Visualization of RL Agents' Learning Process Using Prior-Based Modification Of t-SNE Algorithm

Guy Shkury and Manor Zvi
Supervised By: Tom Zahavy
Electrical Engineering Department
The Technion - Israel Institute of Technology
Haifa 32000, Israel

Abstract

Deep reinforcement learning (DRL) agents are used nowadays in a large variety of applications such as robotics, video games and healthcare. Despite the abundance of applications for DRL agents, the learning process itself remains largely unexplored. In this paper we demonstrate a visualization methodology and use it to analyze the learning process of A2C agents trained on the game Atari Breakout (Atari2600). We present examples for using our method to explore how A2C agents learn to carve side-tunnels, avoid states of losing a game and other useful skills. Our method uses a modified version of the popular nonlinear dimensionality reduction algorithm t-SNE as described in the work of Maaten and Hinton (2008). The modification includes two different kinds of priors that overcome the stochastic nature of t-SNE, and intend to keep the low dimensional representations of the agents' checkpoints comparable along the learning process. The paper also presents comparison between the original t-SNE and its modified version, and prove that our priors exploit t-SNE's capabilities to preserve high dimensional information in a low dimension map, while adding the consistency property to it.

Introduction

Reinforcement Learning (RL) is one of the three fundamental branches of Machine Learning (ML), alongside Supervised Learning and Unsupervised Learning. RL agents learn to behave in an environment by performing different actions while trying to maximize the rewards they receive from that environment. In recent years, many significant achievements were accomplished using RL agents, thus leading to growth in interest in further understanding and exploring these agents. Countless learning algorithms were developed in the field over the years, e.g. Q-Learning, SARSA (Rummery and Niranjan (1994)), DDPG (Lillicrap et al. (2015)), etc. In addition, new techniques such as Experience Replay (ER) (Lin (1993)) were introduced to further optimize the learning process.

Engineering of networks and agents using these methods relies, more often than not, on experimental trial and error rather than fully understanding the way these methods affect the learning process. Having a deeper understanding of how each method affects the learning process will allow us to optimize networks in a way that better suits the problem in hand, and to tackle problems more efficiently, i.e. with less hyper-parameters, simpler networks, and shorter training times. Yet, the field of exploring the learning process itself, i.e. how, why and when does an agent learn a new skill, is generally overlooked.

Proper representation of an agent's perception of the world is fundamental to explore its learning process. Various methods tried to tackle the problem of state representation, for example that of Zeiler and Fergus (2014), which visualized intermediate feature layers of convolutional neural networks (CNN). This visualization method showed how different kernels in different layers of the network learn to respond to specific patterns. Yet, this method can only be applied to CNNs. Other methods, such as the work of Zahavy, Ben-Zrihem, and Mannor (2016), projected the latent space of an agent network onto a 2D space using dimensionality Reduction (DR) techniques. This method represents how an agent perceives similar states in a way that minimizes spatial distance between states of similar properties, and vice versa. It also shows how an agent can distinguish between states that are non-similar in a semantic manner, but similar in classic measures such as pixel-wise L2 norm. Despite all this method's merits, it merely deals with the final state of the agent's network, and completely ignores the learning process itself. A naïve approach to visualize the learning process of an RL agent would be to stack presentations of these 2D maps at different milestones along the learning process. The result of this approach is perplexing, though, as a result of the random nature of the Stochastic Gradient Descent (SGD) algorithm that's frequently applied. One cannot expect the 2D embedding of two consecutive high dimensional maps to be similar or comparable, let alone a whole process of trainig a network. Small changes in the algorithm's inputs can cause large difference in its outputs, causing the consecutive maps to be non-continuous, or inconsistent, and therefore incomparable.

Another major concern of this type of representation techniques is The Curse Of Dimensionality: the hardness to maintain spatial information when reducing the number of dimensions of the data. The hardness to do so increases as the number of dimensions in the original data increases. Many different DR algorithms exist to try and cope with this issue: LLE (Roweis and Saul (2000)), ISOMAP (Tenen-

baum, De Silva, and Langford (2000)), and most notably-t-SNE (Maaten and Hinton (2008)), which is currently considered to outperform most of the other algorithms.

In order to address these two issues of embedding inconsistency and The Curse Of Dimensionality, we used two priors added to the t-SNE algorithm. These priors are intended to keep the low dimensional maps of consecutive stages consistent and therefore comparable, while exploiting t-SNE's superior performance in preserving high dimensional information, in a low dimensional, easy to comprehend, embedding.

In this paper we present a novel approach to investigate the training process of a network by introducing two key components:

- Applying prior knowledge to the t-SNE algorithm, to improve its consistency between runs
- Innovative visualization tool to help exploring the learning process of an agent, and enable intuitive comparison between different agents

Background

RL

The goal of RL agents is to maximize their expected total reward by learning an optimal policy (mapping states to actions). At time t the agent observes a state s_t, selects an action a_t, and receives a reward r_t. Following the agent's decision it observes the next state s_{t+1}. We consider infinite horizon problems where the cumulative return is discounted by a factor of $\gamma \in [0, 1]$ and the return at time t is given by:

$$R_t = \sum_{m=1}^{T} \gamma^{t'-t} r_t \tag{1}$$

where T is the reward horizon we define for the problem.

The action-value function $Q^\pi(s, a)$ measures the expected return when choosing action a_t at state s_t, and following policy π:

$$Q^\pi(s, a) = \mathbb{E}[R_t | s_t = s, a_t = a, \pi] \tag{2}$$

afterwards.

The optimal action-value obeys a fundamental recursion known as the Bellman equation:

$$Q^*(s_t, a_t) = \mathbb{E}[r_t + \gamma \max_{a' \in \mathbb{A}} \{Q^*(s_{t+1}, a')\}] \tag{3}$$

where $\mathbb{A}$ represents the agent's action space.

Deep Q Networks

Mnih et al. (2013, 2015) approximate the optimal Q function using a Convolutional Neural Network (CNN). The training objective is to minimize the expected TD error of the optimal Bellman equation:

$$\mathbb{E}_{s_t, a_t, r_t, s_{t+1}} ||Q_\theta(s_t, a_t) - y_t||_2^2 \tag{4}$$

where:

$$y_t = \begin{cases} r_t & s_{t+1} \text{ is terminal} \\ r_t + \gamma \max_{a' \in \mathbb{A}} \{Q_{\theta_t arget}(s_{t+1}, a')\} & \text{otherwise} \end{cases}$$

Notice that this is an off-line algorithm, meaning that the tuples $\{s_t, a_t, r_t, s_{t+1}, \gamma\}$ are collected from the agents experience, stored in the ER and later used for training. The reward r_t is clipped to the range of $[-1, 1]$ to guarantee stability when training DQNs over multiple domains with different reward scales. The DQN algorithm maintains two separate Q-networks: one with parameters θ, and a second with parameters θ_{target} that are updated from θ every fixed number of iterations. In order to capture the game dynamics, the DQN algorithm represents a state by a sequence of history frames and pads initial states with zero frames.

Actor Critic

As explained in the work of Konda and Tsitsiklis (2000), traditionally the vast majority of RL methods belonged to one of two categories: Actor-only methods and Critic-only methods. In actor-only methods, the gradient of the performance, with respect to the actor parameters, is directly estimated by simulation. Critic-only methods, on the other hand, are indirect in the sense that they do not try to optimize directly over a policy space, but rather calculate a value function approximation, and deduce policy from it.

Actor-critic methods aim at combining the strong points of actor-only and critic-only methods. The critic uses an approximation architecture and simulation to learn a value function, which is then used to update the actor's policy parameters in a direction of performance improvement. Such methods, as long as they are gradient-based, may have desirable convergence properties, in contrast to critic-only methods for which convergence is guaranteed in very limited settings. They hold the promise of delivering faster convergence (due to variance reduction), when compared to actor-only methods.

Mnih et al. (2016) presented a variant of actor-critic methods, called A3C (Asynchronous Advantage Actor Critic). This algorithm maintains a policy $\pi(a_t|s_t; \theta)$ and an estimate of the value function $V(s_t; \theta_v)$. The algorithm operates in a forward view and uses a mix of n-step returns to update both the policy and the value function. The policy and the value function are updated after every arbitrary number of actions or when a terminal state is reached. The update performed by the algorithm can be seen as:

$$\nabla_{\theta'} \log \pi(a_t|s_t; \theta') A(s_t, a_t; \theta, \theta_v) \tag{5}$$

where $A(s_t, a_t; \theta, \theta_v)$ is an estimate of the advantage function.

Yuhuai Wu (2017); Wang et al. (2016) mention a synchronous, deterministic variant of A3C, called A2C (Advantage Actor Critic), which according to Yuhuai Wu (2017), outperforms A3C while being more cost-effective than A3C when using single-GPU machines, and being faster than a CPU-only A3C implementation when using larger policies.

t-SNE

t-SNE (T-distributed Stochastic Neighbor Embedding, Maaten and Hinton (2008)) is a non-linear DR method used mostly for visualizing high dimensional data. The algorithm initiates randomly the low dimensional embedding $\mathcal{Y}$ and

performs the following stages until convergence (or a number of iterations given by user):

- compute low-dimensional affinities q_{ij}
- compute the gradient relative to the embedding of each point $\frac{\delta C}{\delta y_i}$
- set $\mathcal{Y}^{(t)} = \mathcal{Y}^{(t-1)} + \eta\frac{\delta C}{\delta \mathcal{Y}} + \alpha(t)(\mathcal{Y}^{(t-1)} - \mathcal{Y}^{(t-2)})$, where $\alpha(t)$ represents the momentum at iteration t, and η indicates the learning rate.

This technique is easy to optimize, and it has been proven to outperform linear DR methods and non-linear embedding methods such as ISOMAP (Tenenbaum, De Silva, and Langford (2000)), in several research fields including machine learning benchmarks and hyper-spectral remote sensing data (Lunga et al. (2013)). t-SNE reduces the tendency to crowd points together in the center of the map by employing a heavy-tailed Student-t distribution in the low dimensional space. It is known to be particularly good at creating a single map that reveals structure at many different scales, which is particularly important for high-dimensional data that lies on several low dimensional manifolds. This enables us to visualize the different sub-manifolds learned by the network and interpret their meaning.

Method

t-SNE priors

t-SNE algorithm uses stochastic gradient descent to optimise the embedding of the data. Therefore, the algorithm's result is sensitive to initialisation and a random parameter (seed) given by the user. As a result, the t-SNE low dimensional output, may change significantly due to small changes in the data, or to a change of the random seed. In our work, in order to keep the embedding of consecutive learning stages continuous, we applied two different kinds of priors to the original t-SNE gradient calculation:

Initialisation Prior To begin with, we applied a naïve approach where we set a constant seed and ran a single iteration of the regular t-SNE algorithm. Then we substituted the algorithm's random initialisation with the output of the previous learning stage's embedding, for all the following iterations. Starting from the second iteration, the algorithm is first initialised to the output of the last learning stage and then begins to minimize the KL divergence between the pairwise similarity in the low dimensional map q_{ij} to that of the high dimensional data p_{ij}. Assuming small changes between the latent space representations of two consecutive stages, the algorithm still converges to a local minima, but to one that's highly probable to be close (in spatial deployment terms) to the low dimensional map of the last stage. Moreover, this implementation barely affects the performance of t-SNE algorithm in terms of running time, and merely requires saving its last output.

Gradient Prior The second approach is a more direct one. In order to achieve comparability between consecutive outputs of the t-SNE algorithm, we modified the terms that are used to calculate the gradient, to make the consecutive outputs continuous.

In the original t-SNE paper (Maaten and Hinton (2008)), the algorithm minimizes the Kullback-Leibler (KL) divergence between the joint probabilities in the high-dimensional space p_{ij} and the joint probabilities in the low-dimensional space q_{ij}. The cost function is defined as follows:

$$C = KL(P||Q) = \sum_i \sum_j [p_{ij} \log(p_{ij}) - p_{ij} \log(q_{ij})] \tag{6}$$

and under the assumption that $\sum_{k \neq l} p_{kl} = 1$, the gradient turns out to be:

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(1 + ||y_i - y_j||^2)^{-1}(y_i - y_j) \tag{7}$$

In order to keep the outputs somewhat continuous, we defined $Q^{(n)}$ as the set of low dimensional joint probabilities in the n-th iteration, and defined the cost function as:

$$C^* = KL(\alpha Q^{(n-1)} + (1-\alpha)P||Q^{(n)}) \tag{8}$$

This way the algorithm minimizes, in the n-th iteration, the KL divergence between the low dimensional joint probabilities $q_{ij}^{(n)}$ and a combined term of the high-dimensional joint probabilities p_{ij} and the low dimensional joint probabilities of the previous iteration $q_{ij}^{(n-1)}$. As demonstrated in Appendix II: Gradient Derivation, the gradient of this cost function turns out to be:

$$\frac{\delta}{\delta y_i} C^* = 4 \sum_j (1 + d_{ij}^2)^{-1}(\alpha q_{ij}^{(n-1)} + (1-\alpha)p_{ij} - q_{ij}^{(n)})(y_i - y_j) \tag{9}$$

Note also that this implementation affects the performance of t-SNE algorithm in terms of running time by requiring multiplying by a constant two large matrices and summing them up. For our experiments, this slightly affected the t-SNE running, but increased its memory requirements to a certain extent.

Hand-crafted features

In our work, we are looking to model the agent's understanding of the different game states. We expect game states to aggregate in advanced learning stages according to features that are significant for the agent to react. To be able to analyze the meaning of these aggregations (clusters), we manually designed and extracted hand-crafted features for the Atari2600 game Breakout, on which we tested our algorithm, in a similar way to the work of Zahavy, Ben-Zrihem, and Mannor (2016). These features include game data like score and remaining lives, features that we find significant for one to play the Breakout game such as the paddle location, ball location, total bricks left, existence of a tunnel, and the depth of the tunnel in case it isn't fully dug yet. We also introduced features pertaining the agent itself- the estimated value function for each state, its taken action for that state, and the standard deviation (std) of its outputs (as a measure of the agent's confidence to take that action).

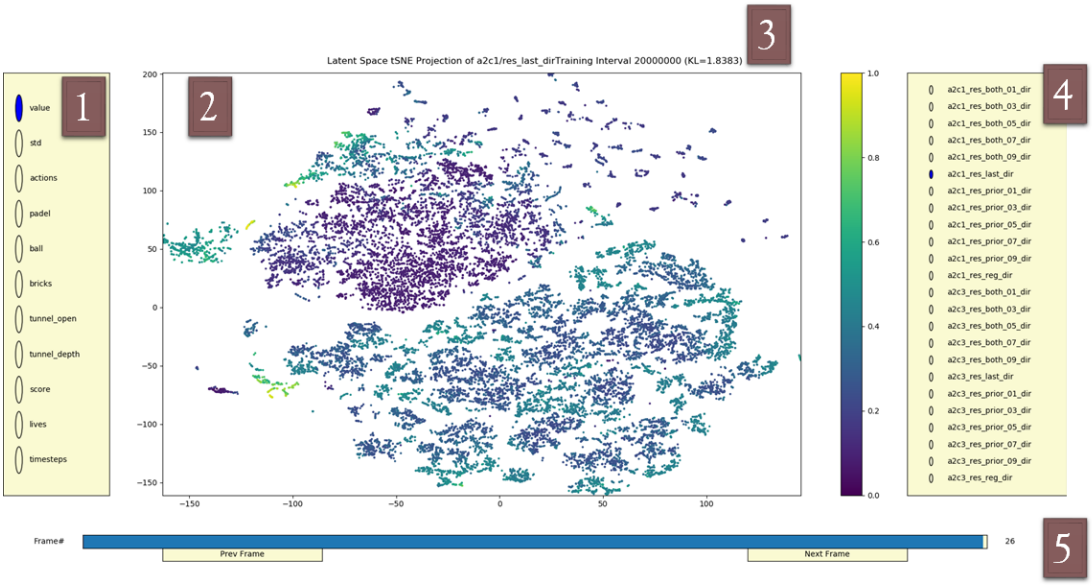


Figure 1: This figure shows an example for the GUI we created for this work. The GUI contains a bar [1] for selecting coloring setup, an interactive presentation of the data [2], a title [3] with useful information such as current agent being presented and KL divergence value. It also contains another bar [4] for choosing an agent and an algorithm, and a sliding window [5] to choose a learning stage from the collected checkpoints. For example, the above figure presents the approximated value function of each state, according to the agent named A2C1. The presented map is that of the final training stage, and the embedding was calculated using the Initialisation Prior alone (shortly named Last in the figure). In this figure, only two agents are presented in bar [4] to avoid cluttering the figure.

Graphical User Interface (GUI)

In order to allow the analysis of agent's learning process, we created an interactive GUI, based on the Matplotlib package for python. The GUI, shown in Figure 1, lets the user to choose an agent in one of the saved stages of the learning process to see the embedding of its latent space. The user can then toggle between agents and learning stages, and present the hand-crafted features on the 2D embedding map to analyze the results. The interface also lets the user to mark data points in one view, and see their embedding in other views (different agents or learning stages). Moreover, it allows the user to present a game frame that corresponds to a point, or view the average image of a bulk of data points.

Experiments

Our work consists of 2 main features: a modification for the t-SNE algorithm, and a GUI for the learning process analysis. To analyze the performance of these two modules, we created several experimental setups:

Priors Analysis

We tested our modified t-SNE algorithm versus the original t-SNE algorithm for both DR performance and consistency between consecutive runs. When applying the Gradient Prior, as α increases, the last run's weight increases in the algorithm's cost function, on the expense of the weight of the high dimensional affinity. Therefore, we expected the prior based approach to improve consecutive runs consistency, at the cost of impacting the DR performance in an extent that's correlated to the α value given by the user. We also expected our Initialisation Prior to maintain runs consistency, but to cause the algorithm to converge to a local minima that is not necessarily as good as the one reached by the original t-SNE algorithm.

First we tested the DR performance of the different priors on both the MNIST[1] and the Fashion-MNIST[2] data sets. To carry out the test we built two CNNs and trained them on the two data sets. For the MNIST data set we built a network composed of a convolution layer with kernel size 1x20x5, followed by another convolution layer with kernel size 20x50x5, and ends with 2 fully connected (FC) layers of size 500 and 10. For the Fashion-MNIST data set we built a network composed of a convolution layer with kernel size 1x32x5, followed by another convolution layer with kernel size 32x64x5, and ends with 2 FC layers of size 512 and 10.

We trained these networks on the two data sets, both for 2 epoches by batches of 64 images, on the 60,000 images train set. The networks reached 98% accuracy on the MNIST data set, and 85% accuracy on the Fashion-MNIST data set. After every 10 batches of training, we ran the 10,000 images of the test set through the networks and saved the latent

[1]The MNIST data set is publicly available at http://yann.lecun.com/exdb/mnist/index.html.

[2]The Fashion-MNIST data set is publicly available at https://github.com/zalandoresearch/fashion-mnist

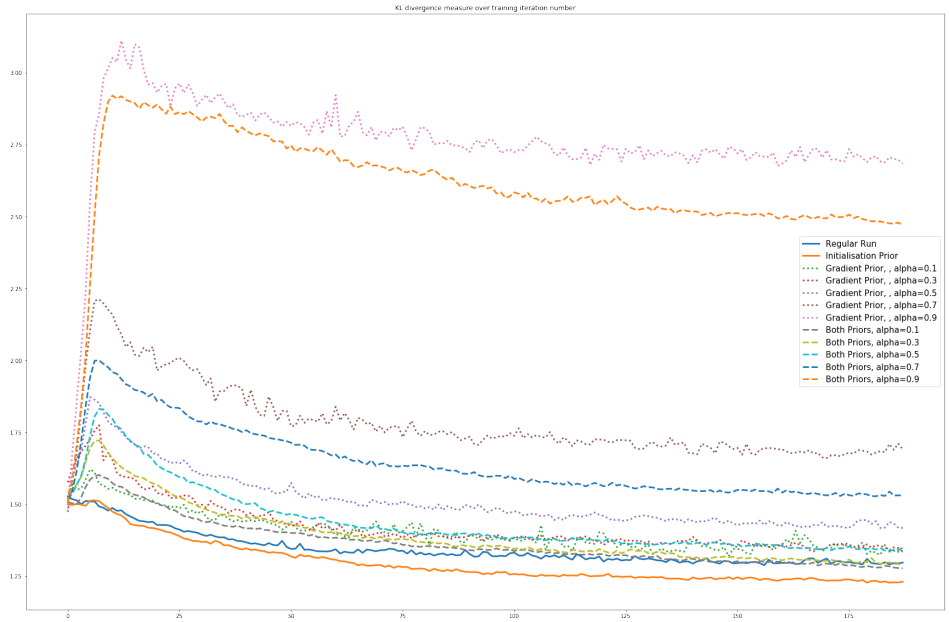


Figure 2: Kl divergence measure over training iteration number, for an agent trained on MNIST data set

space issued by the last layer of the network, for each input image. This way we obtained a total of 188 checkpoints for each image in the test set, along 188 different stages of the training process. These representations ware inserted as an input for the different DR algorithms for further investigation. In this experiment we tested the performance of a regular t-SNE algorithm against t-SNE algorithm with our Initialisation Prior, against t-SNE algorithm with our Gradient Prior with 5 different α values, and finally against t-SNE algorithm with both of our priors implemented at the same time, with the same 5 different α values.

Figures 2 and 3 present the KL divergence of the embedding of each of the algorithms runs, for both of the data sets. The KL divergence in this case acts as a measure of the algorithm's ability to maintain high dimensional information, in a low dimensional map of the data. The KL divergence here is calculated between the pairwise affinity of the data in the high dimensional space and the pairwise affinity of the low dimensional embedding of the data, as calculated in the original t-SNE paper. As one can see, we found that large values of α (where the weight of the previous output is high) maintain the pairwise affinity of the high dimensional data in a poor way. We also found out that for the same α value, using both of the priors simultaneously increases the algorithm's performance slightly compared to using only the Gradient Prior. In addition, when both of the priors are applied with small α values such as 0.1, the algorithm converges to a KL divergence value similar to the original t-SNE algorithm. To our surprise, in both of the experiments the t-SNE algorithm with our Initialisation Prior outperforms the t-SNE algorithm with the regular random initialisation. This result is consistent for the data sets used for our work, but might be data dependent, and requires further investigation. This result is surprising since it restricts the algorithm to a local minima that is close to the one it reached in the previous run (for a similar but different data).

In order to test our algorithms for consistency we calculated the sum of euclidean distances between the embedding of the same point in every two consecutive runs. The results, presented in Figure 7 prove that the regular t-SNE algorithm does not maintain consistency well. They also show that our Gradient Prior, when applied without the Initialisation Prior, does not improve consistency. That being said, applying the Gradient Prior alongside the Initialisation Prior, improves the algorithm's consistency.

Learning Process Analysis

For this experiment we used a setup that is composed of three A2C agents trained separately on the Atari2600 game of Breakout for $2 \cdot 10^7$ time steps, each time step is made up from 4 game frames. All three agents used the same setup as defined in Yuhuai Wu (2017), and based on Mnih et al. (2013): the networks used a convolutional layer with 16 filters of size 8x8 with stride 4, followed by a convolutional layer with with 32 filters of size 4x4 with stride 2, followed by a fully connected layer with 256 hidden units. All three hidden layers were followed by a rectifier nonlinearity. The model had two sets of outputs – a softmax output with one entry per action representing the probability of selecting the action, and a single linear output representing the value function. It used a discount of $\gamma = 0.99$, an RMSProp decay factor of $\alpha_{RMS} = 0.99$, and entropy regularization with a weight $\beta = 0.01$.

We trained the agents independently on the game of Breakout, and saved a checkpoint (a copy the agent with all its parameters at that point in the learning process) every fixed number of time steps, to a total of 27 checkpoints for each agent. Once we had a fully trained agent, we recorded

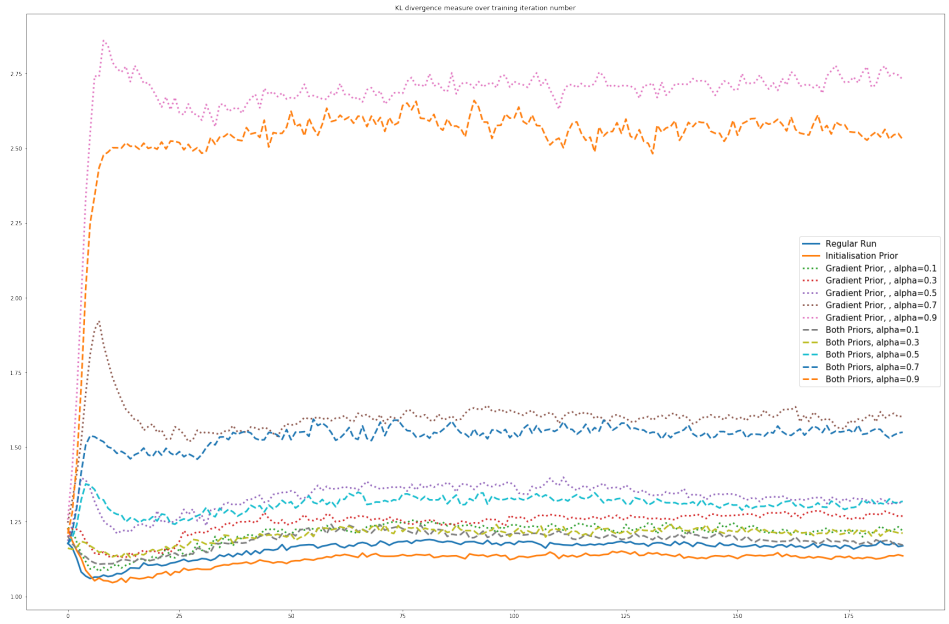


Figure 3: Kl divergence measure over training iteration number, for an agent trained on Fashion MNIST data set

its game states to cover a large variety of different states. These recorded game states were fed as an input to each of the saved checkpoints, to calculate their latent space representations according to the agent in that stage. These 256 dimensional representations were used as an input to our different DR algorithms to receive 2D maps, which we presented in our GUI to analyze the agents' learning process.

Key Game Insights Often we would like to know whether and when an agent "understands" certain key aspects of the problem it faces. In the game of Breakout, as elaborated in Zahavy, Ben-Zrihem, and Mannor (2016), such a key aspect is the existence, or lack, of a tunnel, in which the ball can traverse from the lower part of the board (below the bricks) to the upper part. Passing the ball to the upper part of the board is extremely beneficial and the risk of losing the game is low because the ball tends to hit bricks over and over again until it comes back down to the dangerous area. In our experiment, we flagged all the game states where there is a clear tunnel- a column without bricks, from the bottom to the top of the board. Figure 4 shows that our agents managed to learn the significance of an open tunnel, and therefore their latent space is similar, causing them to aggregate in the low dimensional map.

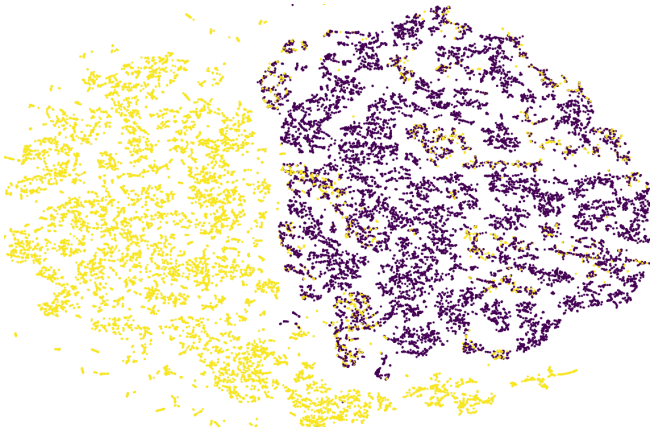
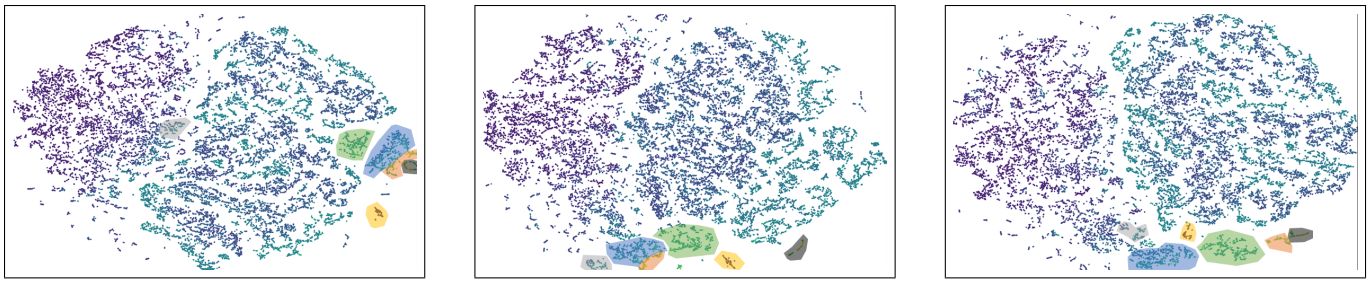


Figure 4: Embedding of the final learning stage, where the game states are colored according to the existence of a tunnel in the state. The map shows the aggregation of game states where a tunnel is open (yellow dots) and the aggregation of states where there is no open tunnel (purple dots). This figure was created using none of the priors, yet the results are consistent with all of our algorithms and the different agents.

Clusters Affinity Not only the affinity between game states, but also the affinity between clusters of them, is important to understand the way an agent perceives the game states. Zahavy, Ben-Zrihem, and Mannor (2016) defined a skill of an agent as its ability to traverse from one significant cluster to another by taking deliberate actions. Clusters proximity may indicate the learning of such skills, and the likelihood of taking them. Figure 5 presents an example of 3 A2C agents, trained separately, at their final, fully trained, state. The figure shows that the same clusters are formed in 3 different agents, and also that most of these clusters are considered by all three of them as related. This information helps us to distinguish coincidental results or proximity between clusters, and deliberate vicinity between them. For example- in the figure, all three agents perceive the clusters of an open side-tunnel, where the ball is above the bricks line, as clusters of similar significance. In contrary, the figure shows that the cluster of an open central tunnel is perceived by agent A2C1 differently than A2C2 and A2C3.



(a) A2C1 (b) A2C2 (c) A2C3

Figure 5: This figure shows clusters of the same game states, and the relative positions of them in the embedding of the final state of 3 different agents. The different clusters are colored in different colors where the yellow area is a cluster of losing states, the dark gray area is a cluster of the highest valued states- where the ball has just traversed a side-tunnel and is above the bricks. The orange and blue areas are clusters similar to the dark gray one, except there are less bricks left (in the area blue there are less bricks than the orange area). The green area is a cluster of states where a side-tunnel is open, but the ball is below the bricks. Finally, the light gray area contains states where a central tunnel is open. These graphs were created using the regular t-SNE maps.

Clusters Formation The GUI enables one to explore different properties of a learning algorithm. One example of such property is the learning of a skill, or an insight, over time. For example, Figure 6 shows how an agent learns over time to interpret game states where the ball is at the same height, or below, the paddle, meaning the player will inevitably lose a life in the upcoming states. The figure presents t-SNE embedding of the recorded game states' latent space, where the marked points belong to these losing game states. The figure shows how the agent learns over time, that these states represent a similar situation in the game, even though the game frames themselves are not necessarily similar: while in figures 6a and 6b, the agent perceives the game states as unrelated to each other, by Figure 6c it has already learned to relate two sub-groups of these states: one with high value function (green dots), where there's an open tunnel, and one subgroup of low value states, without an open tunnel. In Figure 6e the two sub-groups are already clustered together, and stay clustered until the end of the training session Figure 6f. It seems that the agent has learned early in its learning process that digging a tunnel is profitable, but only in much later stages (Figure 6e) realised that an open tunnel is useless when the game is lost. In these late stages, all the losing states are considered very low valued, and are clustered together- meaning that the losing linkage is stronger than that of the open tunnel linkage.

It is important to state that these results vary somewhat from one DR algorithm to another, though the same general results and conclusions can be obtained. Specifically some versions of t-SNE cluster the data points slightly different. Note also that the spatial location of the clusters is insignificant and derives from the first run's initial random placement. The vicinity of the data points and clusters is much more meaningful.

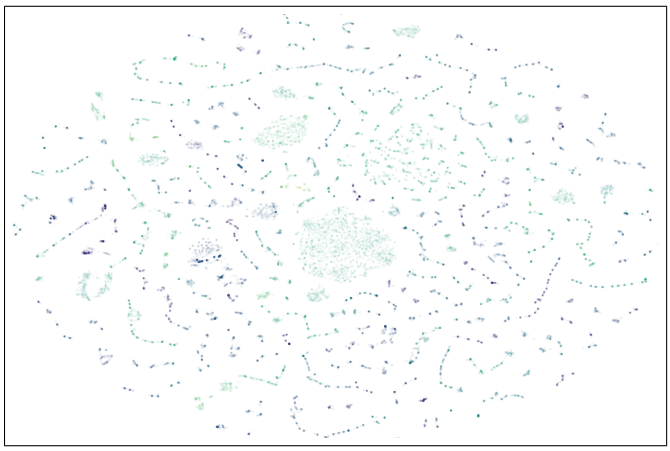
Discussion

In this paper we explored the generally overlooked field of the learning process of DRL agents. First we introduced two types of priors and applied them as a modification for the t-SNE algorithm. We tested these priors for performance in terms of dimensionality reduction capabilities, and in terms of consistency. We found that the naïve approach of the Initialisation Prior, slightly outperforms the implementation of using both of the priors simultaneously, and outperforms by a large margin the Gradient Prior when applied alone. We even found that for the data at hand, forcing the t-SNE initialisation to the previous output, helps it to converge to a better result.

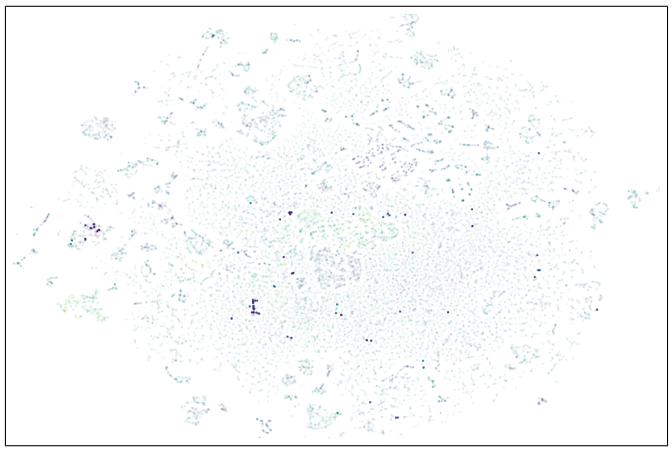
Then, we introduced our innovative visualization tool and used it to analyze the learning process of A2C agents trained on the game Atari Breakout (Atari2600). We exploited our new, consistent algorithm and presented how our method can be used to show that these agents learn useful skills in the environment of the Breakout game. This tool may serve as an aid to compare different training algorithms in order to tackle problems more efficiently, with less hyperparameters, simpler networks, and shorter training times. It may also enhance our understanding of the way these agents tackle problems.

While having many applications and obvious advantages, as elaborated in this paper, our method requires saving numerous checkpoints along the training process of a network, thus is memory and computing demanding. It also requires running the resource-exhaustive t-SNE algorithm over and over, though our priors barely affected its running time, if at all.

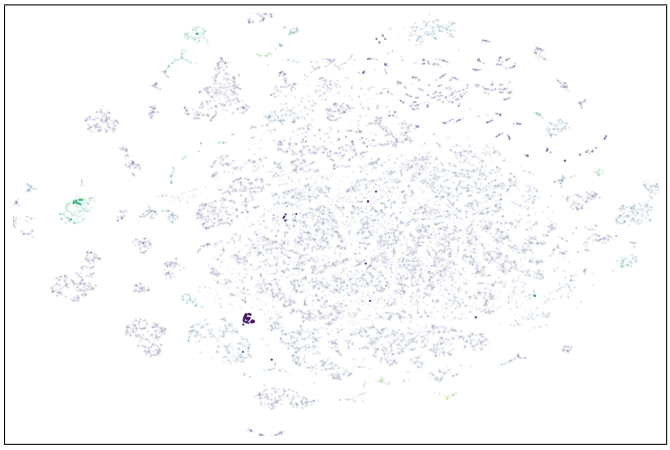
This paper offers a novel way to explore how DRL agents learn, and suggests several directions for using it. That being said, further investigation will be required on comparisons between different training algorithms and other network implementation. We also noted, in our studies, that despite we used a constant number of time steps for collecting checkpoints, the network changes much more rapidly in its early learning stages. Therefore we suggest implementing this method in a way that's increases the gap between consecutive checkpoints as the training process advances.



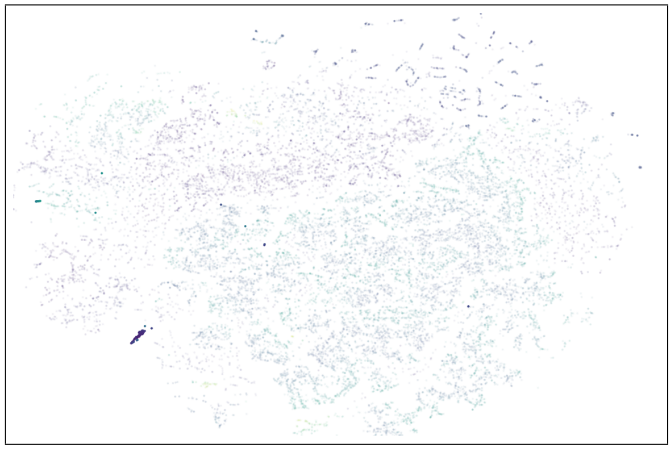
(a) 40 time steps



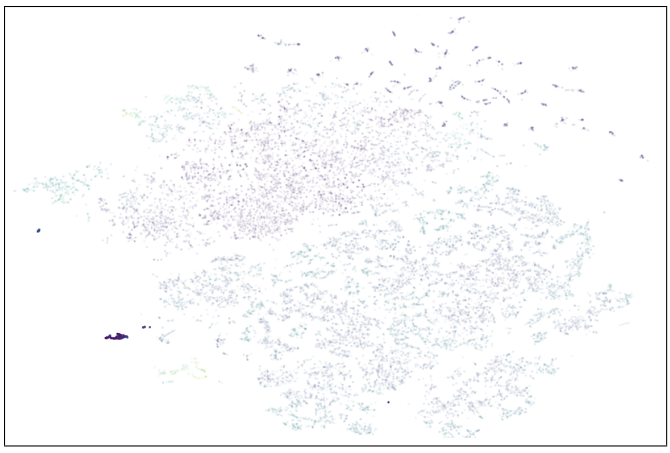
(b) 1,592,000 time steps



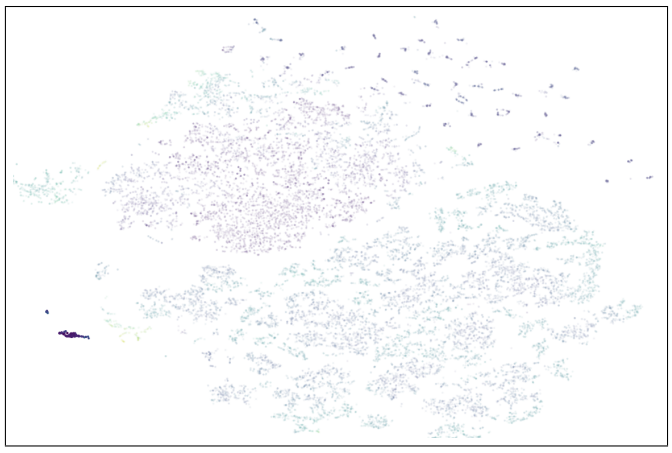
(c) 3,980,000 time steps



(d) 9,552,000 time steps



(e) 15,920,000 time steps



(f) 20,000,000 time steps, final state

Figure 6: t-SNE embedding of the recorded game states' latent space, using Initialisation Prior. In this figure, the marked points represent game states where the ball is at the same height, or below, the paddle, meaning the player will inevitably lose a life in the upcoming states.

References

Konda, V. R., and Tsitsiklis, J. N. 2000. Actor-critic algorithms. In *Advances in neural information processing systems*, 1008–1014.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Lin, L.-J. 1993. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

Lunga, D.; Prasad, S.; Crawford, M. M.; and Ersoy, O. 2013. Manifold-learning-based feature extraction for classification of hyperspectral data: A review of advances in manifold learning. *IEEE Signal Processing Magazine* 31(1):55–66.

Maaten, L. v. d., and Hinton, G. 2008. Visualizing data using t-sne. *Journal of machine learning research* 9(Nov):2579–2605.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, 1928–1937.

Roweis, S. T., and Saul, L. K. 2000. Nonlinear dimensionality reduction by locally linear embedding. *science* 290(5500):2323–2326.

Rummery, G. A., and Niranjan, M. 1994. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, England.

Tenenbaum, J. B.; De Silva, V.; and Langford, J. C. 2000. A global geometric framework for nonlinear dimensionality reduction. *science* 290(5500):2319–2323.

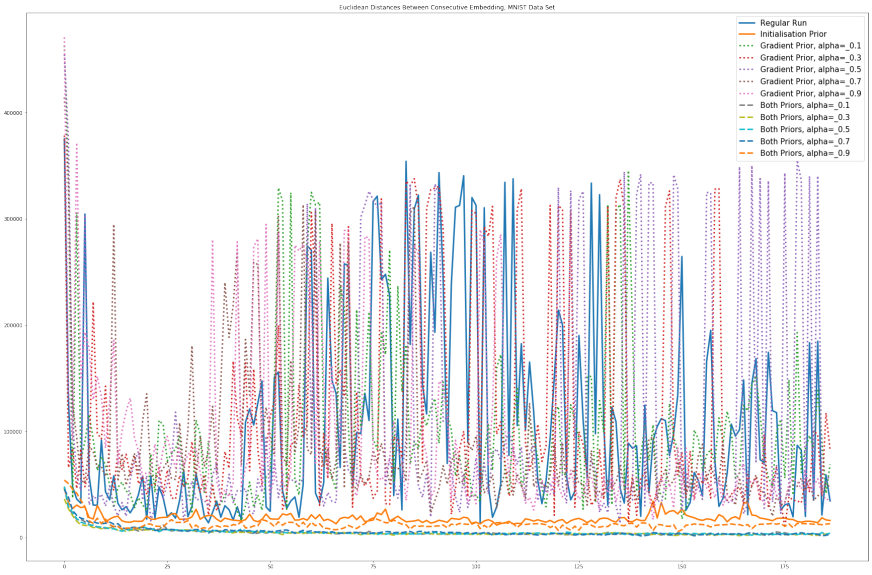
Wang, J. X.; Kurth-Nelson, Z.; Tirumala, D.; Soyer, H.; Leibo, J. Z.; Munos, R.; Blundell, C.; Kumaran, D.; and Botvinick, M. 2016. Learning to reinforcement learn.

Yuhuai Wu, Elman Mansimov, S. L. A. R. J. S. 2017. OpenAI Baselines: ACKTR & A2C. `https://openai.com/blog/baselines-acktr-a2c/`. [Online; accessed 26-October-2019].

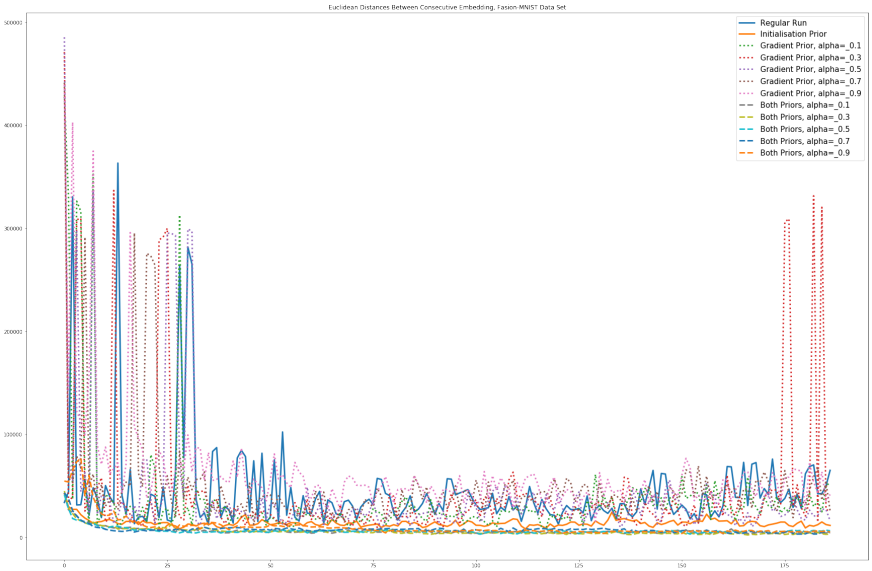
Zahavy, T.; Ben-Zrihem, N.; and Mannor, S. 2016. Graying the black box: Understanding dqns. In *International Conference on Machine Learning*, 1899–1908.

Zeiler, M. D., and Fergus, R. 2014. Visualizing and understanding convolutional networks. In *European conference on computer vision*, 818–833. Springer.

Appendix I: Sum of euclidean distances between consecutive embedding



(a) MNIST data set



(b) Fashion-MNIST data set

Figure 7: This figure shows the sum of Euclidean distances between the embedding of each point in two consecutive runs, for the different t-SNE algorithms. This measure serves as an indicator to the algorithms' consistency

Appendix II: Gradient Derivation

The derivation of the gradient, for our modified algorithm, is almost identical to that of the classic t-SNE algorithm. In this appendix we'll demonstrate why the same calculation can be applied to the modified algorithm, and show each step of the gradient derivation, to prove that none of the assumptions of the original calculation breaks. Our modified t-SNE algorithm minimizes in each iteratiion n the Kullback-Leibler divergence between the low dimensional joint probabilities $q_{ij}^(n)$ and a combined term of the high-dimensional joint probabilities p_{ij} and the low dimensional joint probabilities of the previous iteration $q_{ij}^{(n-1)}$. Note that the values of p_{ij} depend solely on the data and therefore do not change between iterations.

The values of p_{ij} are defined as the symmetrized conditional probabilities of the data points:

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2}$$

where $p_{j|i}$ is the conditional probability of the data point x_j relative to x_j:

$$p_{j|i} = \frac{\exp\left(-||x_i - x_j||^2/2\sigma_i^2\right)}{\sum_{k\neq i}\exp(-||x_i - x_k||^2/2\sigma_i^2)}$$

The values of $q_{ij}^{(n)}$ are obtained from the low dimensional embedding of the data $Y^{(n)}$, by means of a Student-t distribution with one degree of freedom:

$$q_{ij}^{(n)} = \frac{(1 + ||y_i^{(n)} - y_j^{(n)}||^2)^{-1}}{\sum_{k\neq l}(1 + ||y_k^{(n)} - y_l^{(n)}||^2)^{-1}}$$

The values of p_{ii} and $q_{ii}^{(n)}$ are arbitrarily set to zero $\forall n, i$, since we are only interested in modeling pairwise similarities.

The cost function is defined to be:

$$C^* = KL(\alpha Q^{(n-1)} + (1-\alpha)P||Q^{(n)})$$

where α is a user-defined constant, such that $\alpha \in [0, 1]$.

To make the calculation less cluttered, we define $B^{(n)}$ as the set of $b_{ij}^{(n)}$ such that

$$b_{ij}^{(n)} = \alpha q_{ij}^{(n-1)} + (1-\alpha)p_{ij}$$

Note that:

$$\sum_i\sum_j b_{ij}^{(n)} = \sum_i\sum_j[\alpha q_{ij}^{(n-1)} + (1-\alpha)p_{ij}] = \alpha\sum_i\sum_j q_{ij}^{(n-1)} + (1-\alpha)\sum_i\sum_j p_{ij} = \alpha + (1-\alpha) = 1$$

and in the same way, $b_{ii}^{(n)} = 0, \forall i, n$. Also, $B^{(n)}$ is a constant when calculating the gradient for the n-th iteration, since it is completely defined in the $n-1$ iteration. From this point forward, all the terms have the $^{(n)}$ notation, unless specifically stated, and therefore we'll omit using it.

The gradient of the cost function, relative to y_i is:

$$\frac{\delta}{\delta y_i}C^* = \frac{\delta}{\delta y_i}KL(||Q) = \frac{\delta}{\delta y_i}\sum_k\sum_l b_{kl}\log(\frac{b_{kl}}{q_{kl}}) = \frac{\delta}{\delta y_i}\sum_k\sum_l b_{kl}\log(b_{kl}) - \frac{\delta}{\delta y_i}\sum_k\sum_l b_{kl}\log(q_{kl})$$

and since we established that b_{ij} is a constant relative to y_i, we get:

$$\frac{\delta}{\delta y_i}C^* = -\frac{\delta}{\delta y_i}\sum_k\sum_l b_{kl}\log(q_{kl})$$

In order to make the derivation less cluttered, we define two auxiliary variables d_{ij} and Z as follows:

$$d_{ij} = ||y_i - y_j||$$

$$Z = \sum_{k\neq l}(1 + d_{kl}^2)^{-1}, \text{ and therefore:}$$

$$q_{ij} = \frac{(1 + d_{ij}^2)^{-1}}{Z}$$

Note that if y_i changes, the only pairwise distances that change are d_{ij} and d_{ji} for $\forall j$. Hence, using the chain rule, the gradient of the cost function C^* with respect to y_i is given by:

$$\frac{\delta}{\delta y_i}C^* = \frac{\delta}{\delta d_{ij}}C^* \cdot \frac{\delta d_{ij}}{\delta y_i} = \sum_j (\frac{\delta}{\delta d_{ij}}C^* + \frac{\delta}{\delta d_{ji}}C^*)\frac{\delta d_{ij}}{\delta y_i}$$

The last term is easily obtained by:

$$\frac{\delta d_{ij}}{\delta y_i} = \frac{\delta}{\delta y_i}||y_i - y_j|| = \frac{y_i - y_j}{||y_i - y_j||} = \frac{\delta d_{ji}}{\delta y_i}$$

And because of symmetry we can write:

$$\frac{\delta}{\delta y_i}C^* = 2\sum_j \frac{\delta}{\delta d_{ij}}C^* \cdot \frac{y_i - y_j}{||y_i - y_j||}$$

$\frac{\delta}{\delta d_{ij}}C^*$ can be obtained using our auxiliary variables:

$$\begin{aligned}\frac{\delta}{\delta d_{ij}}C^* &= -\frac{\delta}{\delta d_{ij}}\sum_{k\neq l} b_{kl}\log q_{kl} = -\sum_{k\neq l} b_{kl}\frac{\delta}{\delta d_{ij}}\log q_{kl} = -\sum_{k\neq l} b_{kl}\frac{\delta}{\delta d_{ij}}\log(\frac{q_{kl}Z}{Z}) \\ &= -\sum_{k\neq l} b_{kl}\frac{\delta}{\delta d_{ij}}\log(q_{kl}Z) + \sum_{k\neq l} b_{kl}\frac{\delta}{\delta d_{ij}}\log(Z)\end{aligned}$$

First we calculate $\frac{\delta}{\delta d_{ij}}\log(q_{kl}Z) = \frac{1}{q_{kl}Z}\frac{\delta}{\delta d_{ij}}(1+d_{kl}^2)^{-1}$. It is important to mention that this derivative is only nonzero when $k = i$ and $l = j$, and in this case:

$$\begin{aligned}\frac{\delta}{\delta d_{ij}}\log(q_{ij}Z) &= \frac{1}{q_{ij}Z}\frac{\delta}{\delta d_{ij}}(q_{ij}Z) = \frac{1}{q_{ij}Z}\frac{\delta}{\delta d_{ij}}(1+d_{ij}^2)^{-1} = -\frac{1}{q_{ij}Z}(1+d_{ij}^2)^{-2}\cdot 2d_{ij} \\ &= -\frac{1}{(1+d_{ij}^2)^{-1}}(1+d_{ij}^2)^{-2}\cdot 2d_{ij} = -(1+d_{ij}^2)^{-1}\cdot 2d_{ij}\end{aligned}$$

and the derivative of the second term:

$$\frac{\delta}{\delta d_{ij}}\log(Z) = \frac{1}{Z}\frac{\delta}{\delta d_{ij}}Z = \frac{1}{Z}\frac{\delta}{\delta d_{ij}}\sum_{k\neq l}(1+d_{kl}^2)^{-1} = \frac{1}{Z}\frac{\delta}{\delta d_{ij}}(1+d_{ij}^2)^{-1} = -\frac{1}{Z}\cdot 2d_{ij}(1+d_{ij}^2)^{-2}$$

Back to the derivative of the cost function:

$$\begin{aligned}\frac{\delta}{\delta d_{ij}}C^* &= -\sum_{k\neq l} b_{kl}\frac{\delta}{\delta d_{ij}}\log(q_{kl}Z) + \sum_{k\neq l} b_{kl}\frac{\delta}{\delta d_{ij}}\log(Z) = -b_{ij}\frac{\delta}{\delta d_{ij}}\log(q_{ij}Z) + \sum_{k\neq l} b_{kl}\frac{\delta}{\delta d_{ij}}\log(Z) \\ &= b_{ij}(1+d_{ij}^2)^{-1}\cdot 2d_{ij} - \sum_{k\neq l} b_{kl}\frac{1}{Z}2d_{ij}(1+d_{ij}^2)^{-2} = b_{ij}(1+d_{ij}^2)^{-1}\cdot 2d_{ij} - \frac{1}{Z}2d_{ij}(1+d_{ij}^2)^{-2} \\ &= b_{ij}(1+d_{ij}^2)^{-1}\cdot 2d_{ij} - 2d_{ij}q_{ij}(1+d_{ij}^2)^{-1} = 2d_{ij}(1+d_{ij}^2)^{-1}(b_{ij} - q_{ij})\end{aligned}$$

Finally we get:

$$\begin{aligned}\frac{\delta}{\delta y_i}C^* &= 2\sum_j \frac{\delta}{\delta d_{ij}}C^*\frac{(y_i - y_j)}{d_{ij}} = 2\sum_j 2d_{ij}(1+d_{ij}^2)^{-1}(b_{ij} - q_{ij})\frac{(y_i - y_j)}{d_{ij}} = 4\sum_j (1+d_{ij}^2)^{-1}(b_{ij} - q_{ij})(y_i - y_j) \\ &= 4\sum_j (1+d_{ij}^2)^{-1}(\alpha q_{ij}^{(n-1)} + (1-\alpha)p_{ij} - q_{ij}^{(n)})(y_i - y_j)\end{aligned}$$