Self-supervised learning from snaps

Kenneth Blomqvist
kenneth.blomqvist@aalto.fi

Aleksi Hämäläinen
aleksi.hamalainen@aalto.fi

Abstract

This paper attempts to replicate results obtained in the paper 'Time-Contrastive networks: self-supervised learning from video'. (Sermanet et al., 2017) We perform experiments to show that videos taken from a single viewpoint can be used to learn a distance preserving embedding of a scene onto a vector space. We attempt to teach a simulated robotic arm to imitate itself performing different trajectories using reinforcement learning and the learned embedding. The results show that an embedding can be learned in a self-supervised manner. However, learning to imitate proved itself more challenging.

I. Introduction

Learning from others is one of the primary ways in which animals, humans included, learn to perform new tasks. Augmenting robots with such learning capabilities could open up countless new applications for robots. It would be even more beneficial, if such learning could occur from simple videos of humans demonstrating a task. Humans have already mastered countless tasks and collecting videos of humans performing these tasks is very simple to do.

In this project we attempt to replicate results from the paper titled 'Time-contrastive networks: self-supervised learning from video' Sermanet et al. (2017). In the paper the authors propose a way to teach a robot manipulation tasks using videos of a human performing a task without an explicit supervision signal. Differences in the frames across time are used as a learning signal.

A distance preserving embedding is learned using a triplet loss function. This learned function is used to create a reward function for reinforcement learning of the task. The triplets used for training can be constructed using either video from multiple views or by using video from a single viewpoint.

We attempt to replicate a simplified version of the problem. Namely, we try to teach a simulated robot to imitate itself performing different moves in a video taken from a single viewpoint.

Section II presents the main ideas behind the methods used in the experiments. Section III presents the experiments we performed. Section IV presents the results we obtained. Section V provides some subjective conclusions we draw from our experience building this project and our results.

II. Methods

Here we present the main methods utilized by our project.

A. Learning an embedding

The paper Sermanet et al. (2017) presents a way to learn a distance preserving embedding from video frames onto an n-sphere. The goal is to map frames that are close to each other in the video to vectors that are close to each other on the n-sphere as measured by the L_2-norm.

The embedding function is denoted $f(x)$ where x is a frame from a video. We use three types of frames: an anchor frame x_a, a positive frame x_p and a negative frame x_n. The anchor frame is closer to the positive frame than the negative frame.

Essentially, we want the constraint

$$\|f(x_a) - f(x_p)\|_2^2 + \delta < \|f(x_a) - f(x_n)\|_2^2$$

to hold. δ is a constant margin parameter.

f is a convolutional neural network. The network is optimized by optimizing a triplet loss function.

$$L(x_a, x_p, x_n) = \delta + \|f(x_a) - f(x_p)\|_2^2 - \|f(x_a) - f(x_n)\|_2^2$$

The triplets are either from a single or multiple viewpoints. When using multiple viewpoints, the anchor frame is a random frame sampled from the video. The positive frame is a frame from the exact same time-step but another viewpoint than the anchor frame. The negative frame is a frame sampled from the same viewpoint as the anchor frame outside a margin range around the anchor frame. This enables the network to learn a viewpoint invariant representation of the scene. (Sermanet et al., 2017)

In the single viewpoint case, all frames are from the same viewpoint. The anchor frame is again a random frame of the video. The positive frame is within a small margin range of the anchor frame. The negative frame is from outside a larger margin range of the anchor frame. (Sermanet et al., 2017)

B. Learning using reinforcement learning

We learn to imitate using reinfocement learning. The reward function is defined using the embedding shown in the previous section.

We use a huber-style loss:

$$R(\boldsymbol{v_t}, \boldsymbol{w_t}) = -\alpha \left\| \boldsymbol{w_t} - \boldsymbol{v_t} \right\|_2^2 - \beta \sqrt{\gamma + \left\| \boldsymbol{w_t} - \boldsymbol{v_t} \right\|_2^2}$$

Here α and β are scaling parameters. γ is a small constant to make the equation well defined for almost zero distances. $\boldsymbol{w_t}$ is an embedding of an image of the robot itself and $\boldsymbol{v_t}$ is an embedding the example video frame at timestep t.

We learn a policy that optimizes the reward function using the proximal policy optimization algorithm (PPO) (Schulman et al., 2017). PPO is a robust, simple, model free policy gradient algorithm. It optimizes a clipped loss function:

$$L(\theta) = \hat{E}_t[\min r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

$r_t(\theta)$ is the ratio between the probability of the action taken under the current policy and the policy before the previous update. $\hat{A}_t$ is the advantage defined:

$$\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + ... + \gamma^{T-t+1} r_{T-1} + \gamma^T V(s_T)$$

The policy and value functions are neural networks. This surrogate reward function can be updated using multiple epochs of minibatch updates, making it well suited for neural networks. (Schulman et al., 2017)

III. Experiments

We are interested in two questions:

- Is it possible to learn an embedding of video frames in a self-supervised manner using only videos from a single viewpoint?
- Is the learned embedding robust and well-behaved enough to power a reward function to use as part of a reinforcement learning problem?

A. Embedding frames of a robotic arm

We created 200 videos of a robotic arm performing trajectories in a simulated environment. The arm is initialized to random joint positions and it moves to goal joint positions which are also randomly sampled. We use the Bullet 3 physics simulator and a robot modelled after the Kuka iiwa robotic arm.

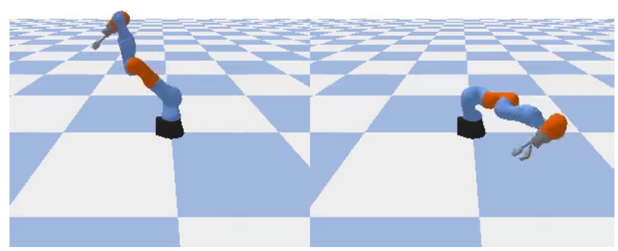

Figure 1: *Two example frames from a recorded trajectory. The left frame is the goal position and the right frame is the start position.*

We split the 200 videos such that 190 are used for training and 10 for validation.

We use a convolutional neural network derived from the Inception architecture as presented in (Szegedy et al., 2016). We use the 8 first layers of the network up until the layer labeled 'Mixed_5d'. We add two batch normalized convolutional layers, one spatial softmax transformation followed by two fully connected layers. This is very similar as the network used in Sermanet et al. (2017).

The network was implemented using the PyTorch deep learning framework. The layers taken from the inception architecture are initialized to values pretrained on the ImageNet dataset. The added layers are randomly initialized using the default initialization scheme of the PyTorch package.

In our experiments, the output of the network is a 32-dimentional embedding constrained to have an L_2 norm of 10. The scaling factor of 10 was motivated by results presented in Ranjan et al. (2017). We use a margin value of 2.0. The positive frame was sampled from within 10 frames of the anchor frame. The negative frame was sampled from outside a range starting from 30 frames before the anchor frame and ending 30 frames after the anchor frame. The size of a video frame is 299 pixels in width and height.

We use the triplet loss presented in the previous section. At each epoch, we create a dataset of triplets sampled from 5 videos with 200 samples per video. We then run stochastic gradient descent with momentum against the triplet loss over this dataset 5 times after which a new triplet dataset is sampled and the process is repeated. The use a batch size of 64 triplets.

We use a learning rate schedule such that we start with a learning rate of 0.1. Each 500 epochs we decrease the learning rate to 1/10th of the previous rate until we reach 0.0001 inclusive. Table 1 summarises the final parameters we used after tuning. The training of the network was performed on a single NVidia Quadro P5000 GPU.

Parameter	Value
minibatch size	64
learning rate	0.1-0.0001
δ	2.0
positive frame margin	10
negative frame margin	30
optimizer	SGD with momentum
momentum	0.9
embedding dimensions	32

Table 1: *The parameters we used for training our embedding function.*

B. Learning to imitate

We used the learned embedding to teach the same simulated robot to imitate itself in a video performing different trajectories. We use the proximal policy optimization algorithm to optimize the loss function.

The observation at each time step is a concatenation of the robot joint states, joint velocities, the TCN embedding of an image of itself and the TCN embedding of the video frame at that timestep.

The reward is calculated using the huber loss presented in section II using the embedding of an image of the robot and the embedding of an example video frame. Actions are torques applied to the 12 joints of the robotic arm.

#TODO policy and value network specs.

We trained the imitation agent by running 16 processes in parallel. We evaluate the policy network to get an action for each process at each time step. We use a rollout storage to sample updates from. An update is done on the policy and value networks every 1000 time steps. All the parameters we used are summarized in table 2.

Parameter	Value
γ	0.99
τ	0.95
clip ϵ	0.2
epochs each update	4
update interval	1000 time steps

Table 2: *The parameters we used for proximal policy optimization.*

IV. Results

We monitored the accuracy of our embeddings over the course of training our CNN. The validation set is a dataset of 1000 triplets such that 100 triplets are created from each of the 10 validation videos. After the end of training, 736/1000 samples fulfilled the triplet constraint. 930/1000 fulfilled the contrained without the added margin. I.e. $\|x_a - x_p\| < \|x_a - x_n\|$.

For comparison after 10 epochs of training the values were 467/1000 with margin and 894/1000 without the margin.

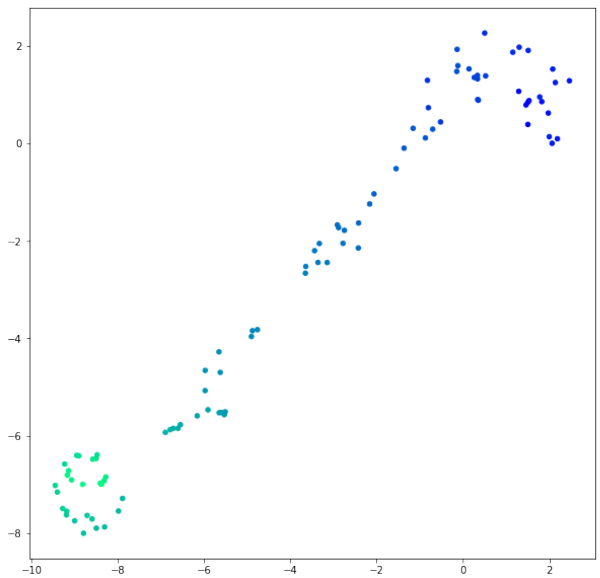

Figure 2: *A t-SNE plot of a validation trajectory. Perplexity 30, learning rate 200 and 1000 iterations. The color goes from blue to cyan as a function of the frame indices.*

The t-SNE plot of a validation trajectory suggests that the network has learned a meaningful and well-behaved embedding of the video frames. Embeddings of frames that are close to each other in the video are close to each other in the dimensionality reduced regime.

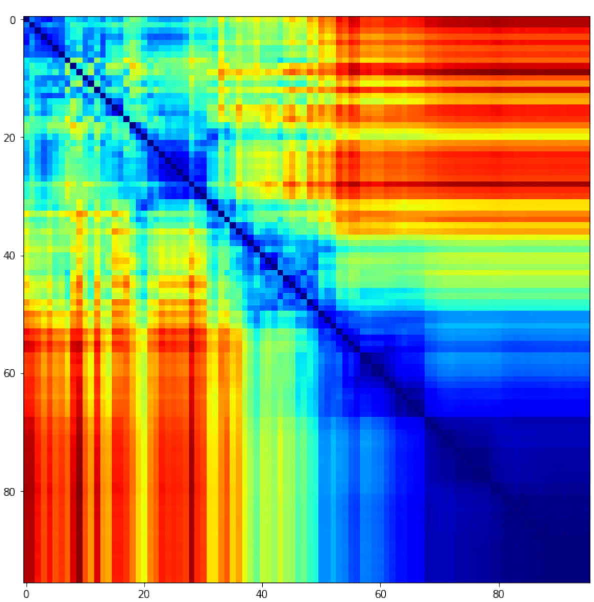

Figure 3: *Visualization of the L_2 distance matrix of the same validation video as in figure 2. Blue means close and red is far away.*

Each row and column in the distance matrix corresponds to a frame of the video. The value is the euclidean distance between the embedding of the frame. I.e. the diagonal is 0 and the matrix is symmetric. This visualization seems to confirm that subsequent frames are close to each other in the embedding space.

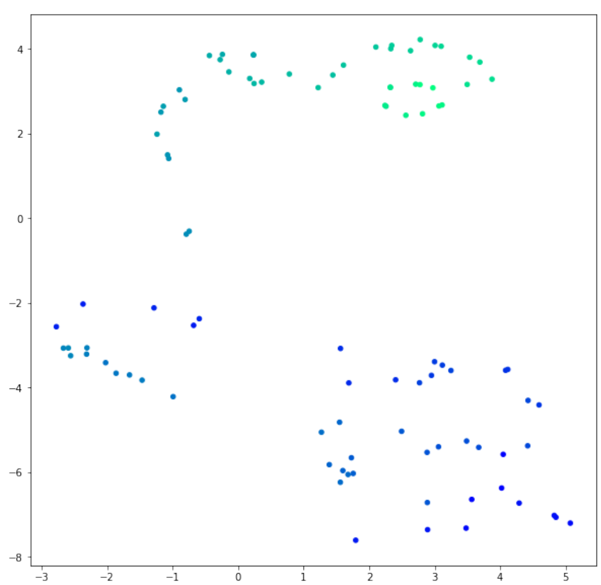

Figure 4: *For comparison, a visualization of a trajectory after 50 epochs of training. Visualized using the same t-SNE parameters as figure 2.*

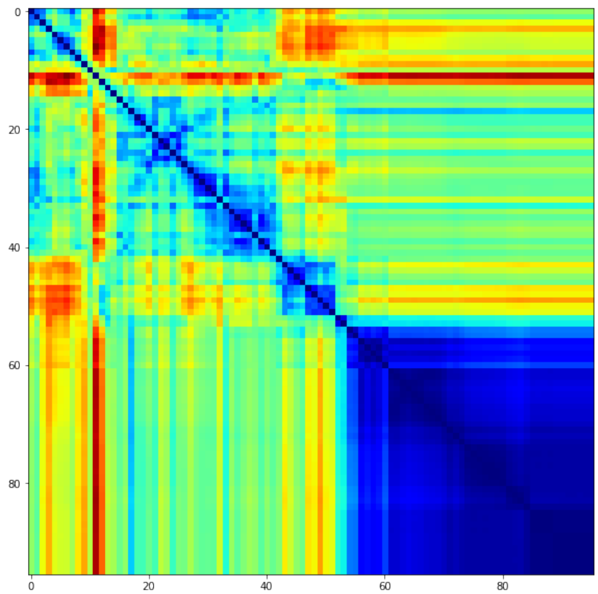

Figure 5: *Distance matrix for the same network as in figure 4.*

Comparing figure 2 to figure 4 and figure 3 to figure 5 we can see that early on in training the network has not yet learned a meaningful embedding of the entire joint space of the robot.

V. Discussion

As we have seen, it is possible to learn a meaningful embedding of video frames taken from a single viewpoint onto a vector space. Computer vision has progressed a lot in recent years and methods such as convolutional neural networks have become very usable with tools such PyTorch. Implementing the neural network and tuning the parameters did not prove to be very difficult.

Learning to imitate proved itself to be more difficult. We performed many runs of our experiments, but ultimately failed to learn any useful behaviour. This might be due to areas in the domain of our embedding function which are not properly mapped onto the codomain. It might be that these areas were not properly represented in our training data.

Proximal policy optimization is known to not be very sample efficient. One hypothesis might be that we simply did not train our policy and value functions for long enough or that we did not manage to find suitable parameters for the algorithm. We can not entirely rule out that there are no bugs in our code. However, we did try our implementation on more simple environments from the Openai gym (Brockman et al., 2016). We also tried the PPO algorithm in our simulation on a different environment where the objective is to move the robotic arm into a randomly sampled goal position. The state in this environent was the goal position, the robot joint states and velocities. In these environments the algorithm appeared to learn adequately.

It would be interesting to run our imitation experiments using more computational resources than we had available. Another interesting experiment would be to try and use videos taken from multiple viewpoints.

References

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

Rajeev Ranjan, Carlos D Castillo, and Rama Chellappa. L2-constrained softmax loss for discriminative face verification. *arXiv preprint arXiv:1703.09507*, 2017.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Pierre Sermanet, Corey Lynch, Jasmine Hsu, and Sergey Levine. Time-contrastive networks: Self-supervised learning from multi-view observation. *arXiv preprint arXiv:1704.06888*, 2017.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. pages 2818–2826, 2016.