Complexification Through Gradual Involvement in Deep Reinforcement Learning

Eugene Rulko
Minsk, Belarus
eugeni1533@gmail.com

Abstract

Training a relatively big neural network that has enough capacity for complex tasks is challenging. In real life the process of task solving requires system of knowledge, where more complex skills are built upon previously learned ones. The same way biological evolution builds new forms of life based on a previously achieved level of complexity. Inspired by that, this work proposes a way of training neural networks with smaller receptive fields and using their weights as prior knowledge for more complex successors through gradual involvement of some parts. That allows better performance in a particular case of deep Q-learning in comparison with a situation when the model tries to use a complex receptive field from scratch.

1 Introduction

Reinforcement learning (RL) offers a powerful framework for decision-making tasks, where agents learn from interactions with an environment to improve their performance over time. The agent observes states and rewards from the environment and acts with a policy that maps states to actions. Deep Reinforcement Learning (DRL) denotes the combination of deep learning with RL. DRL uses deep neural networks to train powerful function approximators to address complicated domains [1]. But DRL still faces difficulties, especially when convergence of deep neural networks requires learning complicated concepts in environments with sparse feedback. That difficulty has some intuitive explanation. Imagine a human baby behind a wheel with the target to drive home and the amount of attempts that it would take to achieve some positive feedback. Or the task of getting some chemical substance in a lab through the pure trial and error method by a person who is totally unfamiliar with chemistry. These processes require learning consecutive sets of skills, where each set is built upon previously learned ones. It's especially attributable to humans and high-level animals. For a person it requires learning how to control bodily functions, getting basic knowledge from parents, kindergarten, and school and so on. A fox baby isn't able to hunt, until acquiring all the necessary skills. In addition to learning procedure, defining a structure of a neural network of sufficient capacity, that can learn the set of consecutive tasks and effectively converge, also represents a hurdle. The process of evolution in nature generally goes from simple neural structures to more complex. Inspired by that, this work describes an example of *Complexification Through Gradual Involvement* used for the game of Snake within the framework of deep Q-learning.

2 Related Work

Training a model on examples of increasing difficulty, progressively providing more challenging data or tasks as the policy improves, is called *Curriculum Learning* (CL) [2]. As the name suggests, the idea behind the approach borrows from human education, where complex tasks are taught by breaking them into simpler parts. This is used now in advanced spheres like teaching quadrupedal robots to perform complex movements [3], quantum architecture search [4] and many others. There are a lot of strategies of CL. An approach that uses separate policies for each skill [5] and a similar one that distils the specialist controllers into a single generalist transformer policy [6] both seem to be closest to the approach described in the current paper because of connection between successive teaching and allocating some network capacity for newly formed skills. Another related approach is called *Progressive Neural Networks* [7]. A progressive network is composed of multiple columns, and each column is a policy network for one specific task. It starts with one single column for training the first task, and then the number of columns increases with the number of new tasks. While training on a new task, neuron weights of the previous columns are frozen and representations from those frozen tasks are applied to the new column via a collateral connection to assist in learning the new task [1]. Also the idea of *Distillation Model* involves training a smaller model first and then building a big one that will imitate the first one in order to kick start the large model's learning progress [8]. In spite of some similarities, the suggested approach unlike others uses successive allocation of the network capacity for a current single task through increasing the perception field, i.e. the state space.

3 Complexification Through Gradual Involvement And Experimental Studies

Complexification can be performed in different directions: just network capacity, the amount of sensory information (state vector) and subsequent network capacity, and the action vector. In the current work only the case of increasing the size of a state vector is considered. Suggested approach is described based on the game of Snake [9], which is a modified version of the original game [10]. The screen of the game is represented on Figure 1.

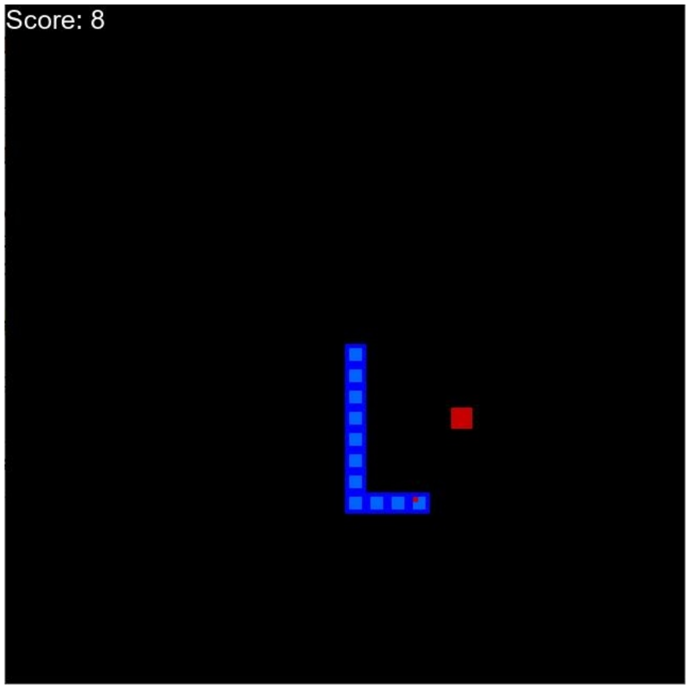

Figure 1: The Snake game

Initially as an input vector the snake takes the following 11 parameters that are relative to its head's position [9]:

- danger straight within 1 step
- danger right within 1 step
- danger left within 1 step
- moving left
- moving right
- moving up
- moving down
- food left
- food right
- food up
- food down

The result of training is represented on Figure 2. It takes approximately 100 games to converge and the average result is about 35 scores. Also it requires epsilon-greedy strategy for exploration during the first 80 epochs. Without it the network doesn't converge at all.

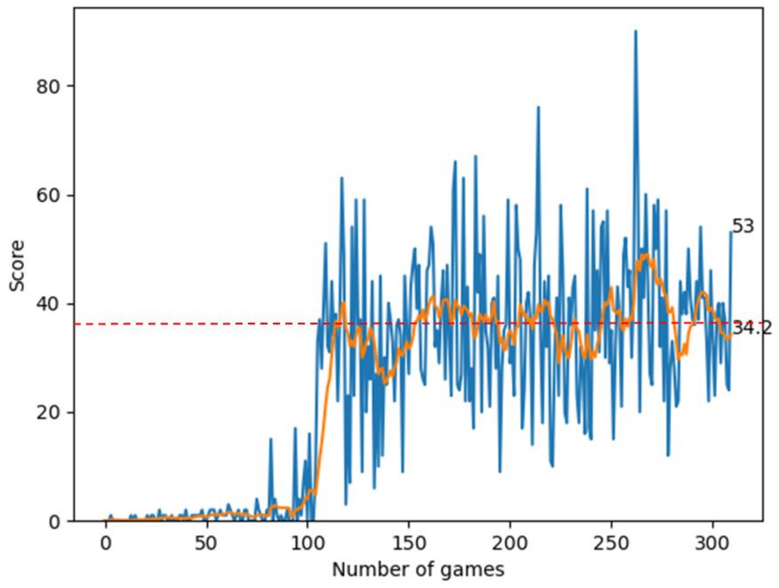

Figure 2: Training result for 11 values

The analysis of the way the snake ends up shows that it tends to coil in itself – Figure 3.

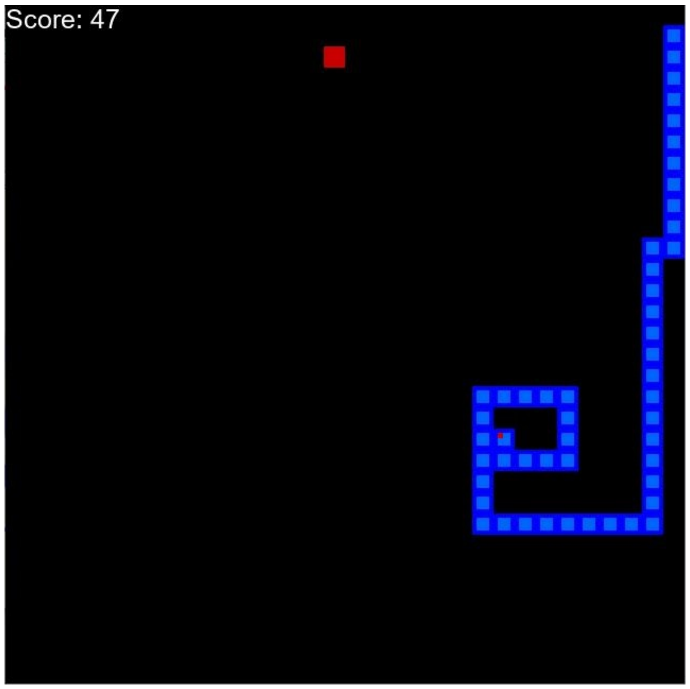

Figure 3: Coiling up

That situation is supposedly attributable to the inability of the snake to get understanding of the location of its own parts. Let's increase the input vector by adding the following parameters:

- snake tail to the right of the head

- snake tail to the left of the head
- snake tail to the front of the head
- relative distance to the right wall
- relative distance to the left wall
- relative distance to the front wall
- last turn left
- last turn right

Now the input vector is comprised of 19 parameters. The result of training is represented on Figure 4.

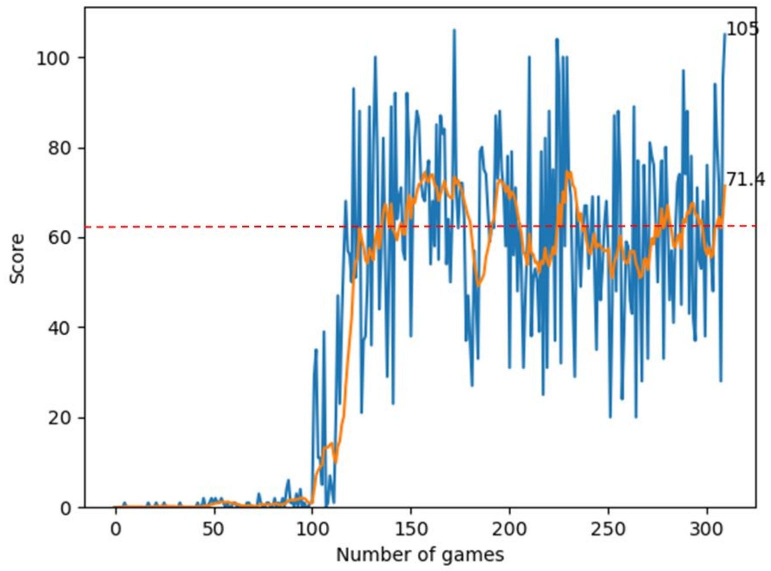

Figure 4: Training result for 19 values

Now it takes approximately 150 games to converge and the average result is about 62. If we want to start the learning process of a model with a bigger input vector not from scratch, but with weights of a smaller one, the procedure in this case is straightforward. For the current network architecture it requires copying of the weights of the second fully connected layer (FC 2) and the weights of the first fully connected layer (FC 1) concatenated with a tensor of random values of shape (8, 256) in order to fit the new formed FC 2 layer. The scheme of the process is presented on Figure 5. The number of input and output features of each layer is specified in parenthesis.

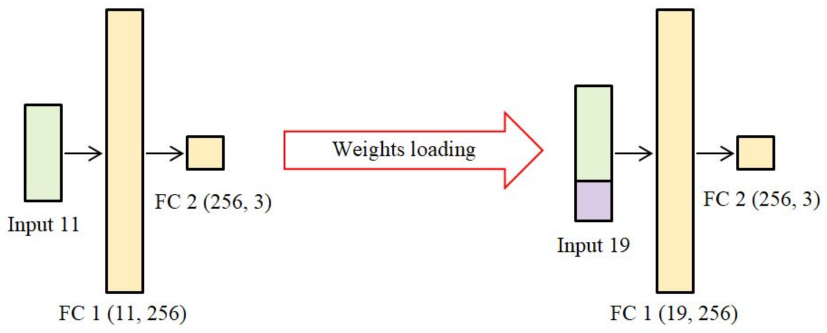

Figure 5: Weights loading

Modern frameworks like PyTorch provide a convenient way of loading weights from one model to another. The result of training of the network with loaded weights from the experiment of Figure 2 is represented on Figure 6. It demonstrates that with prior knowledge it takes approximately 50 games to converge to even better scores in comparison with the

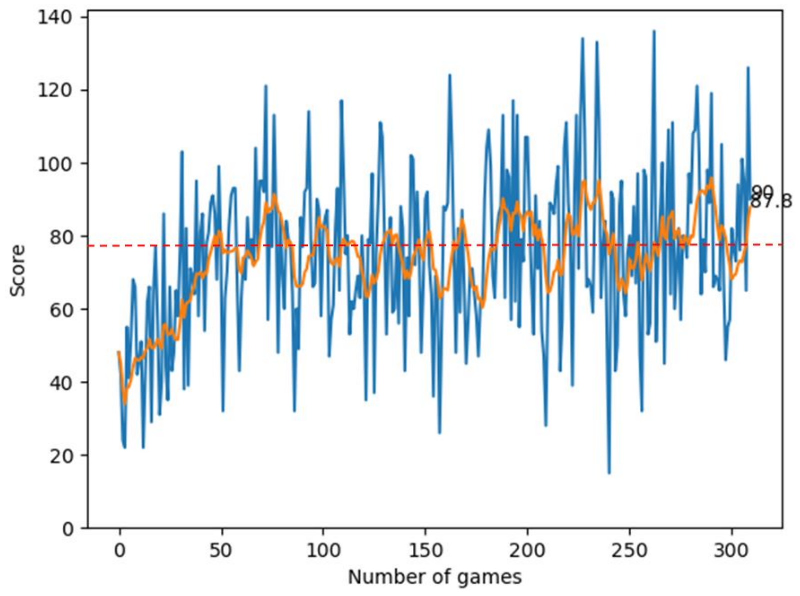

Figure 6: Training result after weights loading

experiment on Figure 4, which is about 3 times less in terms of count of games. Due to the nature of neural networks, they can rely only on known part of the input vector, performing rational activity in terms of the environment and simultaneously figuring out the way of applying newly added part of the input vector. It's important to note that it doesn't require any initial exploration as it were in both cases with a smaller and bigger vectors starting from scratch. But it seems that in more complicated scenarios a way of exploring possibilities that come with added input vector might be required. The next step is to add a convolutional (2d) head to the neural network that will partially observe the environment. For this case a bit different approach will be demonstrated, which involves turning off some advanced parts of the neural network, like the convolutional head in this example, while training the initial smaller parts. In essence, this process is similar to training a smaller neural network and loading its weights into a correspondent part of a bigger one. In order to make it easier for the agent to learn, the convolutional head is provided not with the full environment, but with black and white cropped fragment of shape (8, 8) around snake's head, rotated according to its current direction – Figuire 7.

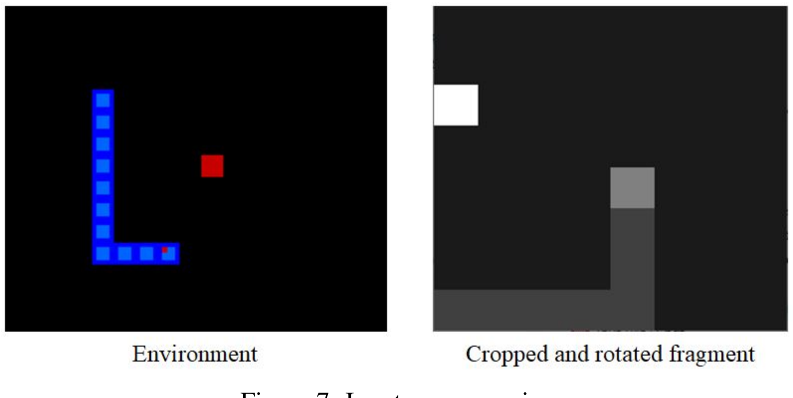

Figure 7: Input preprocessing

Architecture of the neural network (without ReLU layers) is presented on Figure 8. There are several sequential stages of training. During a "Zeros" stage the output of the convolutional head is always a tensor of zeros and the head is frozen. In this case the agent is supposed to rely only on the 1d head. A "Noise" stage involves processing the image by the frozen convolutional head with randomly initialized weights. The absence of any structured useful information about the environment from 2d head supposedly will make the rest of the network insensitive to any information from that head. The initial intent of that is to prevent possible sporadic behavior of the network on the transition between the previous stage and involving the 2d head, when the network has been trained with the constant tensor of only zeros and

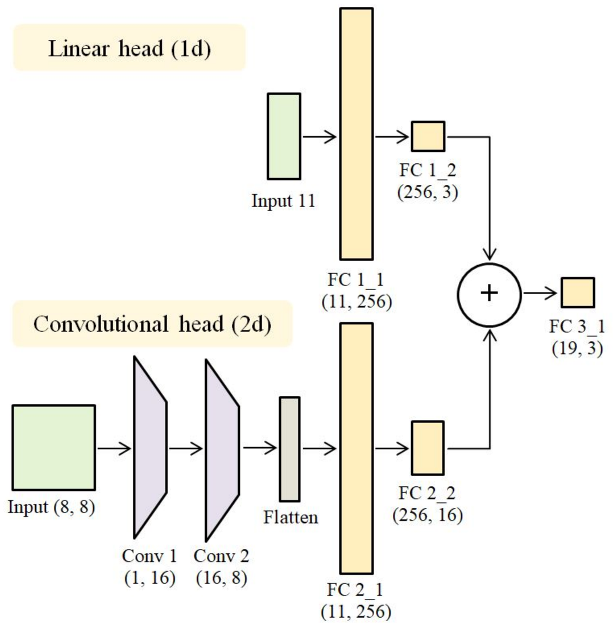

Figure 8: Network architecture

it unexpectedly gets a tensor of random values. An "Involving" stage implies freezing the 1d head and unfreezing the 2d head in order to provide some prior knowledge and kick start the learning process of 2d head. A "Both heads" stage involves simultaneous training of both 1d and 2d heads. A set of experiments has been conducted in order to practically evaluate performance, depending on redistribution of the entire amount of 3000 games between different stages using fixed hyperparameters. Every experiment the agent uses epsilon-greedy strategy during first 280 games and then the greedy one. The first experiment involves training the agent during all of the episodes (games) using a "Zeros" stage, which means it effectively uses only 1d head – Figure 9.

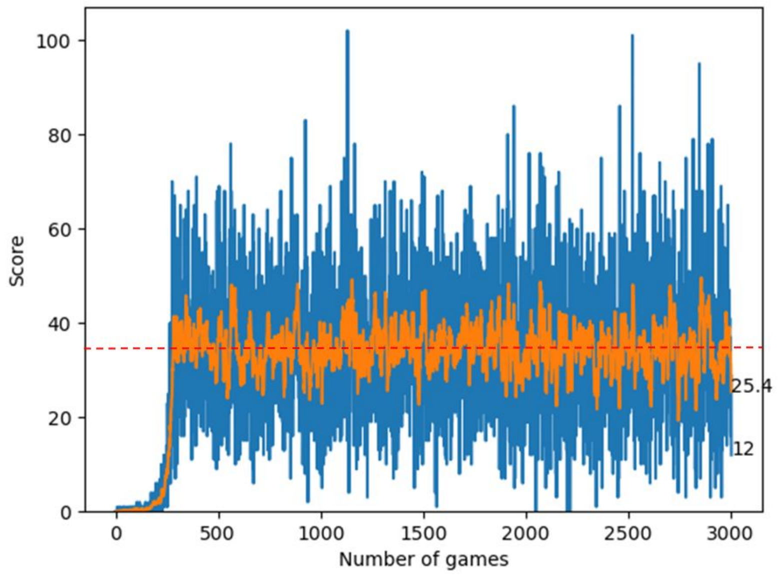

Figure 9: Training using only "Zeros" stage

Unsurprisingly, the result doesn't seem much different from Figure 2. It has the average score of 33 over 100 last games. The network just learns to ignore a tensor of zeros from the 2d head and rely only on 1d part that uses 11 values, the same as in case on Figure 2. It's necessary to mention that in spite of pretty stable average score, the dispersion of scores for each game (blue color) is pretty high. The second experiment involves training the agent during all of the episodes using the "Both heads" stage, which means it uses both heads from the beginning. The result of training is presented on Figure 10.

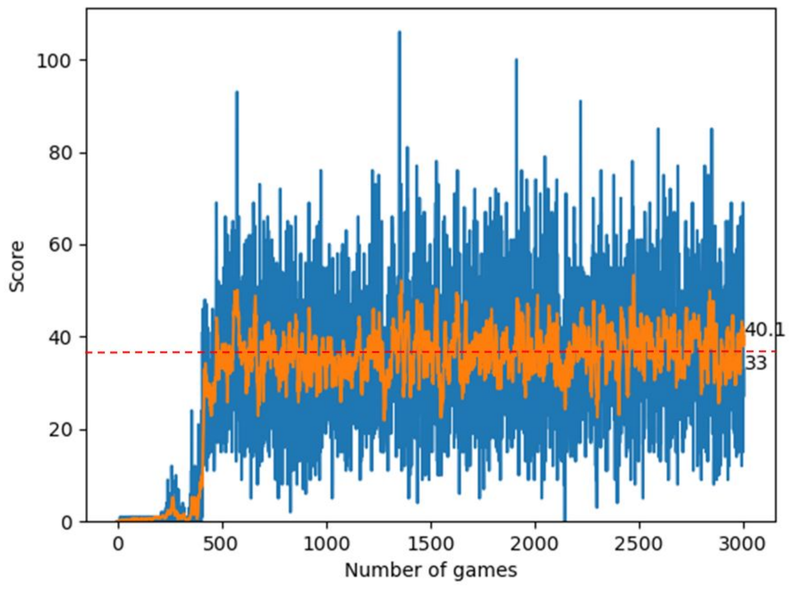

Figure 10: Training using only "Both heads" stage

It has the average score of 36 over 100 last games. The result is not far from the previous experiment, which means that the network isn't able to utilize data from the 2d head, "turned that head off", and still relies only on 1d head as in the case with "Zeros" stage. The third experiment involves training the agent on 500 games using "Zeros" stage and 2500 games using "Both heads" stage – Figure 11.

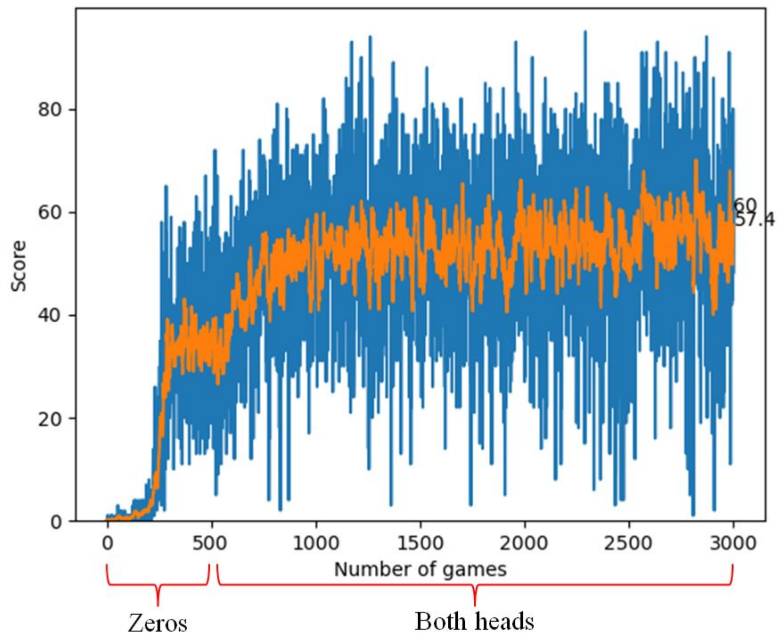

Figure 11: Training using "Zeros" and "Both heads" stages

In this case during the first stage the network learns how to utilize the 1d head and then, with its weights trained, involves the second one in the training process. The average score over 100 last games is 54, which is better than in the previous cases. The important point here is that such a score can't be achieved by training of two heads simultaneously. The forth experiment involves training the agent on 500 games using "Zeros" stage, 1000 games using "Involving" stage, and 1500 games using "Both heads" stage – Figure 12.

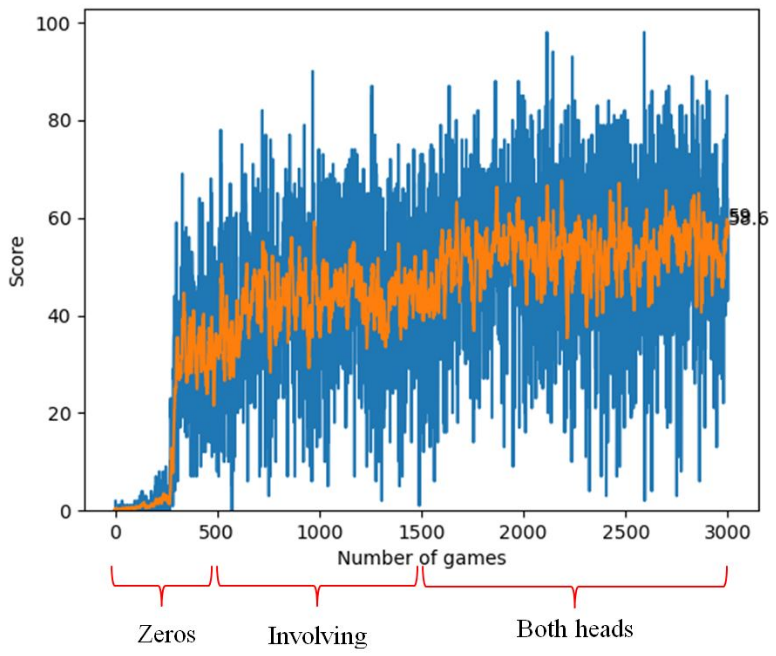

Figure 12: Training using "Zeros", "Involving" and "Both heads" stages

The final average score is 54 and is the same as in the previous experiment, which means that using "Involving" stage doesn't improve the results. The fifth experiment involves training the agent on 500 games using "Zeros" stage, 500 games using "Noise" stage, 500 games using "Involving" stage, and 1500 games using "Both heads" stage – Figure 13.

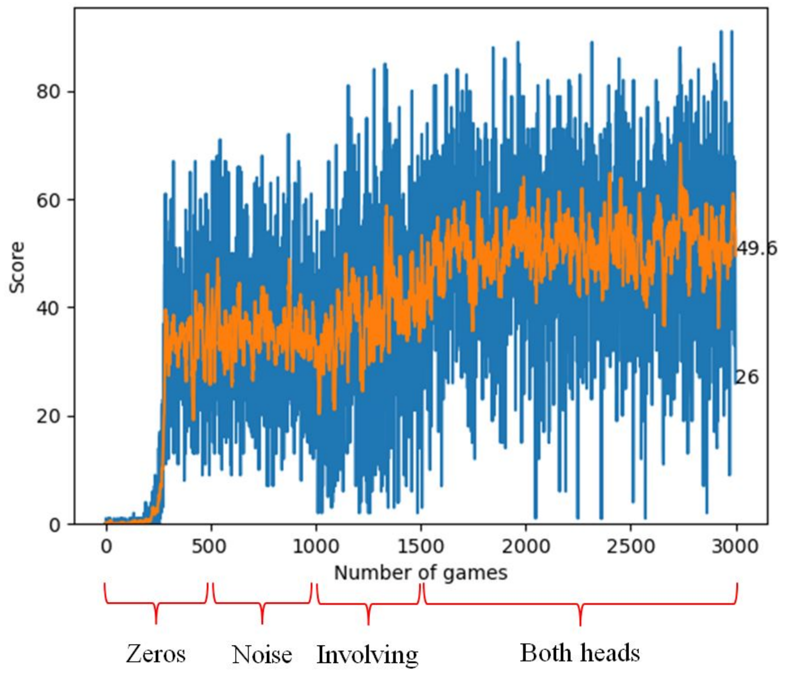

Figure 13: Training using all possible stages

Here also the final average score of 52 doesn't deviate too much from the third experiment, which means that using "Involving" stage also doesn't improve the results.

4 Conclusions And Future Work

This work is based on considering a pretty trivial example of the Snake game and describes the process of using weights of a previously trained network as prior knowledge for a more complicated one. It was shown that the suggested approach provides a way of achieving higher score without any hyperparamenter search in comparison with the case of training a complexified network from scratch. Future work requires conducting a more extensive set of experiments, including different environments and RL algorithms for getting conclusive information about applicability of the approach. It's necessary to consider different possible dimensions of increasing complexity, not only what's directly connected with a receptive filed, i.e. a state vector. It seems that in this particular case of the Snake game we can use not a single current state of the game, but also several previous states and gradually add some recurrent part to the network. It also seems that a sense of feeling a distance of food, not just direction of that, may also improve results. In this case it requires a gradual transition from a Boolean value to a value between 0 and 1. Future research can also be dedicated to automatic finding the necessary directions of extending the network capacity, unlike it was done manually in the current work.

Acknowledgments

The author would like to thank Patrick Loeber for the original project of Snake game [10] and a correspondent video tutorial.

References

[1] Zhuangdi Zhu et al. *Transfer Learning in Deep Reinforcement Learning: A Survey*. 2023. arXiv: 2009.07888.

[2] Petru Soviany et al. *Curriculum Learning: A Survey*. 2022. arXiv: 2101.10382.

[3] Vassil Atanassov et al. *Curriculum-Based Reinforcement Learning for Quadrupedal Jumping: A Reference-free Design*. 2024. arXiv: 2401.16337.

[4] Yash J. Patel et al. *Curriculum reinforcement learning for quantum architecture search under hardware errors*. 2024. arXiv: 2402.03500.

[5] David Hoeller et al. *ANYmal Parkour: Learning Agile Navigation for Quadrupedal Robots*. 2023. arXiv: 2306.14874.

[6] Ken Caluwaerts et al. *Barkour: Benchmarking Animal-level Agility with Quadruped Robots*. 2023. arXiv: 2305.14654.

[7] Andrei A. Rusu et al. *Progressive Neural Networks*. 2022. arXiv: 1606.04671.

[8] Enric Boix-Adsera. *Towards a theory of model distillation*. 2024. arXiv: 2403.09053.

[9] Eugene Rulko. *Complexification Through Gradual Involvement in Deep Reinforcement Learning*. https://github.com/Eugene1533/snake-ai-pytorch-complexification. 2024.

[10] Patrick Loeber. *Reinforcement Learning With PyTorch and Pygame*. https://github.com/patrickloeber/snake-ai-pytorch. 2021.