An Implementation of Contrastive Predictive Coding

Sterling Suggs

August 2020

1 Introduction

We implement Contrastive Predictive Coding (CPC) as described by van den Oord et al. [3]. This is a self-supervised representation learning algorithm that learns features that will maximize the ability to predict future states. We use this algorithm to learn embeddings of pixel observations in the Atari video game Space Invaders.

Our motivation for this work is the characteristic inefficiency of reinforcement learning. Most RL models trained in visual domains must learn to interpret state data at the same time as learning a policy for behavior. If, however, an untrained RL agent had a pre-trained Encoder that could already produce useful feature embeddings, this would greatly decrease the burden of learning on the agent, which could then focus only on learning the policy. We hypothesize that this would increase sample efficiency of an RL agent, but in this work we only implement and test one example of such an Encoder.

In the next section we briefly review Contrastive Predictive Coding. We then outline our implementation, discuss the training procedure, and present results. Before concluding, we describe a casual experiment we performed with a reinforcement learning agent.

Note: This report was prepared some time after the work was done. Many existing graphs were sloppy, missing axis labels or legends, or with the text poorly sized. Our data and saved models were no longer extant, therefore we did not take time to recreate these graphs. We use them as they are, and apologize for the untidiness.

2 Contrastive Predictive Coding

For details of this algorithm, please refer to the original paper [3]. The following overview will serve to refresh memories which have had prior exposure.

The intuition behind CPC is to learn high-level features, like phonemes in audio or objects in a picture, that maximize the mutual information between the present context c and future data x, defined as $I(x;c) = \sum_{x,c} p(x,c) \log \frac{p(x|c)}{p(x)}$. We will do that by essentially training a model to predict future states given the current context c.

An Encoder $\mathcal{E}$ creates feature representations $z_t = \mathcal{E}(x_t)$, where x_t is the data at time-step t; and an Autoregressor $\mathcal{A}$ summarizes all past $z_{\leq t}$ into a context vector, $c_t = \mathcal{A}(z_{\leq t})$. Rather than training a generative model to predict states directly, which is computationally intensive, we model a density ratio proportional to the mutual information $I(x;c)$. For this, the paper suggests a log-bilinear Score function, $\mathcal{S}_k(z_{t+k}, c_t) = \exp(z_{t+k}^{\mathsf{T}} W_k c_t)$, with a different W_k for each step k in the future we want to predict.

Finally, the loss is defined by comparing the score for state $t+k$ to the scores of a set of false candidates. Let $X = \{x_i\}_{i=1}^N$ be a set containing one true sample from $p(x_{t+k}|c_t)$ and $N-1$ negative samples. Then we want to minimize

$$\mathcal{L}_N = -\mathbb{E}_X \left[\log \frac{\mathcal{S}_k(x_{t+k}, c_t)}{\sum_{x_j \in X} \mathcal{S}_k(x_j, c_t)} \right]. \tag{1}$$

Doing so will result in the Score function $\mathcal{S}$ estimating a density proportional to the mutual information $I(x,c)$, causing the Encoder to learn high-level "slow" features, which are, it is hoped, the features of interest.

3 Implementation

Our Encoder is a three-layer convolutional network (CNN), with hyper-parameters as specified in Table 1. Input is pre-processed as for DQN [2], with four stacked frames grayscaled and downsampled to 84×84. The Autoregressor is a gated recurrent unit (GRU), with a hidden size equal to the embedding size of the Encoder. The Scorer is a bilinear layer as defined earlier, using a different

weight tensor W_k for each step k into the future that we are predicting.

The three components are trained jointly; once trained, the Encoder and Autoregressor can each be used in subsequent tasks independent of the other components.

Hyper-parameter	Value
Channels	32, 64, 64
Filter size	$8 \times 8, 4 \times 4, 3 \times 3$
Stride	4, 2, 1
Embedding sizes	16, 32, 63, 128, 256

Table 1: Hyper-parameters for the Encoder.

4 Training and results

We trained our model on pixel data collected by a randomly acting agent playing the Atari video game Space Invaders. Data from the collection is sampled randomly, and so to create each context vector c_t we stoked the Autoregressor with $M = 20$ prior frames, $\{x_i\}_{i=t-21}^{t-1}$. We predicted up to $K = 30$ steps in the future, and used $N = 15$ false samples for each step. The learning rate was 10^{-4} for all model components.

We were curious how embedding size would affect learning, and so we trained on five different embedding sizes, $\{z_t \in \mathbb{R}^{2^n}\}_{n=4}^{8}$. Figure 1 shows train and test loss for each embedding size.

5 RL experiment

We performed only a cursory experiment using a pre-trained CPC Encoder as the embedding layer for an RL agent. The RL agent used was Model-Free Episodic Control [1], or MFEC. Our baseline MFEC model does not learn representations at all, but uses a random projection from the state space down to the desired embedding dimension. Using the CPC Encoder instead slightly hurt performance (see Figure 2, black vs. blue lines).

One issue is that as the agent learns, it sees states that are increasingly different from those the Encoder trained on. Ideally, the Encoder will be able to generalize to an extent, but if later game states differ significantly from what the Encoder was trained on, we expect to see a reduction in

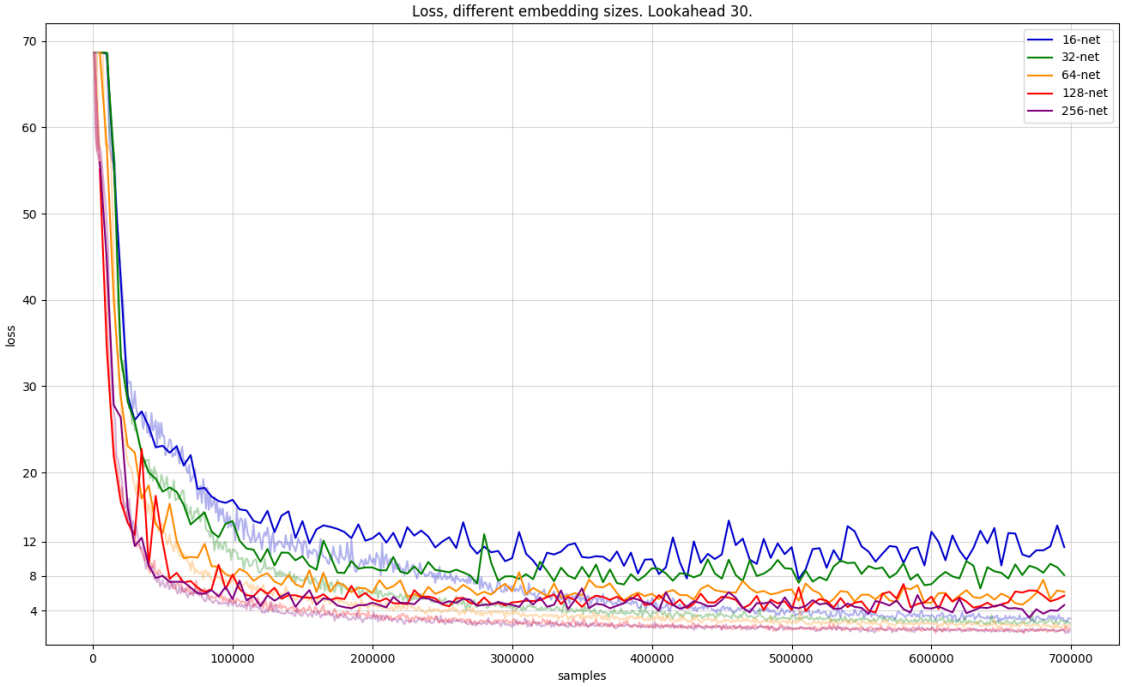

Figure 1: Loss by embedding size. For each size, the light curve depicts training loss while the dark curve is test loss.

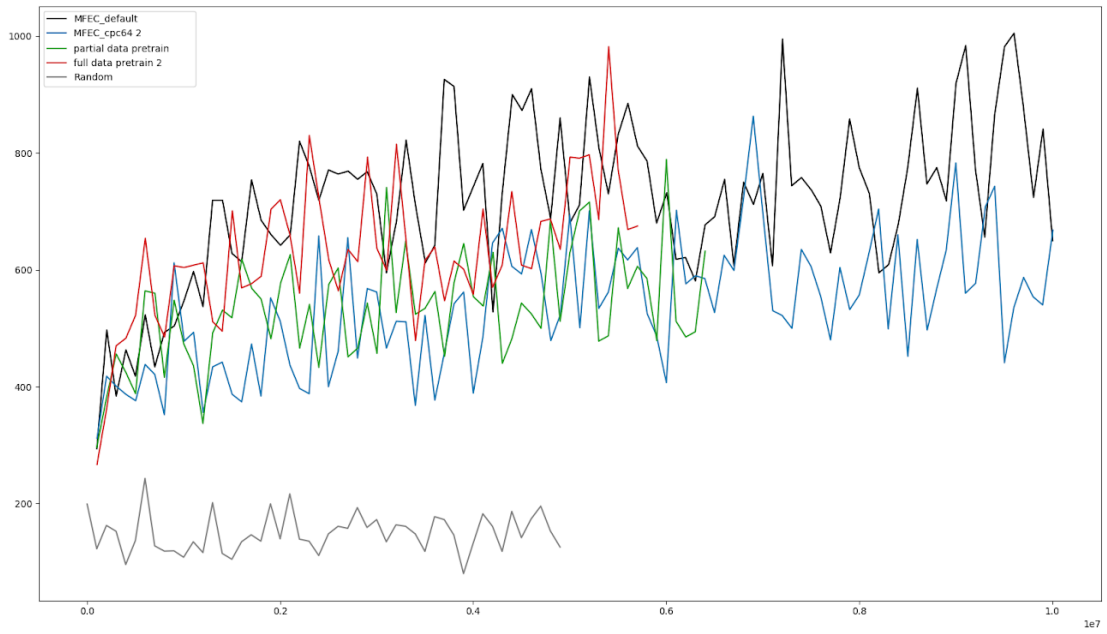

Figure 2: MFEC agents with various Encoders. **Black**: random projection. **Blue**: Encoder data collected by random agent. **Green**: Encoder data collected by partially trained agent. **Red**: Encoder data collected by fully trained agent. **Gray**: A randomly acting agent.

agent performance compared to the baseline. Therefore, we also trained a CPC Encoder on data collected by a fully trained Rainbow DQN agent, which would visit a greater area of the state space. Finally, we also trained an Encoder on data collected by a partially trained Rainbow DQN agent, to see which dataset provided the most representative sample of states for CPC training. MFEC agents using these Encoders are represented in Figure 2 by the red and green lines, respectively. They appear to do slightly better than the agent with the Encoder trained on randomly collected data (blue), but not as well as the baseline (black). However, each line represents only a single test run, and a more thorough experiment should repeat each test several more times.

To confirm our intuition that a learning agent visits states that are different from what the Encoder was trained on, we plotted CPC loss during pre-training and then continued to evaluate periodically on new data after the agent starts training. Figure 3, top, shows this for an Encoder trained on data from a partially trained agent; bottom, on data from a fully trained agent. In both cases, we see loss on the Encoder jump rapidly once the agent starts learning and visiting new states.

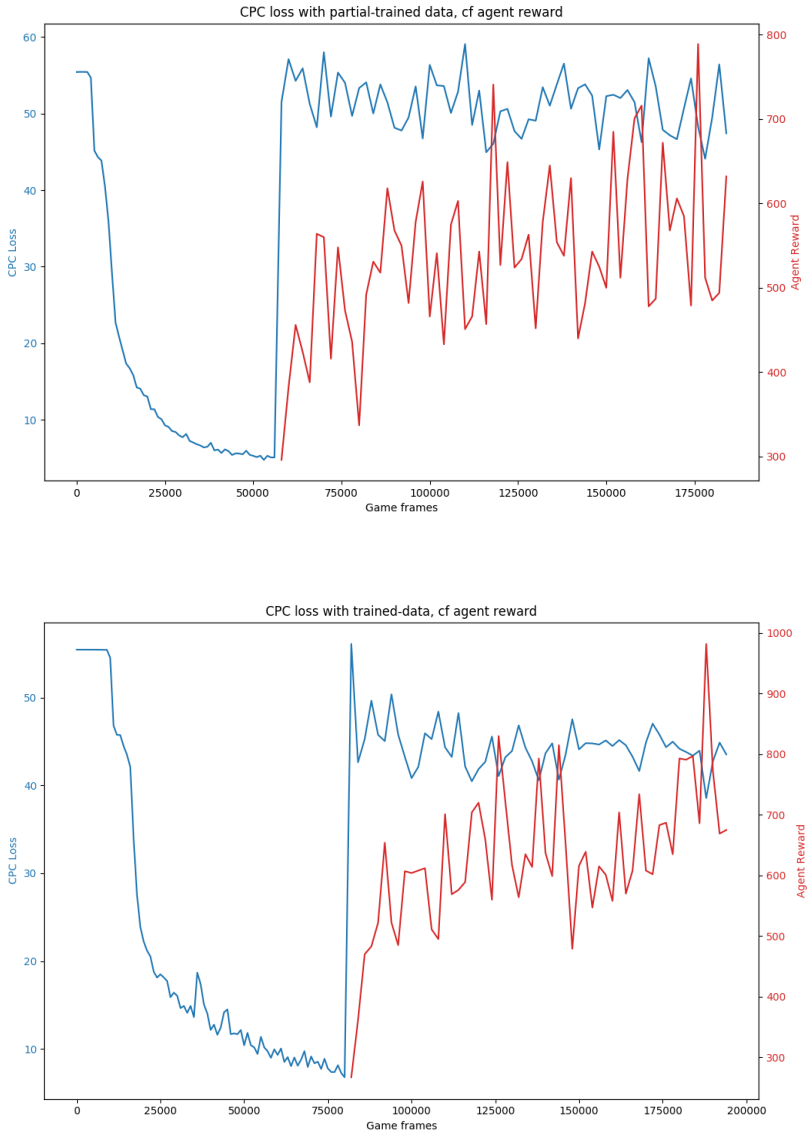

Figure 3: Encoder (CPC) loss vs. agent reward. CPC loss increases after the agent starts learning, due to the agent visiting unfamiliar states. **Top**: CPC training data from a *partially trained* agent. **Bottom**: CPC training data from a *fully trained* agent. **Blue**: CPC loss during training, and during evaluation after the agent starts learning. Left y-axis. **Red**: Agent reward, right y-axis.

One possible route would be to periodically retrain the Encoder as the agent sees new data. However, this would defeat the purpose of pre-training an Encoder. It is interesting to note that in spite of the CPC loss returning to un-trained levels, the agent is still able to learn. Perhaps the features learned by the Encoder were not particularly useful after all? Or is it possible that the Encoder continues to extract useful features on new data, just not ones which maximize mutual information with future states?

6 Conclusion

We implemented Contrastive Predictive Coding and tested it on a visual domain. Our comparison of embedding sizes shows that this unsupervised approach can learn effective features even at relatively low-dimension embeddings. Pre-training an RL agent's feature encoder could be a viable way to improve data efficiency, but it will require some dedicated research to figure out how.

References

[1] Charles Blundell, Benigno Uria, Alexander Pritzel, Yazhe Li, Avraham Ruderman, Joel Z Leibo, Jack Rae, Daan Wierstra, and Demis Hassabis. Model-free episodic control, 2016.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. ISSN 00280836. URL `http://dx.doi.org/10.1038/nature14236`.

[3] Aäron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *CoRR*, abs/1807.03748, 2018. URL `http://arxiv.org/abs/1807.03748`.