
|   32|   16|    4|     |    |   32|   16|    4|     |
|-----|-----|-----|-----|    |-----|-----|-----|-----|
```

Fig. 1. On the left is a random grid configuration and on the right is the new configuration after the action $left$ has been chosen. Note that the two 2-tiles have merged into a 4-tile and a new 2-tile has appeared.

A. Representing the states

There are two possible ways to represent the states:

1) an array of size 16 where each cell contains 0 if it's empty or the log of the tile it contains otherwise. The value of the log can never exceed 18 on a 4x4 grid. We can even suppose that it's bounded by 15 if we settle for the 32768 tile.

[1]Code can be found using this link: https://github.com/mmcenta/left-shift. Please follow the instructions in "README.md" to set up the environment.

2) we can represent each cell i by a vector $u_i \in \{0,1\}^{16}$ where $u_i(j) = 1$ if, and only if, the cell i contains the 2^j-tile (or is empty if $j = 0$). A grid configuration is then represented as a $\{0,1\}^{256}$ vector. This representation is often refered to as "one hot enconding".

The second option is more widely used and has the advantage of making the cell values equidistant. Depending on what algorithm we use one representation or the other.

Another possibility is, instead of defining the state as just the current grid configuration, to consider a collection of grid configurations that can be attained by 1 or 2 moves from the current configuration. In [6], this representation resulted in improved performance. According to the author, he was inspired by a similar representation used by the Alpha Go Zero[7] agent.

B. *Reward function*

The score a normal player would want to maximize is the sum of values of all the generated tiles as stated earlier. The reward at a given time step is the difference between the new state value and the previous state value.

It is best to avoid non-valid actions as they have little learning significance and can pollute the replay buffer. Furthermore, when using off-policy methods, the agent might get stuck in long sequences of such actions. As a result, we add a negative penalty for choosing a non valid action and we terminate the episode when the proportion of invalid non-valid actions surpasses a threshold - which we defined to be 10% of all performed actions by default. Some of our earlier experiments also used the strategy of stopping the game with a low probability after each non-valid move. Compared to instantly ending the episode after a single invalid action, these methods allow agents to recover from mistakes (such as an exploration step that resulted in no changes) and explore more states.

IV. The Agent

As we mentioned before, the large state space makes the usage of tabular methods unfeasible due to computational limitations. Instead, we opted to concentrate our efforts on approximate algorithms. Most of the approximate methods we found in the literature estimate the policy directly, many by imitation learning from a human agent. We chose to focus on state-action value estimation techniques. We take inspiration from [8] and use DQN agents to tackle this problem, because they achieve good performance on Atari games. Moreover, the only examples of their usage on the 2048 game we found were personal projects that didn't explore the novel DQN extensions (see section IV). We experiment with both Multi-Layer Perceptron (MLP) and CNN policies. We expect CNNs to achieve better performance because they can take advantage of the board organization and learn local features. Furthermore, there are three extensions to the DQN algorithm that we consider for our agent:

- **Dueling DQN:** introduced in [9], this architecture explicitly separates the state value estimation from the state-dependent action estimation. The motivation came from the realization that there are games for which it is not necessary to know the value of each action at every time step.
- **Double Q Learning:** proposed in [10], this technique aims at reducing Q-value overestimation that is common with DQNs.
- **Prioritized Experience Replay:** presented in [11], this framework allows agents to remember and reuse experiences from the past by prioritizing more significant experiences. It is an improvement on experience replay [12], which samples experiences uniformly from memory.

For our network architecture, we take inspiration from [3] for agents with CNN policies and we compare them against MLP architectures. We implement the agents using the Stable Baselines[5] library, which in turn is implemented with TensorFlow[13]. We use the ϵ-decay method. We start by favoring exploration over exploitation by setting $\epsilon = 0.6$ then it progressively goes down to 0.01 during the first 10% of the training time. Then, it stays fixed at 0.01.

V. Results and Discussion

A. *Reward function*

Early experiments confirmed what we discussed in subsection III.B and showed that agents tend to loop on non valid actions while the board is half empty if no penalty is added when the agents takes a non valid action. We tested two values for this penalty: 32 and 512. Higher penalty values improved overall performance.

B. *CNN structure for DQN*

Then, our goal was to find the best CNN structure to use in the DQN algorithm. We varied the number of convolutional layers and obtained the results shown in Table I. We use the same number of channels for each layer as in [3]. All CNNs use a kernel of size 2x2 with stride 1 and padding to ensure the input size stays the same. We use the one-hot encoding to represent the states. We used a penalty of 32 for non valid actions and a probability of 5% to terminate after an invalid action.

TABLE I
Performance of DQN algorithm using a CNN depending on the number of convolutional layers

# layers	# channels per layer	Avg. score	# timesteps
2	436	1555	10^6
3	312	1063	10^6
4	256	2364	10^6
5	222	**2591**	10^6
6	200	1760	$3 \cdot 10^6$
7	182	1203	$3 \cdot 10^6$
8	168	1707	$3 \cdot 10^6$

Using a CNN with 5 layers seem to be the best option as it outperforms all other CNNs even when trained with much less timesteps (CNN with 5 layers is trained for 1 million timesteps while CNNs with 6, 7 and 8 layers have been trained for 3 million timesteps). For the remaining experiments, all CNNs have 5 layers, each layer having 222 channels.

C. Architectures

For the next subsections, we are going to use 8 different architectures. They were all trained for 10 million timesteps with Prioritized Experience Replay, Double Q and Dueling enabled. The Dueling of stable-baselines applies a MLP only to the state value prediction, not to the action value prediction. We have modified the code to apply it to both of them, and the models with this modification have a "_FC" appended to their name (*).

- **MLP:** MLP with layer sizes [512,512,512];
- **CNN_5L:** the winner of the previous subsection, a 5-layer CNN with 222 channels per layer. Dueling MLP with layer sizes [256,64,64];
- **CNN_5L_4:** a modification of CNN 5L, with the first kernel of size 4 instead of 2, keeping the same channels per layer. Dueling MLP with layer sizes [512,128];
- **CNN_5L_4_v2:** a modification of CNN_5L_4 with Dueling MLP with layer sizes [256,64,64];
- **CNN_5L_4_FC:** modification of CNN_5L_4 as explained in (*). Dueling MLP with layer sizes [512,128];
- **CNN_ALT:** an alternate CNN we designed. Refer to the appendix for the architecture. Dueling MLP with layer sizes [512,128];
- **CNN_ALT_FC:** a modification of CNN_ALT as explained in (*). Dueling MLP with layer sizes [512,128];

D. State encoding

As discussed in subsection III.A, we have tried two different state-encodings, the default 4x4 grid and the one-hot encoding. We compared the results using different architectures as summarized by table II. The scores were averaged over 1000 episodes. Results show that using one-hot encoding makes a huge difference and improves performance of our agent.

TABLE II
COMPARISON OF ONE-HOT ENCODING EACH CELL VERSUS USING A DEFAULT GRID

Architecture	One-Hot Encoding	Avg. score	Largest Tile
MLP	✗	1851	512
MLP	✓	**5563**	**1024**
CNN 5L	✗	2315	512
CNN 5L	✓	**8925**	**2048**
CNN 5L_4	✗	2047	512
CNN 5L_4	✓	**10261**	**2048**

E. Comparing architectures

We compare the results of each of the mentioned architectures. In table III we can observe the average score of each architecture and the histogram of maximal tiles reached. Figure 2 shows the evolution of the reward function of 4 of our architectures during training. Figure 3 does the same for the episode length.

Figures 4, 5, 6 and 7 show the maximum tiles achieved during training for the same 4 architectures.

Results show that while CNN_ALT performs better without a MLP, CNN_5L_4_FC outperforms CNN_5L_4.

Table III. shows that overall best performance is achieved by CNN_5L_4_v2. It reaches the tile 1024 at least 49.4% of the time. It also reaches the 2048 tile 3.8% of the time.

TABLE III
HISTOGRAM OF MAXIMUM TILE ACHIEVED BY EACH ARCHITECTURE

Architecture	Score	32	64	128	256	512	1024	2048
MLP	5563	1	17	85	270	533	94	0
CNN_5L	8925	2	9	59	126	460	311	33
CNN_5L_4	9495	0	1	15	162	474	326	22
CNN_5L_4_v2	**10261**	0	4	8	94	400	**458**	36
CNN_5L_4_FC	9788	2	11	26	106	415	402	**38**
CNN_ALT	7931	7	53	116	170	348	274	32
CNN_ALT_FC	8430	0	1	15	176	578	227	3

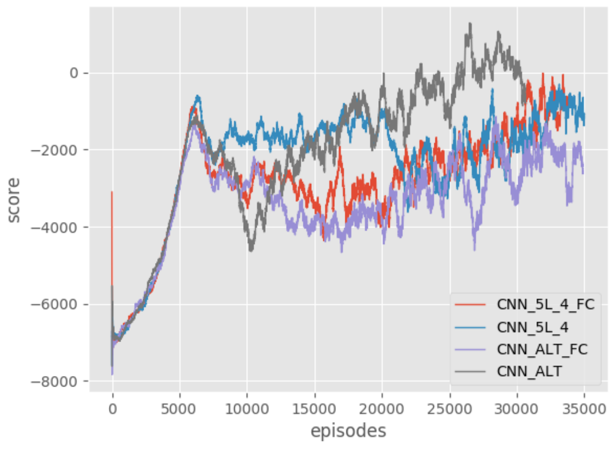


Fig. 2. Evolution of reward function during training. Each time the score is averaged over the last 500 episodes.

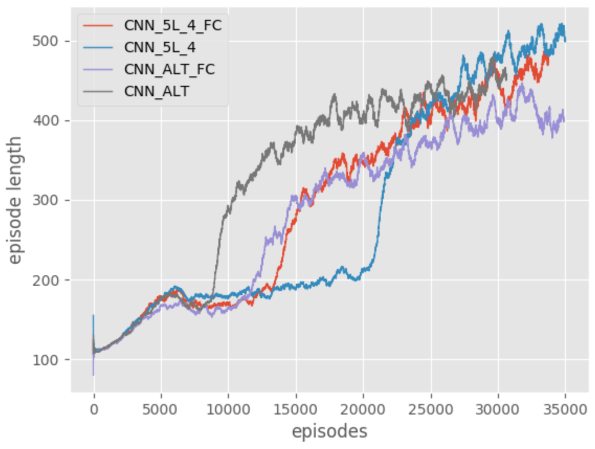


Fig. 3. Evolution of episodes length function during training. Each time the lengths are averaged over the last 500 episodes.

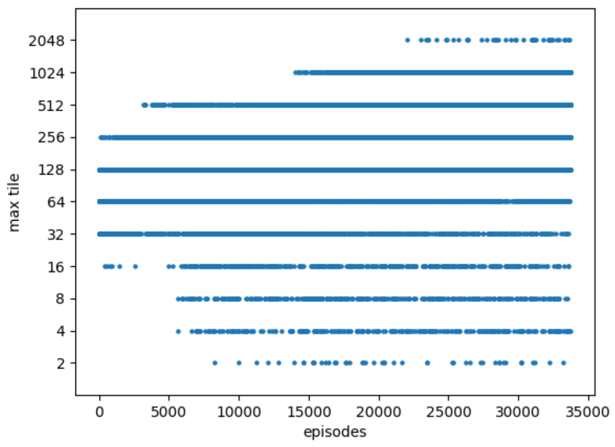


Fig. 4. Maximum tile achieved during training for CNN_5L_4_FC.

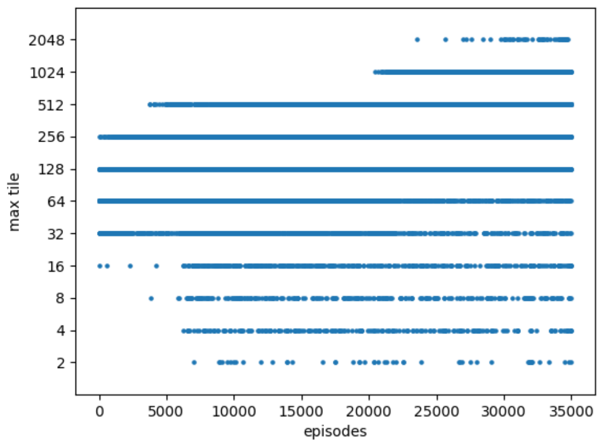


Fig. 5. Maximum tile achieved during training for CNN_5L_4.

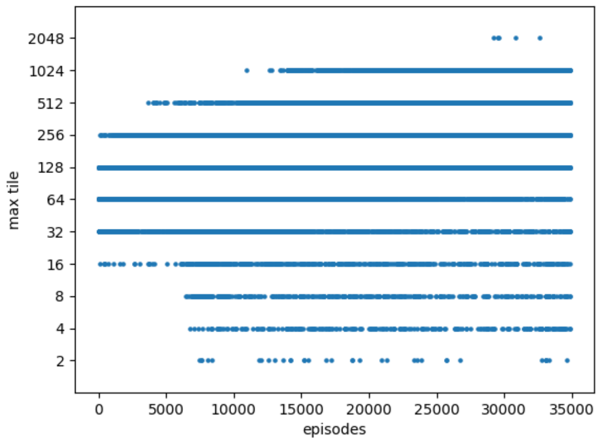


Fig. 6. Maximum tile achieved during training for CNN_ALT_FC.

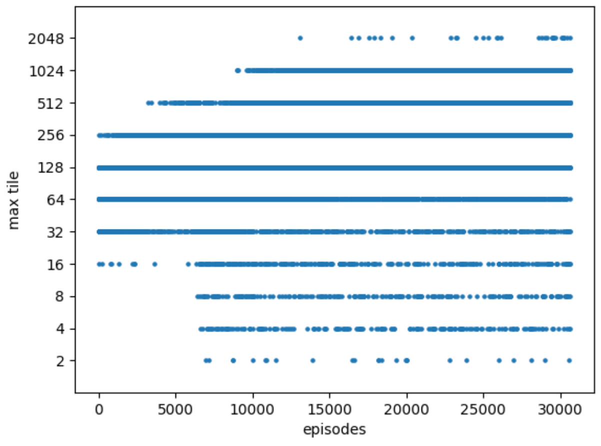


Fig. 7. Maximum tile achieved during training for CNN_ALT.

VI. Conclusion and Future Work

DQN with a 5-layer CNN using one-hot encoding and DQN extensions such as Dueling DQN, Double Q learning and Prioritized Experience Replay showed best results on 2048. Our agent is subject to further improvement. Training the agent with a learning rate decay might improve performance. This option is not currently supported by Stable Baselines. We could also change the state representation to include more information such as the configurations that can be reached after one or two timesteps.

References

[1] R. Xiao, "2048-ai." https://github.com/nneonneo/2048-ai, 2013.

[2] M. Szubert and W. Jaśkowski, "Temporal difference learning of n-tuple networks for the game 2048," in *IEEE Conference on Computational Intelligence and Games*, (Dortmund), pp. 1–8, IEEE, Aug 2014.

[3] N. Kondo and K. Matsuzaki, "Playing game 2048 with deep convolutional neural networks trained by supervised learning," *Journal of Information Processing*, vol. 27.

[4] J. Amar and A. Dedieu, "Deep reinforcement learning for 2048," in *31st Conference on Neural Information Processing Systems*, (Long Beach, CA, USA), 2017.

[5] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines." https://github.com/hill-a/stable-baselines, 2018.

[6] S. Iommi, "Dqn-2048." https://github.com/SergioIommi/DQN-2048, 2018.

[7] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–, Oct. 2017.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

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

[10] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015.

[11] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2015. ICLR 2016.

[12] L. ji Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," in *Machine Learning*, pp. 293–321, 1992.

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

VII. Appendix

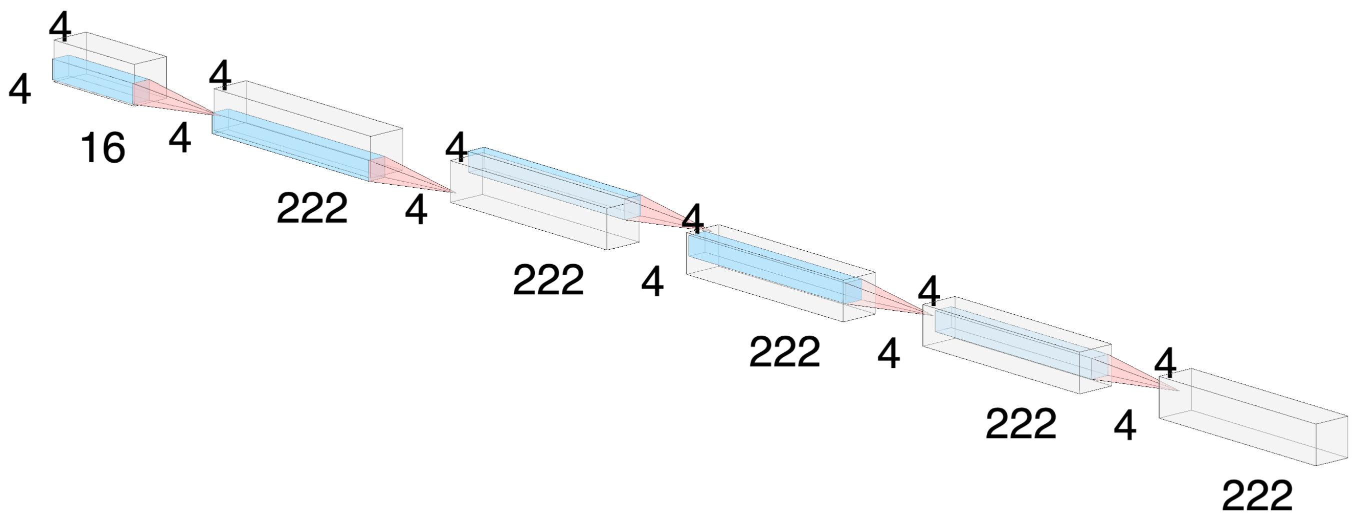


Fig. 8. Base CNN feature extractor for CNN_5L.

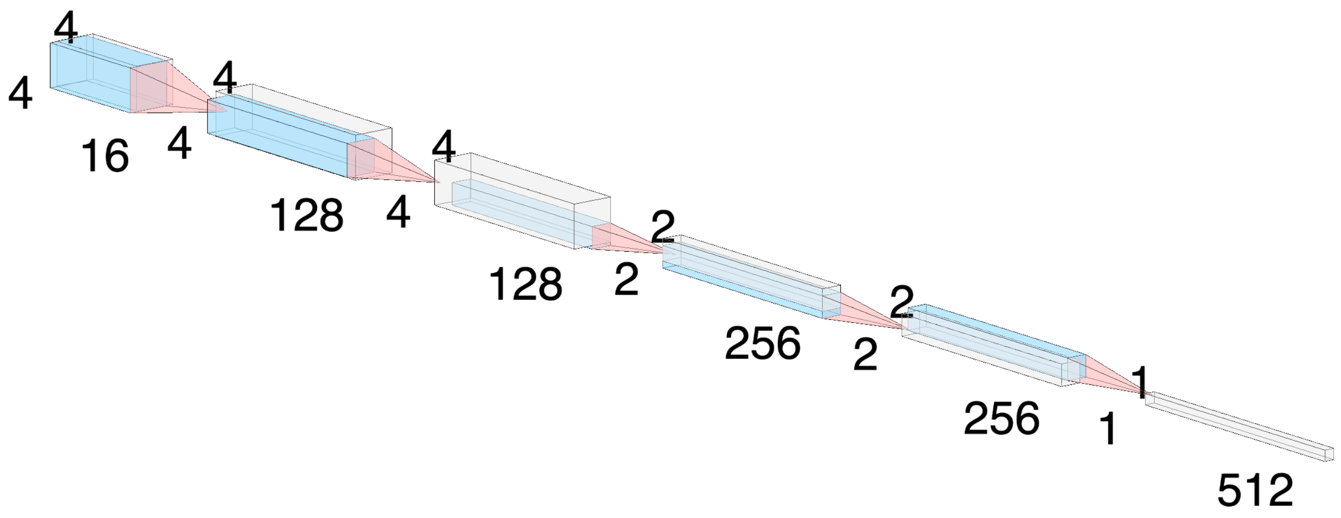


Fig. 9. Base CNN feature extractor for CNN_ALT.