Exploring Autoencoders and Contrastive Learning in application to Deep Reinforcement Learning

Ildar Abdrakhmanov [1] **Pavel Kopanev** [1] **Timur Akhtyamov** [1] **Samir Mohamed** [1] **Anna Nikolaeva** [1] **Ilia Ivanov** [1] **Yerzhaisang Taskali** [1] **Anastasia Chernysheva** [1]

Abstract

Deep Reinforcement Learning is a combination of deep neural networks and policy-based reinforcement learning algorithms that facilitates learning the optimal set of actions agent takes in a virtual environment to attain the maximum possible reward. Training reinforcement learning agent using high-dimensional images is very time inefficient and often requires a huge amount of actions to learn the optimal policy. In our work, we aimed to enhance this process by learning from hidden representations obtained by training multiple autoencoders and contrastive learning architecture from raw images mined from the virtual environment. We compare various approaches to learn efficient data encodings by linking the maximum reward and the number of actions taken by the agent.

1. Introduction

Deep neural networks in reinforcement learning(RL) recently became the foundation of state-of-the-art approaches in different RL application areas such as robotics, computer vision, video games, and many others. In our work, we focus on autonomous driving, which is one of the most exciting computer vision applications nowadays. Convolutional neural networks(CNN) are the primary tool in modern autonomous driving applications, and state-of-the-art RL algorithms utilize CNNs to recognize the agent's state when the input is visual. In our case, the visual context is represented by a car driving on a marked road in a virtual environment. The environment where the agent takes actions can be very complex, and the most complex one can imagine is the real-world car driving on public roads. The one crucial issue with deep neural networks is that extracting characteristic features from the environment requires training of wide and deep convolutional networks with a lot of neurons and layers. Thus, there are two main limitations of using deep RL in real-world applications when the learning process takes place on embedded systems. First, training CNNs is very time-consuming and requires powerful GPUs which consumes a lot of electrical power. Second, efficient learning requires a huge amount of data to be collected while the agent takes actions in the environment(Prakash et al., 2019). To overcome these limitations, we propose a rather different deep RL approach. We move step of feature extraction and learning CNNs out of the embedded system. Firstly, the images from the virtual environment were gathered, then the different feature learning techniques were implemented and learned on GPUs using obtained data. These learned representations are used to learn optimal RL policy with much smaller networks, which requires much fewer number of iterations to converge. Finally, the networks with learned weights are used on embedded systems to perform data compression and real-time reinforcement learning.

In our project, we mined 15K pictures from an open-source self-driving Donkey Car platform where the environment represents a visual field of the mini-car driving on a track in a warehouse. The track and its surroundings are shown in Figure 1. The 10K images are sampled in the autonomous

Figure 1. Track environment

driving setup, and 5K are sampled by manually driving the car. It is used to model real-life driving, which can potentially increase the variability of visual representations.

In the feature learning or representation learning part, we

[1]Skolkovo Institute of Science and Technology, Moscow, Russia. Correspondence to: Ildar Abdrakhmanov <ildar.abdrakhmanov@skoltech.ru>.

Final Projects of the Deep Learning 2020 Course, Skoltech, Moscow, Russian Federation, 2020.

experimented with six different autoencoders in unsupervised setting and contrastive learning framework in a self-supervised setting. In the case of autoencoders(AE) we observed the reconstructions produced by the decoder and compared the different types of AE visually and using the reconstruction loss. In the case of self-supervised contrastive learning approach, we could not observe the reconstructions. Therefore the approach is validated in a different way, which will be discussed in section 4. Furthermore, we compare all these methods with the number of actions the agent has to perform to achieve a certain level of reward and learn an effective policy. Additionally, we compare our approach to the end-to-end method when RL agent is trained on raw images.

Real-time reinforcement learning is implemented using Soft Actor Critic(SAC) algorithm, which is discussed in subsection 3.4. The input images are forwarded through encoder part of the AEs and the separate convolutional architecture of contrastive learning network. All the encoders are initialized with the pre-trained weights. The obtained hidden vectors are transferred to the RL agent which is a shallow fully-connected neural network. The RL agent's goal is to drive as fast as possible while staying within the track and maximizing the cumulative reward. The obtained rewards and the number of episodes needed to achieve them are compared for different encoding methods. We believe that the optimal encoding architecture can help to learn the effective policy by the minimum amount of episodes needed to learn an agent, thus minimizing the amount of power consumption and maximizing the rate of learning an agent.

2. Related Work

Deep reinforcement learning(DRL) is becoming the widely popular scientific area among researches from all over the world. That is because a combination of deep learning architectures with RL algorithms is capable of scaling to previously unsolvable problems(Arulkumaran et al., 2017). Indeed, convolutional neural networks in recent years enhanced the computer vision area of the reinforcement learning domain, which embraces such applications as autonomous driving and robotics. Although the potential applications are not limited to those cases. The breakthrough in DRL which attracted the interest to the subject was the development of deep learning model which were capable of learning control policies from high-dimensional input images. This model is the first successful application in learning the game policy, which outperformed humans in playing Atari arcades (Mnih et al., 2013). This paper also introduced the novel idea in deep Q-learning algorithm by combining experience replay memory idea and stochastic minibatch updates to facilitate learning of the deep network. Despite the high-interest growth, the real-world applications are incomplete nowadays (Schulman et al., 2017). In our project, we make a step towards the RL applications on embedded systems in autonomous driving.

Reinforcement learning algorithms learned from images as inputs are typically end-to-end models with deep convolutional networks used directly on visual inputs. It means that they do rely on powerful hardware and a huge amount of input data to learn the effective policy. Thus, such models are not suited to be integrated into onboard systems. Here the idea of autoencoders and self-supervised approaches comes in. There are different solutions to embed the RL model and make it faster, robust, and energy-efficient. For instance, in this article, (Li et al., 2017) authors address the problem of energy saving by introducing the cloud computing resources allocation and residential smart grid scheduling. In this way, they achieve a significant total energy cost reduction and energy saving of the embedded computing platform for online RL applications.

Another limitation of online DRL is the amount of data needed for training. Typically, the larger the dataset, the more efficient the whole approach is. However, the amount of needed data varies for different RL algorithms. In our work, we implemented Soft Actor Critic algorithm (Haarnoja et al., 2018) since it can learn an effective policy with a smaller number of data points. Also, SAC is an off-policy algorithm that is essential for embedded systems where the ability to learn from a random set of interactions is crucial (Prakash et al., 2019). Furthermore, the sample efficiency of this algorithm is improved by the ability to learn from the previous set of actions, which decreases the number of dangerous actions the agent undertakes.

The idea of shifting the training stage of deep convolutional architectures in a supervised setting offline on powerful GPUs using different RL simulations is not new. Extracting the hand-crafted features from high dimensional environments are being widely used for different RL applications. Although these approaches are very time-consuming and cost-inefficient, they aim at decreasing the depth and the number of parameters of the network which can be trained online using extracted representations. Furthermore, supervised approaches fail to generalize to complex environments and require human expertise and time. Unsupervised and self-supervised approaches are much more robust to the diverse nature of the real-world environment and do not require labeled data to learn (Chen et al., 2020).

The state-of-the-art research work on using autoencoders for RL (Prakash et al., 2019) used variational autoencoder(VAE) in three different configurations to learn from different simulation environment for autonomous driving. The authors explored the effect of changing the number of filters in VAE on the performance of the RL algorithm and energy required to encode a video frame and the latency associated

with this encoding. Then, they performed inference on the encoder at each time step and trained RL agent on embedded Nvidia Jetson TX2 platform. In our work, we rather focus on different approaches for data encoding than on different configurations of one kind of autoencoder. However, we also compare the performance of VAE with the state-of-the-art approach.

3. Models and Algorithms

In this section we firstly describe the simulation environment and datasets used in our work. Then, we explore the autoencoders used to compress the data and contrastive learning framework. Finally, we discuss the RL agent training setup. The experiments we conducted in our project is in section 4. The general pipeline is shown in Figure 2.

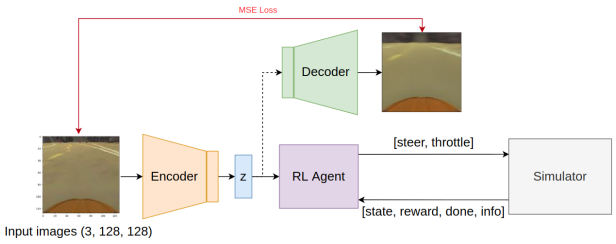

Figure 2. Project architecture.

3.1. Simulation environment and Data

3.1.1. Simulation environment

We utilized a simulation environment for collecting data and further train autoencoders as well as RL agent. Gym toolkit developed by OpenAI is commonly used nowadays for designing and comparing reinforcement learning algorithms. In our case it was important to pick an environment with observations - images, also observations should have been visually complex enough otherwise dimensionality reduction with autoencoders could have become redundant.

We examined three different gym environments which you can see on the Figure 3. We started from gym core environments and tried the CarRacing simulator and its improved more advanced version (`https://github.com/NotAnyMike/gym`) first. But we were not satisfied with the visual complexity of the images (RL agent observations) and began to look at gym wrapped simulators which were more complex.

We tried Duckietown environment but as a pure python environment it was not easy to make modifications in the scene and images from it still looked primitive.

After that, we tried the Donkeycar simulator which was used in the reference paper (Prakash et al., 2019). The simulator is based on the Unity game scene and could be

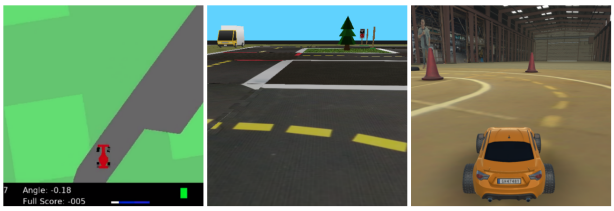

Figure 3. Examined simulation environments: CarRacing simulator (left picture), Duckietown simulator (middle picture), and Self Driving Sandbox donkey simulator (right picture).

modified using the Unity editor. Its architecture you can see on the Figure 4. The simulator is wrapped by the gym environment and they are communicating through JSON messages via TCP sockets. The Unity engine is able to provide realistic-looking virtual camera images suitable for our project. Therefore we choose the Donkeycar simulator. Also, the Unity engine provides server builds for the game scenes which could be utilized for RL networks training on powerful servers. But for image rendering from the virtual camera server should have a GPU.

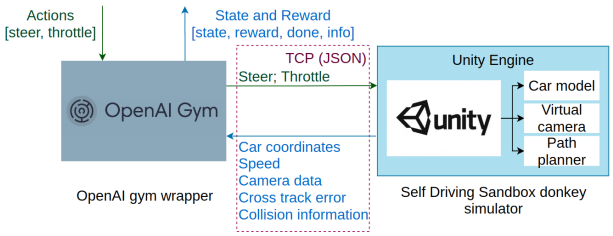

Figure 4. The architecture of the simulation environment.

After choosing the simulator we were examining scenes in it. A simple road environment was used in the reference paper (Prakash et al., 2019). But we decided to utilize the Warehouse environment. It was more visually complex and our approach with autoencoders as dimensionality reduction tool for RL network should have shown its full power.

RL agent steer and throttle actions were controlling the car. The Unity environment was providing feedback with car coordinates, speed, virtual camera data, cross-track error (the measure of deviation from the defined path), and objects collision status. Than gym wrapper was converting this data into observations: state (camera data), reward, done, and info (speed and other information) values. The top view of the track and predefined paths is shown on the Figure 5.

We were changing camera position and angle as well as image resolution. For RL network it was important to have human control under the car (not just RL agent control like it was implemented in the original Donkeycar simulator), therefore we added teleoperation mode.

Figure 5. The top view of the track where blue lines refer to possible car paths.

3.1.2. DATA

For training autoencoders, 14 235 pictures were taken, 30% of which were given for the validation part. Each snapshot was a 128x128 RGB matrix. Samples from the dataset are shown on the Figure 8.

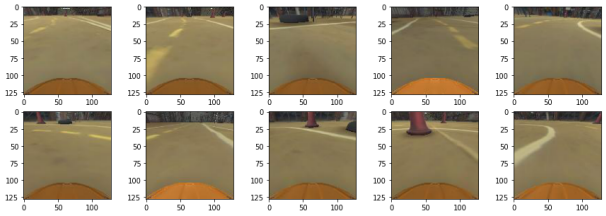

Figure 6. Data samples from the simulation environment.

All data was taken from the simulation environment via the Unity scene virtual camera. It was possible to apply postprocessing operations (fog, ambient occlusion, motion blur, antialiasing, and others) as well but we settled on the default image data. Data samples were taken during self-driving mode which is provided in the Donkey simulator (in this mode car follows the optimal path) and during manual driving with attention to track objects like cones or people's legs. The 10K images were sampled in the autonomous driving setup, and 5K were sampled during manual driving.

In the beginning, we tried to collect data via observations provided by Donleycar simulator gym wrapper but timestep between images was substantial and we switched to direct data extraction from the Unity simulation scene which was more efficient.

3.2. Autoencoders

3.2.1. ADVERSARIAL AE

Adversarial Autoencoder is a probabilistic autoencoder that uses the recently proposed generative adversarial networks (GAN) to perform variational inference by matching the aggregated posterior of the hidden code vector of the autoencoder with an arbitrary prior distribution. Matching the aggregated posterior to the prior ensures that generating from any part of prior space results in meaningful samples. As a result, the decoder of the adversarial autoencoder learns a deep generative model that maps the imposed prior to the data distribution.

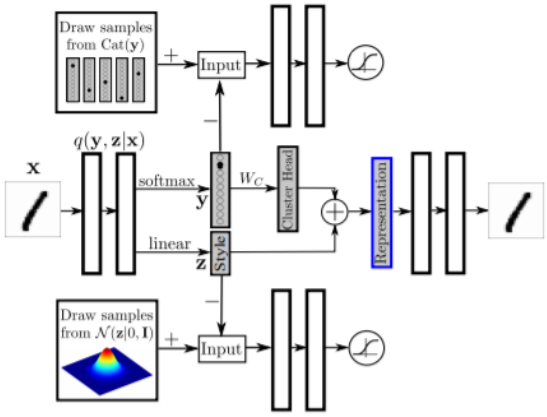

Figure 7. AAE architecture.

There are two separate adversarial networks in AAE that impose Categorical and Gaussian distribution on the latent representation. The final n dimensional representation is constructed by first mapping the one-hot label representation to an n dimensional cluster head representation and then adding the result to an n dimensional style representation. The cluster heads are learned by SGD with an additional cost function that penalizes the Euclidean distance between of every two of them.

3.2.2. BETA-AE

We use beta-variational auto-encoder (Loic Matthey, 2018) where that progressively increases the information capacity of the latent code during training to get concentrated information in each latent space as shown in figure 8.

So it acts like normal variational auto-encoder and the trick part happens in the loss function where we have to use this equation to get the loss value:

$$\mathrm{L}(\theta, \phi; \mathbf{x}, \mathbf{z}, C) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}|\mathbf{z})\right] - \gamma \left| D_{KL}\left(q_\phi(\mathbf{z}|\mathbf{x}) \| p(\mathbf{z})\right) - C \right| \quad (1)$$

So the model learns an axis-aligned disentangled representation of the generative factors of visual data compared to the standard VAE objective.

3.2.3. VANILLA VAE

Vanilla VAE (Kingma & Welling, 2013) uses a stochastic algorithm of variational inference and training, which scales

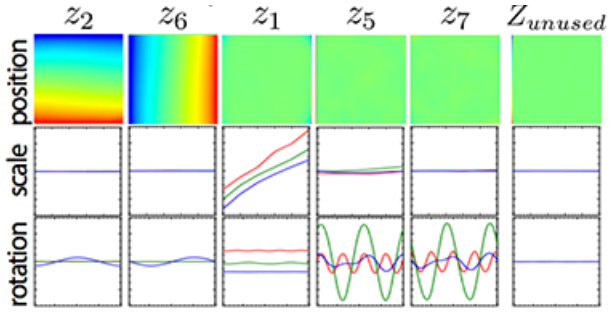

Figure 8. Latent space that output from beta-variational autoencoder where the information is concentrated in the first dimensions

to large data sets and, under certain conditions of soft differentiability, even works in the insoluble case. Its advantages are two points:

- using reparametrization of the variational lower bound, an estimate of the lower bound is given, which can be optimized using standard stochastic gradient methods
- data set with continuous hidden variables for each data point, the conclusion can be made especially effective by fitting an approximate output model (also called a recognition model) to an intractable posterior using the proposed lower bound estimator.

The article constructs an estimate of the lower boundary of the probabilistic limit of a complete data set based on minibatches:

$$\mathcal{L}(\Theta, \phi; X) \simeq \widetilde{\mathcal{L}}^M(\Theta, \phi; X^M) = \frac{N}{M} \sum_{i=1}^{M} \widetilde{\mathcal{L}}(\Theta, \phi; X^{(i)})$$

where the minibatch $X^M = x^{(i)}{}_{i=1}^{M}$ is a randomly drawn sample of M datapoints from the full dataset X with N datapoints. Derivatives $\nabla_{\Theta,\phi}\mathcal{L}(\Theta; X^M)$ can be taken, and the resulting gradients can be used in conjunction with stochastic optimization methods such as SGD or Adagrad. See Algorithm 1 for a basic approach to compute the stochastic gradients.

The article shows an estimator of the variational lower bound, Stochastic Gradient VB (SGVB), for efficient approximate output with continuous hidden variables. The proposed evaluator can be directly differentiated and optimized using standard stochastic gradient methods. For the case of i.i.d. data sets and continuous hidden variables for each data point, an efficient algorithm for efficient output and training, Auto-Encoding VB (AEVB), which studies an approximate output model using the SGVB evaluator, is presented.

Algorithm 1 Minibatch version of the Auto-Encoding VB (AEVB) algorithm. Either of the two SGVB estimators can be used.

$\Theta, \phi \leftarrow$ Initialize parameters
repeat
  $X^M \leftarrow$ Random minibatch of M datapoints (drawn from full dataset)
  $\epsilon \leftarrow$ Random samples from noise distribution $p(\epsilon)$
  $g \leftarrow \nabla_{\Theta,\phi}\widetilde{\mathcal{L}}^M(\Theta, \phi; X^M, \epsilon)$ Gradients of minibatch estimator
  $\Theta, \phi \leftarrow$ Update parameters using gradients g (e.g. SGD or Adagrad)
until convergence of parameters (Θ, ϕ)
return Θ, ϕ

3.2.4. Variance Constrained AE

Adding noise to latent codes makes the latent space better, because in this way we add a continuity condition to our optimization scheme (i.e. close latent codes correspond to close decoder outputs). But if we just add noise, the encoder will try to spread latent codes as wide as possible to don't be disturbed by the noise with a certain variance. This is not what we want. Therefore, the authors of (Braithwaite et al., 2020) add noise with a fixed variance. To ensure that latent codes work and the encoder does not suppress them by increasing the distance between latent codes, dispersion regularization is added. Since they do not optimize the code distribution form, they pass the desired distribution form to the normalization flows after training. This allows us to sample from the model, see Figure 9.

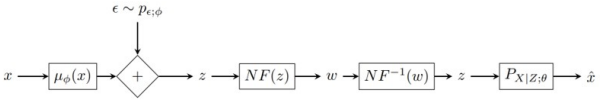

Figure 9. VCAE Architecture

What is normalization flows? It is a series of invertible transformations modeled by neural networks. They also require the ease Jacobin counting to calculate the density conversion. It can be applied to some distribution and expect that the output will be the right one, and then optimize it using the maximum likelihood method Also it is has to train normalization flows separately from the main part. To do that, it has to be divided in two stage: training encoder/decoder as the first part, and NF as the second.

Algorithm 2 describes an implementation of the VCAE, where is assumed that $P_{X|Z;\theta}$ is a symmetric Gaussian permitting the use of the mean squared error (MSE) at the decoder output. As normalization flows were used Masked Autoregressive Flows.

Algorithm 2 VCAE algorithm with the assumption that $P_{X|Z;\theta}$ is a symmetric Gaussian. Let L_m and L_n be the number of minibatches used to train the encoder/decoder and normalising flows respectively

Input: data x_i
Output: Optimized parameters θ^*, ϕ^*
set noise distribution $P_{\epsilon;\phi}$
set weight λ
set parameter υ
initialise parameters θ, ϕ
for each minibatch $l \in L_m$ **do**
  $\bar{x_l} \leftarrow$ current minibatch
  $\bar{z} \leftarrow \mu_\phi(\bar{x_l}) + \bar{\epsilon}, \bar{\epsilon} \sim P_{\epsilon;\phi}$
  $\hat{x} \leftarrow \mu_\theta(\bar{z})$
  $L \leftarrow MSE(\hat{x}, \bar{x_l}) + \lambda|var(\bar{z}) - \upsilon|$
  $(\theta, \phi) \leftarrow$ +Adam update of θ, ϕ to minimise L
end for

initialise $f_1, ..., f_m$
$q_\psi(\omega) \leftarrow p(\omega) \prod_{n=1}^{m} |det \frac{\partial f_i(\omega_{i-1})}{\partial \omega_{i-1}}|^{-1}$
for each minibatch $l \in L_n$ **do**
  $\bar{x_l} \leftarrow$ current minibatch
  $\bar{z} \leftarrow \mu_\phi(\bar{x_l}) + \bar{\epsilon}, \bar{\epsilon} \sim P_{\epsilon;\phi}$
  $L_{nf} \leftarrow -\frac{1}{|\bar{x_l}|} \sum_{j=1}^{|\bar{x_l}|} \log q_\psi(\bar{z_j})$
  $\psi \leftarrow$ +Adam update of ψ to minimize L_{nf}
end for

3.2.5. JIGSAW-VAE

It is important to notice that the polygon includes attributes, the reconstruction of which plays a key role in the outcome of a race. Such objects are cones, table legs, and people. One of the main problems of traditional auto-encoders is that when learning on a large and meanwhile unbalanced dataset, sensitivity to smaller classes disappears, which in our case can lead to a race loss.

Theoretically, this problem has a solution, that has been recently published in the article (Taghanaki et al., 2020). The idea is to use the well-known Jigsaw puzzle approach on the top of an autoencoder. In particular, we are going to perform sampling permutations form a uniform distribution of each input image as it is shown in the Figure 10.

The results of the article show that such a modification proved to be effective in experiments on generative and other downstream tasks such as clustering. It is achieved due to the exploitation of a mixture prior and a stochastic permutation layer. The mixture prior helps to smooth the conflict between the likelihood and KL terms while the permutation layer enforces the VAE to learn spatial relationships between the tiles (object parts). Thus, we consider this approach to be key when constructing the auto-encoders.

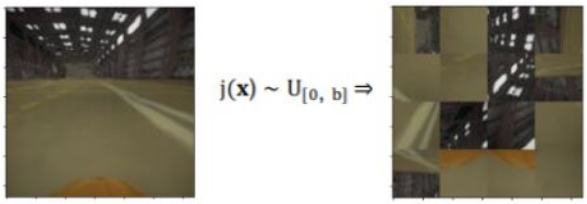

Figure 10. Sampling permutations form a uniform distribution for window size of (4x4), where b is the total number of permutations.

3.2.6. WASSERSTEIN AE

Another autoencoder we explored in our project is Wasserstein Auto-Encoder(WAE) (Tolstikhin et al., 2017). The WAE introduces a new type of regularization based on a penalized form of Wasserstein distance between the target distribution and the model distribution. In the original work, authors showed that using this type of regularization allows WAE to generate better quality samples than variational autoencoders (VAE). Also, this approach encourages global training distribution to match the prior more closely. The main idea of WAE is shown in Figure 11. Using the same notation as in the article, we denote: χ - input space; P_X - input distribution; P_Z - prior distribution; $Q_{WAE}(Z|X)$ - distribution induced by encoder; $P_G(X|Z)$ - latent variable model, specified by P_Z. As well as VAE, WAE minimizes the reconstruction loss and regularizing term. However, unlike VAE, WAE minimizes discrepancy between the continuous mixture $Q_Z = \int Q_{WAE}(Z|X)dP_X$ and target distribution P_Z, whereas VAE matches P_Z for all input samples drawn from P_X. The regularizing term is a Wasserstein distance between P_Z and $Q_{WAE}(Z|X)$.

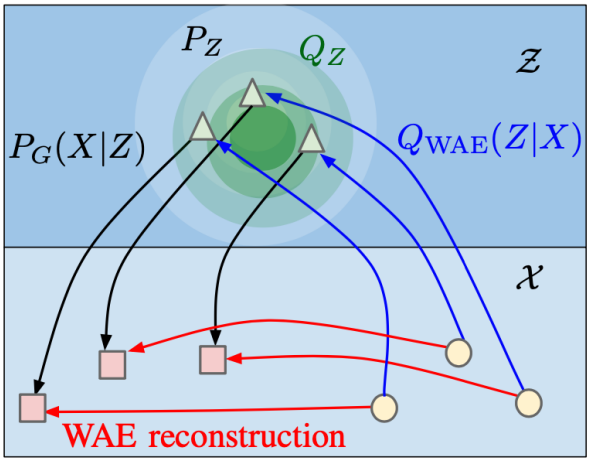

Figure 11. WAE idea

Algorithm 3 WAE-MMD

Input: $\lambda = 100$; IMQ: $k = \frac{c}{\epsilon + c + \|x_1 - x_2\|^2}$, where c is calculated using the dimension of latent space and latent variable as $c = 2 z_{dim} z_{var}$. z is sampled from prior(Gaussian) distribution, $\epsilon = 1e-7$; Initialize parameters of encoder Q_ϕ and decoder G_θ networks.
while (ϕ, θ) not converged **do**
Sample $\{x_1, ...x_n\}$ from the training set
Sample $\{z_1, ...z_n\}$ from P_Z
Sample $\hat{z}_i$ from $Q_\phi(Z|x_i)$ for $i = 1, ..., n$
Update weights of the networks by descending:

$$\frac{1}{n}\sum_{i=1}^{n} c(x_i, G_\theta(\hat{z}_i)) + \frac{\lambda}{n(n-1)}\Big(\sum_{l \neq j} k(z_l, z_j) + \\ + \sum_{l \neq j} k(\hat{z}_l, \hat{z}_j)\Big) - \frac{2\lambda}{n^2}\sum_{l,j} k(z_l, \hat{z}_j)$$

end while

The corresponding objective: $D_{WAE}(P_X, P_G) = \inf_{Q(Z|X) \in \Theta} \mathbb{E}_{P_X} \mathbb{E}_{Q(Z|X)}[c(X, G(Z))] + \lambda \cdot D_Z(Q_Z, P_Z)$, where D_Z - divergence between Q_Z and P_Z, Θ - set of encoders, $\lambda > 0$ - hyperparameter, $c(X, G(Z))$ - loss function. The authors of original paper proposes two different approaches to penalize objective with different D_Z divergence terms, leading to a different objectives: WAE-GAN and WAE-MMD. In our project we used WAE-MMD variant of WAE because it doesn't require adversarial training. MMD stands for maximum mean discrepancy: $MMD_k(P_Z, Q_Z) = \| \int_Z k(z, \cdot) dP_Z(z) - \int_Z k(z, \cdot) dQ_Z(z) \|_{H_k}$, where $k : Z \times Z \to \mathbb{R}$ - positive definite kernel(PDS), H_k - reproducing kernel Hilbert space of real-valued function mapping Z to $\mathbb{R}$. In our work we chose the inverse multiquadratic PDS kernel: $k(x, y) = (c^2 + ||x - y||_2^2)^\beta$ for some $\beta < 0$ and $c > 0$. It produces better mean estimates than the Gaussian kernel.

Finally, the WAE-MMD algorithm we used is summarized in Alg. 3.

The decoder and encoder networks architectures are the same as other autoencoders have. It is described in details in section 4.

3.3. Contrastive Learning

Autoencoders is not the only way to learn useful representations. In our project, we explored the completely different idea of contrastive learning in a self-supervised setting. This idea is a very new state-of-the-art framework that matches the performance of a supervised Resnet-50 when a linear classifier trained on features learned from ImageNet pictures (Chen et al., 2020). Contrastive learning of visual representations (SimCLR) model learns to maximize the agreement between two different latent vectors of the same image corresponding to two different transformations of the same image sampled from the dataset. The main idea of (SimCLR) is summarized in Figure 12.

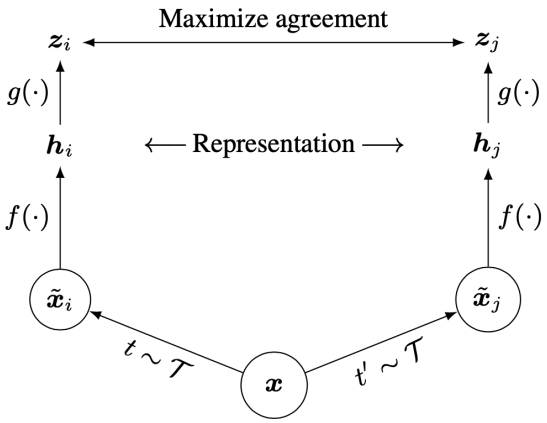

Figure 12. SimCLR framework

The following points covers the basic ideas behind the SimCLR framework and describes the Figure 12.

- *Image Transformation*. Given the image, two different transforms are applied to obtain two correlated views $\hat{x_j}$ and $\hat{x_i}$, which are considered the positive par. The example of positive par is shown in Figure 13. In our project, we experimented with two different configurations of transformation pipeline, the details of which are presented in section 4.
- *Encoder network*. The transformed images are then passed to the convolutional network to extract the feature vectors. We experimented with two different CNNs for efficient data encoding.
- *Projection network*. Latent representations are forwarded through a fully-connected shallow network with ReLU nonlinearities. It maps the feature vectors to space where the contrastive loss is applied. This network is reported to be beneficial in terms of the quality of low-dimensional manifold learned by the model.
- *Contrastive Loss*. The loss maximizes the agreement between hidden projections of the same image. Given a set $\{\hat{x_k}\}$ the contrastive loss aims to identify $\hat{x_j}$ in $\{\hat{x_k}\}_{j \neq i}$ for a given $\hat{x_i}$.

To make the encoder network learn to compress images in a meaningfull manner, the contrastive loss is used. Given

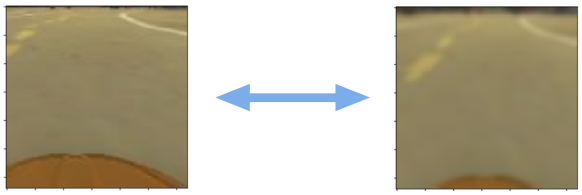

Figure 13. Different transformations of the same image

Algorithm 4 SimCLR algorithm

Input: batch size $N = 512$; $\tau = 0.5$; Initialize parameters of the encoder CNN.
for sampled batch $\{x_k\}_{k=1}^{N}$ **do**
  for all $k \in \{1, ..., N\}$ **do**
    Draw two augmentations t, t'
    $\hat{x_{2k-1}} = t(x_k)$
    $h_{2k-1} = f(x_{2k-1})$
    $z_{2k-1} = g(h_{2k-1})$
    $\hat{x_{2k}} = t'(x_k)$
    $h_{2k} = f(x_{2k})$
    $z_{2k} = g(h_{2k})$
  end for
  for all $i \in \{1, ..., 2N\}$ and $j \in \{1, ..., 2N\}$ **do**
    $s_{i,j} = \langle z_i, z_j \rangle / (||z_i|| ||z_j||)$
  end for
  $l(i,j) = -\log \frac{\exp(s_{i,j}/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(s_{i,k}/\tau)}$
  Minimize $Loss = \frac{1}{2N} \sum_{k=1}^{N} [l(2k-1, 2k) + l(2k, 2k-1)]$ by updating the networks f and g
end for
Return encoder network f weights

a batch of N images two different transformations yield the batch twice as big as the initial one. Among these $2N$ images two images are treated as positive pair and remaining $2N - 1$ samples as negative examples. The contrastive loss function between positive examples is then defined as:

$$l_{i,j} = -\log \exp \frac{\frac{\langle z_i, z_j \rangle}{||z_i|| ||z_j||} / \tau}{\sum_{k=1}^{2N} \mathbb{1}_{[j \neq j]} \exp(\frac{\langle z_i, z_k \rangle}{||z_i|| ||z_k||}) / \tau}$$

, where τ is a temperature parameter. The cumulative value of loss is calculated across all positive pairs. The final algorithm with some hyperparameters we used in our implementation is summarized in Alg. 4.

In the referenced article, the authors introduced a lot of useful insights to learn the high-quality representation of the input space. However, those are adapted for a very diverse dataset of natural images ImageNet, where each picture contains some unique object. The critical point is that these real-world objects are not repeated across the data. On the contrary, our dataset's images are very similar to each other, and it is tough to find the images which would look completely different. This fact is a severe constraint on learning high-quality representations. The experiments that were conducted and the exact architectures and parameters used in the contrastive learning framework are discussed in section 4.

3.4. Reinforcement Learning

As it was mentioned in introduction, as Reinforcement Learning Algorithm we have chosen the Soft Actor-Critic (SAC) algorithm, proposed in (Haarnoja et al., 2018), and this algorithm was used in (Prakash et al., 2019). SAC is related to Policy Gradient based Actor-Critic algorithms. This is an off-policy algorithm which means that it is able to use previous experience for learning process. Also here we use continuous action and observation space.

The goal of the RL algorithm is to find an optimal policy - the policy that will maximize total cumulative reward. In Markov Decision Process (MDP) theory, which founds most of the RL algorithms, one way to find such policy is the *policy iteration* process which assumes two-step iterations that include policy evaluation and policy improvements steps. The policy evaluation can be done with *Bellman operator* T_π of current policy π, applied to the Value function $V(s)$:

$$[T_\pi V](s) = \mathbb{E}_{a \sim \pi(\cdot|s)} \left[r(s,a) + \gamma \mathbb{E}_{s'|s,a} \left[V(s') \right] \right],$$

where a corresponds to action, produced by policy, $r(s,a)$ corresponds to obtained reward and γ is a discount factor. The improvement step thus can also be defined with Bellman operator:

$$[T_{\pi_{new}} V] = \max_\pi [T_\pi V]$$

In this case there are theoretical guarantees of convergence to the true value function of MDP in case if we do not assume any function approximators.

In work (Haarnoja et al., 2018) it was proposed an extension of the policy iteration, applied to the action-value function $Q(s,a)$. This is done by augmenting Bellman operator with additional term that is corresponds to the policy entropy that in fact acts as a regularization term:

$$T_\pi Q(s_t, a_t) = r(s_t, a_t) + \\ + \gamma \mathbb{E}_{a' \sim \pi} \left[Q(s_{t+1}, a') - \log \pi(a'|s) \right]$$

In this case there are guarantees of convergence to optimal *soft Q-function*. Authors of (Haarnoja et al., 2018) propose to update policy towards the softmax distribution of current Q-function, which means minimizing KL divergence between two distributions:

$$\pi_{new} = \text{argmin}_\pi J(\pi)$$
$$J(\pi) = \mathbb{E} \left[KL \left(\pi(\cdot|s) || \frac{Q(s,\cdot)}{\sum_a Q(s,a)} \right) \right]$$

In case of finite state and action space with no function

approximation (this case is called *tabular*) the policy will be being improved in monotonic way.

Since we are working with continuous observation and action spaces, we can't find exact solutions for MDP, and thus function approximators have to be used. In Deep RL, artificial neural networks are used as function approximators. In case of SAC, we approximate the policy with neural network that plays as actor, and Q-function with neural network that plays as critic. The actor network receives current state (observation) as an input and produces mean and variance of corresponding action, and the actual action is then sampled from this distribution and passed to the environment; the critic network receives a pair of state and action and produces value of approximated Q-function $Q(s, a)$. In case when dimensionality reduction is used for environment, the state is a vector, and the networks architecture are simple multilayer perceptrons. Otherwise, the state is a jull image, and thus networks have additional convolutional part, consisting of convolutional layers and batch normalization layers. While training, each set of corresponding state, action, reward and new state after applying action are put into the *replay buffer* D which then used for training networks.

The Q-function approximator parameters (weights) θ can be optimized w.r.t. MSE loss between the predicted action value and target action value:

$$J_Q(\theta) = \mathbb{E}_{s_t,a_t\sim D}\left[\frac{1}{2}\left(Q_\theta(s_t, a_t) - q_t\right)\right]$$

$$q_t = \mathbb{E}_{a\sim\pi_\omega(\cdot|s_{t+1})}\left[r(s_t, a_t) + Q_\theta\left(s_{t+1}, a\right) - \alpha \log \pi_\omega(a'|s_{t+1})\right.$$

where ω corresponds to the parameters (weights) of the policy, and α is an entropy parameter called *entropy temperature*.

For policy update step, authors in (Haarnoja et al., 2018) propose the "reparametrization trick" where the idea is following. In training step, action is represented as hyperbolic tangent of the value u sampled from distribution on the output of policy network. In this case, $\log \pi(a|s)$ can be computed as:

$$\log \pi(a|S) = \log CDF(u|s) - \sum_{i=1}^{D} \log(1 - a_i^2)$$

This trick helps to reduce variance of the policy estimator. Now, policy can be optimized w.r.t. simplified form of KL divergence:

$$J_\pi(\omega) = \mathbb{E}_{s_t\sim D}\left[\mathbb{E}_{a'\sim\pi_\omega(\cdot|s_t)}\left[\alpha \log \pi_\omega(a_t|s_t) - Q_\theta(s_t, a_t)\right]\right]$$

Also by the authors of original paper also was proposed automatic adjusting the entropy temperature with optimization w.r.t. to objective:

$$J_\alpha = \mathbb{E}\left[-\alpha \log \pi_t(a_t|s_t, \alpha) - \alpha\overline{H}\right],$$

where $\overline{H}$ is the desired minimum entropy, usually set to zero.

To avoid overestimation of Q-functions, two networks are used in the model. In policy update step, the minimal of outputs of those networks is used. To maintain target Q-function, we maintain special target Q-networks. Those networks are defined as copies of initial Q-networks with Polyak averaging.

Also some words about training process. Our framework is to do training step after the end of each episode, and also there is additional training step after some iterations of episode (usually, this step is done for long episodes). Before starting actual training phase, we have additional phase of "manual exploration". To improve exploration and reduce training time, we collect experience in replay buffer during some episodes in following way: agent makes random actions, and then if action is close to fail the episode, we switch to the manual control word and correct the position of a car, and those actions and corresponding states are also added to replay buffer. Thus, when starting training, we already have an experience of possible good and bad actions. In some sense, this approach is close to behaviour cloning methods.

4. Experiments and Results

4.1. Autoencoders

In our project we implemented the universal pipeline for training any autoencoder. We used batch size of 1024 images and trained all models for 3000 epochs. Most of the models required additional hyperparameters tuning to achieve good performance. These parameters are described in detail for each autoencoder separately in Section 3.2. Some of the autoencoders additionally required adversarial training. All models used different types of losses, but the reconstruction component remained the same for all models. We used mean squared error(MSE) as a reconstruction loss. Overall, without hyperparameters tuning, only autoencoder models were trained for more than 50 hours. The train loss curves and validation loss curves are shown in Figure 14 and Figure 15 respectively. The final loss values are summarized in table 2. The number of model parameters is shown in table 1. As encoder and decoder parts of each autoencoder, we used the same architecture and number of filters. Specifically, the encoder consists of four convolutional layers, followed by three linear layers. We used rectified linear unit ReLU function as nonlinearity in the network. The architecture of the decoder is symmetrical to one of the encoders. We also tried to increase the number of channels of the WAE autoencoder aiming at lower model loss and better quality, the number of parameters of the network was increased by the factor of 9. However, this network required more than twenty-four hours of training

Table 1. Number of models' parameters

MODELS	NUM. PARAMETERS
AUTOENCODERS	1,9 M
SIMCLR18	11,7 M
SIMCLR50	25 M

on Tesla V100 GPU and we decided that it is impractical. All the code is fully-reproducible and available in the github repository [1].

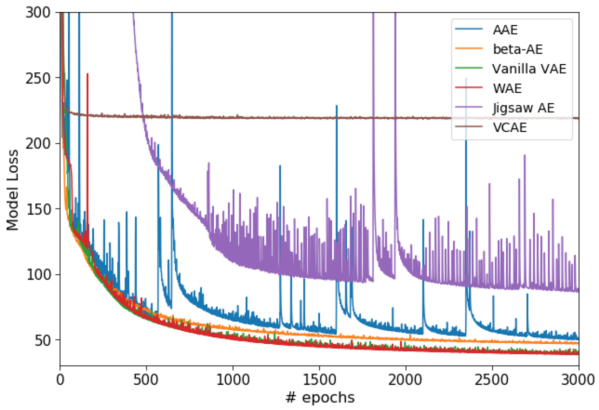

Figure 14. Training loss curves

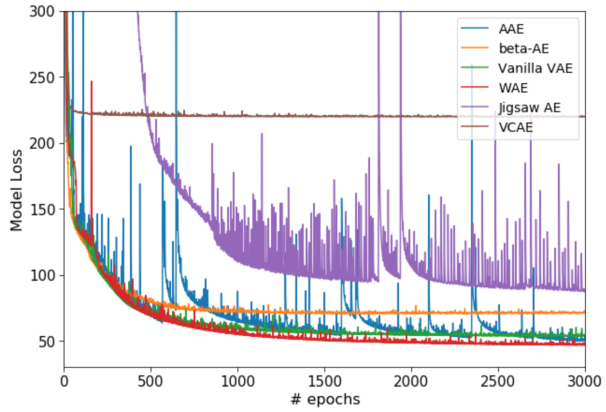

Figure 15. Validation loss curves

4.2. Contrastive Learning

The data augmentation is crucial for representation learning. In our experiments, we used two different sets of transformations. The first one used strong transformations, as the original paper proposes. However, we quickly figured out that such strong transformations yield to indistinguishable

Table 2. Loss on validation set

MODELS	MODEL CUMULATIVE LOSS
ADVERSARIAL AE	49.8
BETA-AE	70.9
VANILLA VAE	53.6
VAR. CONS. AE	220.1
JIGSAW-VAE	88.3
WASSERSTEIN AE	48.3

pictures. The visual environment in our task is scarce on representative features, breaking them down influenced the performance of compressing in a bad manner. Thus, the transformation pipeline was changed in the direction of softer transform. The scale limits of the random crop were adjusted to ensure the safety of representation and probabilities was adjusted downwards. The optimal transformation we found is summarized in the following points.

- Random crop and resize to the original size, the scale limits: [0.6 - 1].
- Random horizontal flip.
- Random change of the brightness, contrast and saturation of an image.
- Random Gaussian blur as proposed in the original paper.

Concerning the encoding network, the first experiment was conducted using ResNet-50 as a backbone. It was initialized with pre-trained weights taken from the official repository[2]. As a projection head, we used fully connected layers, followed by batch normalization layers and ReLU non-linearity between linear layers. The projection head stayed the same for all experiments. The whole model was finetuned on the target dataset and then used in the inference phase for training RL agent. However, it turned out that the forward pass of this network takes too much time on a CPU unit. The time of forward pass is crucial for online learning because the encoded vector can become outdated if it takes too much time to compute. That is why, in the next experiments, we focused on ResNet-18 encoding architecture. The ResNet-18 appeared to be faster in compressing the data, however, it is still slower than all the autoencoders we experimented with. Using this CNN architecture, we conducted different experiments varying hyperparameters of the model and training setup. Firstly, we tried training the model from scratch and compared the loss with the model finetuned on ImageNet. As expected, the latter performed better, and we used it as our final SimCLR network. Authors of the

[1] https://github.com/SamirMoustafa/auto-features-extraction-for-RL

[2] https://github.com/google-research/simclr

Table 3. SimCLR model perfomance with different hyperparameters

τ/BS	512	1024
0.3	3.74	3.96
0.5	3.91	4.16

original article reported that contrastive learning benefits from larger batch sizes and longer training. Also, representation learning with contrastive cross-entropy loss benefits from normalized embeddings and an appropriately adjusted temperature parameter. Following these insights, we varied these hyperparameters with the outcomes summarized in table 3.

The best combination we found is: $\tau = 0.3$, $BS = 512$. Additionally, the optimizer's hyperparameters and learning rate scheduling setup was tuned to achieve better performance. The learning curves for training and validation of the final model is presented in Figure 16.

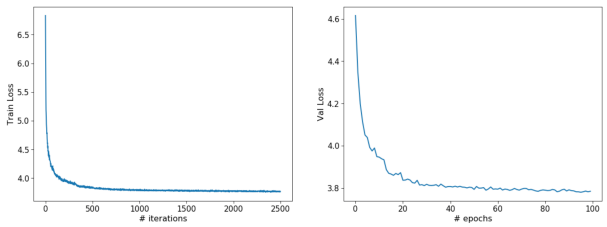

Figure 16. Final SimCLR learning curves

The critical issue we faced with the SimCLR approach is the problem of model verification before it is used for the inference in training the RL agent. In the case of autoencoders, we can observe the reconstructions produced by decoder network and reconstruction loss value. With SimCLR self-supervised learning, we do not have any labeled data to verify the quality of learned representations. That is why the verification made manually by sampling two sets containing similar images inside each but different in different sets. We also tried to do four different clustering algorithms in the latent space to explore if any groups of vectors correspond to the groups of similar images. However, as expected, the clusters overlap, and the highest silhouette score we achieved is 0.1. Due to the curse of dimensionality, we could not rely on Euclidian distance between vectors, and that is why we used cosine similarity between 64-dimensional vectors, which provides a bit stronger notion of distance in high-dimensional space. We compared this metric for these two sets taking images' latent vectors. It appeared that images inside one set are closer to each other in terms of cosine similarity than images taken from different sets. Based on this fact, we used the SimCLR network for training RL agent online. The cumulative reward over 130 episodes is presented in Figure 17. We can see that SimCLR performed better than training on raw images did, however, it is worse than all the autoencoding techniques we experimented with. We consider the three following reasons for that.

- Non-representative visual environment of the track makes the model unable to distinguish images inside a batch.
- One-track environment almost without any objects on images except for the road and car hood, which look the same way on different pictures.
- The high computational cost of the forward pass makes the online training performance suffer.

Summing up, we can say that autoencoders suit better for this particular environment, simulation configuration, and embedded system setup.

4.3. Reinforcement Learning

The methodology of experiments was following. We run training of our SAC implementation in several modes: without any dimensionality reduction (use full frames as observations), with encoder and in some cases bottleneck parts of different types of autoencoders and with representation learning model. Training was run for 130 episodes. Episode terminates when car overcomes the threshold of deviation from the road's boundaries or if it bumps the object around the track. For each episode total reward for that episode was calculated. For good policy, this reward should tend to increase with the number of episodes. Since the learning process for us was not very stable, we represent this metric through another metric - the average reward that was obtained for last 10 episodes (decades). The final results are shown in Figure 17.

From that plot we can figure out following insights. First of all, the usage of the autoencoders dramatically improved the performance of the RL algorithm. Training the critic and policy networks with convolutional layers seems to be very complicated process that will obtain good results in a very long terms. The performance difference between different autoencoders is not very much, but we can consider that standard VAE, Adersarial AE and Wasserstain AE showed better performance.

Also we can see that Representation Learning technique didn't get such good results as autoencoders. We can consider following results for that. The one reason is the computational complexity for such networks that leads to quite high delay between moment of the time when network receives current frame and when it produces output vector. The higher that latency, the less relevant the output vector

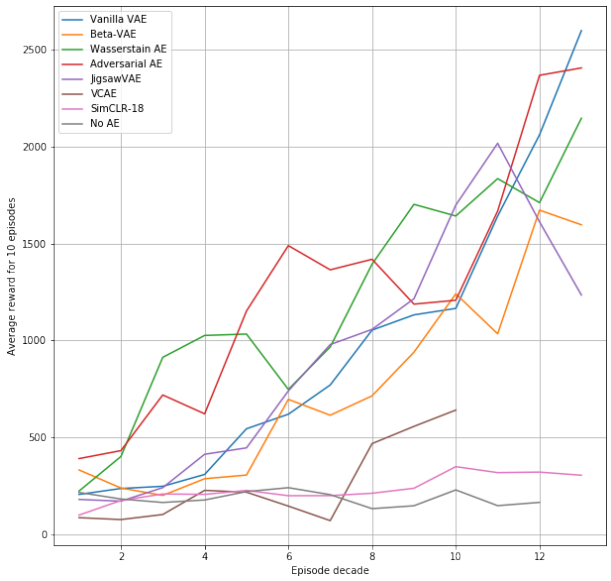

Figure 17. Results for different methods

will be. For SimCLR-50 model that latency was so high that we couldn't train our agent at all. For SimCLR-18 things are better, but still it is slower than encoders. The second possible reason is that the autoencoders and representation learning networks have different objective functions, and thus objectives of autoencoders seem to be more suitable in our case. We elaborated on this topic in the previous subsection.

Overall, results seems to be consistent with the results from state-of-the art paper (Prakash et al., 2019), despite we use different environment, conditions and value function setting. In this work authors achieved cumulative reward of 2500 for 130 episodes using their best configuration of VAE. In our case we achieved almost the same performance with Vanilla VAE, Adversarial AE and Wasserstein AE. The more advanced tuning of hyperparameters for these models would certainly allow to achieve higher rewards.

5. Conclusions

In our work, we provided an in-depth study and exploration of unsupervised approaches and self-supervised state-of-the-art approach for leaning the useful features from input images. The explored encoding techniques allowed us to achieve a much higher cumulative reward for online RL agent training than training directly from raw images. We showed that such models could be successfully incorporated into the embedded systems of autonomous cars or robots.

References

Arulkumaran, K., Deisenroth, M. P., Brundage, M., and Bharath, A. A. A brief survey of deep reinforcement learning, 2017.

Braithwaite, D., O'Connor, M., and Kleijn, W. Variance constrained autoencoding. *arXiv preprint arXiv:2005.03807*, 2020.

Chen, T., Kornblith, S., Norouzi, M., and Hinton, G. A simple framework for contrastive learning of visual representations, 2020.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

Kingma, D. P. and Welling, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Li, H., Wei, T., Ren, A., Zhu, Q., and Wang, Y. Deep reinforcement learning: Framework, applications, and embedded implementations, 2017.

Loic Matthey, Nick Watters, G. D. A. L. Understanding disentangling in -vae. 2018.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning, 2013.

Prakash, B., Horton, M., Waytowich, N. R., Hairston, W. D., Oates, T., and Mohsenin, T. On the use of deep autoencoders for efficient embedded reinforcement learning, 2019.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms, 2017.

Taghanaki, S. A., Havaei, M., Lamb, A., Sanghi, A., Danielyan, A., and Custis, T. Jigsaw-vae: Towards balancing features in variational autoencoders. *arXiv preprint arXiv:2005.05496*, 2020.

Tolstikhin, I., Bousquet, O., Gelly, S., and Schoelkopf, B. Wasserstein auto-encoders, 2017.

A. Team member's contributions

Explicitly stated contributions of each team member to the final project.

Ildar Abdrakhmanov

- Responsible for Wasserstein Autoencoder. Implementation and validation of WAE. Training WAE big configuration on the target dataset.
- Implementation of some auxiliary functionality for the general AE pipeline.
- Implementation, training and validation of Contrastive Learning Framework. Conducting all the corresponding experiments.
- Preparing Sections 1, 2, 4, 5 of the report and corresponding subsections in other parts of the report.

Pavel Kopanev

- Examine possible core and custom gym environments suitable for the project
- Integrate the Donkeycar simulator into the project and modify Unity scene
- Collect dataset for training all feature extractors (Autoencoders and Contrastive Learning Framework)
- Contribution in implementation of Contrastive Learning Framework, results validation
- DDQN implementation without validation

Timur Akhtyamov

- Responible for project proposal
- Organizing and distributing tasks
- Responsible for Soft Actor-Critic implementation
- Summarizing all the results for RL

Samir Mohamed

- Build and construct the GitHub repo.
- Design the auto-encoders infrastructure (base)
- Responsible for beta-variational auto-encoder
- Train all the auto-encoders on our dataset
- Summarize all the results for feature extraction
- Add utilization functions to help the team

Anna Nikolaeva

- Building Jigsaw auto-encoder
- Training Jigsaw auto-encoder on our dataset
- Implementation of loading part in the infrastructure up to DataLoader

Ilia Ivanov

- Implementing vanilla auto-encoder
- Training vanilla auto-encoder on MNIST
- Also tried to implement Modified VAE (from https://arxiv.org/pdf/1812.07352.pdf)

Yerzhaisang Taskali

- Responsible for Adversarial auto-encoder
- Training AAE on MNIST
- Also tried to implement and train another autoencoder on MNIST to compare results

Anastasia Chernysheva

- Responsible for implemetation of Variance Constrained Autoencoding
- Train VCAE encoder/decoder part of model on MNIST (it achieves 0.1 val loss)
- Train VCAE NF part of model on MNIST (it achieves 0.2 train loss)

B. Third-party code list

- https://github.com/google-research/simclr - official github repo for Contrastive Lerning (TensorFlow). We didn't use their code in any way but some implementation insights were taken from there.
- https://github.com/AntixK/PyTorch-VAE/blob/master/models/vanilla_vae.py - we didn't use the code from here directly, but we borrowed some ideas
- https://github.com/joeybose/pytorch-normalizing-flows - normalization flows MAF
- https://github.com/lksmllr/beta-vae-3D-shapes - beta Variational Autoencoder learning the latent representation
- https://github.com/JasonPlawinski/AutoEncoderTest - taking some ideas to implement Adversarial Autoencoder.