CS230

Universal Representation Learning for Faster Reinforcement Learning

Alex Loia
Department of Computer Science
Stanford University
alexloia@stanford.edu

Alex Nam
Department of Computer Science
Stanford University
hjnam@stanford.edu

Abstract

Traditional refinforcement learning (RL) approaches, like Q learning, excel when environment states are provided for training agents. However, many RL applications do not have access to hand-crafted state features and need to work with pixel data. Naively running these images through Q learning, however, performs quite poorly across most environments. Instead, we explored a variety of representation learning methods, based on Contrastive Unsupervised Representation Learning (CURL) [7], to provide relevant state encodings learned from task-aware image data that are then used to learn optimal actions via Q learning. We found that our encoding approaches enable RL agents to outperform purely pixel-based deep Q learning but leave room for improvement when compared to the state-based baseline.

1 Introduction

Reinforcement learning (RL) is a type of unsupervised learning, where an agent learns to act optimally through interactions with the environment, which returns a next state and reward given some current state and the agent's choice of action. Thus, a policy, a reward signal, and a model of the environment define RL learning systems [13]. The learning objective is to learn a policy, mapping states to actions, in order to maximize rewards from the environment.

In this project, our goal is to learn lower-dimensional representation for RL from pixel data. In many realistic settings, such as mobile robots and self-driving vehicles, RL agents do not have access to underlying state features and instead need to work directly with high-dimensional pixel inputs. To mitigate the burden of handcrafting state encodings, we have applied several deep learning techniques for learning representations directly from the pixels jointly with the RL objective. Specifically, we focused on state-of-the-art technique, Contrastive Unsupervised Representation Learning (CURL) [7].

Our Contributions and Highlights:

- Our work improves on CURL by exploring different data augmentation techniques on pixel inputs since the CURL paper only presents results from random shifts.
- We apply CURL encoder to deep Q network instead of Soft Actor Critic.
- We employ ResNet18 architecture for convolutional representation encoder.
- We incorporate limited knowledge about the underlying state features (e.g., cart's speed and angular velocity in CartPole domain) to speed up representation learning for RL.

CS230: Deep Learning, Fall 2021, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

2 Related Work

2.1 Double Deep Q Network learning

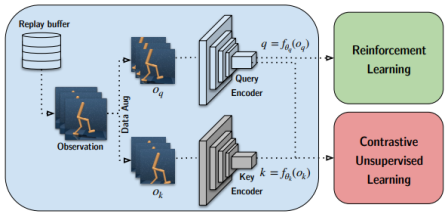
Double deep Q learning is an improved version of deep Q learning, which uses a neural network to estimate action values in current states. Then for selecting optimal actions, the argmax action is selected after predicting the Q value for each action. Deep Q learning uses an experience replay, or replay buffer, to disentangle temporal correlations between state-action-next state tuples [10]. Additionally, double Q learning uses two value functions with separate weight parameters, to mitigate over-optimism, so that the Q function used for selecting the argmax action is different from the Q netwrk used for estimating the value of the chosen action [15]. We used double DQN in our project.

2.2 Experimental Domain

Our work uses CartPole environment from the OpenAI Gym package [3]. The Gym environment constructs both the well-defined low-dimensional states, representing the cart's position, speed, pole angle, and pole angular velocity, and also gives access to the raw pixel data. With low-dimensional states, each dimension is discretized into 7 possible ranges, so the total size of the state space is $7^4 = 2401$. The actions are 0 for pushing the cart left and 1 for pushing right. Our goal is to train the agent to keep the pole standing for as long as 200 timesteps, with each surviving step rewarded by one.

2.3 Contrastive Representation Learning through Data Augmentation

We derive the our implementations' primary structure from CURL [7]. CURL assembles a frameset of 4 images to encode from each batch and apply different random shifts to create anchor and positive pairs. These framesets are inputted to the CNN encoder that outputs a feature embedding. All other framesets comprise negatives. Thus, the loss represents how much the encoder cam maintain anchor-positive similarity and anchor-negative dissimilarity, promoting distinct embeddings. The CURL approach is derived as a simplification of van den Oord et al.'s LSTM-based encoding method [11]. The CURL loss[1] is optimized with Soft Actor Critic (SAC) [12] with the architecture seen in Fig 1.



(a) CURL Architecture [7]

$$\mathcal{L}_q = \log \frac{\exp(q^T W k_+)}{\exp(q^T W k_+) + \sum_{i=0}^{K-1} \exp(q^T W k_i)}$$

(b) CURL Encoder Loss Function

Figure 1: CURL Specifications

Since CURL's loss is minimized alongside RL losses, CURL's encoder trains along with the DQN of Mnih et al. [10]. CURL's implementation atop other RL methods informed our applications of the technique across experimental formulations. As we experimented with generating anchor, positive, and negative observations with different augmentations besides random cropping, the Kornia package [2] was instrumental in producing transformed images.

3 Dataset & Features

We used the OpenAI Gym CartPole environment to train and evaluate the agents online on the final learned policies. CartPole frames are 400x600x3 (Fig 4). The agent applies forces to the cart to keep the pole upright for a maximum of 200 timesteps.

[1] In this case, q represents the anchor set of frames, k_+ represents the positive frames, W is the parameter matrix for a bilinear product used to compute logits, and K is the set of frame-sets in the batch. This loss function objective can be generalized as a differentiable dictionary lookup task.

Using the Kornia library [2], we explored eight different transforms in addition to random shift (as in original CURL) to construct positive and negative pairs for contrastive learning, including color jittering, color inversion, color equalization, fisheye warping, Gaussian blur and noise, perspective shifting, and horizontal flipping.

4 Methods

As in Fig 5, we implemented 5 different methods: A is the baseline DQN with a convolutional header for mapping pixel inputs to scalar Q values; B uses CURL to learn lower dimensional state representations from high-dimensional pixel observations; C uses ResNet18 to generate hidden features of size 512 from pixel data, then trains these 512-dim vectors with the CURL objective.

We explored two different scaffolding methods. Scaffolding is providing agents with limited access to state by augmenting embeddings with cart speed and pole angular velocity. In method D, instead of feeding in a single frame, we stacked two frames representing current/past observations to provide implicit temporal information. Method E appends the cart's and the pole's velocities to the embeddings. While CURL alone does not guarantee that the embeddings include all relevant features, we can force the inputs to the DQN to contain necessary state information by augmenting with the true state features. While we understand this assumption does not hold in domains where true states are completely unknown, we proposed them as alternatives to our previous methods which make no assumption about the environment but perform poorly compared to state-based Q learning.

In methods A-C and E, only one observation frame was used as the encoder input. Using the CURL architecture, we then tried a variety of augmentations as described in the Dataset & Features section, comparing results from different augmentation techniques. Some of the more disorienting augmentations like fisheye or elastic transform would only be applied with probability 0.5 to avoid modified data from diverging too far from the original pixel distributions.

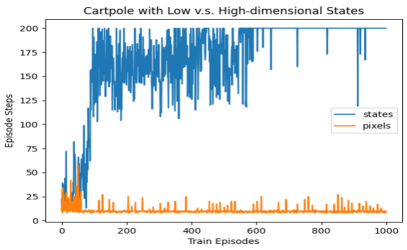
As shown in Fig 7, method C uses a pretrained ResNet18 with a fully connected last layer mapping 512 intermediate features to 1000 classes. Instead of using the last 1000 outputs, we used the hidden 512 features and unfroze the weights of the last 4 layers to continue trainining the ResNet model with CURL.

5 Experiments

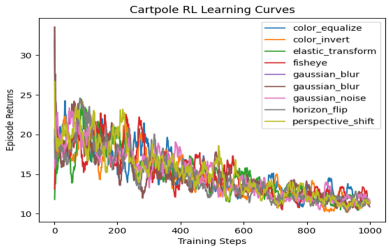
5.1 Neural Network Architecture

Our goal is to extract embeddings from inputs and map these embeddings to a scalar Q value per action. The RL agent consists of (i) convolutional layers (without or without pretrained ResNet18 weights) to process pixel data, and (ii) fully connected layers with 1 output representing the Q value estimate.

- Methods C and D using ResNet18 models. CartPole pixel inputs are mapped to 512 embeddings before feeding into fully-connected Q network.
- Method B without ResNet18. Used embedding size of 50, which were then mapped to action values through the fully connected Q network. We used smaller embedding size because the true states were only 4 so allowing the encoder to output higher-dimensional features might include too much extraneous information that hurts the model's performance.
- Method E with low-dimensional state scaffolding. Used embeddings of size 48 which were outputted by convolutional layers. Then the embeddings were appended by the cart's speed and the pole's angular velocity into a sized 50 embedding vector.
- Method A with convolutional layers plus DQN (baseline using raw pixel data): Used the following NN structure: input dimensions were [400, 600, 3], passed through [32 kernels of size 5 with stride 5], followed by [64 kernels of size 4 with stride 2], and [64 kernels of size 4 with stride 1]. The results were flattened into 13056-dimensional vectors, which were fed into fully connected layers of [512] hidden units. Then the final Q value estimate was scalar per action.
- Q network from the learnable embeddings to action value estimates: Used 2 fully connected layers of 50 hidden units followed by ReLU activation. The final Q value estimate was also scalar per action.

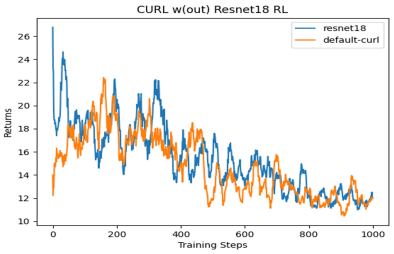


(a) Pixel versus State Features

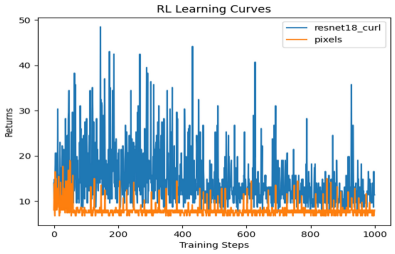


(b) Image Transforms

Figure 2



(a) ResNet18



(b) Ours versus Default DQN with Pixel Inputs

Figure 3

5.2 Reinforcement Learning Training Parameters

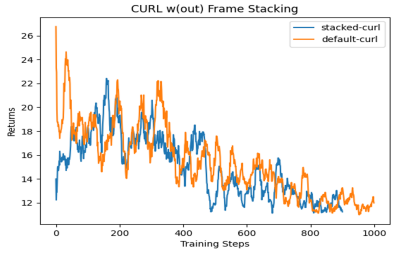
RL training parameters were applied consistently to all models. The adjustable parameters included: ϵ for determining how often to take greedy versus random actions, γ for discounting future rewards relative to the immediate reward at the next time step, batch size (i.e., how many samples to update the Q network parameters with), and replay buffer size (how many tuples to keep in memory before swapping out with newer ones). Based on our preliminary runs on state-based CartPole, we concluded that if the agent can keep the pole standing for over 195 steps, then learning is successful; and decided on the following RL parameters to use for pixel inputs as well:

- ϵ: 0.5, with a decaying factor of 0.999
- γ: 0.99
- Batch size: 32 (how many state-action-next state-reward tuples are used per Q update)
- Replay memory size: 200
- 1000 max training episodes and 200 steps per episode.
- Double DQN parameter: online Q network weights are copied to the target network every 100 episodes

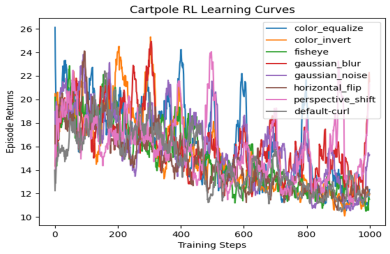
We tried increasing ϵ because we suspected if ϵ was decaying too quickly, then the Q function might learn a policy that repeats the sub-optimal actions without improving its action value estimates. However, we observed that a higher ϵ, 0.8 instead of 0.5, with 0.2 as the minimum value, did not make a difference.

6 Results and Discussion

We evaluated our models based on (1) episode rewards from the final learned policy, and (2) number of training steps required to achieve rewards above 195. Figures 2 and 3 show the number of training episodes on the x-axis and cumulative returns per episode on the y-axis. The higher the returns, the better the RL agent is performing. These returns were observed by running the models directly in the online CartPole environment while taking ϵ-greedy actions.



(a) CURL with 2 Stacked Frames



(b) Image Transforms with Added Velocity Feature

Training ResNet18 with CURL (shown in Fig 3(a)) did not outperform CURL with convolutional layers. While none of our approaches worked as successfully as tabular Q learning, our method of combining CURL with pretrained ResNet18 outputs outperformed the baseline pixel-based DQN (Fig 3(b)). Tabular Q learning achieved maximal rewards of 200 within 480 training episodes (Fig 2(a)) whereas DQN with pixel inputs performed abysmally, never reaching rewards above 50. Fig 2(b) shows the learning curves of training with 8 additional augmentation techniques (see Dataset & Features). However, we observed that none of our methods performed significantly better than the other ones, and all performed poorly compared to tabular Q learning.

Lastly, we tested our two scaffolding models: Method D using two stacked frames and Method E augmenting the learned embedding with the cart's velocities. However, these approaches also did not lead to any visible learning improvement.

7 Conclusion

In summary, we tried a variety of different approaches in a range of divergence from original CURL. The CURL architecture integrated with ResNet18 with some unlocked layers allowing for transfer learning performed marginally better than the rest of our models, and markedly so when compared to the pixel DQN baseline. While the ImageNet dataset the ResNet model was initially trained on may not correspond well to CartPole environment, we speculate that the earlier layers of the pre-trained network have the ability to distinguish importance aspects of the environment, like angles within the image and the position of important elements. This arrangement could potentially operate even better in real-world environments that more closely match the ImageNet dataset than OpenAI's classical Gym domains.

Henderson et al. [6] point us toward some of the inherent problems in reproducing "state-of-the-art" RL findings, such as environments' non-deterministic nature along with variance encountered in RL training. Even though we tried double DQN [13], which employs a replay buffer and double Q networks for less over-optimism, we were unable to mitigate RL training instability. Additionally, we suspect possible sources of suboptimal learning are: (1) cartpole pixel inputs having significant whitespaces that provide little to no state information and (2) CURL failing to encode the true underlying state features. As shown in Fig 8, our pixel inputs are 400x600, but the relevant part is only located in the center sized 50x100. Also, while CURL attempts to encode differences between distinct frames, it has no access to the underlying states and therefore must rely on minute differences. In CartPole, the cart's speed and angular velocity are required states for RL learning that CURL may not successfully learn to encode under the contrastive loss objective.

In the future, we would continue to tweak various hyperparameters of both CURL and underlying RL models to produce better performance. We can also expand our scope to different RL environments to evaluate how the encoder performs with varying RL tasks and input pixel data. We could also explore what effects more image transforms and combinations of different augmentation techniques has on RL agent's performance. Though our experiments did not approach state-trained performance, we still recognize the importance of improving pixel-based RL to tackle complex, ill-defined problems common in the real world.

8 Contributions

Alex Loia set up the remote computing infrastructure on AWS and adapted the code to work with GPUs accordingly. He also implemented and conducted the image augmentation and velocity state scaffolding experiments.
Alex Nam implemented the baseline pixel-based DQN, the pretrained ImageNet implementations, the frame stacking scaffolded approach, and processed the outputs of all experiments. She also found and adapted the state-of-the-art CURL implementation and contributed her existing knowledge of reinforcement learning techniques.
Together, both worked to setup the base CURL implementation from which modifications were made to experiment with different approaches. Both split work on the final video and researched this area to find applicable approaches and guidance.

Our code can found in this GitHub repository (we used open-sourced code from the following repositories for tabular Q learning and DQN [2, 4, 5, 7–9]). Our video can be found here.

9 Appendix

Here are additional figures that are referenced above.

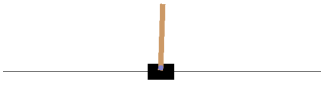


Figure 4: CartPole Environment

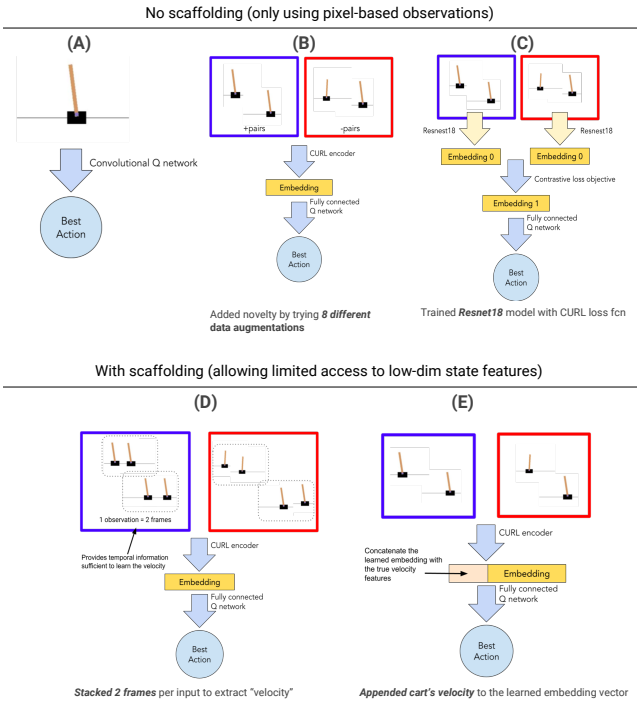


Figure 5: Methods A - E

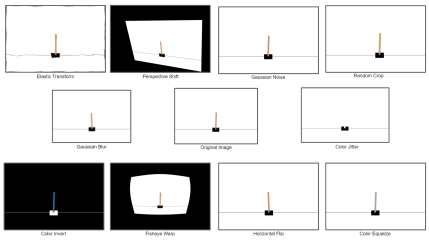
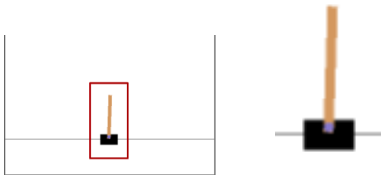


Figure 6: Applied Augmentations

Layer Name	Output Size	ResNet-18
conv1	$112 \times 112 \times 64$	7×7, 64, stride 2
conv2_x	$56 \times 56 \times 64$	3×3 max pool, stride 2
		$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$
conv3_x	$28 \times 28 \times 128$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$
conv4_x	$14 \times 14 \times 256$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$
conv5_x	$7 \times 7 \times 512$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$
average pool	$1 \times 1 \times 512$	7×7 average pool
fully connected	1000	512×1000 fully connections
softmax	1000	

Figure 7: ResNet 18 Architecture



(a) (b)

Figure 8: Original versus relevant pixel inputs

References

[1] Bellemare, M. G. et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". *Journal of Artificial Intelligence Research* 47 (2013): 253–279. Print.

[2] Bradski, G. et al. "Kornia: an Open Source Differentiable Computer Vision Library for PyTorch". *Winter Conference on Applications of Computer Vision*. N.p., 2020. Web.

[3] Brockman, Greg et al. "OpenAI Gym". 2016: n. pag. Print.

[4] Chen, Tao. "Taochenshh/DQN-Pytorch." GitHub, https://github.com/taochenshh/dqn-pytorch.

[5] Guts, Yuriy. "YuriyGuts/Cartpole-Q-Learning." GitHub, https://github.com/YuriyGuts/cartpole-q-learning.

[6] Henderson, Peter, et al. "Deep Reinforcement Learning That Matters." *ArXiv:1709.06560 [Cs, Stat]*, Jan. 2019. arXiv.org, http://arxiv.org/abs/1709.06560.

[7] Laskin, Michael, Aravind Srinivas, and Pieter Abbeel. "CURL: Contrastive Unsupervised Representations for Reinforcement Learning". *Proceedings of the 37th International Conference on Machine Learning*, Vienna, Austria, PMLR 119 (2020): n. pag. Print.

[8] Marcel, Sébastien, and Yann Rodriguez. "Torchvision the Machine-Vision Package of Torch." *Proceedings of the 18th ACM International Conference on Multimedia*, Oct. 2010, pp. 1485–1488., doi:10.1145/1873951.1874254.

[9] Paszke, Adam, et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019.

[10] Mnih, Volodymyr, et al. "Human-Level Control through Deep Reinforcement Learning." *Nature*, vol. 518, no. 7540, Feb. 2015, p. 529.

[11] van den Oord, A., Li, Y., and Vinyals, O. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

[12] Yarats, Denis, et al. "Improving Sample Efficiency in Model-Free Reinforcement Learning from Images." *ArXiv:1910.01741 [Cs, Stat]*, July 2020. arXiv.org, http://arxiv.org/abs/1910.01741.

[13] Sutton, Richard, and Barto, Andrew. *Reinforcement Learning: An Introduction*, The MIT Press, 2017.

[14] Mnih, Volodymyr, et al. “Playing Atari with Deep Reinforcement Learning.” *ArXiv:1312.5602 [Cs]*, Dec. 2013. arXiv.org, http://arxiv.org/abs/1312.5602.

[15] Hado van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-learning.” 2015, CoRR. *https://dblp.org/rec/journals/corr/HasseltGS15.bib*.