Space Invaders World Models Implementation

Even Tang (z5165320), Ian Park (z5163307), Justin Or (z5163950)

Abstract

Over the recent years, reinforcement learning (RL) has been a popular field of research due to its potent, versatile real-world applicability in decision-making and motor-control. Historically, however, its progress has been hampered by the lack of standardised measure to test and compare different RL algorithms. To alleviate this problem, OpenAI gym provides a set of benchmarks in the form of open-source RL environments ranging from 3D robot control to playing Atari 2600 games. In particular, Atari games challenge the researchers and their RL algorithms with its high-dimensional video input of size 210 x 160 at 60Hz frequency, which paves a meaningful step towards decision-making applications in the real-world where inputs are high-dimensional, such as self-driving vehicles.

As apparent on the leaderboard [https://www.endtoend.ai/envs/gym/atari/], the current state-of-the-art RL methods for the atari games are Deep Q-Networks (DQNs) and its variations, and thus it is the current focus of the research. However, in this paper, we present another promising direction of research that is based on the “Recurrent World Models” (WM) framework developed by Ha et al. [https://worldmodels.github.io/] and we will apply this idea to the classic arcade game Space Invaders. Here is a link for those who are not familiar with the game: https://en.wikipedia.org/wiki/Space_Invaders.

There has been numerous attempts to model the environment of the agent in reinforcement learning, but WM builds upon the existing ideas to produce a framework that consists of three components each with distinct roles: V Model to model the observations as an encoded latent vector of significantly smaller size, an M Model to interpret the phenomena of time in this encoded environment, and finally a simple C Model agent which tries to play the game from this latent space-time representation of the game. C is not trained with DQN, but rather with an evolutionary algorithm CMA-ES which is known to excel in problems with decently small number of weights. An extension to this framework has been suggested by the author, known as “dreaming”, where the generative power of V and the predictive power of M are combined to simulate the real environment, and the C model is trained within this “dream” environment instead.

The authors of the paper evaluated their World Model algorithm on two arcade environments, namely “Car Racing” and “Doom: Take Cover”. Their agent achieved state-of-the-art results for “Car Racing” and successfully solved the designated task of surviving for 750 steps in “Doom” by a large margin of 350 steps. However, we hypothesise that their RL framework may not be generalise well to different game environments for one particular reason: VAE is an unsupervised algorithm that is oblivious of which features are relevant or useful to the overall task of the game, and so it may remove or obscure important features of the game that is essential to the controller’s decision making and thus its performance. This idea is supported by the author who mentions that in the Doom environment, the VAE has failed to replicate the number of enemies in the input image.

In particular, Atari 2600 Space Invaders exhibits interesting properties that differentiates it from the two arcade environments used by the authors. As shown in the figure right, the game begins with a 6 x 6 grid of aliens that moves across and down the game screen, and at the bottom of the screen is a player's ship, which can move horizontally and shoot lasers vertically at the aliens.

Although aliens stay fixed in their grid positions throughout the game, some of the aliens may be taken out of the grid by the player as the game proceeds, and thus the grid can be in any of $2^{36} = 69$billion binary configurations. For a framework that has been shown to fail at replicating about at most a dozen of enemies, representing exact configuration of all 36 aliens in a grid could be a great challenge.

Figure 0.1: Initial screen of the game Space-Invaders

Another distinguishing feature of Space Invaders is the fact that the bullets are extremely under-emphasised. In Doom, the bullets can cover more than half of the game screen, whereas in Space Invaders, the bullets are only a pixel-wide and a few pixels long. To make the situation worse, bullets can exist at almost any point on the game frame, and they frequently *shrink and disappear* in some frames and reappear in the later frames. Combined with the fact that autoencoders are widely used for noise removal [https://dkopczyk.quantee.co.uk/dae-part2/], the above points suggest that VAE may be inclined to treat the bullets as noise, and thus fail to encode bullets into its latent vectors. This is catastrophic for the learning framework as bullets are clearly crucial elements of the gameplay, and failure to encode them would be analogous to making the controller blindfolded, thus degrading its performance.

This is highly relevant problem as approximately half of the Atari 2600 environments available at OpenAI Gym such as Assault, Jamesbond, and Phoenix display similar characteristics and it poses generalisation issues if we are considering real-world applications where some under-emphasised elements in the environment are important to the task. Furthermore, there is an informally reported instance of the issue on Reddit (https://www.reddit.com/r/MachineLearning/comments/aaw4m4/d_making_autoencoders_care_about_small_details/) for Atari 2600 Breakout, concerning the issue of VAE not being able to represent the bouncing ball in the game.

Therefore, in this project, we aim to:

1. Investigate how well the World Model framework generalises to Space Invaders;
2. Extend the original framework to alleviate its aforementioned weaknesses
3. Develop an agent that can exceed the median human performance of 1652 points [https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf].

Related Work

The Mixture Density Network combined with a Recurrent Neural Network in the M model has been applied before to many sequential problems involving generating unseen information such as generating handwriting (https://arxiv.org/abs/1308.0850) to predicting pen strokes of a sketch (https://arxiv.org/abs/1704.03477).

The Architecture

The World Models architecture is inspired by how a human-being perceives and learns in an environment. Just like how humans have an internal representation of the environment around them, this structure aims to train the agent using its own internal representation of time and space. The structure utilises three main components:

Variational Auto-Encoder - (V model)

Given the large amount of information in the form of high-dimensional colour video input, the Variational Autoencoder (VAE) learns a latent compressed representation of the input, such that from this latent representation it can reproduce the original input as accurately as possible.

Mixture Density Gaussian Network with Recurrent Neural Network - (M model)

Similarly, the Mixture Density Network (MDN) and Recurrent Neural Network(RNN) aims to produce a compressed representation of time in the environment, just like the VAE. This is accomplished by the RNN's ability to model and learn spatial and temporal representations of data. Furthermore, due to the stochastic nature of the environment, instead of merely predicting the next latent vector z_{t+1} itself, the MDN seeks to estimate the probability distribution of the next latent vector z_{t+1}, $P(z_{t+1}|z_t, a_t, h_t)$, where a_t is the action at the previous time step, and h_t is the previous hidden state of the Long Short Term Memory (LSTM) RNN cell. By sampling from the MDN, the M model will encode the stochastic nature of the environment, rather than provide a deterministic representation.

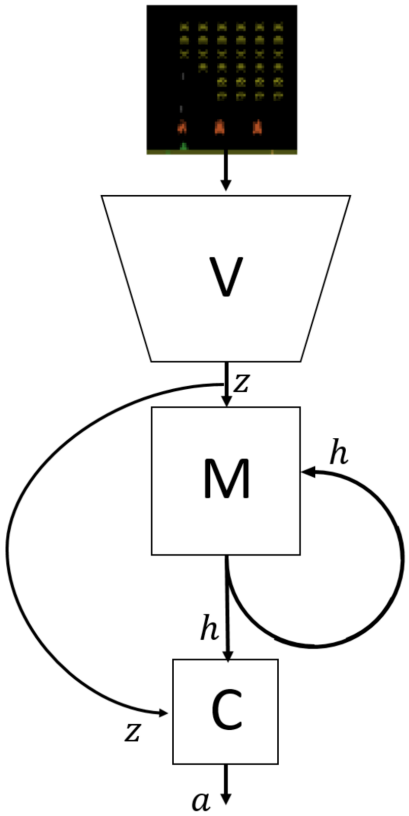


Figure 0.2: The overall architecture

Controller (C model)

The complexity of other components of the architecture de-couples some of the complex time-dependent features which allows for the controller to be rather simple in comparison. We have chosen for a simple linear model for the actual agent, then defined its input to be the latent vector z_t and the hidden unit states of the MDN-RNN h_t to predict the action at a particular time step a_t. With weight matrix w and bias b parameterising the model, each action at time step t is determined by the equation

$$a_t = w[z_t, h_t] + b$$

However, this simple linear model is learned by employing an evolutionary algorithm known as CMA-ES, details of which can be found here: https://en.wikipedia.org/wiki/CMA-ES.

Implementation

In this section, we discuss the final implementation of the learning algorithm we used to solve the task of “obtaining high scores in Space Invaders” [https://gym.openai.com/envs/SpaceInvaders-v0/]. In the sections that follow, we discuss the experiments that led to this particular implementation.

The learning algorithm is conducted in steps, where we train each of the three components VAE, MDN-RNN and CMA-ES in sequence with some data preprocessing in between.

1. Training Data Collection and Selection

In order to train the VAE and the MDN-RNN, we first collected some training data by recording the gameplay of a random agent, whose actions are randomly selected from one of the six actions: noop, shoot, move right, move left, shoot and move right, shoot and move left. For each round/episode of the game, we recorded the following set of observations at every time step of the episode:

1. The game screen as a 210 x 160 x 3 tensor (height x width x RGB)
2. The action taken by the random agent at that frame

However, because the random agent often performed so poorly, the majority of the observations were taken during the animations of the agent dying, each of which persisted for about 80 time steps. Hence, we discarded the episodes if either the agent did not survive for longer than 600 time steps or if the agent did not progress far enough to see a commander ship flying across the screen. Using this selection procedure, we have collected 10,000 episodes worth of observations.

2. Downsampling and Preprocessing

The raw observation of the game screen has the dimensions 210 x 160 x 3, which is not suitable for a convolutional autoencoder that expects the height and width to be equal. Thus, we cropped and downsampled each frame into a 64 x 64 square, while preserving the three colour channels.

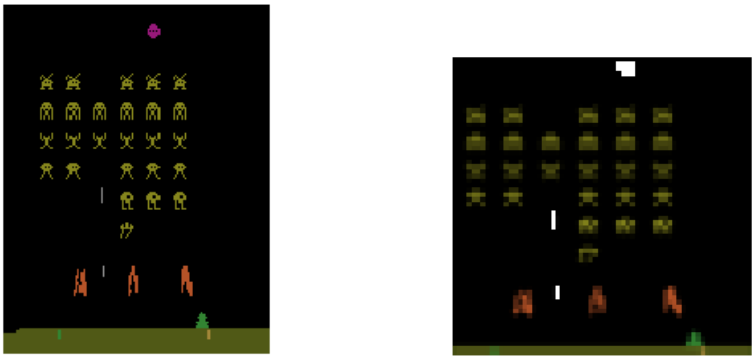


Figure 1.1. (Left) A raw observation that features all elements of the game.
(Right) The downsampled version of the image with emphasis on the bullets and the commander ship.

As it can be seen in the figure, further preprocessing has been applied to the images where bullets and commander ships are highlighted white, as this has been shown to improve the VAE's performance on reconstructing these core game elements.

3. Training VAE

The convolutional VAE is fed in the preprocessed images of size 64 x 64 x 3 and is trained to reconstruct the same image that it was given. However, as shown in the figure below, the VAE is constrained by a hidden dense layer "z" of size 64, which beneficially acts as a bottleneck that forces the network to learn a succinct, latent vector representation of the images in the training set.

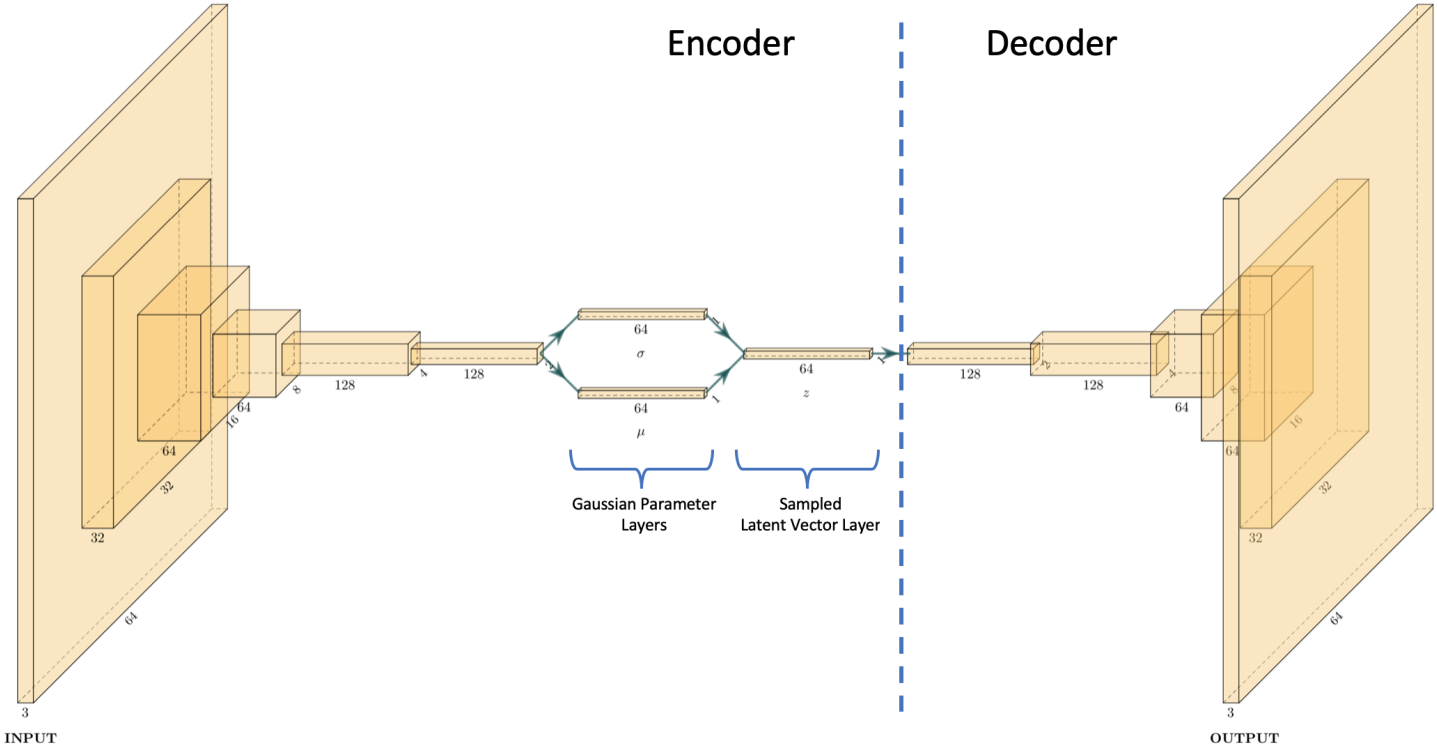


Figure 1.2: The layers view of the convolutional VAE. Both the encoder and the decoder components consist of 5 convolutional layers, each of which uses stride 2 and filter size 3.

For each convolutional layer of the model, we employed he-uniform initialiser with relu activation, whereas the dense layers used glorot-uniform initialisers with no activation.

We used Adam optimiser to minimise the loss function, which is a weighted sum of the reconstruction loss $R(\hat{Y})$ and the ELBO (Evidence Lower BOund) regularisation $ELBO(\theta)$:

$$L_\theta(\hat{Y}) = 8R(\hat{Y}) + \mathrm{ELBO}(\theta)$$

In the above formula, $\hat{Y}$ is a 64 x 64 x 3 tensor representing the image reconstructed by the model and $ELBO(\theta)$ approximates the KL-divergence loss of the model parameterised by θ with respect to the standard normal distribution $N(0,\ 1)$. In the implementation, θ corresponds to the set of weights of the dense layers μ and σ. In short, KL-divergence loss is higher when the Gaussian distributions represented by the two dense layers are further away from the standard normal distribution. The weighting 8 in front of the reconstruction loss has been selected through hyperparameter tuning.

On the other hand, the reconstruction loss is given by:

$$R(\hat{Y}) = \sum_{\hat{y} \in \hat{Y}} (y - \hat{y})^2 \cdot e^{(y-\bar{y})^2}$$

where $\hat{y}$ represents an individual pixel value of the reconstructed image, $y$the corresponding pixel value of the original input image, and $\bar{y}$ is the average pixel value over the training set, but instead approximated by the average of the batch. Further discussion about this handcrafted loss function can be found in 'Experiments' section.

The training was conducted for 100 epochs where each epoch used the data from 500 randomly selected episodes. In order to assess the model's ability to generalise to unseen input images, we reserved 10% of the training set as a validation set.

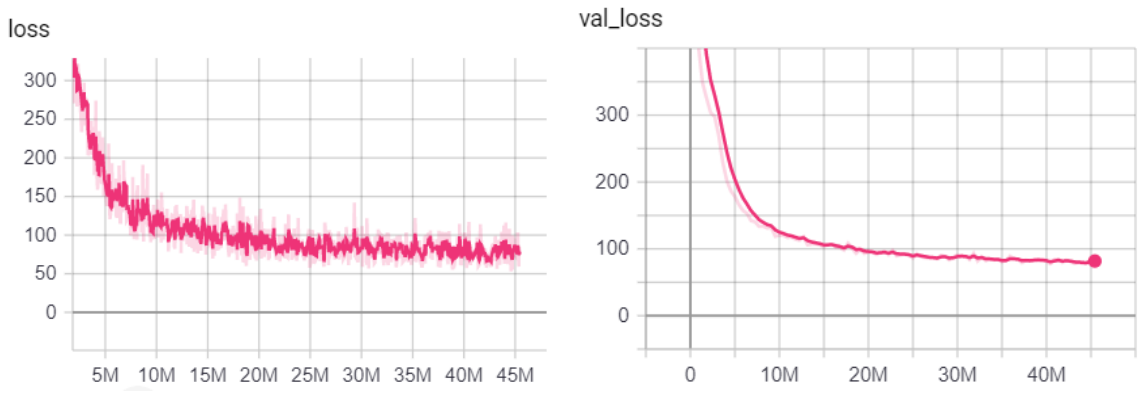


Figure 1.3: Graph of the training loss (left) and the validation loss (right) over the training session.

We did not train further even though the graph of the loss indicated some room for further convergence because the random agent clearly had not explored all the possible states of the game, and so we wanted to maintain a satisfactory level of generalisation by stopping early.

Since VAE is an unsupervised algorithm, the exact value of the loss is not useful for analysing how well it encodes the input images. Thus, we generated a new set of images which we fed into the trained model and we assessed the quality of the VAE by manually comparing the reconstructed images to their original.

Comparison of Original vs Reconstructed Images

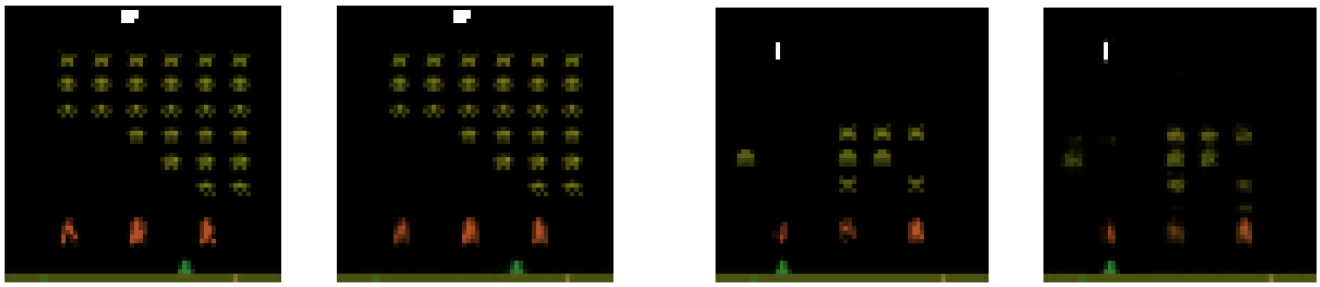
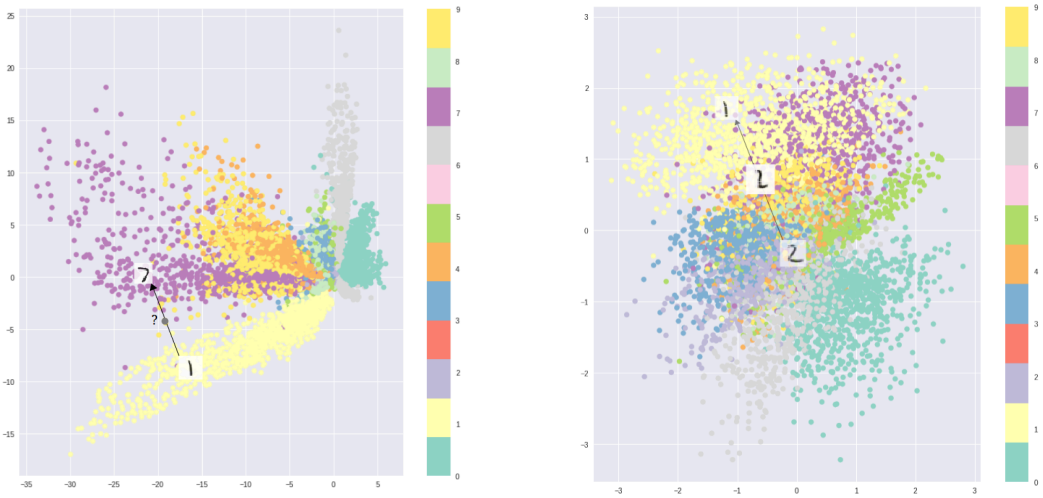


Figure 1.4: (Left) Near-perfect reconstruction of an early-game observation with a commander ship at the top of the screen. (Right) Late-game observation with a bullet and few aliens left; this illustrates the degrading quality of reconstruction as the game progresses, but the quality is still (arguably) satisfactory by the human eyes.

A core property of variational autoencoders that differentiates them from the regular autoencoders is their capability to generate new images. This quality is characterised by the shape of the cluster formed by plotting the “true” latent vectors produced from the real images, as shown in the figure below.

2D projected latent spaces of hand-written digits from the MNIST dataset



Source: https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf

Figure 1.5

(Left) The latent space produced by a “regular” autoencoder. Interpolating between two digits in the latent space will result in an unrealistic image.

(Right) The latent space produced by a variational autoencoder. The VAE incorporates Gaussian assumptions on the data and KL-divergence loss to encourage homogeneity and zero-centredness of the space.

The homogeneous latent space learned by the VAE means when you perturb a true latent vector by some small amount, then the reconstructed image will still look realistic. This essentially allows for smooth transition between the true latent vectors, which in turn allows the RNN to make predictions more easily because consecutive frames of the game are engineered to be close to each other in the latent space. Thus, to analyse the generative quality of our VAE, we carried out the following steps:

1. Randomly select 5 episodes from a test set
2. For each image from the selected episodes:
    a. Feed the image into the encoder part of the VAE and obtain its output, which is in the form of two vectors of size 64 that represent the means and variances of 64 independent Gaussian distributions.
    b. Sample 5 latent vectors of size 64 from the Gaussian distributions. We sampled from the distributions rather than plotting the mean values so that the variances of the distributions could be captured by our plot.
3. Concatenate the latent vectors generated from all 5 episodes into a single batch and perform a PCA dimensionality reduction on that batch to plot the latent vectors onto a 2D plane, as follows:

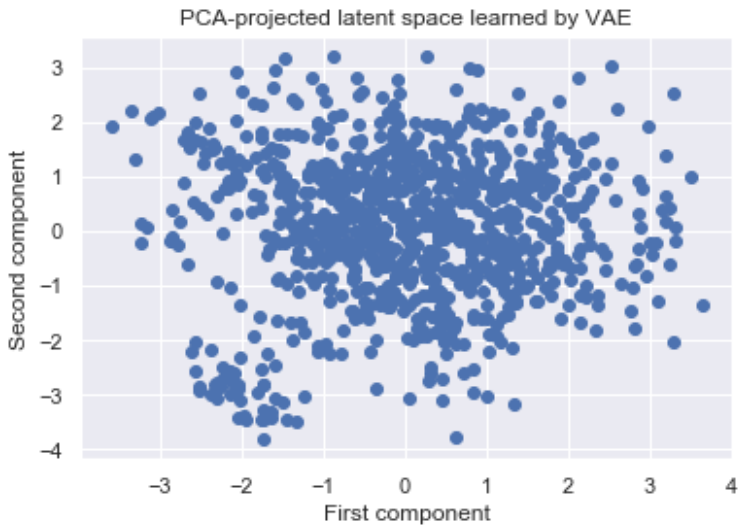


Figure 1.6. Plot of the latent space learned by the trained model. Note that only 2% of the points have been randomly selected for display in order to reduce clutter and to illustrate the spread of the latent vectors in the space.

4. Training MDN-RNN

After collecting the input from the VAE, $(\mu,\ \log(\sigma^2))$, factorised Gaussian vectors mu and log-variance, we sample the latent vector z_t from these parameters. This is required to model the stochasticity of the environment and improve explorability. The input is then fed into the MDN-RNN, implemented with 512-layer LSTM into the MDN layer. The MDN layer is effectively a Dense fully-connected layer of size $n_{mixtures} * 3 * (n_{latent}) + n_{categorical}$, where the number of mixtures is a hyperparameter defining the number of Gaussian distributions used to model the latent variable as a mixture of these distributions. Each distribution in the mixture is represented by 3 variables $(\mu,\ \sigma, \pi)$ for the mean, standard deviation and mixing coefficient respectively. Additionally for the dreaming extension, we may also predict some categorical variables such as the type of action taken at the next time step, hence the reason for the extra $n_{categorical}$ parameters.

5. Training Controller

To train the final component of the framework, we employed the evolutionary technique known as the Covariance Matrix adaptation evolution strategy (CMA-ES). At the start of each evolution, we generate or mutate a certain number of performers. Then, a fitness level of each performer is determined by letting them play 3 rounds of the game and taking the average of the final score acquired by the performer. At each frame of the game, however, the performer decides its next action based on the latent vector from the VAE and the hidden state from the RNN. The relative fitness scores are then used to determine the probability of reproduction and survival for the next stage of evolution - better performers reproduce with some mutation and recombination, whereas worse performers are likely to go extinct.

The results of this implementation are discussed fully in the ‘CMA-ES Experiments’ section.

VAE Experiments

We began our project by implementing the exact convolutional variational autoencoder described in the original authors’ paper, assuming Keras’ default kernel initialiser ‘glorot uniform’ and relu activations for the convolutional layers. The loss function employed was (squared) L_2 distance for reconstruction and ELBO loss as regularisation. In training, we used the same methods as in the final implementation, using 10,000 episodes (but without discarding any episodes with poorly performing agents) worth of training set for 100 epochs, and 500 steps per epoch. This training scheme was employed for all subsequent experiments as well since it allowed for sufficient level of convergence.

To assess the model’s performance, we supplied a random selection of input images from a separately generated test set and visualised the model’s output against the original images. As shown below, the results clearly demonstrated the model’s failure to capture the bullets as we hypothesised.

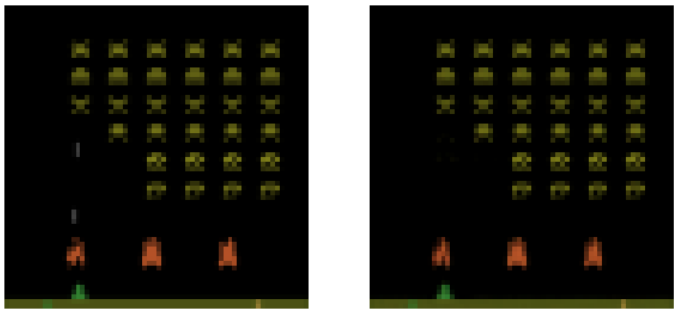


Figure 1.7. The input image (left) vs the reconstructed image (right).

In order to alleviate this issue, we have explored numerous modifications to the original VAE architecture, few of which have been summarised below.

Downsampling method

Two different types of resizing methods have been explored: “blocky” and “blurry”, where they share similarities to max-pooling and average-pooling mappings in convolutional neural networks. Blocky resizing produced clear and sharp images that may be more human-friendly, yet blurry resizing allowed for smoother images that we could expect the network to fit with more ease.

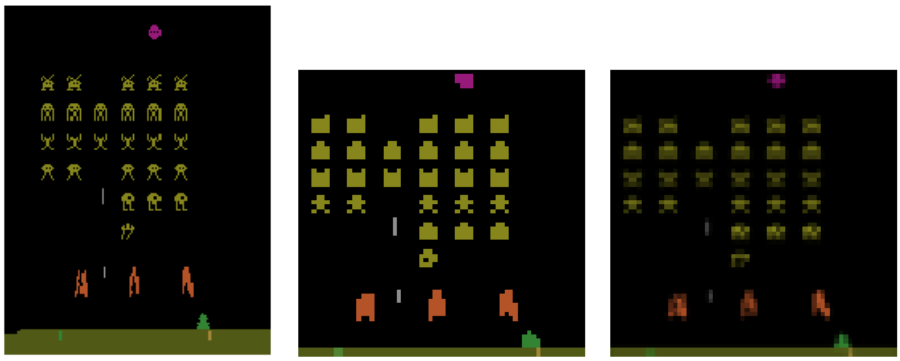


Figure 1.8. (Left) original image, (middle) blocky resized image, (right) blurry resized image.

Moreover, in an attempt to reduce the number of parameters in the VAE network and to help the network focus on reconstructing the bullets correctly rather than the colours of other game elements, we tried converting the images into grayscale, reducing the dimensions of each image from 64 x 64 x 3 to 64 x 64 x 1.

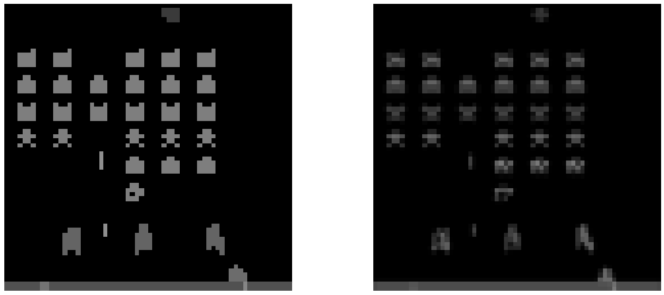


Figure 1.9. (Left) blocky grayscale image, (right) blurry grayscale image.

In most of the experiments that follow, we trained our model on all four different downsampling methods, and the results consistently showed best performance by the model trained using full-colour, blurry downsampled images; henceforth, the subsequent images will only feature this particular resizing method.

Cross-Entropy Loss

As Altosaar, J. explains in https://jaan.io/what-is-variational-autoencoder-vae-tutorial/, cross-entropy serves as a powerful reconstruction loss function for VAEs due to their strong theoretical basis. From a probability theory viewpoint, a variational autoencoder consists of two probability distributions: the encoder $q_\theta(z \mid x)$ and the decoder $p_\phi(x_i \mid z)$, which are parametrised by the weights of the network θand ϕrespectively. Given an input image x_i, the latent vector z^* is sampled from the encoder distribution $q_\theta(z \mid x_i)$, which can then reconstructed as an image using the decoder distribution $p_\phi(x \mid z^*)$. Thus, for a datapoint x_i, the cross-entropy is the expected negative log-likelihood $-E_{z \approx q_\theta(z|x_i)}[\log p_\phi(x_i \mid z)]$, which essentially measures how much information is lost by compressing the high dimensional vector x_i into a lower dimensional latent space, which is in turn related to how well the VAE reconstructs the original image.

Motivated by its theoretical profoundness, we replaced L_2 distance by cross-entropy for the reconstruction loss, only to find out its disappointing performance:

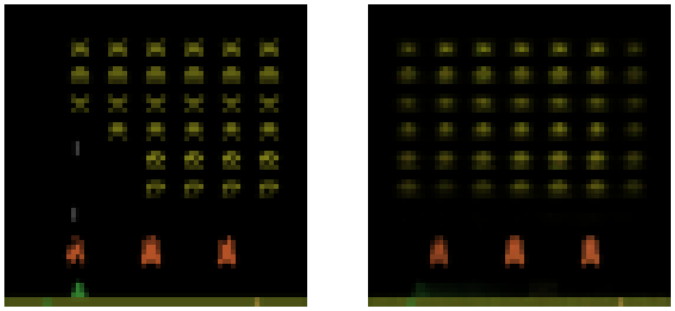


Figure 1.10. (Left) input image, (right) image reconstructed by cross-entropy VAE.
The aliens seem to be replicated across the screen in blobs.

Further research [https://arxiv.org/pdf/1708.08487.pdf] showed that this behaviour comes from the fact that cross-entropy loss is theoretically relevant mostly when each pixel is either black or white, or more specifically, when each pixel represents a binary random variable. We also experimented with L_1 distance and a weighted mixture of L_1 and L_2 distances, and they seemed to exhibit almost the opposite property where the loss tends to remove some aliens from the reconstructed image. Therefore, we continued to use the L_2 loss.

Bullet loss amplification by brightening its colour

In most of the instances where bullets vanished in the VAE's output, the bullets have been replaced by the black background. Thus, we specifically penalised the learning algorithm for ill-representing bullets by changing their colours from grey (#8E8E8E) to white (#FFFFFF), which in turn amplified the L_2 loss incurred by missing the bullets.

This method seemed to improve the quality of the outputs by the VAE, as it allowed the model to successfully encode bullets at least in the bottom-left quadrant of the screen. This behaviour can be explained by the fact that during the data collection stage, each episode started with the random agent on the bottom-left corner of the screen, and thus the distribution of the bullets were concentrated in that quadrant, especially for short-living agents. As this way of preprocessing displayed some promising results, we have decided to continue using these datasets for the subsequent experiments.

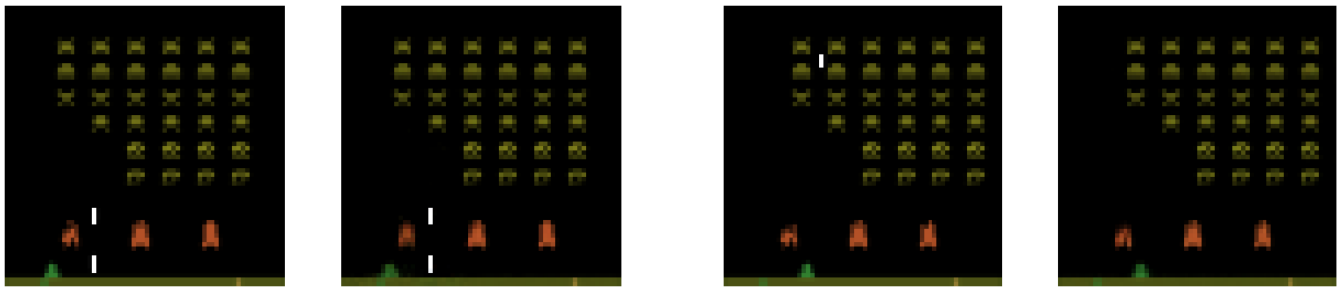


Figure 1.11. Input-output comparisons of bullet-emphasised VAE. (Left) The case where the bullets reside in the bottom-left quadrant. (Right) The case where the bullet lies outside the quadrant.

Extreme loss amplification for bullets

We extended the above method by adding extra loss specifically for misrepresenting the white pixels (i.e. bullets) in the image:

$$R(\hat{Y}) = \sum_{\hat{y} \in \hat{Y}} (y - \hat{y})^2 (1 + \lambda I[\text{y is a white pixel}])$$

where λ represents a scaling-hyperparameter.

Due to the usage of colour masking in the loss computation, the training time doubled many other experiments and the result was much below our expectation; it caused the model to focus too much on representing the bullets that other game elements became ill-represented. Another explanation could be the use of highly non-smooth loss function defined by the masking that caused difficulties in gradient descent.

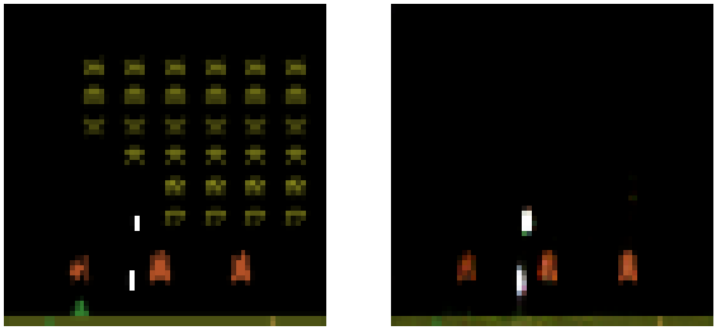


Figure 1.12. Input-output comparison of extra-loss-amplified VAE for $\lambda = 10$.

Double VAE

We developed a new preprocessor that separates every input image into two images of the same size as the original, where the first consisted of only the bullets and the second contained the remaining elements. Then, we trained two VAEs of different sizes, where the smaller network was trained to encode the bullets into a latent vector of length 24, and the larger network of latent size 64 was designed to encode the remaining elements.

However, depending on how much we amplified the reconstruction loss, the output of the bullet-encoding VAE was either completely black or an extremely noisy one as shown below.

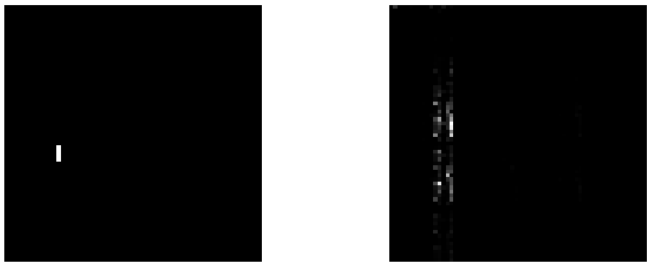


Figure 1.13. Input-output comparison of bullet-encoding VAE.

Deeper Convolutional Network with Smaller Filter Sizes

Inspired by the simplicity of the revolutionary VGGNet (2014), we decided to modify the network to be deeper but with smaller filter sizes of 3 x 3.

This new architecture uses only 654,339 parameters as opposed to 4.5M parameters of the original architecture. Yet, it surpassed the original architecture by a large margin as it successfully encoded nearly all (97 out of 100 when manually counted) of the bullets on the screen.

Hence, not only that it halved the training time of the model, it also allowed the model to solve the problem, but only with such a simple modification.

However, one more issue remained. That is, the commander ships were never being rendered on the reconstructed image, even with this new architecture. The explanation for this behaviour is similar to that for the bullets, except the commander ships are much rarer in the game; they appeared in roughly about 40% of the episodes in the training set and in those episodes, they appeared for only about 10% of the frames on average, thus leading to meagre 4% appearance rate.

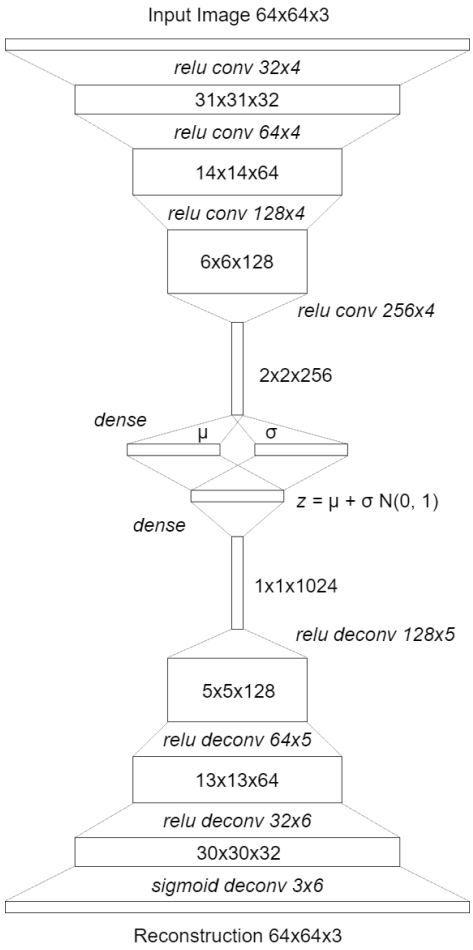


Figure 1.14. The architecture used by the original authors.

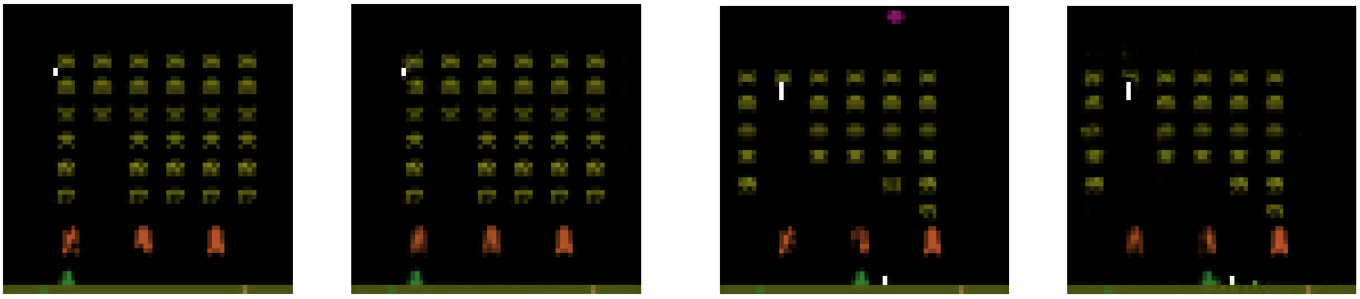


Figure 1.15. Input-output comparison of the new VAE. (Left) Near-perfect reconstruction of the original image including the miniature bullet near the top of the screen. (Right) The VAE is still not able to reconstruct the purple commander ship.

Surprise-Scaled MSE

The final experiment was the capping stone of all our discoveries throughout numerous experiments. In this experiment we produced a smooth loss function that amplified the loss for the pixels that are “surprising”, that

is, radically different to the average value of the pixel. As aforementioned in the ‘Implementation’ section, the new reconstruction loss is given by

$$R(\hat{Y}) = \sum_{\hat{y} \in \hat{Y}} (y - \hat{y})^2 \cdot e^{(y - \bar{y})^2}$$

Furthermore, we also improved the quality of our training set by whitening the commander ships (just like with bullets) and also discarding all episodes that did not contain a commandership. Essentially, this increased the appearance rate of the commander ships from 4% to 10%.

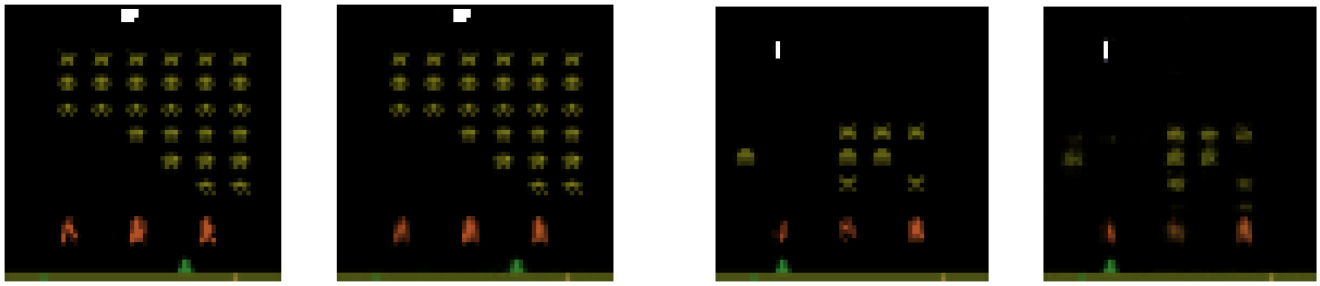


Figure 1.16. Input-output comparison of the final VAE. (Left) Near-perfect reconstruction of the original image involving the commander ship. (Right) Arguably decent reconstruction of the late-game observation.

RNN Experiments

Initially, we looked at existing keras implementations of a MDN for the M model, to match the choice of infrastructure for the other models and ease of coding generalised concepts. The inputs consisted of the latent vector z from the V Model, the action and the previous hidden state, and we used a Long Short Term Memory (LSTM) layer of size 256, initially without dropout. The MDN layer implemented on Keras provided both the loss function for training and the sampling function for prediction. The data was obtained from 10,000 random rollouts of the environment, which were carefully chosen to show commander ships, and reach a specified timestep. The data was processed with the VAE above to produce the μ and σ to sample our z_t latent vector from. We also tried batch learning by concatenating all games together and cutting the games into (Batch_size, Seq_length, Latent_dims) slices. It was here where we hit our first hurdle.

Use of Time Distributed Layer in Keras

At first we had used the Time Distributed Layer Wrapper provided by Keras to "(apply) a layer to every temporal slice of an input."(https://keras.io/layers/recurrent/) Furthermore, since SketchRNN, a recurrent network from which WM's MDN-RNN M Model is based on, was implemented with a time distributed layer, our initial thoughts were to use it for this implementation. However, after looking through the implementation and proposed architecture, we decided to remove the Time Distributed Layer and instead use a simple LSTM layer. This was almost no work, and the model trained just fine without the Time Distributed Layer. We found immediate improvements with the loss as soon as we made the change.

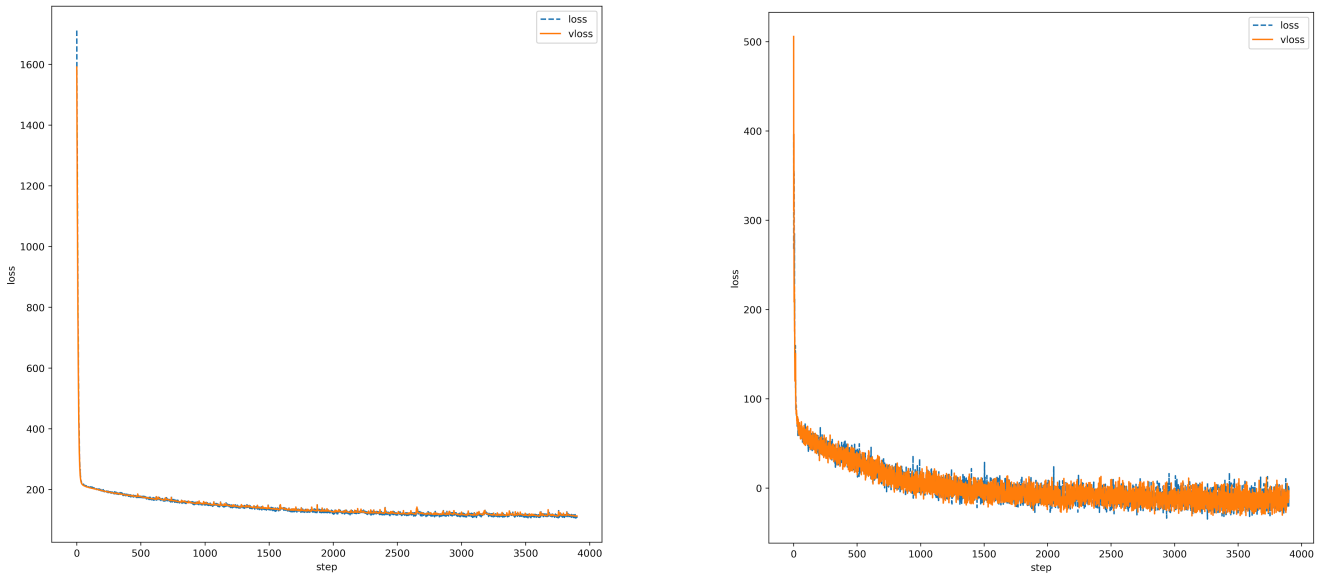


Figure 2.1: On the left, a model trained with the Time distributed layer plateaus at around 200 loss, on the right, the model trained with a simple LSTM instead, but faces another problem of negative loss.

Negative Loss

Following the change to the architecture, we were surprised to see the emergence of negative loss from the model. Because we were using the loss function from a package from Keras, we were unable to change the code to remedy this problem. Following some more research in the area online, it appeared that the Negative Logarithmic Likelihood loss from an MDN network can actually be negative.

$$L(\theta) = \sum_{i=1}^{N} - log(p(y_i|x_i, \theta))$$

Equation 2.2: NLL loss for discrete random variables y_i

Since the y_i's are discrete random variables, $p(y_i|x_i\theta) \leq 1 \; \forall\theta$, and the existence of a minimizer θ^* for $L(\theta)$is guaranteed. However, since the y_i's are continuous random variables in the Gaussian Mixture Density Network's case, there could be a possibility for the pdfs to be greater than 1 (as long sa it integrates to 1 with respect to y). This can be explained intuitively as having some of the mixture components collapse in a data point, making the loss decrease to arbitrary small values. (https://stats.stackexchange.com/questions/294567/negative-loss-while-training-gaussian-mixture-density-networks/381694)

To seek more control over this phenomenon we tried a few strategies:

- Dropout applied in the LSTM inputs. This also lead to more generalisable performance, but the decrease in performance was far too great.

- Change the loss function and architecture so we would have more control of how the loss was calculated. There may have been a problem with the keras package we used, and after changing the loss function we did in fact not encounter this phenomenon anymore.

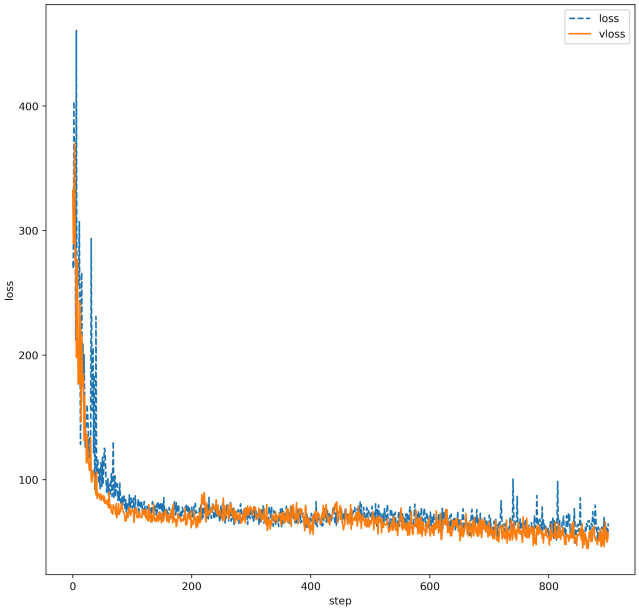


Figure 2.3: the effect of dropout on loss.

Choice of Inputs and Outputs

From the general architecture of WMs, the MDN-RNN takes (z_t, a_t, h_t) as input and produces (z_{t+1}, h_{t+1})at each timestep. To be precise, the MDN layer produces the Gaussian Mixture parameters to model the probability distribution of z_{t+1}.

$$P(z_{t+1}|z_t, a_t, h_t)$$

Equation 2.4: The probability distribution the M Model

However, for the overarching goal to create a dream environment, we also need to model extra parameters in order to use this M model itself as an environment.

- Predict d_{t+1}, whether the game is done at timestep t+1. This is needed as a game-ending condition for the dream environment. Since this is a boolean variable, the loss for this variable should not be NLL, and should be treated like a categorical variable. This was a shortfall of our current architecture, which required all the predicted variables to be mixtures. However, retrospectively speaking, we could have built a model specifically with this purpose using Keras' Functional API for more advanced Model architectures with multiple outputs [https://keras.io/getting-started/functional-api-guide/]
- Predict a_{t+1}, what action the agent will take at time step t+1, in order to do iterative learning. However, due to the training time, we decided not to pursue this idea.
- Predict r_{t+1}, what reward the agent will receive at time step t+1, for creating the dream environment. This is an environment specific requirement, as the examples from WM utilising dreams had rewards based on time alive (https://gym.openai.com/envs/DoomTakeCover-v0/). Since the time alive in Space Invaders does not directly correlate to score, we would require the dream environment to return the reward.

Ultimately, the final probability distribution the model was approximating was

$$P(z_{t+1}, a_{t+1}, r_{t+1}, d_{t+1}|z_t, a_t, h_t).$$

Equation 2.5: desired model

New Loss Function

Due to our requirements to have further control of the loss function, especially now that we want to predict more variables, we needed to separate the output dimension into different sections to calculate loss differently. In particular, we would use only one neuron to model d_{t+1}as it was a boolean discrete variable. The others were continuous so they could be modelled more precisely with the mixtures model. Thus, we separated the loss for each of these outputs, and made actual loss calculated a weighted sum of these losses. At first we equally weighted these different losses.

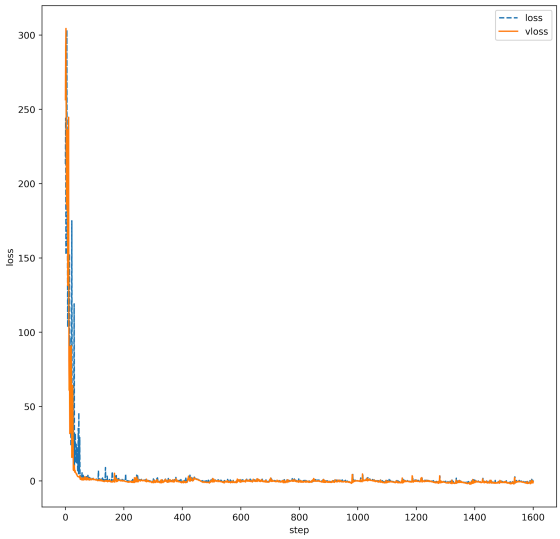
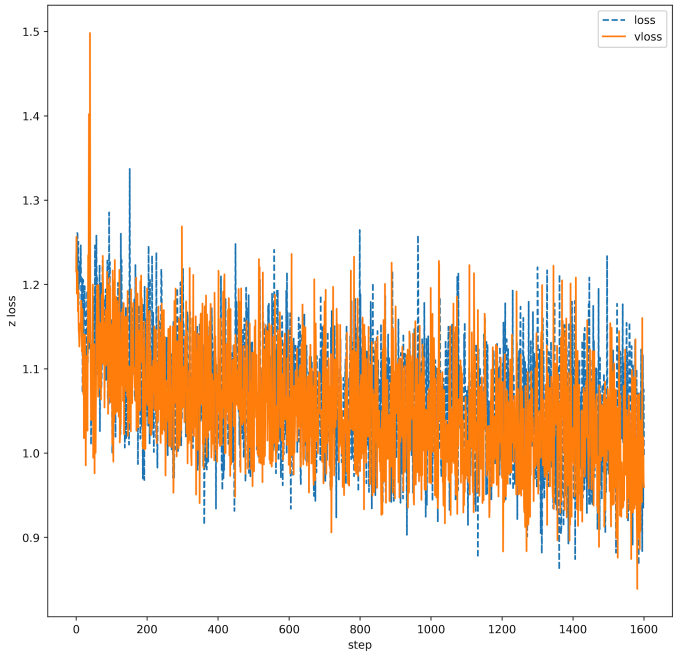
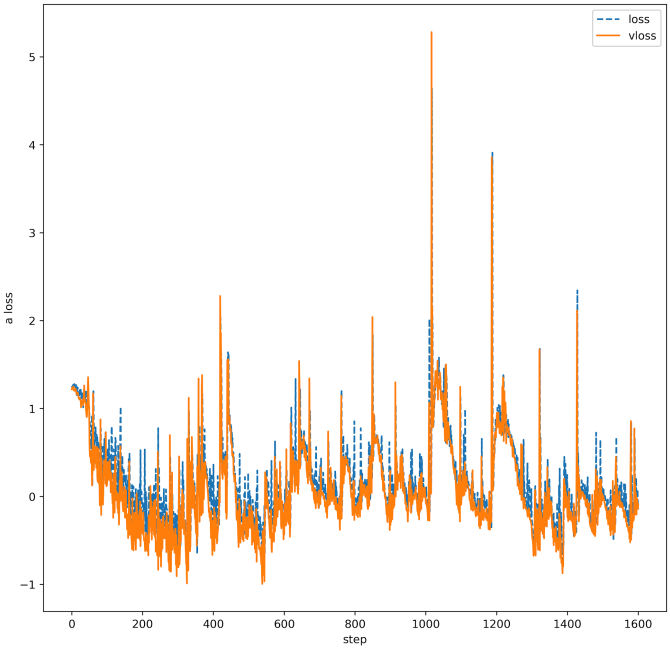
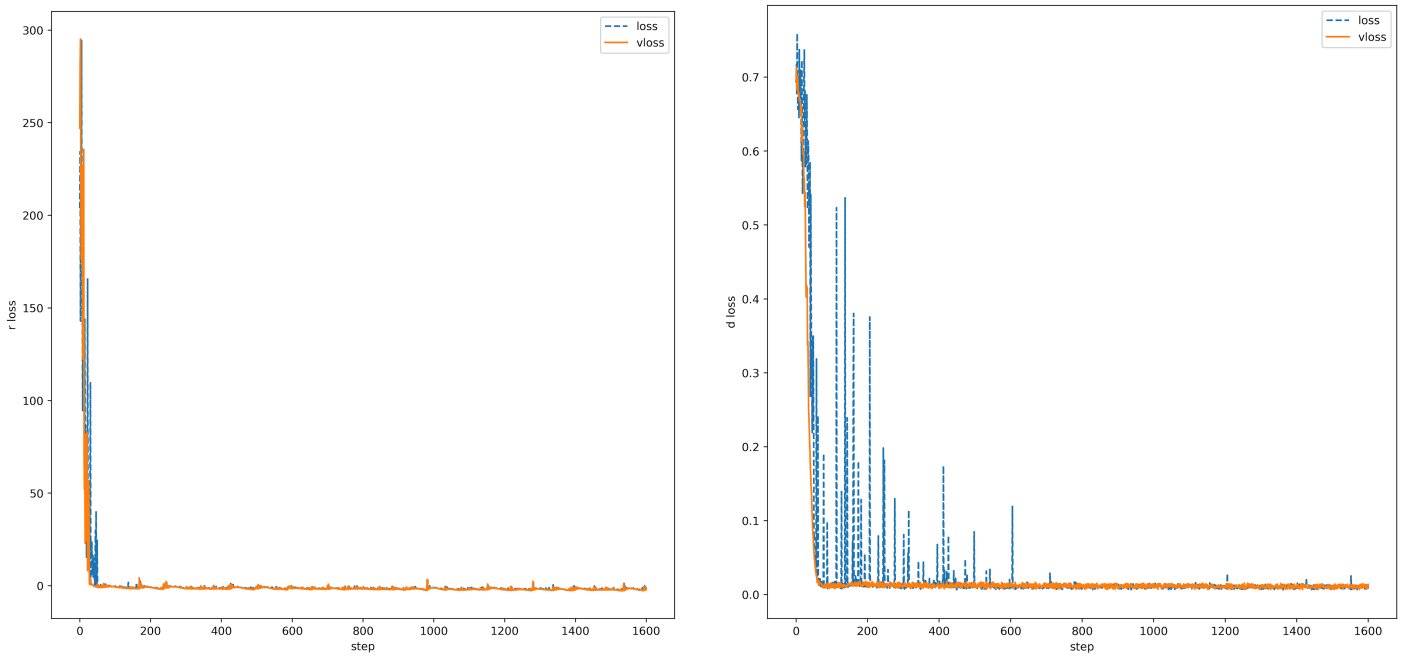


Figure 2.6: loss = $loss_z + loss_a + loss_r + loss_d$







Figures 2.7.a-d: Top Left is z loss. Top Right is action loss. Bottom Left is reward loss. Bottom Right is done loss. Everything except the bottom right used the MDN NLL loss, done loss was calculated with Binary Crossentropy.

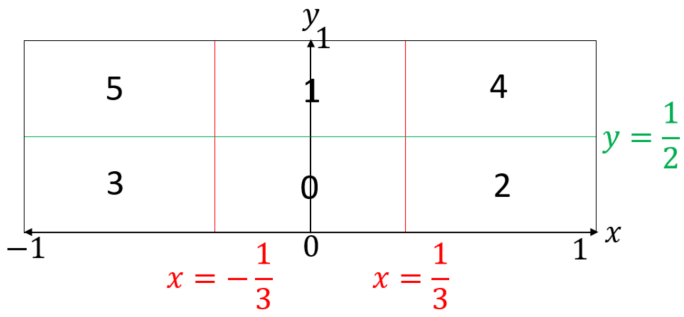
Here is a summary of the interesting features of the new loss function evident in the loss graphs above:

- Z loss is converging, but very slowly and with a high variance. This may be due to the online learning applied. Due to the model architecture and design of data, we decided against batch learning.
- Action loss is very sporadic and the loss did not converge on the lowest value possible, hovering near 0. One reason for this is due to the fact that the actions chosen randomly to produce the random rollouts, and the agent was trying to learn a random distribution.
- Reward and Done loss converged relatively quickly. Reward loss was significantly higher than the other losses and thus the final loss was heavily scaled by the reward. We also tried using cumulative reward instead of normal reward, which significantly increased the reward loss. The intuition was that since reward is zero 95% of the time, the model might collapse at 0 leading to negative loss.
- We tried different scalings of the losses, at first trying to scale reward down in order to prioritise the minimising of the other losses. We also tried scaling up the z loss by 10, but found the difference to be quite minimal, and showing quite similar trends. **(Check Appendix 1)**

Continuous Action Space

The choice of action representation a_t changed a few times throughout our experimentation. Here is a table detailing the changes and reasoning behind them.

Representation	**Reasoning**
Initial: [0,1,2,3,4,5] [do nothing, shoot, right, left, right+shoot, left+shoot]	This was the representation given by the Space Invaders environment on OpenAI gym. We quickly realised it was not a good idea to leave the action ordered numerically in no intuitive order.

One Hot Encoded: [1,0,0,0,0,0], …,[0,0,0,0,0,1]	One hot encoded representation. This correctly displayed the action vector as a discrete variable, but due to the loss function, we would need to calculate the loss of the action separate to the z latent vector
2D continuous space: [-1,1] x [0,1]  Figure 2.8: representation of action space in 2D	By converting the discrete action space into a 2D continuous space, we can determine which action is specified by the boundary lines in red and green. For example, if a point is above the green line (y > ½) and greater than the right red line (x > ⅓), the action chosen is 4 (move right and shoot) As a continuous space, the action can be modeled by the MDN RNN.

Issue of Batching with Done

One issue with batching that we dealt with was how exactly to deal with the done datapoint. Ideally, we would like to reset the states of the LSTM when Done is reached to simulate how the model will be used, but with how keras is designed, this is technically impossible, without delving into lower level libraries like tensorflow, like the source code for WM. Initially we naively used batch learning, but without enforcing the reset of LSTM hidden and cell states, the overall quality of learning was lower, as referenced by our higher loss. Thus, for most of the experimentation, we decided to stay with online learning on data point (rollouts of the game of various lengths), implemented by calling the fit function on one game, and training on one epoch, and repeating this step N times. We also experimented with dropout, which allowed us to control overfitting.

Prediction and Sampling

Following WM's implementation of the MDN, we decided upon 5 Gaussian Mixtures, and modelled every output variable with 3 variables μ(the mean), σ(the standard deviation) and π(the mixing coefficient). Thus the output of our model had a shape $(5 * 3 * (z_{dim} + a_{dim} + r_{dim}) + d_{dim},)$.

N.B. this is with respect to the output, not the input. So for a simpler solution, the output would just be z_t so our model output would have the shape $(5 * 3 * z_{dim},)$.

Furthermore, this implementation of having a mixing coefficient for every mixture of every neuron in the MDN as opposed to a mixing coefficient for each component of the mixture model like in the keras implementation of an MDN may be the reason behind the difference in training loss.
(https://github.com/cpmpercussion/keras-mdn-layer/issues/14)

Prediction also needed an extra step of sampling from the Mixture of (μ, σ, π) per variable, which adds extra work after every prediction to sample a possible z_t.

Temperature determines the stochastic nature of the sampling, by raising the temperature, you may sample increasingly unlikely and random outcomes for z_t - this function will be useful for creating the Dream environment.

Dream Environment

With the V model and M model, we can extend their functionality in order to build a completely virtual environment, separate from the actual environment. The hope is to have a simplified environment which represents the actual environment so well that policies learned in this environment can be carried through to the actual environment. The functionality required is for the MDN RNN to model the $reward_t$ and $done_t$ variables. As discussed earlier, we need the binary event $done_t$ to determine when the agent dies in the dream, and $reward_t$ to approximate the reward of a given time step.

After training a model with these new parameters, we needed to create our own custom environment for openAI's gym package. This required 4 functions to be self implemented: __init__, step, reset and render. Due to the design of our MDN-RNN, the outputs of the M model match the outputs needed after each step. Here are the results of a particular rollout:

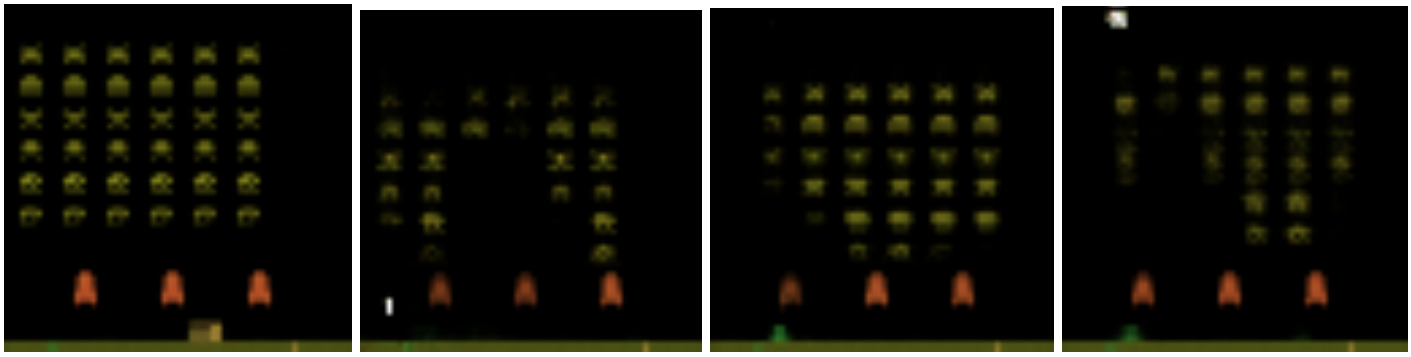


Figure 2.9: first 4 frames of rollout, decoded by V model, temperature 1.0

Although the VAE was able to encode the environment fairly well, and the MDN RNN learnt some general concepts of time evident from the starting position to the general movement of the enemies, important information like bullet travel as well as the movement of the agent itself were not encoded well. Below are some other rollouts with different values of temperature.

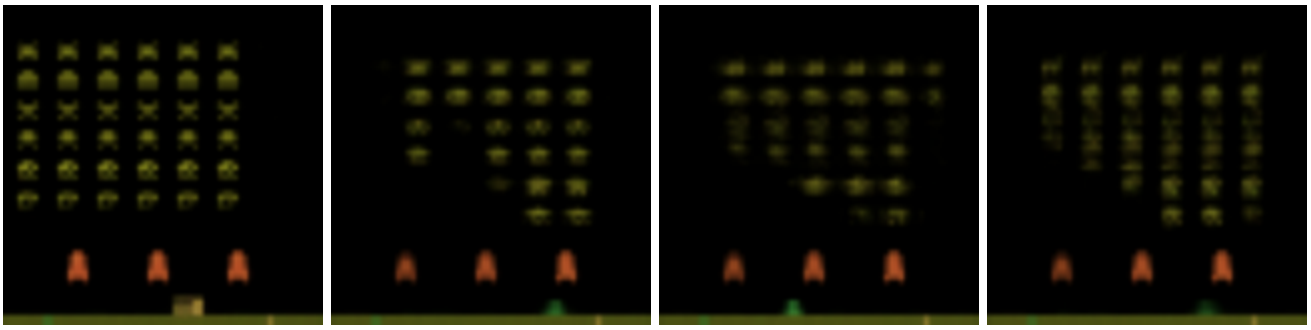


Figure 2.10: first 4 frames of rollout, decoded by V model, temperature 0.2

We can see evidence of mode collapse, as referenced in WM, as the M model fails to jump to another mode of Gaussian model where bullets appear. The inconsistency in movement of the agent at the bottom is also apparent even at the lowest temperature.

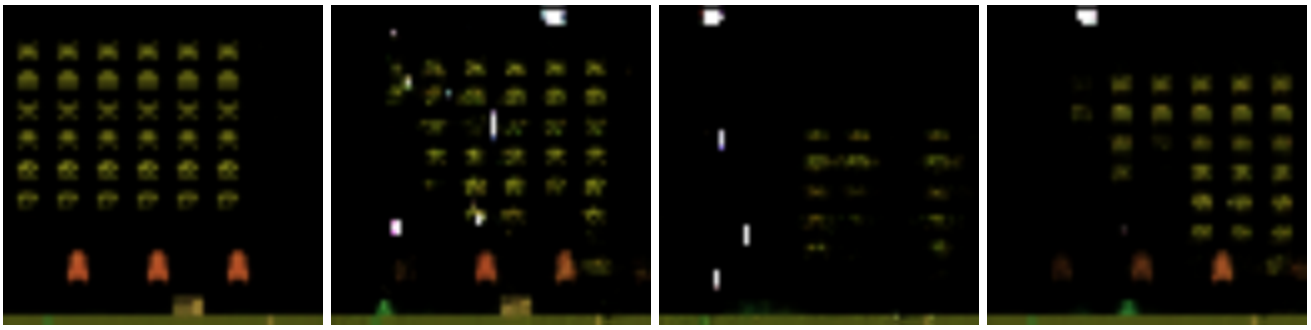


Figure 2.11: first 4 frames of rollout, decoded by V model, temperature 3

At very high temperatures, the image quality degrades as the V model is sampled from latent vector space that would not normally appear in the actual game. Ultimately, due to time constraints and the problems encountered, we did not pursue Dream training further.

Focussing on Z loss

Seeing how the z latent vector loss was not converging very well, I decided to scale back to the original WM architecture by only predicting $P(z_{t+1}|z_t, a_t, h_t)$. The effect on the convergence of loss on the z latent vector is clearly visible.

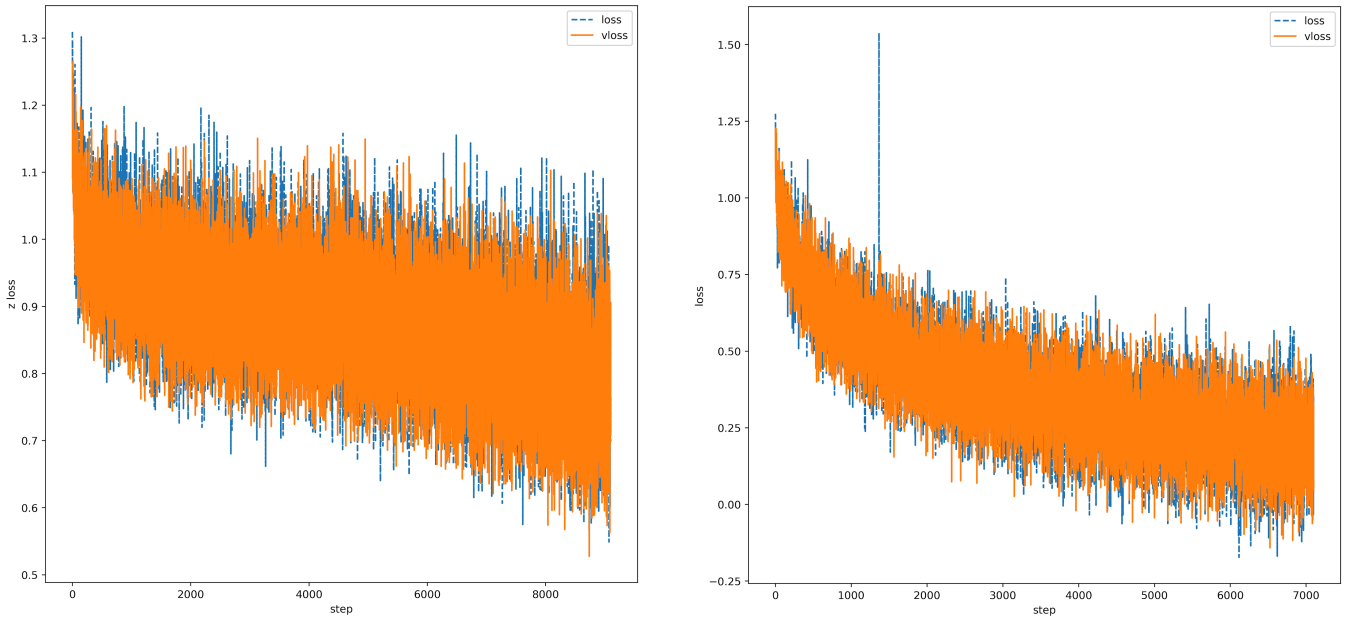


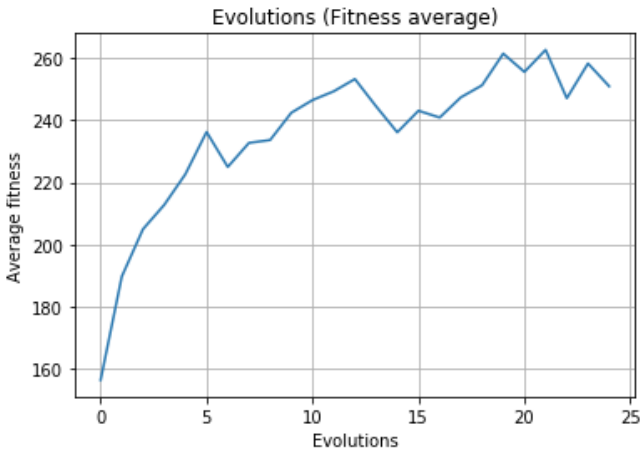
Figure 2.12: The left is z_loss equally weighted with the other losses. The right is only z_loss. The loss has less noise and reaches a lower value.

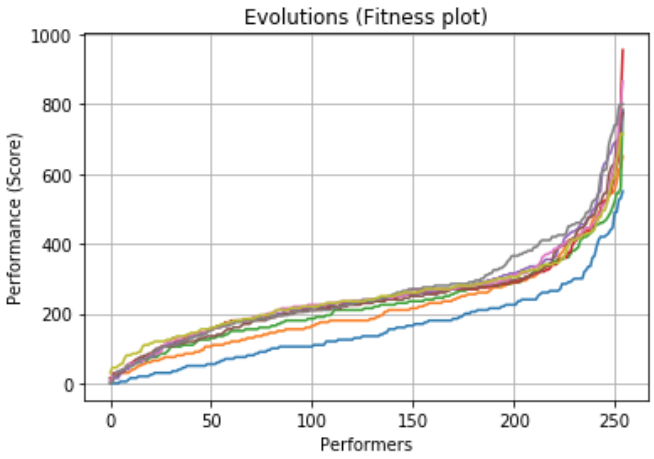
Controller Experiments

We have selected to use CMA-ES for general evolution strategy and cumulative score of each of the mutated model for fitness. Training the controller has a rather simple architecture to it, which also bring a natural limitation on the possible hyperparameters unless we introduce a new architecture. The training is also heavily dependent on the stages of development of the MDN-RNN and the VAE, along the process, we were able to conduct 2 successful evolution that produced some very promising results despite the time constraint.

First trial

In the initial trial, we have selected a population of 255, an initial $\sigma = 0.1$ for mutation along with the fitness set as the score of a performer at only 1 game. The results are shown below:

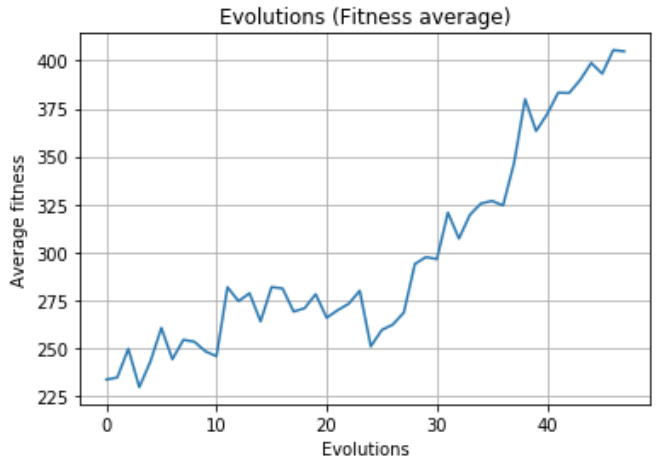


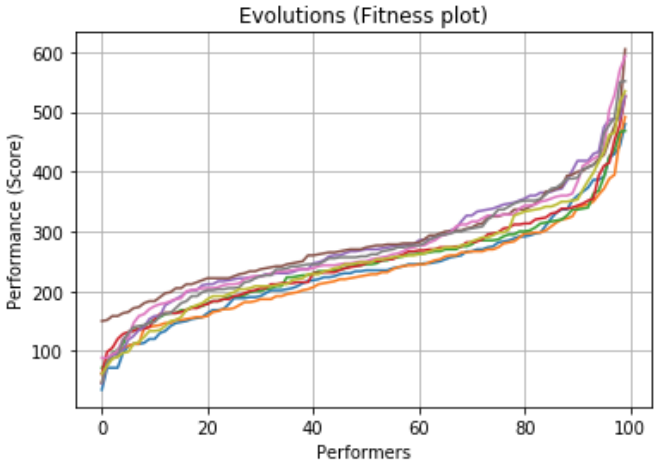


After 24 evolutions, the average performance among all of the population shows a general trend of increasing, but however there exists some issues. Since the game environment was stochastic, it is not a sensible strategy to use the performance of 1 game only as fitness. Furthermore, population size of 250 is taking too long for generating the fitness, therefore we proceed to run another trial with better hyperparameters.

Second trial

For the second trial, we have re-adjusted the population size to 100 and used an average of 3 games as the fitness to avoid certain performer getting lucky, the results a more steady increase and a more consistent performance as no winner of the evolution will be picked simply due to random events. We now realize that the action of those performers aimed more and more towards the commander ship, which all its mutation seems to have preferred one approach which is to simply stay and wait for the commadership, causing a local maximum in the process of optimization.





Final remark

As the training of the controller requires other parts to be functional first, the training process did not start until the later stages of development, giving a constraint on time. We have only experimented limited evolutionary strategies and model architecture. Analysis have shown that for games (like space invaders) that involves

continuous reward along the time step of the game, architecture like Q-learning etc. tends to perform well but such structure was not possible in the given time frame.

However, one very interesting experiment we conducted was to train the controller on the RAM states of the game supplied by the OpenAI Gym [https://gym.openai.com/envs/SpaceInvaders-ram-v0/], instead of the input from the VAE and the RNN. Since the RAM states fully describe the current state of the game as well as some aspects of its future (e.g. the direction in which the aliens will move in the next time step, when the commander ship will appear), it serves as a perfect baseline to test the performance of the VAE-RNN combination. From quick experimentation, CMA-ES trained on the RAM states achieved 1200 points in the game on average after 20 evolutions, which seemed to be much higher than the scores achieved by our framework. This suggests that more improvements can be made to the VAE and RNN in future research.

Conclusion

In this work, we applied the World Model Framework to an Arcade game Space Invaders. Our model can encode the Space Invaders environment into a latent vector space, and it has learnt to predict the probability distribution of the next state of a given latent vector. We demonstrate that the seemingly difficult to encode 2D environment of Space Invaders can be encoded into a compressed latent vector representation preserving important information such as the agent and enemy positions and bullet travel. We applied the World Model's Dream idea to Space Invaders to generate a dream environment which attempts to emulate the real environment. We also trained a controller network that used the V and M models to compress the information and showed that an agent can be trained in this way. We hope to encourage further research and development in the area of World Models Reinforcement Learning, especially in the area of atari games.

References

1. **Ha and Schmidhuber, "Recurrent World Models Facilitate Policy Evolution", 2018.**
   For the basic architecture
   https://worldmodels.github.io/

   TF-Implementation for Gym (9.0, note: uses deprecated APIs, found near the end of the project, so we could not utilise the resources fully)
   https://github.com/hardmaru/WorldModelsExperiments

2. **David Foster: Hallucinogenic Deep Reinforcement Learning Using Python and Keras**
   Keras implementation for Car Racing (Gym 10.0; uses Windows-incompatible libraries). Used for streamlining and visualising the training processes, after VAE experiments have been completed with code written from scratch.
   https://medium.com/applied-data-science/how-to-build-your-own-world-model-using-python-and-keras-64fb388ba459

3. **Pluralnet**
   For visualisation of VAE layers
   https://github.com/HarisIqbal88/PlotNeuralNet

4. **Atari Environment State of the Art leaderboard**
   https://www.endtoend.ai/envs/gym/atari/

5. **MDN negative loss discussion**
   https://stats.stackexchange.com/questions/294567/negative-loss-while-training-gaussian-mixture-density-networks/381694

6. **MDN paper**
   https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/bishop-ncrg-94-004.pdf
7. **Generating handwriting**
   https://arxiv.org/abs/1308.0850

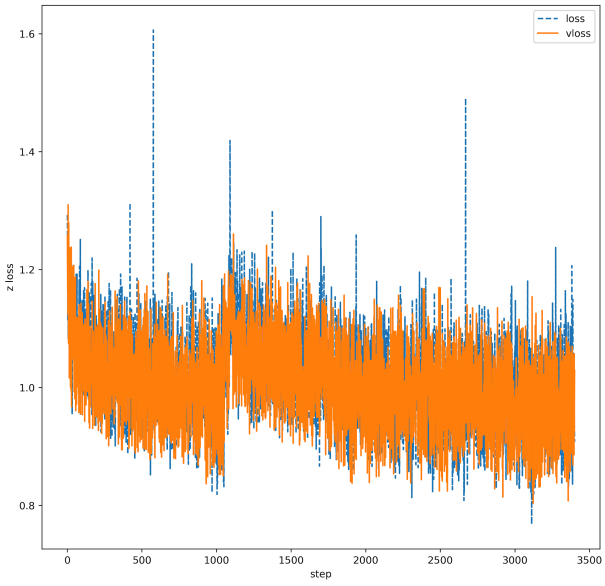
8. **SketchRNN**
   https://arxiv.org/abs/1704.03477

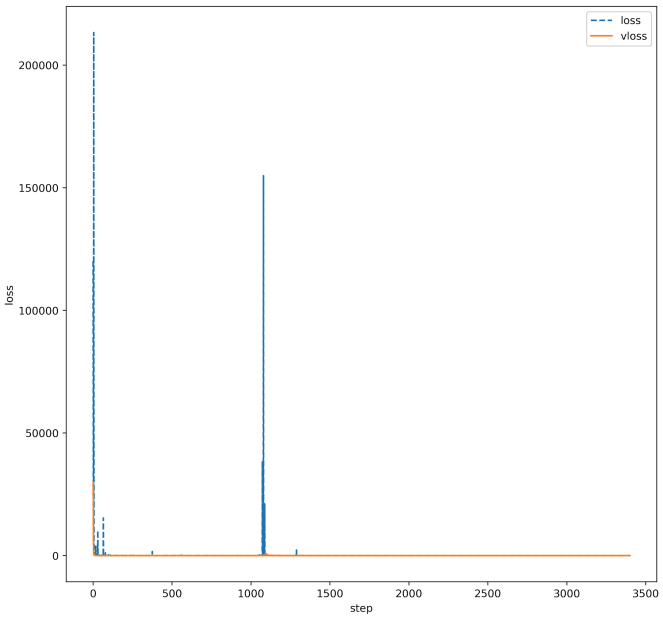
9. **MDN layer keras implementation**
   https://github.com/cpmpercussion/keras-mdn-layer

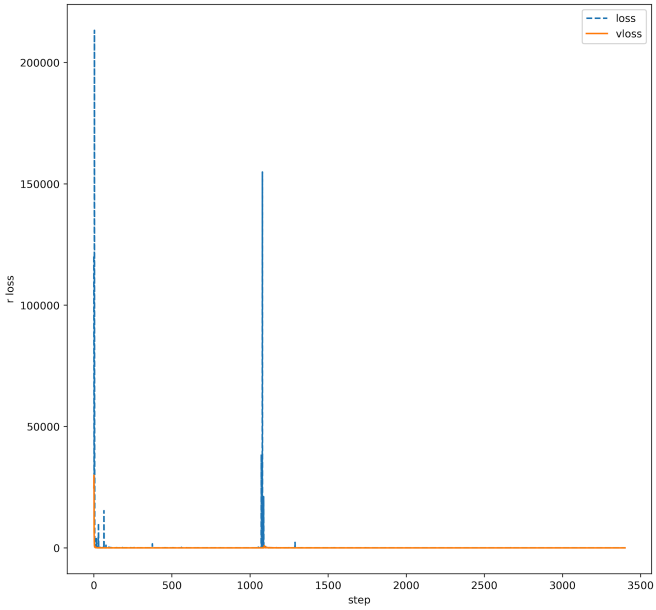
10. **World Models basic implementation**
    https://github.com/AppliedDataSciencePartners/WorldModels

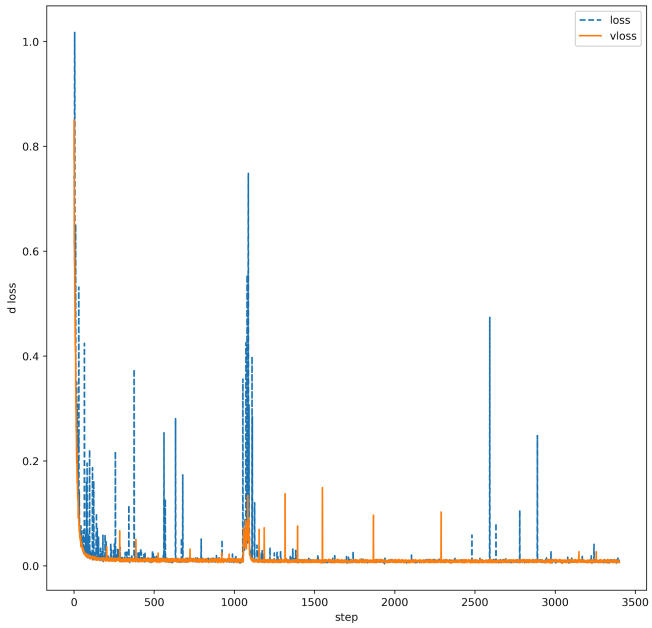
11. **Paquette, Philip**
    Doomtakecover-v0 2016
    https://gym.openai.com/envs/DoomTakeCover-v0/

Appendices









Appendix 1: Top Left Z loss, Top Right total loss, Bottom Left reward loss, Bottom right done loss. Z loss does not converge well, when using a loss= $loss_a + loss_d + 10loss_z$, in an attempt to amplify the loss to z.