Transfer learning from image segmentation to reinforcement learning

Mathieu Béligon (mathieu.beligon@mail.mcgill.ca; ID: 260983485)

April 30, 2020

1 Abstract

The purpose of this project is to study the possible impact of transfer learning between an image segmentation task, and a reinforcement learning one, based on the same space of images. I demonstrated this approach on the NES version of Mario Bros. A first model is built to perform an image segmentation on frames from the game. Then, I transferred this knowledge into a Double DQN agent. I showed that it lowers the number of training steps required, compared to a Double DQN agent trained from scratch.

External links: Youtube Video - Github repo

2 Related Work

Gamrian and Goldberg [2018] explored the idea of transfer learning among pixel based RL tasks. They showed that it was it was inefficient to try to do a direct transfer learning between different RL environments. Even in the case of the states are slightly modified (like for a different level in the game, or by adding perturbations to the images), it was better to re-train from scratch than to try to fine-tune. Nonetheless, they managed to resolve this issue using GANs in order to map the new space of images into the space of images in which the first RL model was trained.

3 Models

3.1 Double DQN

For the Double DQN agent that served as a baseline for my comparison, I used the same architecture and training strategy as described in this article.

As shown on Fig.5, the network for the Double DQN agent is composed of 3 Convolutional layers, followed by two dense layers. The input of the network is a stack of $84 \times 84 \times 4$ grayscale image, with values between 0 and 1, and the output are the predicted Q-values for each of the 5 possible actions that the agent can take. The convolutional layers have respectively 32, 64 and 64 filters, kernels of 8, 4, 3 and strides of 4, 2, 1. The hidden dense layer is composed of 512 rectifier units. All activation function for hidden layers are ReLU.

I've also used a double QDN as proposed by van Hasselt et al. [2015]. In addition to the online network, we keep a previous copy of it, in a target network. This second network is copied from the other at a low frequency. It resolves the issue of having Q values too high like we can observe in a standard Q-learning, but is more simple to implement than the double Q-learning. The q-value we are targeting for a state S and action A are: $R_{t+1} + \gamma Q_{target}(S_{t+1} argmax(Q_{online}(S_{t+1}, A_t)))$, where R_{t+1} and S_{t+1} are the the rewards and next states observed after taking action A on state S.

Lastly, I used experience replay (Lin [1992]) to train the agent.

3.2 POC

As a proof-of-concept for what I wanted to do, I trained a second Double-QDN agent, with the exact same architecture and settings. The only difference is that, instead of taking the actual frame as input, I first performed an image segmentation on it. The input of the Double DQN agent is the a stack of 4 images, each detected object on the image being replaced by a specific gray color (Fig.3), depending on the type of object.

The idea of this POC is to show that, by condensing and simplifying the information contained on the picture, we can improve the learning process.

3.3 FCN

Fully Convolutional Networks are a relatively simple architecture used to perform image segmentation through deep learning. In a first part, convolutional layers are applied on the image. Then, a chain of deconvolutional

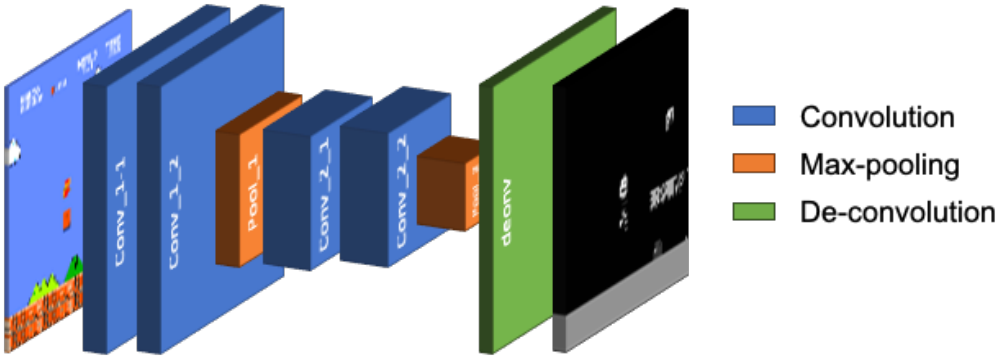

Figure 1: Architecture for my simplified FCN, with 2 poolings

layers are used to predict the classes. In the original paper (Long et al. [2014]), 3 different architectures are proposed. But the number of convolutions doesn't change between those architectures, and each require 5 pooling, and 13 convolutional layers (see Fig.6).

I simplified the architecture, keeping only 2 pooling layers (see Fig.3.3), with a maximum of 2 convolutional layer before each, and 1 deconvolution, as our segmentation problem is far more simple than real world images.

For the naming of my architectures, I used 'f' before the number of filters, 'k' for the number of kernels for the convolutional layers, and 's' for the number of strides of the pooling layer. The kernel of the deconvolutional layer is prefixed by 'd'. Thus, 'f16-k3' refers to a convolutional layer with 16 filters and a kernel of size 3. 'fcn__f16-k3_s2__f32-k3_f32-k3_s2__d8' refers to a chain of 1 convolutional layer, 1 pooling, 2 convolutional, 1 pooling, and finally 1 deconvolutional.

3.4 Double-DQN FCN based

Once the FCN model trained, we can remove the deconvolutions, and replace them with two dense layers, one hidden of 512 rectifier units, and the last output layer. As I froze the convolutional and pooling layers trained by the FCN, I choose to add another convolutional layer after them, to help the agent to gain contextual information.

To achieve a good quality in the segmentation with the FCN, I had to train it on colored images. Thus, the input of the model is now $84 \times 84 \times 3 \times 4$, higher than the previous ones.

4 Experiments

The repo for my experiments can be found on my public Github, along with instructions to reproduce in the README. No specific infrastructure is needed, as I ran it on my computer on CPU (but it takes time).

4.1 Environment

For my experiments, I used the gym-like environment gym-super-mario-bros (Kauten [2018]), and restricted the actions Mario could perform to right only.

I changed the rewards as follow, so that Mario's objective is to finish the level as fast as possible, avoiding death and collecting points along the way.

- -300 if dead;
- -1 per second (in the game time);
- +1 if mario's x position increased by 2 since last frame;
- +1 per each 25 points gained by mario (by collecting points or killing an enemy).

I also pre-processed the frames, skipping 3 out of 4 (to avoid too close data points), resizing them to squared grayscale images of 84×84 pixels.

4.2 Double DQN

The Double DQN agent was trained over 10K episodes, with a learning rate of 0.00025. I used an exploration ratio decaying, from 1 to 0.02, with a decay factor of 0.999975. For the double-q learning, I took $\gamma = .9$,

name	accuracy	fps
fcn__f32-k3_f32-k3_s2__f64-k3_f64-k3_s2__d8	99.3%	14.1
fcn__f32-k3_s2__f64-k3_f64-k3_s2__d8	99.2%	15.5
fcn__f16-k3_s2__f32-k3_f32-k3_s2__d8	99.2%	32.6

Table 1: Performance for various FCN architectures[1], on the test set.

and the target model was updated every 10K steps. For experience replay, I used a memory of the last 100K steps, replaying a batch of 32 experiences every 4 steps. Finally, I waited 10K steps before starting to train the networks, to avoid a too high correlation of data at start.

For the architecture and the parameters, I was influenced from the settings used by Mnih et al. [2013] on the Atari games. I have reduced the minimum value of the exploration rate, because I found that Mario died too easily with a higher value, since a jump at wrong time can cause it, thus it was difficult for the agent to ever reach the end of the level.

4.3 POC

For this first segmentation, I used a simple sprite matching rule, checking the value of the pixels in each blocks.

For the moving objects, I used the *matchTemplate* function from OpenCV (Bradski [2000]).

I managed to achieve a 40 FPS segmentation of the images.

The same Double QDN architecture, with the exact same parameters as in 4.2, were applied to train the agent on the segmented images.

4.4 FCN

The images were generating by letting the random agent play. To avoid having data points too much correlated, I saved only 1 out of 10 stack of frames. I also added some frames generated with a Double DQN agent, to have samples from the end of the level.

696 stacks were used to train the models, using a randomized batch generator, and 134 were kept for testing purpose.

The learning rate used to train the models was of 0.001, and 30 training epochs were done.

4.5 Double DQN FCN-based

This model uses colored images. Thus, I reduced the memory size to 50K. Also, the loss computation was taking too much time. To take better advantage of the batch processing, I reduced the frequency of learning, to compute the loss only every 16 frames, and increased the batch size to 128, to keep the same ratio.

5 Results

5.1 FCN segmentation

Table.1, show some metrics for the 3 best models that I've tried out. As expected, we can obtain a very good accuracy, since the images are tiles based. See Fig.8 for some examples of the segmentation obtained.

The performance is similar if 1 or 2 convolutional layers are used before the first pooling. The number of filters can also be halved without drawback. Since it was faster, I thus kept the fcn__f16-k3_s2__f32-k3_f32-k3_s2__d8 architecture: a first convolution of 16 filters, then a pooling of stride 2, followed by 2 convolutions of 32 filters, a final max pooling of 2 strides, and, in the end, a deconvolution with a kernel of size 8.

5.2 Double QDN agents

Fig.2 presents the evolution of the x position achieved by the agents before the end of the episode, during training. It should be compared to the random agent, that achieved a x position of 475.75 on average over 20 episodes.

As we can see on those graphs, both the model trained on the ground truth images (blue), and the model obtained through transfer learning (red), achieved better results than the original Double DQN agent (orange). Furthermore, we can see that they converge faster, in about 2K episodes, while the latter took more than 7K episodes.

[1] See 3.3 for details about notation

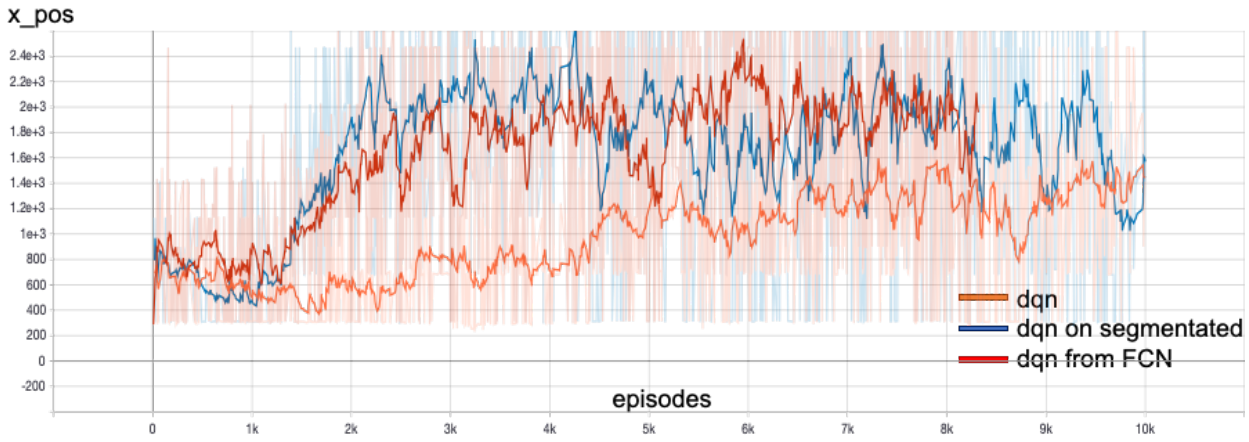

Figure 2: Evolution of the x position achieved by the agents (3162 is the position of the end of the level)

The drawback is that the time of each episode is increased due to the complexity of the model. Thus, the first DQN agent took only 12 hours to train, while the one obtained through transfer learning took more that 2 days.

5.3 What didn't work

At first, I tried to train a lighter version of FCN8s in order to fasten the inference. To do so, I tried to skip the last VGG pooling layers, since they are a macro-representation of the image (32 pixels). The FCN training failed, and I observed a divergence during the first epochs. I'm not completely sure why this failed, but it may be because I used the pre-trained weights of the VGG network, without going through all the convolution process. Anyway, I found later that I could construct my own, more simple, architecture, with very good performance and speed.

I tried to train a FCN to use grayscale images as input, but the results were really bad, so I kept colored images.

6 Future work

The comparison between the Double DQN agent trained from scratch and the QDN agent obtained from transfer learning is not completely fair. Indeed, the complexity of the models are not the same. The former uses only 3 convolutional layers on grayscale images, while the later uses 4 convolutional layers and 2 pooling layers. We should try to compare with the same architecture.

Another thing I would have like to do is to explore unfreezing the FCN layers, when we train the agent.

Finally, it could be interesting to see if the model obtain through a transfer learning from image segmentation generalize better than what Gamrian and Goldberg [2018] obtained with regular models. Would a DQN trained only on light levels, but based on a FCN trained on both dark and light levels, be re-usable on dark levels, after fine-tunning ?

7 Conclusion

The results I obtained with my setup are encouraging for transfer learning between image segmentation and reinforcement learning. We can indeed reduce the amount of experience needed in order to have a usable agent.

The main drawback is that the time needed to process the experiments is bigger, due to the increase in models complexity. In the case of games, that can be played at any speed, it tends to increase the training time.

Nonetheless, we can think of cases were this drawback wouldn't be an issue. For instance, for real world robots, there are time constraints that don't depend upon the agent, but that are inherent to the environment. In that case, reducing the number of experiences needed would reduce the training time.

References

G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

Shani Gamrian and Yoav Goldberg. Transfer learning for related reinforcement learning tasks via image-to-image translation. *CoRR*, abs/1806.07377, 2018. URL http://arxiv.org/abs/1806.07377.

Christian Kauten. Super Mario Bros for OpenAI Gym. GitHub, 2018. URL https://github.com/Kautenja/gym-super-mario-bros.

Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3):293–321, 1992. doi: 10.1007/BF00992699. URL https://doi.org/10.1007/BF00992699.

S. Liu and W. Deng. Very deep convolutional neural network based image classification using small training sample size. pages 730–734, 2015.

Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *CoRR*, abs/1411.4038, 2014. URL http://arxiv.org/abs/1411.4038.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A survey on deep transfer learning. *CoRR*, abs/1808.01974, 2018. URL http://arxiv.org/abs/1808.01974.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL http://arxiv.org/abs/1509.06461.

8 Reproducibility checklist

For all models and algorithms presented, check if you include:

- **A clear description of the mathematical setting, algorithm, and/or model.** See 3(Models).
- **A clear explanation of any assumptions.** My only assumption was that image segmentation was more simple in my case than with real-world images. See 3.3(FCN).
- **An analysis of the complexity (time, space, sample size) of any algorithm** No such analysis was needed.

For all datasets used, check if you include: (my only dataset is the one used to train a segmentation model)

- **The relevant statistics, such as number of examples** See 4.4(FCN).
- **The details of train / validation / test splits.** See 4.4(FCN).
- **An explanation of any data that were excluded, and all pre-processing step.** No data was excluded. For pre-processing of images from the environment, see 4.1(Environment).
- **A link to a downloadable version of the dataset or simulation environment.** All the images and sprites used for the segmentation can be found in my github repo.
- **For new data collected, a complete description of the data collection process, such as instructions to annotators and methods for quality control.** Not relevant in this case.

For all shared code related to this work, check if you include:

- **Training code.** See src folder in the github repo.
- **Evaluation code.** See the src folder in the github repo.
- **Pre-trained model(s).** See the exp folder in the github repo.
- **README file includes table of results accompanied by precise command to run to produce those results.**

For all reported experimental results, check if you include:

- **The range of hyper-parameters considered, method to select the best hyper-parameter configuration, and specification of all hyper-parameters used to generate results.** Due to limitation of my setup (I only have access to the CPU on my computer, and can't use any GPU), and the time required to run one experiment (in dozen of hours), I wasn't able to do an hyper-parameter search.
- **The exact number of training and evaluation runs.** Due to the same limitation as above, I was able to perform only 1 run per experiment.
- **A clear definition of the specific measure or statistics used to report results.** The metrics I used (accuracy for segmentation, and x position of Mario at end of episode) were clear enough.
- **A description of results with central tendency (e.g. mean) variation (e.g. error bars).** I couldn't with only 1 run.
- **The average runtime for each result, or estimated energy cost.** Time for each experiment is given in 5(Results)
- **A description of the computing infrastructure used.** See 4(Experiments).

Figure 3: Inputs received by the first (left) and second (right) Double DQN models

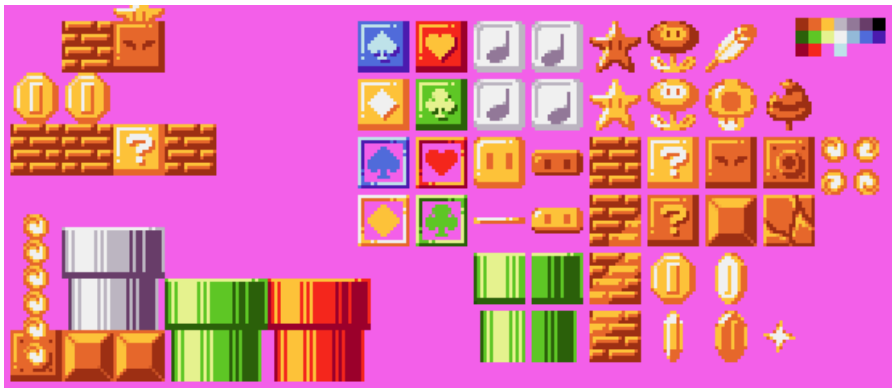

Figure 4: Sprites used in Mario

9 Appendix

9.1 Additional Figures

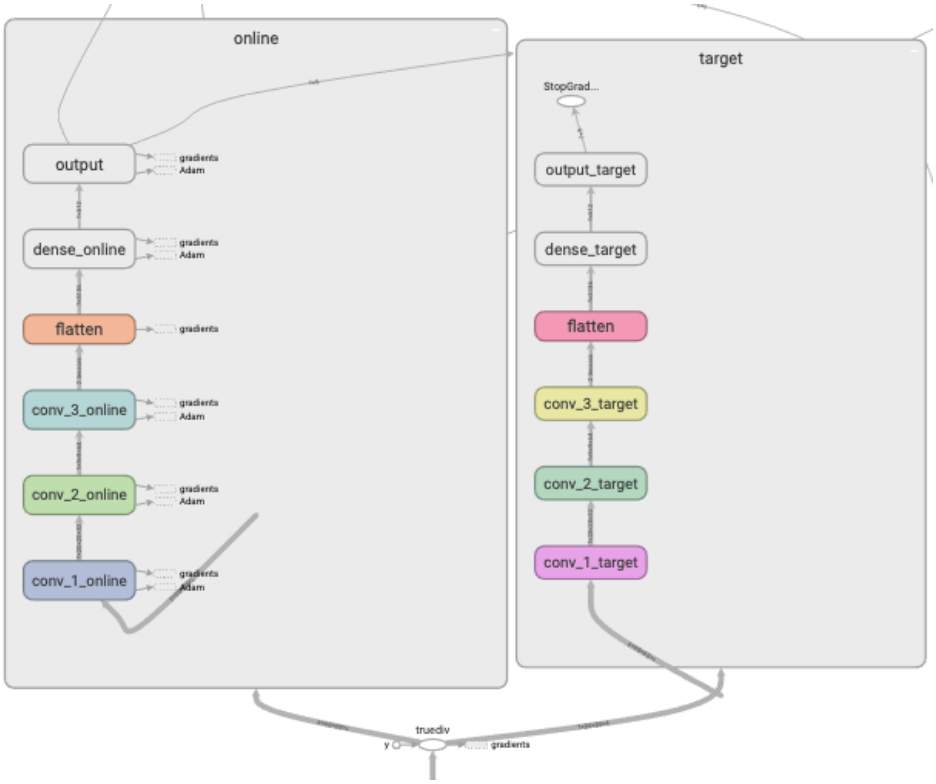

Figure 5: Architecture for the Double DQN agent

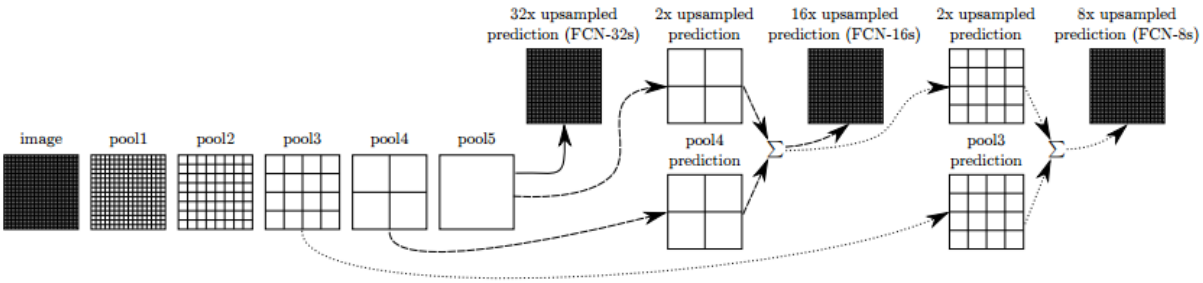

Figure 6: FCN architecture

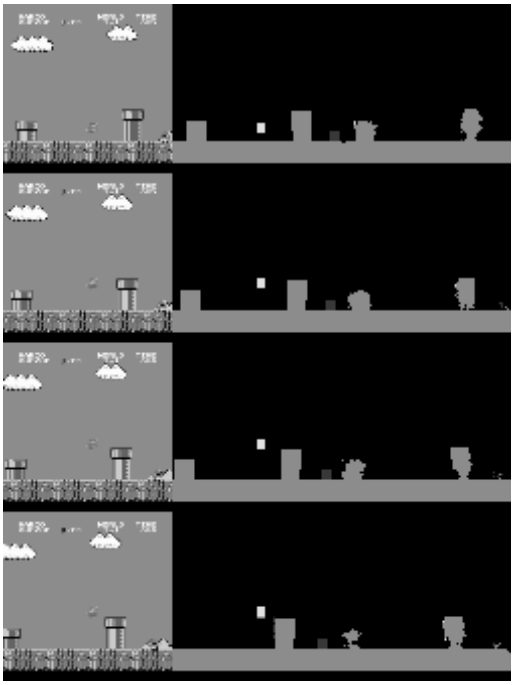

Figure 7: Results of segmentation using grayscale inputs, wit a f16-k3_s2__f32-k3_f32-k3_s2__d8 architecture. From left to right: input, expected and predicted.

Figure 8: Results of 3 different FCN architecture, on the same set of stacked frames.