Neural Network Surgery in Deep Reinforcement Learning

Matej Buljan **Oleguer Canal** **Federico Taschin**

Abstract

Exploration on neural network architectures is still a highly time-consuming task. When testing different models, weights need to be re-trained from scratch each time, severely limiting the amount of tests one can run. While this may not be critical in supervised learning tasks, it can become prohibiting in a deep reinforcement learning (DRL) setup, where obtaining data samples is extremely slow. Furthermore, it is often the case that researchers want to try modifications of the state-action spaces or network architectures. Transferring knowledge from an agent who has previously interacted with an environment to a reasonable modification of it can thus become very handy.
In this work we present an approach for transferring weights between two similar architectures. We get inspiration from [26], where they successfully applied Neural Network Surgery to continuously train an agent to play Dota2 [22] during 11 months.
We first validate our approach in simple supervised learning tasks to later assess its performance on reinforcement learning environments. Results show significantly faster training when the new model's weights are transplanted from a previously trained one. Interestingly, this was observed even when models labelled distinct sets of classes. Moreover, it even showed beneficial when the old model was performing poorly.

1 Context

While training DRL agents, we constantly found ourselves needing to re-train from scratch policy networks each time we wanted to test state-action space or architecture changes. Since each training can take from several hours up to several months, using some kind of knowledge-transfer tool becomes essential. To our knowledge, there is no standard library which is able to perform it across distinct architectures composed by different layer types specially designed for DRL algorithms.
Researching on the topic, we found a recent article from OpenAI [26] describing a methodology for performing that task. However, OpenAI has decided not to disclose their developed code. Therefore, we set as a long-term goal to provide an equivalent open-source implementation of their algorithm. This implementation should enable to transfer knowledge across similar policy architectures of the most well-established deep learning frameworks - TensorFlow [3] and PyTorch [24] and be compatible with current DRL libraries such as Stable Baselines [16], KerasRL [25], Tensorforce [18], etc.
This project is a a first step towards this goal, where we apply some naive transfer learning techniques and evaluate their potential on simple DRL tasks.

2 Introduction

The main goal of this work is to explore *surgery* on weights across feature and architecture changes for DRL policies. We wanted to identify the main challenges and key properties of that process. Given

Project report for DD2424

a trained network and an untrained modification of it, we provide an implementation of transferring learned weights from the trained network to the untrained one. First, we pair each layer of both models while identifying removed or added layers. For paired layers, we copy the weights of the old layer to the new one accounting for input-output order consistency. New layers are initialized so that they modify the inputs they receive as little as possible. We have implemented and tested weight transplanting and initialization for networks composed by both, dense and convolutional layers for Keras [10] (with Tensorflow 1 backend) and Tensorflow 2 [3].

3 Related Work

3.1 Transfer Learning

Conventional machine learning and deep learning algorithms are trained to solve specific tasks. The models have to be rebuilt from scratch once the feature-space distribution changes. Transfer learning [23] allows to overcome the isolated learning paradigm and utilizes the knowledge acquired for one task to solve related ones.

The practice of transfer learning was popularized in the field of DL after successful attempts of transferring feature extractors across Deep Convolutional Networks (DCNN) [27]. Studies about the transferability conditions [5] [30] of their generic representations [4] show that transferring features even from distant tasks can be better than using random features. This applies especially for weights from early layers which are not specific to a particular dataset or task.

Since then, it has been applied to a wide variety of DL applications. Some examples in natural language processing (NLP) are [15] [9] [11]. In automatic speech recognition (ASR), [19] shows how to use knowledge of understanding English for understanding German.

More recent developments study domain pre-processing (domain confusion) [28] [13] where they try to represent the domains to be as similar as possible to ease the process of knowledge transfer.

Overall, transfer learning shines when modelling complex tasks with small datasets available. Off-the-shelf models pre-trained on huge datasets significantly boost performances, even with frozen weights. This makes it really useful for industry applications.

The problem we are tackling here is a slightly different one in nature compared to the ones mentioned above. The techniques that were previously mentioned focus on copying weights across big networks for supervised and unsupervised learning, sometimes for vastly different tasks. However, they do not describe how transferring weights works for DRL network policies. The main question we want to answer is: If I have trained a DRL agent, but want to try a modification of its input/output or architecture, can I train faster or get higher rewards if I do not train from scratch? In this case, the inductive transfer learning [29] paradigm should be useful as neither source nor target domains will change drastically.

3.2 Knowledge Transfer in DRL

Research in classical reinforcement learning (RL) suggests [17] [6] that transfer learning (TL) can greatly boost the agent's performance. However, when using deep neural networks (DNN) to encode policies, the transferability of the weights becomes more challenging [14], especially between drastically different tasks. Successful attempts of doing so rely on manually splitting feature extractor layers of CNN parts of the policy and only transferring those [12].

3.3 Neural Network Surgery with Sets

In 2019, OpenAI:s DRL-trained agents defeated top professional players at a game of Dota2 [21]. Training for this task is highly time consuming, even with their huge computational resources. It took the team 11 months to train an AI capable of defeating best human players. During these 11 months, they performed 20 architectural changes in the main network policies, either to adapt for different state-space changes, different actions available, or just changing the network architecture to improve its performance. They estimate that if they had to re-train from scratch each time the network needed a change, the whole process would have taken around 3 years and 4 months. They designed an algorithm based on NN surgery using sets [26] which transferred learned weights from a network to the modification without damaging performance.

4 Approach

Inspired by [26], we designed a naive algorithm to transfer weights from a network to a modification of it. Trainable parameters in DRL policies are often part of dense, convolutional and LSTM layers. For that reason, we focused our efforts on transferring knowledge between those types of layers. In contrast to some TL tasks, we opt for fine-tuning the weights, instead of freezing them for two main reasons. First, the most time-consuming part in DRL is not the forward and backward pass of the network, but rather the sampling of the environment. Second, changing state-action space may yield new relations between inputs and outputs which need to be accounted for in the modification of the model. Given two similar architectures - one trained and the other untrained, our algorithm performs the following steps:

1. **Identify trainable layer pairs** between the trained and untrained models. As for now, we use a system of layer id:s, which the user assigns to each layer of the architecture. The user must set the same id to the layers he wishes to transfer the weights from and a new id to those he wishes to reinitialize the weights. This step outputs a list of pairs of layers (trained_model_layer, new_model_layer) to transfer the weights, as well as a list of layers which are to be re-initialized.

2. **Transfer weights between layer pairs**. As for now, we have implemented this functionality with Dense and Conv2D layers. The basic idea relies on propagating through the network a series of input-output id:s to correctly copy pre-trained weights to the untrained layer. If the layer increases in shape, new weights are left untouched from whichever initialization the user set at the beginning. For this to work, the user must provide the first layer input id:s and last layer output id:s. This both ensures similar operations are applied to each corresponding input feature, and its output is correctly matched to the desired one.

    - **Dense Layers**: Thanks to input-output id:s transmission along the new and old network, the transfer job of Dense layers is very simple. A change (addition, removal or permutation) in input features' id:s is translated into the columns of the weight matrix. Similarly, a change of output id:s is translated into the rows of weight and biases matrices. When adding new inputs or outputs, weights are left untouched from the untrained initialization.
    - **Conv2D Layers**: In this case, we use the same technique but inputs' id:s refer to input channels and output id:s to the layer filters. A change (addition, removal or permutation) between the trained layer and new layer of input id:s is applied into each filter's channel. A change of output ids is translated into the same change in the filters of the layer. This is done to maintain the operations that each input receives through the network.

3. **Initialize new layers**. When the user adds a new layer into the model, we need it to change the overall behaviour as little as possible. To achieve this, we initialize the new layers to be as close to identity as possible.

    - **Dense Layers**: In this case, we first set the weight matrix to be as close to identity as possible and then apply the needed transformations according to input-output id:s. Biases are always initialized to zero.
    - **Conv2D Layers**: For convolutional layers, we set the filters to be zero but a one in top-right positions for all channels. This ensures the input and output are similar enough so the weights previously learned in posterior dense layers are not compromised.

Notice that all these operations are agnostic to the deep learning framework used. To cope with that, we have a "core" implementation which performs the described tasks. On top of this, we adapt the code to work for the different frameworks available.

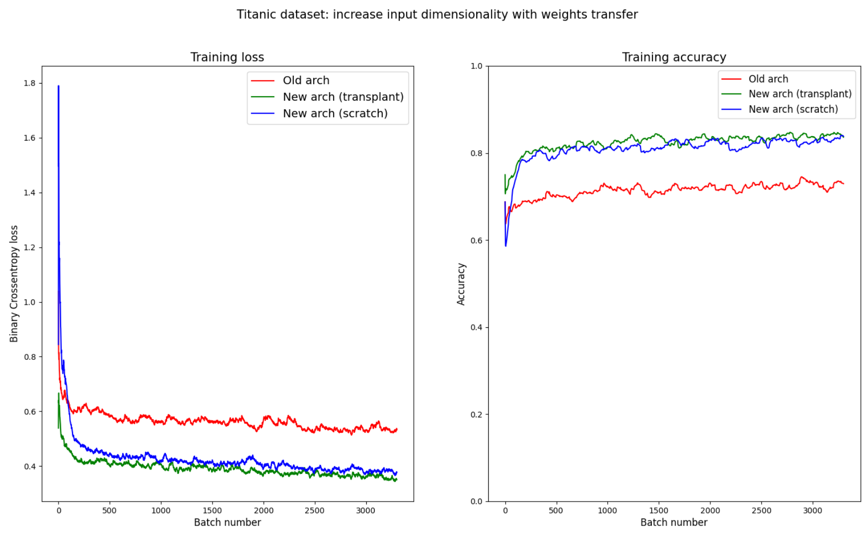

Figure 1: Comparison of pre and post weight transfer on the titanic dataset. In red, the old model that uses only a subset of the inputs. In green, the new model that uses all the inputs, and received the weights with the transfer. In blue, the same architecture of the new model, but trained from scratch.

	Old model	New model (transfer)	New model (scratch)
Loss	0.615	**0.5**	0.52
Accuracy	0.68	0.78	**0.79**

Table 1: Loss and accuracy on the Titanic test dataset

5 Results

5.1 Supervised Learning

In this section, we test our implementation on well-known supervised learning datasets. While these results are not very interesting on their own, it was important for us to perform a sanity check of our surgery before tackling a more complex task such as DRL. Since extensive research 3 on this area has already been done we will not get into much detail.

5.1.1 Titanic Dataset

Titanic dataset [2] contains data about survival of passengers in the famous sinking of the RMS Titanic ship. Recently, the dataset has been used as benchmark in the Kaggle Titanic Challenge. The goal is to predict the survival of passengers based on their available data. We use this dataset to test the weight transfer approach. Our pre-processed dataset contains 1309 17-dimensional samples, where categorical data has been one-hot encoded.

Increasing input size In this experiment we train a model by using only a subset of the input, the first 8 columns. Model architecture: Input(8), Dense(500), Relu, Dense(1), Linear. We train the model for $n = 100$ epochs. Then, we create a new model, this time taking all the 17 inputs. New model architecture: Input(17), Dense(500), Relu, Dense(1), Linear. We transfer the weights from the old trained model to the new one, and we train for $n = 100$ epochs. Figure 1 shows the results, together with the same architecture of the new model, but trained from scratch for $n = 100$ epochs. Results show that although the old model performed poorly, the model that received its weights starts its training from a lower loss and higher accuracy, and performs slightly better than the same architecture trained from scratch. Table 5.1.1 show the evaluation of the three models on the test dataset. The model achieves smaller loss, but has slightly lower accuracy in the test dataset.

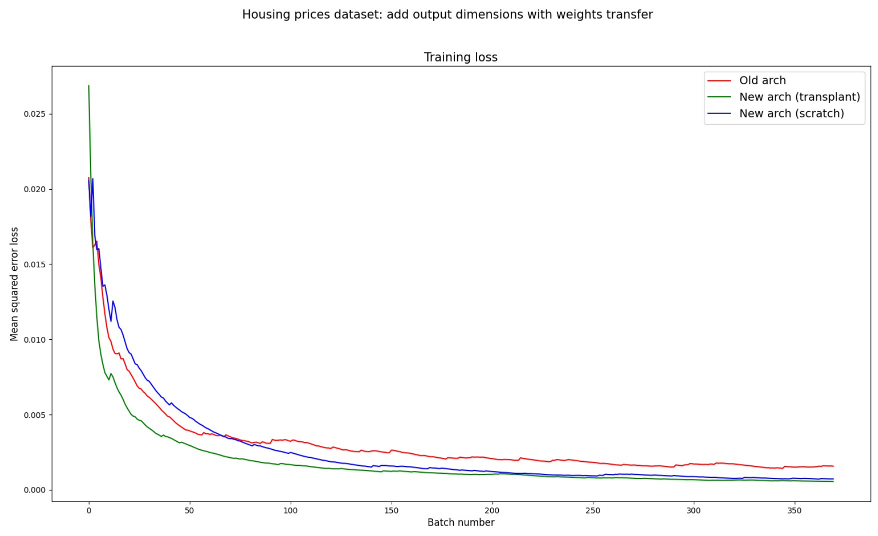

Figure 2: Comparison of pre and post weight transfer. In red the old model. In green the new model that received the weights with the transfer operation. In blue the same model but trained from scratch.

	Old model	New model (transfer)	New model (scratch)
MSE	0.00179	**0.00086**	0.00094

Table 2: Mean Squared Error on House Pricing Dataset test set

5.1.2 House Prices Dataset

The House Prices Dataset [1] contains data about 1460 houses in Ames, Iowa (US). Each sample is described by 80 parameters, such as number of floors, dimension, position, etc. After a preprocessing phase, our dataset contains 1460 rows and 35 columns. In particular, all columns describing the dimension of the floors have ben merged into a single column "*Total Square Meters*".

Adding outputs We train the model to predict the *Price* column using 33 of the remaining 34 columns, leaving *Total Square Meters* apart. The model architecture is: Input(33), Dense(500), Relu, Dense(1), Linear. We train the model with MSE loss for $n = 10$ epochs. Then, we create a new architecture equal to the previous, but with an additional output, that we will use to predict the *Total Square Meters* column. The new model architecture is: Input(33), Dense(500), Relu, Dense(2), Linear. We transfer the weights from the previous model and we train for $n = 10$ epochs. Figure 2 shows the results, together with the same architecture but trained from scratch Results show that the model that received the weights trains faster, and Table 5.1.2 shows that it also achieves better performances on test data.

5.1.3 MNIST Dataset

Different classes To check our transferring algorithm on ANN:s with convolutional layers we decided to run the following experiment on the MNIST [20] dataset: First, we train a model to classify the numbers from 0 to 3, then we design a new architecture, apply our transfer function and use it to classify the numbers from 4 to 9. Both model architectures are: Conv2D(32, shape=(3, 3)), Relu, Conv2D(16, (3, 3)), Relu, MaxPool2D((2, 2)), Flatten, Dense(100), Relu, Dense(number of classes), SoftMax. In this case, as is common practice in TL between CNN:s [27], we just transferred the weights of the first convolutional layers. Nevertheless, we did not freeze these weights for the reasons described in 4. As figure 3 shows, transferring the learned feature extractors from first two layers significantly speeds up the training process even for a different set of classes. Table 5.1.3 shows the test results.

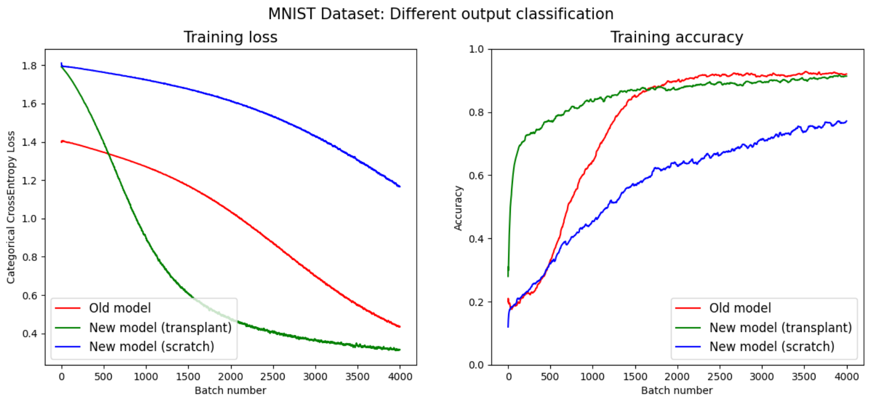

Figure 3: Comparison of pre and post weight transfer on the MNIST dataset. Notice how the model with transferred weights achieves higher accuracy values much faster than a model with random initialization.

	Old model	New model (transfer)	New model (scratch)
Loss	0.352	0.311	1.17
Accuracy	0.94	0.91	0.79

Table 3: Loss and accuracy on the MNIST test set.

5.2 Reinforcement Learning

5.2.1 Cartpole Problem

In the Gym [8] CartPole problem we aim to balance a pole by controlling the cart on which it is attached. We obtain a positive reward for each time step in which the pole is tilted less than a certain angle θ. The state space is a 4-dimensional vector containing the cart position, cart velocity, pole angle, and velocity of the pole at the top. There are two possible actions which correspond to applying force in the left or right direction. We perform experiments by training the agent with the DQN algorithm for n steps, then we modify the network architecture and transfer the weights. We then train the new model for n steps and finally compare it to the same model trained from scratch.

Increasing network inputs. We pretend that our agent is only able to observe the cart velocity and pole angle. This first architecture is: Input(2), Dense(16), Relu, Dense(2), Linear. We train the model for $n = 8000$ steps. Then, we create a new model with the same hidden and output layers, but this time taking all 4 observations as input. We perform the weight transfer operation and we train the new model for $n = 8000$ steps. Figure 4 shows the results together with the new architecture trained from scratch for $n = 8000$ steps. The first model performs poorly, not being able to properly observe the environment. After transfering the weights with the surgery technique, the new model learns faster and better than the same architecture trained from scratch. This lets us infer that, even not performing correctly, the agent understood some basic dynamics of the environment which were later key to boost its performance.

Adding hidden units. We start from a model that takes all observations as inputs. The architecture is: Input(4), Dense(4), Relu, Dense(2), Linear. We train the model for $n = 8000$ steps. Then, we transfer the weights to a new architecture in which the 2 hidden layers now have 16 units each, and we train the new model for $n = 8000$ steps. Figure 5 shows the results, together with the new architecture trained from scratch (using He initialization) for $n = 8000$ steps. Again, we see that the model that received the weights trough the surgery technique trains faster, despite the first model poor performances.

Adding a layer. We start from a model with a single hidden layer of 8 units and we train it for $n = 8000$ steps. We then transfer the weights to a model that has an additional hidden layer of 8

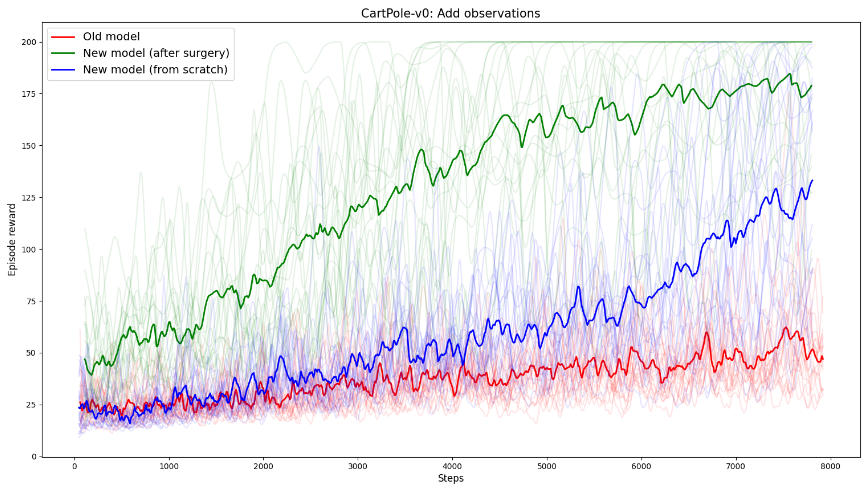

Figure 4: Comparison of pre and post surgery performances in the CartPole problem. In red, the first model, that has only two inputs. In green, the new model that has four inputs and received the weights from the previous. In blue, the same architecture of the new model but trained from scratch. Plots are the mean of 20 runs. Results of each run are plotted in transparency. Notice that despite performing poorly, the first agent grasped some underlying dynamics of the environment which where later key to boost the learning capabilities of the transferred agent.

units, and we train it for $n = 8000$ steps. Figure 6 shows the results. The model that received the weights trains faster than the same model trained from scratch.

5.2.2 Acrobat Problem

In the Gym [8] Acrobot-v1 problem we have two links connected with two joints. At the beginning the links point downwards, and our goal is to make them swing up to a given height. We have three actions to control the first joint, that correspond to applying torque in the clockwise or counter-clockwise direction, or to not apply any force. We observe a 6-dimensional state made of the sin and cos of the two angles of the two joints, and the two angular velocities.

Adding one action We train the first model by allowing it to only perform the actions *Apply force clockwise* and *Do nothing*. The model architecture is: Input(6), Dense(16), Relu, Dense(16), Relu, Dense(2), Linear. We train the model for $n = 20000$ steps using DQN algorithm. Then, we allow the model to perform also the action *Apply force counter-clockwise*, by adding a third unit in the last layer. We transfer the weights from the previous model and we train for $n = 20000$ steps. Figure 7 shows the results. In this case, the results do not show any clear difference between the model that received weights and the same model trained from scratch, but the transfer operation did not damaged the learning capabilities of the model.

6 Discussion and Future Work

More thorough theoretical reasoning and further empirical testing is needed to do take any general conclusion. Specially we should evaluate our transfer method on bigger DRL policies and more complex learning algorithms. Nevertheless, analyzing the presented results it seems that weight transfer in DRL presents good potential to be exploited. Other interesting ideas to consider are network prunning techniques [7] to better identify which weights are more useful than others.

In addition, a deeper understanding about when it actually harms to transfer the weights in DRL is needed. While we didn't encounter a case in this project, it is very plausible to think that

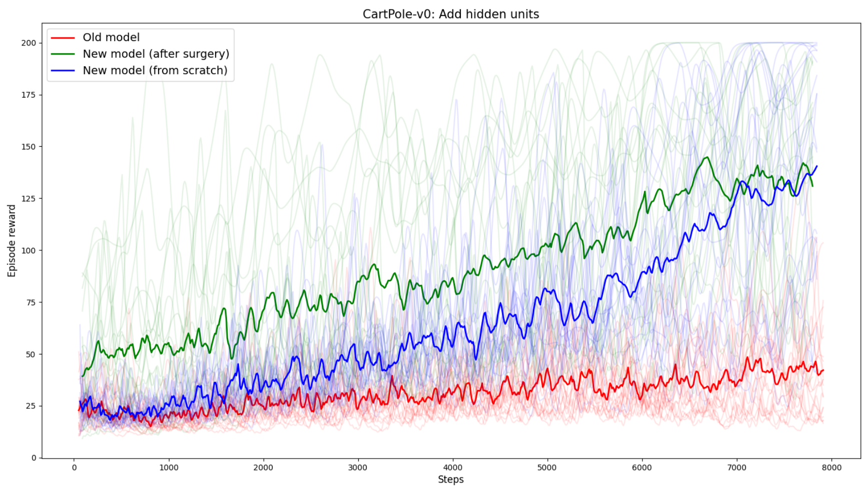

Figure 5: Comparison of pre and post surgery performances in the CartPole problem. In red the first model, with 2 hidden layers of 4 units each. In green, the new model with 2 hidden layers of 16 units each, that received the weights from the old model. In blue, the same architecture of the new model but trained from scratch. Plots are the mean of 20 runs. Results of each run are plotted in transparency.

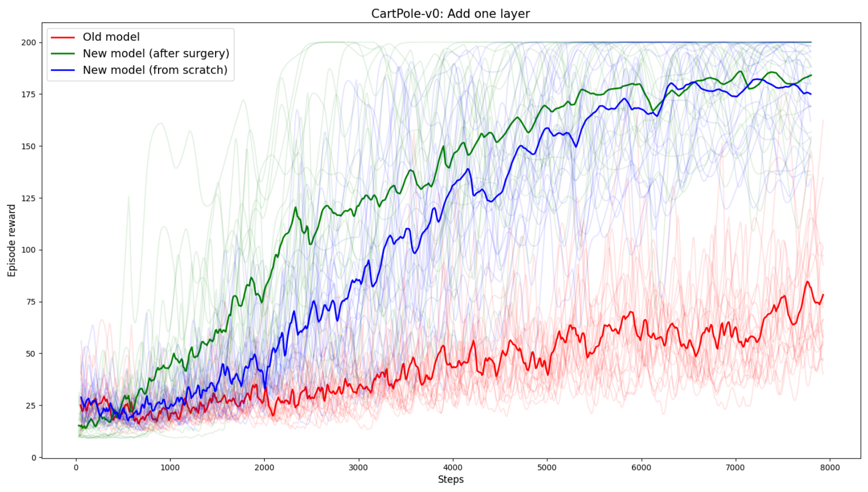

Figure 6: Comparison of pre and post surgery performances in the CartPole problem. In red the first model, with a single hidden layer of 8 units. In green the new model with two hiddel layers of 8 units each, that received the weights from the first. In blue the same architecture of the new model, trained from scratch. Plots are the mean of 20 runs. Results of each run are plotted in transparency.

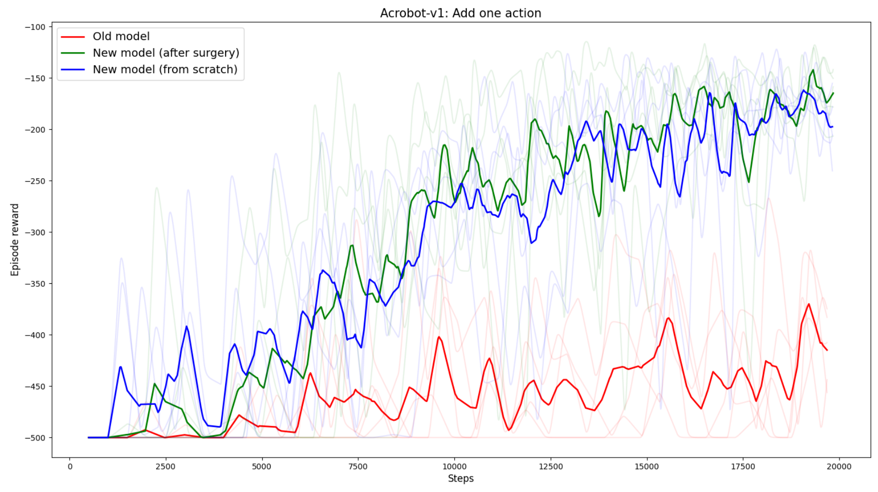

Figure 7: Comparison of pre and post transfer operation. In red, the old model, that used only 2 actions. In green, the new model that uses 3 actions and received weights with the transfer operation. In blue, the same model trained from scratch. Plotted lines are the averages of 5 runs. Results of each run are plotted in transparency.

transplanting weights can bias the initial conditions of the policy in a way that makes it under-perform.

Tacking a step further, is it possible to transfer knowledge across totally different tasks? As [5] [30] mention, it looks very hard unless only re-using non task-specific feature extractors such as the ones present in early layers of the model. Still, is there something we can do to the environment to decrease this task dependency? Can there be a way of extracting general cognition relationships rather than directly copying weights?

7 Conclusion

This work studies the application of transfer learning in a deep reinforcement learning framework. We first present an algorithm to detect which weights of a neural network can be salvaged after a slight modification of its architecture. Later, we run a successful sanity check of our method on three well-known supervised learning tasks, checking how it performs when:

- modifying the input space on the Titanic classification task,
- adding a new output in the House Prices regression task,
- distinguishing different classes in a the MNIST visual classification task.

Finally, we analyze its performance transferring the weights between policies of two basic DRL benchmark environments, empirically testing the effect of:

- changing state-space by adding new observations in Cartpole Gym environment,
- changing hidden layers number of nodes in Cartpole Gym environment,
- adding hidden layers in Cartpole Gym environment,
- adding possible actions in Acrobat Gym environment

In neither of the above mentioned examples the model with transferred weights under-performed the randomly-initialized one. While these studies are not conclusive, we can at least say that this approach has the potential to significantly speed up iteration time when debugging DRL problems. We will continue to explore this idea and will try to provide an open-source tool to ease this iteration process.

References

[1] House prices kaggle dataset. https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data.

[2] Titanic kaggle dataset. https://www.kaggle.com/c/titanic/data.

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[4] Hossein Azizpour, Ali Sharif Razavian, Josephine Sullivan, Atsuto Maki, and Stefan Carlsson. From generic to specific deep representations for visual recognition. Technical report, 2015.

[5] Hossein Azizpour, Ali Sharif Razavian, Josephine Sullivan, Atsuto Maki, and Stefan Carlsson. Factors of Transferability for a Generic ConvNet Representation. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 38, pages 1790–1802, 2016.

[6] Bikramjit Banerjee and Peter Stone. General game learning using knowledge transfer. Technical report, 2007.

[7] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning?, 2020.

[8] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[9] Daniel Cer, Yinfei Yang, Sheng yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Universal sentence encoder, 2018.

[10] François Chollet et al. Keras. https://keras.io, 2015.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[12] Shani Gamrian and Yoav Goldberg. Transfer learning for related reinforcement learning tasks via image-to-image translation. 2018.

[13] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-adversarial training of neural networks. 2015.

[14] Ruben Glatt, Felipe Silva, and Anna Costa. Towards Knowledge Transfer in Deep Reinforcement Learning. pages 91–96, 2016.

[15] Yoav Goldberg and Graeme Hirst. *Neural Network Methods in Natural Language Processing*. Morgan Claypool Publishers, 2017.

[16] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[17] George Konidaris and Andrew G. Barto. Autonomous shaping: knowledge transfer in reinforcement learning. In *ICML '06*, 2006.

[18] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. Tensorforce: a tensorflow library for applied reinforcement learning. Web page, 2017.

[19] Julius Kunze, Louis Kirsch, Ilia Kurenkov, Andreas Krug, Jens Johannsmeier, and Sebastian Stober. Transfer learning for speech recognition on a budget, 2017.

[20] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[21] OpenAI. Openai five. https://blog.openai.com/openai-five/.

[22] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with Large Scale Deep Reinforcement Learning. Technical report, 2019.

[23] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[25] Matthias Plappert. keras-rl. `https://github.com/keras-rl/keras-rl`, 2016.

[26] Jonathan Raiman, Susan Zhang, and Christy Dennison. Neural Network Surgery with Sets. dec 2019.

[27] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. CNN features off-the-shelf: An astounding baseline for recognition. Technical report, 2014.

[28] Baochen Sun, Jiashi Feng, and Kate Saenko. Return of frustratingly easy domain adaptation, 2015.

[29] Dongrui Wu and Thomas D. Parsons. Inductive transfer learning for handling individual differences in affective computing. Technical Report PART 2, 2011.

[30] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.