Continual Learning for Video Games

Nenya Edjah
Harvard University
nenya_edjah@college.harvard.edu

May 2020

1 Introduction

In the past few years, deep neural networks have demonstrated excellent results in solving many of the challenges pertinent to artificial intelligence. However, after successfully training on one task, it is difficult to get them to train on another task without degrading performance on the original. This is because the network weights will become updated to maximize performance of the second task which often causes the network to "forget" how to perform the original. This is a phenomenon known as **catastrophic forgetting**, and it is a primary inhibitor in the goal of continual learning: having a network learn to perform multiple tasks *in sequence* without ever retraining on any of the earlier tasks [12].

Prior work has attempted to address the continual learning problem. Approaches such as Learning without Forgetting [7] have used ideas inspired by distillation networks and fine-tuning to maintain performance on earlier tasks, while other approaches like Elastic Weight Consolidation [4] have used custom loss penalties to prevent the network's weights from deviating too far from the values that give good performance for earlier tasks. The biggest downside of these methods is that performance on earlier tasks is not necessarily guaranteed to be retained – especially so if the tasks are radically different.

Another approach inspired by the popularity of neural weight pruning [3] is PackNet [8] which trains sparse models on earlier tasks, and re-uses the weight capacity to train new tasks without compromising *any* performance on the earlier tasks.

In this work, we extend the ideas from PackNet and apply them to a deep reinforcement learning context. We show that it is possible to train a single neural network that is capable of achieving high performance on multiple distinct Atari video games [10] without compromising *any* performance on earlier tasks.

2 Background

2.1 Deep Reinforcement Learning

Reinforcement learning (RL) is a scenario in which there is a controllable agent and an environment that cannot be directly controlled. The agent interacts with the environment by performing actions. Actions change the environment's internal state, and the environment responds to the agent by providing it with an observation and the agent's reward for the action. The goal of the agent is to determine a *policy* that maximizes it's expected long term cumulative reward. Reinforcement learning has many applications, but for our purposes, note that its serves as a particularly good framework for teaching a machine to learn to play a video game.

Several techniques for deriving optimal RL policies have been developed. Traditional algorithms such as Q-Learning [14] and SARSA [13] are based on the idea of *temporal differencing*, and they are effective when the state space is discrete and small. However, whenever the state space is discrete and large (such as in video games), or even continuous, these algorithms are ineffective. Consequently, techniques such as Deep Q-Learning [10] have been developed to extend the traditional Q-Learning algorithm by using deep neural networks to process the state and predict actions and by using stochastic gradient descent to minimize the temporal differences.

Since the original paper, Deep Q-Learning has been greatly extended, and many other deep reinforcement learning algorithms have been developed. One particularly effective technique, and the one that we use further in this work, is the Advantage Actor-Critic [9] algorithm. At a high level, this technique maintains a *policy network* called the actor which determines the distribution over actions and a separate *value network* called the critic which evaluates those actions. Note that this means we only need the value network during training.

2.2 Weight Pruning

Neural networks are a classic example of an overparameterized machine learning algorithm. This means that they often have far more parameters than they actually need in order to learn their objective functions. Weight pruning [3] is a technique that has been used to greatly reduce the number of parameters in networks without a noticeable drop in accuracy.

Pruning works by fixing to zero the parameters that are of little importance to the network. A simple method for identifying unimportant parameters is magnitude – the parameters with the smallest magnitudes will end up having the least impact on the output of the network.

Weight pruning can be performed in an iterative fashion. The network is first trained to some level of accuracy before pruning begins. The network is then allowed to continue training, but some percentage of its weights will be periodically pruned up until some desired level of sparsity is reached or until accuracy begins to suffer. Depending on the initial size of the model, sparsity levels as high as 90% [3] can be achieved without any drop in accuracy.

Pruning has shown great effectiveness, and it has typically been applied for the purpose of model compression, but some researchers, such as the ones who developed PackNet [8], have also recognized its value for continual learning.

3 Methods

3.1 Continual Learning via Pruning

To perform continual learning for a sequence of video games, we take a similar approach to the one described by PackNet [8]. We begin by training a network on the first task. After a certain amount of training, we start to periodically prune the weights in the network based on their magnitude until we reach a desired sparsity level. After training for the desired number of iterations on the first task, we repeat the same process on a second task, and then a third, etc. Each later task will share the unpruned weights learned by the earlier tasks, but it will be unable to modify them during training. Each earlier task will be prevented from using any of the weights learned by the later tasks.

This effect is accomplished by maintaining a lightweight versioning data structure as follows. For every parameter w_i, we have a corresponding version v_i. Every v_i is initially set to 0. Whenever we begin training on a new task, we increment a version counter c, and we update all versions where $v_i = 0$ to $v_i = c$. The pruning operation for a weight w_i corresponds to resetting v_i to 0.

We use the versions to compute parameter masks that ensure that during a task's evaluation, it will only ever use the weights that it used during it's training. This is accomplished by setting the mask for w_i to zero if $v_i = 0$ or $v_i > c$ and setting it to 1 otherwise. This prevents earlier tasks from being affected by the weights trained for later tasks, and it allows later tasks to share the weights trained during earlier tasks.

However, for this case, later tasks may not modify the weights trained for earlier tasks. To ensure this, we have another mask for the gradients of the weights that is set to 1 if and only if $v_i = c$.

Because of this versioning strategy, we guarantee that performance is maintained for earlier tasks, and because of the pruning, we allow for some residual capacity to train later tasks.

Our method is space efficient and only requires the storage of a single additional number per parameter. If we have fewer than 256 continual tasks, then we can choose v_i to be an 8-bit integer which gives us a very small overhead of 1 byte per parameter. Note that this space requirement does not grow with the number of tasks being learned.

3.2 Implementation

We implemented our system in PyTorch [11]. We use the OpenAI Gym [1] environment to learn to play 3 different Atari video games: Pong, Breakout, and Space Invaders in 10 million frames. Our reinforcement learning agent was based on the Advantage Actor Critic algorithm [9], and we used an open-source implementation of the algorithm by Kostrikov [5] as a starting point for our code.

The actor in our advatange actor-critic was a 3 layer CNN with ReLU activations followed by a linear layer to produce some hidden state. The critic consisted of a simple linear layer that took in the hidden state of the actor. The final probability distribution over the actions was generated by another 2-layer MLP that took in hidden state of the CNN.

With the exception of the critic, all of the parameters for these networks were eligible pruning. We do not prune the critic since it is only necessary for training. In fact, we randomly re-initialize the critic at the beginning of each task's training session.

Proper random initialization is extremely important to convergence in neural networks [6], so during the step above where we reset all versions $v_i = 0$ to $v_i = c$, we also randomly reinitialize the corresponding weights w_i. This helps the network to escape local minima that corresponded to solutions from previous tasks.

4 Results

To evaluate our system, we ran four different experiments. In each of the experiments, we train on Space Invaders, and then Breakout, and finally Pong. For each of these training sessions, we start off with the previous set of model weights. All training sessions lasted 10 million frames.

The first experiment is the baseline experiment which made no attempt to perform continual learning, so rather than re-using weights, the weights are randomly re-initialized before training each game.

The second experiment is a naive attempt to perform continual learning. In this experiment, we perform no pruning, and simply attempt to update the model's previous weights to adapt to the new tasks.

In the third experiment, we perform pruning as described in Section 3. We use pruning to make 60% of all parameters sparse while training Space Invaders, 60% of the remaining parameters (36% of the total) sparse while training Breakout, and we do not perform pruning while training Pong.

The fourth experiment is identical to the third except that we strive for 80% sparsity.

The results are reported in Figure 1. These results represent periodic evaluations of all of the games during the training process. The scores are normalized based on the results of a random agent (which were mapped to a 0), and the results of a trained baselined agent (which were mapped to a 1).

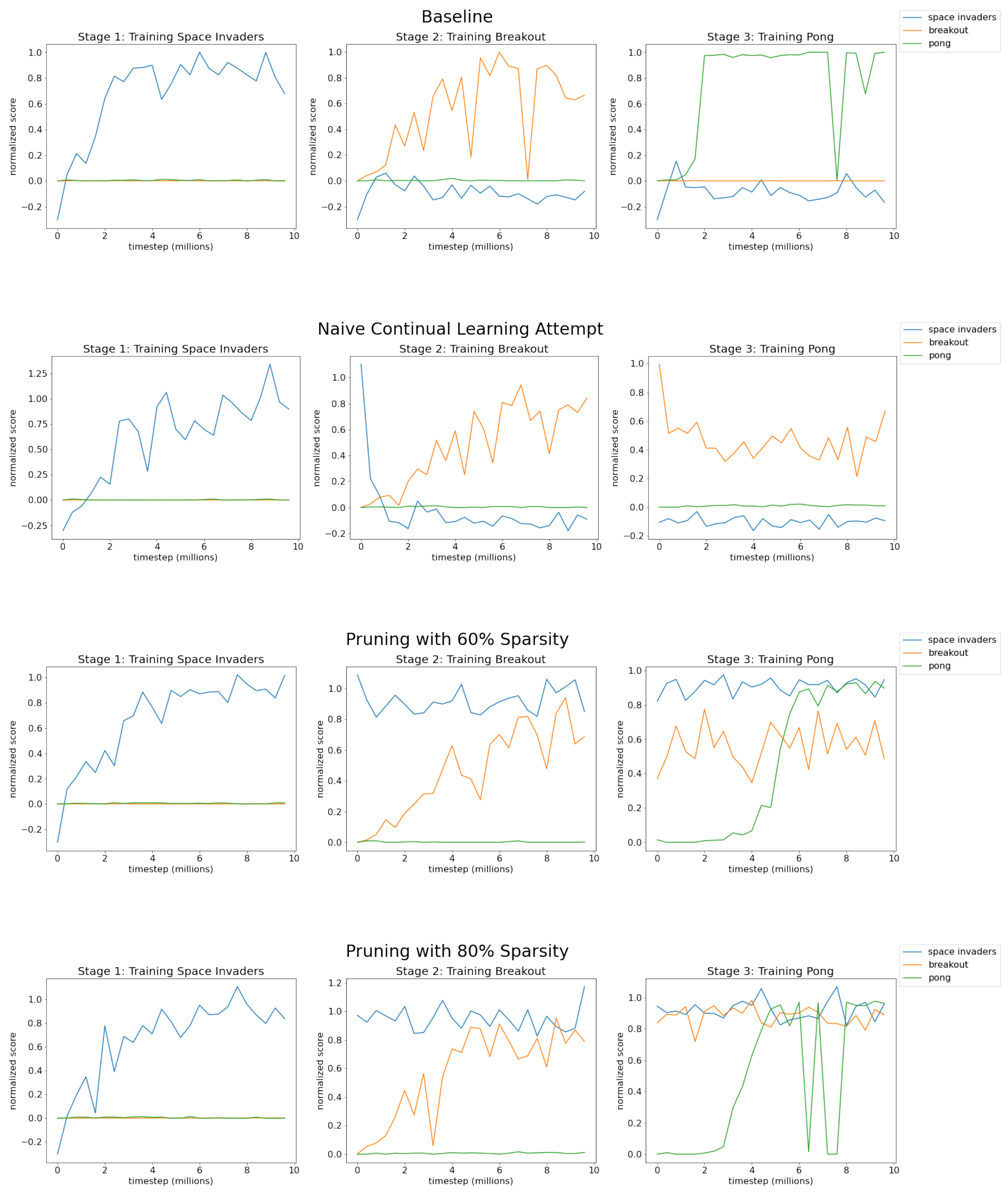

Figure 1: Results of the 4 experiments. Pruning begins at frame 4 million, and happens periodically until frame 6.4 million for the 60% case and frame 8 million for the 80% case.

The results first show that the baseline models are not intrinsically capable of playing multiple games which motivates the need for continual learning.

The second experiment in which we attempt naive continual learning shows evidence of catastrophic forgetting. The model learns to play Space Invaders during the first stage, but it quickly forgets how to play it as it learns Breakout in the second stage. Interestingly, the model seems to retain much of its performance for Breakout while being incapable of learning how to play Pong in the third stage. A possible explanation for this is the lack of random re-initialization between stages 2 and 3. Because of this, it is possible that the solution to Breakout is near to a local minimum for Pong.

The third experiment demonstrates the viability of our approach. We are able to achieve near baseline performance for Space Invaders and Pong, but performance on Breakout suffers.

The fourth experiment improves on the third and shows that the RL models can become as sparse as 80% without sacrificing any accuracy. This increase in sparsity gives greater opportunity for later tasks to learn better models. So, we ultimately end up with a model that achieves baseline level performance for all three tasks.

5 Conclusion

In this work, we have shown that it is possible to train multiple reinforcement learning agents in a continual fashion by using weight pruning without *any* loss in performance for earlier tasks. Moreover, this training was accomplished on a fixed parameter budget that did not scale with the number of tasks.

One might ask: why not simply train a smaller model to begin with? Why do we need to train large models only to prune them later? It is a well-known phenomenon that neural networks are able to train much more effectively when they have a larger number of initial parameters. A potential theory for why this is the Lottery Ticket Hypothesis [2] which posits that large randomly initialized networks are more likely to contain subnetworks whose random initializations allow them to train effectively. In our case, this allows us to learn performant sparse models and to randomly reinitialize the pruned weights to afford the same opportunities to later tasks.

Our source code can be found here: https://github.com/edjah/neuro140-final-project.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[2] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[3] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[4] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

[5] Ilya Kostrikov. Pytorch implementations of reinforcement learning algorithms. `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail`, 2018.

[6] Siddharth Krishna Kumar. On weight initialization in deep neural networks. *CoRR*, abs/1704.08863, 2017.

[7] Zhizhong Li and Derek Hoiem. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence*, 40(12):2935–2947, 2017.

[8] Arun Mallya and Svetlana Lazebnik. Packnet: Adding multiple tasks to a single network by iterative pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7765–7773, 2018.

[9] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[11] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[12] Roger Ratcliff. Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. *Psychological review*, 97(2):285, 1990.

[13] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

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