Erica Chou, Max Christ, and Steve Yang 05/04/2021
Artificial Intelligence Professor Pantelis

Task 1:World Models: An in Depth Explanation

In the World Models paper, the authors attempted to set up an efficient framework for creating Agents which can react to and perform actions in an environment in a reinforcement learning (RL) context. Inspiration for the strategy was taken from Neuroscience, and previous papers in the AI field. Essentially, RL algorithms become unmanageable on large models because the number of weights that need to be adjusted becomes very large, causing the credit assignment problem. However, in order to be able to learn complex environments, a large RNN model is required. To reconcile these two conflicting qualities, the authors "divid[ed] the agent into a large world model and a small controller model," (Ha & Schmidhuber, 2018) so that the agent can retain the expressiveness of the RNN in the large world model, and allow the controller to handle the RL algorithm on a smaller set of weights. This strategy was implemented and tested on two different virtual environments to evaluate its performance. Additionally, due to the nature of the agent, the authors were also able to simulate the virtual environments using the world model. This interesting development led to increased performance in the original environments after training in the simulated, "dream," environments.

Agent Model

In a way that mirrors the way the human brain works, the authors divided the world model into two distinct models. The first is the Variational Autoencoder (VAE) Model, and its purpose is to translate the environment it sees into a smaller, simplified representation, in much the same way the human brain represents visual inputs as internal symbols. Specifically, in this paper, the V model takes images of the environment as input, which the VAE converts into latent vectors, which they named z. The second, is the Mixture Density Network - RNN (MDN-RNN, or just M) Model, which aims to predict the next latent vector, in the same way that the human brain anticipates changes to our surroundings in order to make decisions. The M model predicts the next latent vector as a probability density function approximated as a mixture of Gaussians. In addition to the two models comprising the World Model, the agent also has the controller model (C) to make decisions on what actions will maximize its rewards. The authors used a simple linear model to ensure that the RL algorithm did not get bottlenecked, as described earlier. Since the number of weights needed for this model is relatively small, the authors decided to use the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) for parameter optimization. The agent was then constructed using the V, M, and C models, before being trained on the two test environments.

Car Racing

The first experiment performed in the paper involves a car racing environment. The goal in this experiment is to use the world models generated by the Vision and Memory model to derive the actions that will result in the most rewards. In the game itself, the way to earn a reward is by going to as many of the road tiles as possible in the least amount of time. The only actions the car can perform are steering left/right, accelerating, and stopping. The maps are also randomly generated for each car racing trial.

The first step of this process involves training the vision model, V, using a dataset of random rollouts of the environment. This dataset is produced by allowing the agent to act randomly to explore its environment and recording its actions. Each frame of the data is then encoded into latent vectors that are produced by the decoder. These latent vectors can then be used to train the Memory model, M, as a mixture of Gaussians. Note that only the controller has access to how the rewards work in the real world, and using CMA-ES (Covariance matrix adaptation evolution strategy) allows the controller to optimize the amount of rewards earned given the data from V and M.

The results of this experiment were split into two parts. The first part handicapped the controller by not using the information from the memory. The agent was still able to generally navigate the track; however, the car tended to wobble and miss sharp turns. On the other hand, using both the predictive power of M along with the V model allowed the controller to perform remarkably better. Since the Memory model contains probability distributions of the future, the agent can merely query the RNN to guide its decisions. This allowed the agent using world models to be the first to solve the car racing game and achieve the highest average score recorded.

VizDoom

The second of the two experiments in the paper used a Doom-based AI research platform called ViZDoom. For this experiment, the goal of the agent was to learn how to avoid fireballs for the longest amount of time, by moving either left or right. To achieve this goal, the agent was trained purely inside of a "dream" environment rather than the real environment. The ViZDoom agent was able to train inside this dream environment because the set up for the experiment had all the components necessary to "make a full RL environment," (Ha & Schmidhuber, 2018). Unlike the Car Racing experiment, the VizDoom experiment has a death state along with predictions about the next frame. The death state, a punishment, and steps taken, a reward due to how the experiment is set up, are both needed to set up this imaginary RL environment to train in.

Being able to train in the dream environment proved to be very beneficial to the results of this experiment. Because the world model is internalized and was made by observing raw images from random episodes, the virtual environment could also purposely be changed. Though the virtual environment should simulate the basic parts of the game such as game logic, it is not required to to be an exact replica of the game. Therefore, the game can be adjusted to be easier or harder. In the paper, it is noted that having extra uncertainty added into the virtual environment made the dream game more challenging and improved the overall performance of the agent. This improvement occurred because the agent would be more prepared going into the original, more organized environment after training in the harder, less certain virtual environment.

One major concern that was addressed in the paper, was the "adversarial policies," that would plague the experiments (Ha & Schmidhuber, 2018). These policies would come up during the experiment because the controller was finding ways to exploit the systems of the game. Even though the world model should be following the rules of the game, the model does not have to follow these rules as stated before. This results in the model exploiting hidden variables in the game to maximize the amount of points the agent got. For example, in the experiments, the agent found a policy where if the agent moved in a certain way, the monsters in the virtual environment would not shoot fireballs. This could be detrimental to training the agent because the agent could very easily find and use these adversarial policies in the virtual world and then perform worse in the real environment.

To counter these policies, MDN-RNN was used as the dynamic model to train the Controller model inside a "more stochastic version of any environment," (Ha & Schmidhuber, 2018). This, in turn, allowed for the adjusting of the temperature parameter to control how random the model could be. After varying the temperature of the virtual environment, it was found that generally, higher temperatures will allow the agent to achieve the highest scores. However, the temperature cannot be too high because this will make the virtual environment too hard for the agent to learn anything. Overall, the experiment that was conducted allowed the agent trained in the virtual world model environment to not only succeed in surviving for a longer period of time than intended, but the agent was also able to achieve the best average score on the OpenAI Gym leaderboard.

Iterative Training Procedure

The first two experiments involved tasks that had somewhat simple actions and rewards. However, to implement this method on more difficult tasks, the iterative training procedure is required. For this procedure to work, an agent must be able to continuously explore its world while collecting data on the new or unknown parts. Using this new information, the agent would be able to continuously improve its world model over time. According to the World Models paper, the process works like this:

1. Initialize M, C with random model parameters.
2. Rollout to actual environment N times. Save all actions at and observations xt during rollouts to storage.
3. Train M to model P(xt+1, rt+1, at+1, dt+1|xt, at, ht) and train C to optimize expected rewards inside of M.
4. Go back to (2) if task has not been completed.

To encourage the agent to explore new parts of the world, it is recommended to flip the sign of the model's loss function in the real environment. This is because when the model is having a hard time doing its job, the agent is exploring unknown territory of the world, and this situation happens when we encounter more loss.

This type of iterative training also requires the model to be able to predict the action and reward for the next step on top of already predicting the next frame. Making the model predict actions allows the model to continually add information that the controller does into its world, which also allows the controller to focus on constantly learning new skills.

Conclusion

Using our understanding of a human's perceived world model has helped further the capabilities of AI models. World Models, which was derived from this idea of having unique perceptions of the world, was able to successfully train agents using a Vision Model to allow the agent to visualize input frames, a Memory Model to predict probabilistic distributions of future frames, and a Controller model to determine the action to take for the best reward together. The two experiments that used these world models not only showed how to implement these models into different environments, but it also showed how effective this model was. Furthermore, it seems that this topic could be used for more complex tasks and has a great deal of potential for future applications. .

References

Bhatt, Shweta. "Reinforcement Learning 101." *Medium*, Towards Data Science, 19 Apr. 2019, towardsdatascience.com/reinforcement-learning-101-e24b50e1d292.

Ha, David R and J. Schmidhuber. "World Models." ArXiv abs/1803.10122 (2018): n. pag.

Erica Chou, Max Christ, and Steve Yang 05/04/2021
Artificial Intelligence Professor Pantelis

Task 2: Results of Attempting to Reproduce Benchmarks Seen in the Original Paper

Replicating the results of the car racing experiment in the World Models paper proved a very difficult task. Especially given the tight time constraints, limited computing resources in AWS, and the amount of troubleshooting involved. Specifically, there were bugs in the source repos, and learning how to use Docker/AWS came with many challenges. Additionally, the first repository that was given had to be discarded in favor of the latest repository from the professor, which took even more time to learn. Despite all of this, we were able to run the entire pipeline twice successfully, not including what was done in task 3. The results are not nearly as good as the original paper. However, given that we were only able to use CPU, the improvements were significant.

The first time all of the models were trained, was on the second repository. Unfortunately, the VAE training ended in a premature termination, and this fact went unnoticed. We went ahead with the rest of the training until completion. This first run took about 21 hours, and it was interrupted because it trained for longer than the original paper, and was not improving with time. All of the notebooks which “check” the output were run in order to evaluate performance. However, none of these notebooks had the same plot which was present in the original repository, which plotted the original performance of the paper next to the experimental performance. Therefore, we recreated this plot in the new repository, and visualized our output next to the original. This notebook was named “06_plot_results.ipynb,” and is located in the root folder of the repository.

Unfortunately, the performance of the first run was abysmal, and this can be seen in Figure 1. Essentially, the score continuously hovered around zero. However, this is not surprising given that the VAE was not trained fully. However, some of the 'Check' notebooks provided results that suggested that the individual models worked better than anticipated, given that the VAE was imperfect. It is important to note that we had to use 300 rollouts, or episodes, due to the space and computing restraints of the AWS virtual machines. The number of time steps used to generate the data was 300. Additionally, for VAE training we used N = 1,000 (where N is the number of episodes to use to train) and 3 epochs. It turned out that the large value of N was the reason why the VAE terminated early - it ran out of space. Therefore, we experimented with other values of N, and settled on N = 100, because it did not terminate early. We used this value for our second run and were able to run the full pipeline without any errors. For further clarification, batch size was set to 100, and number of steps was set to 300 for the RNN training. In the controller training, we used n = 4, t = 1, and e = 1, with a max length of 1,000.

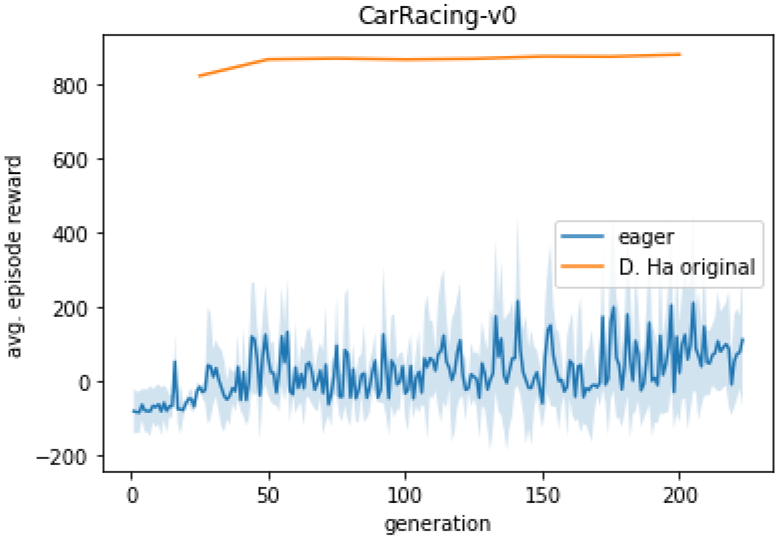


Figure 1: Output after training the controller, but with a VAE that was not completely trained.

The results for our second run were much more promising than the first. This can be seen in Figure 2. We used all of the same parameters as run 1, except we changed N to be 100 during VAE training. This coupled with the fact that we only could use 300 rollouts (as opposed to

10,000 in the original paper) severely limited the performance of our models. The second run seemed quite promising up until about the 50th generation, because the score was steadily increasing. However, after this point, the performance waned and did not continue its upward trek. It took a similar amount of time to run the second run as the first, and it was allowed to run until its roughly 200th generation. The performance steadily declined, and this might be due to overfitting of the data. However, considering that the performance improved in the beginning so quickly, this mirrors the large initial jump in performance seen by the original paper. Therefore, it seems that once the pipeline is properly set up, the controller trains quite quickly.

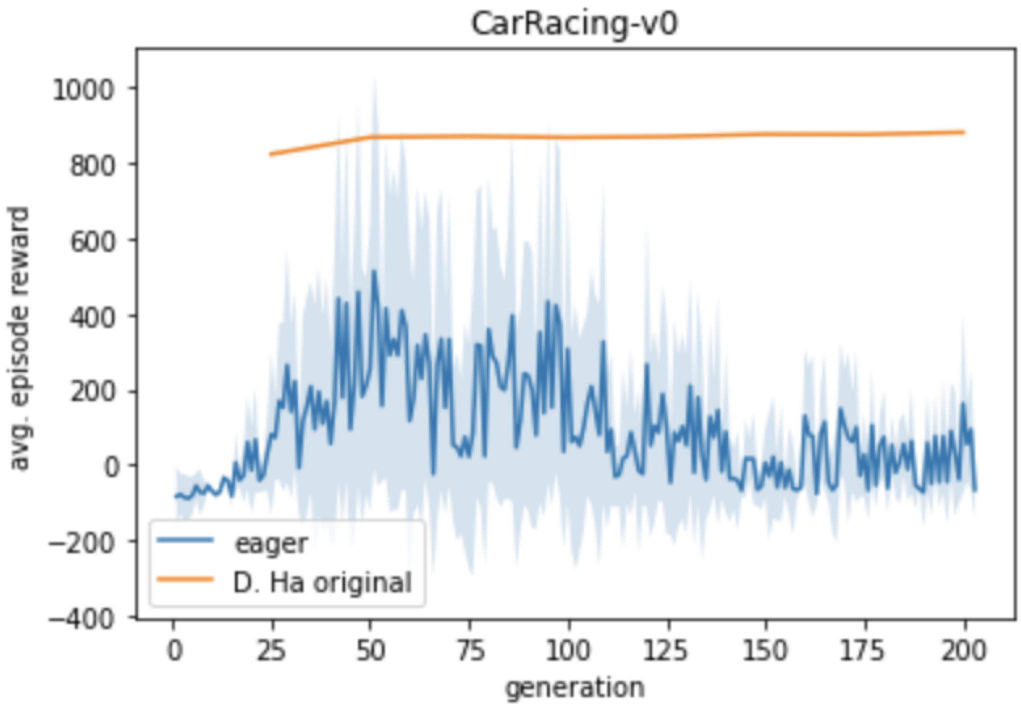


Figure 2: Output after training where all of the models were fully trained.

Therefore, given adequate time, computing power, and resources, it seems plausible based on this project and the World Models paper that it does not take very much for a controller to learn a policy to drive around a race track. That is, if we consider about 50 generations to be quick. This is an extraordinary result, for the future of AI and the world.

Erica Chou, Max Christ, and Steve Yang 05/07/2021
Artificial Intelligence Professor Pantelis

Task 3:Implement VAE-GAN into the car-racing experiment.

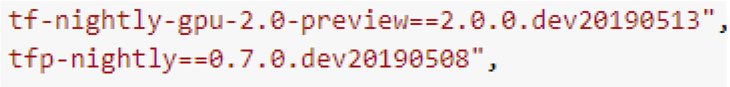
Introduction:

For task 3, we read through the GAN tutorial on tensorflow website and the VAEGAN article to familiarize ourselves. VAEGAN basically presents an autoencoder that leverages learned representations to better measure similarities in a data space. By combining a variational autoencoder with a generative adversarial network we can use learned feature representations in the GAN discriminator as the basis for the VAE reconstruction objective [1]. We are going to implement VAE-GAN to our car racing experiment to improve its performance.

Run Sample:

Firstly, we tried to successfully run a sample of VAE-GAN. We chose https://colab.research.google.com/github/timsainb/tensorflow2-generative-models/blob/master/6.0-VAE-GAN-fashion-mnist.ipynb#scrollTo=M[…]u4ZvNCTTScV . That is a colab notebook of VAE-GAN.

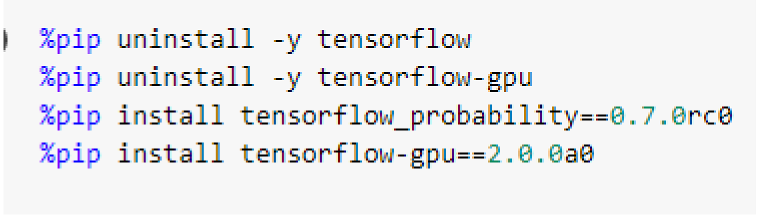
The main challenge we faced to run sample code is setting the proper environment. The requirement of the colab notebook is



```
tf-nightly-gpu-2.0-preview==2.0.0.dev20190513",
tfp-nightly==0.7.0.dev20190508",
```

Fig 1: colab environment

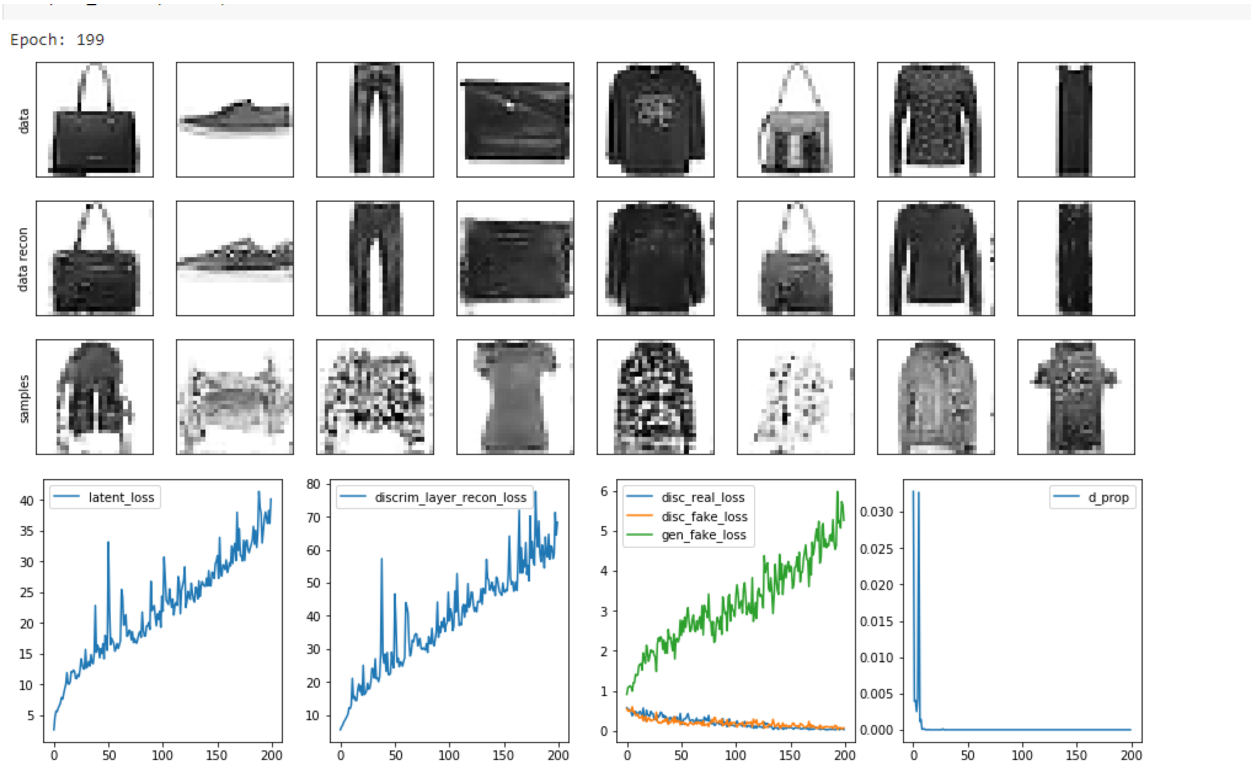
However, the default environment is tensorflow 2.4.0 and tensorflow-probability 0.7.0.dev20190510, which are not compatible. At first, we simply used "pip install tensorflow==2.0.0.dev20190513 and pip install tensorflow==2.0.0" to get the correct version of tensorflow. But what we got is still incompatible. After doing some research about the colab environment, we figured out that the version of tensorflow and tensorflow-probability is kind of unique, so we have to use the following lines:



```
%pip uninstall -y tensorflow
%pip uninstall -y tensorflow-gpu
%pip install tensorflow_probability==0.7.0rc0
%pip install tensorflow-gpu==2.0.0a0
```

Fig 2: setting environment

That brings us the proper environment to run the notebook. And we successfully ran the sample of VAE-GAN.



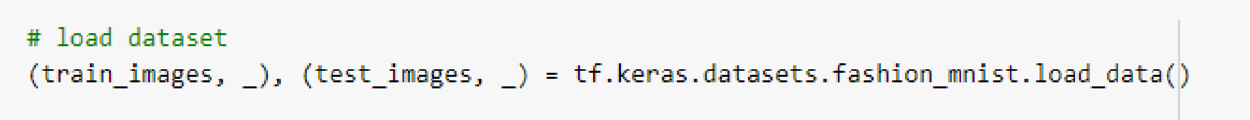
```
Epoch: 199
```

Fig 3: sample VAE-GAN result

Implement Car-racing Data:

Convert Data:

The second step is to train VAE-GAN with our own dataset, so we have a look at the data format the colab is using. We found that in the colab:

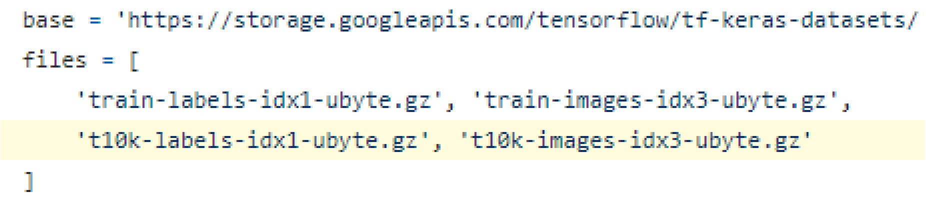


```
# load dataset
(train_images, _), (test_images, _) = tf.keras.datasets.fashion_mnist.load_data()
```

Fig 4: How the original colab imported the dataset

To understand it we tried to read the code in the keras packages: https://github.com/tensorflow/tensorflow/blob/v2.4.1/tensorflow/python/keras/datasets/fashion_mnist.py#L30-L91

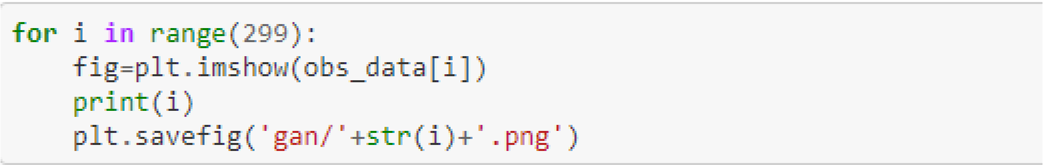
In the original tensorflow package we found that it is using the idx-ubyte data format.



```
base = 'https://storage.googleapis.com/tensorflow/tf-keras-datasets/
files = [
    'train-labels-idx1-ubyte.gz', 'train-images-idx3-ubyte.gz',
    't10k-labels-idx1-ubyte.gz', 't10k-images-idx3-ubyte.gz'
]
```

Fig 5: Source code of keras; the data format part.

As a result, we have to convert our data into that idx-ubyte format. To do that, we are using https://github.com/gskielian/JPG-PNG-to-MNIST-NN-Format. We used this to convert all images into the idx-ubyte format. As a result, we used the data generated by 01_generate_data.py and used the following scripts to decode each .npz file into png form: (Our total_episodes is 300)



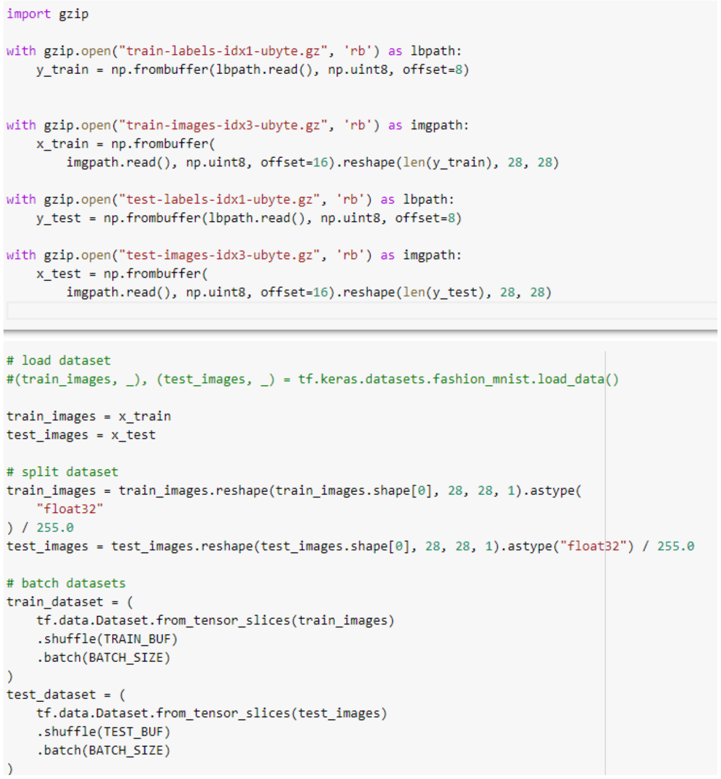
```
for i in range(299):
    fig=plt.imshow(obs_data[i])
    print(i)
    plt.savefig('gan/'+str(i)+'.png')
```

Fig 6: Script to get png from .npz data generated from 01_generate_data.py

After that, we got png data generated from the car-racing model. We converted this data into idx-ubyte format by using the previous github repo. For now, we get the training dataset, but that is not enough, we have to figure out how to import these datasets into the colab notebook.

Import Data:

According to the notebook, we can see that the original dataset is loaded through the keras package. So, if we want to import our own data, we have to write our own import function. https://colab.research.google.com/drive/1kdqA_3M_4qQJtRNd0oMb5w9J5aq75RjR?authuser=1#scrollTo=K1QmKBaoh4Z5



```
import gzip

with gzip.open("train-labels-idx1-ubyte.gz", 'rb') as lbpath:
    y_train = np.frombuffer(lbpath.read(), np.uint8, offset=8)


with gzip.open("train-images-idx3-ubyte.gz", 'rb') as imgpath:
    x_train = np.frombuffer(
        imgpath.read(), np.uint8, offset=16).reshape(len(y_train), 28, 28)

with gzip.open("test-labels-idx1-ubyte.gz", 'rb') as lbpath:
    y_test = np.frombuffer(lbpath.read(), np.uint8, offset=8)

with gzip.open("test-images-idx3-ubyte.gz", 'rb') as imgpath:
    x_test = np.frombuffer(
        imgpath.read(), np.uint8, offset=16).reshape(len(y_test), 28, 28)
```

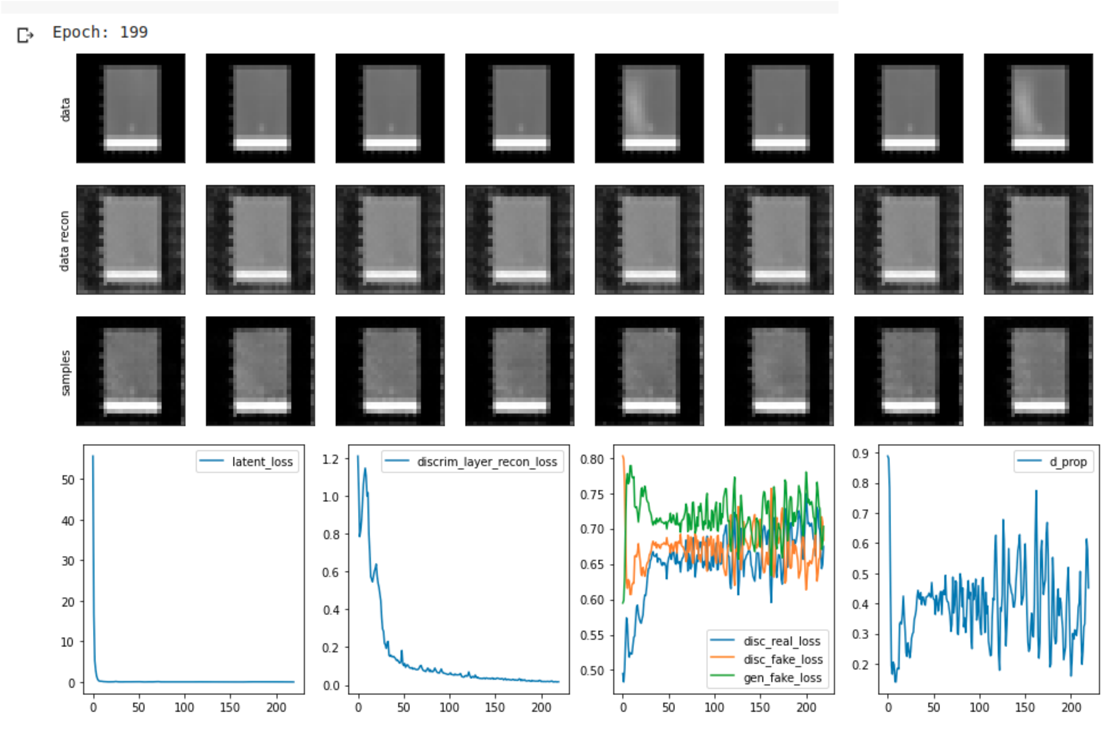
```
# load dataset
#(train_images, _), (test_images, _) = tf.keras.datasets.fashion_mnist.load_data()

train_images = x_train
test_images = x_test

# split dataset
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype(
    "float32"
) / 255.0
test_images = test_images.reshape(test_images.shape[0], 28, 28, 1).astype("float32") / 255.0

# batch datasets
train_dataset = (
    tf.data.Dataset.from_tensor_slices(train_images)
    .shuffle(TRAIN_BUF)
    .batch(BATCH_SIZE)
)
test_dataset = (
    tf.data.Dataset.from_tensor_slices(test_images)
    .shuffle(TEST_BUF)
    .batch(BATCH_SIZE)
)
```

Fig 7: Our script to import data

After that, we successfully imported our car-racing dataset into the colab notebook.

Train Model:

After hours of training with 200 epochs, we got the final result:



```
Epoch: 199
```

Fig 7: Our VAE-GAN training result. The notebook is dealing with grayscale images so we converted out images to grayscale before training.

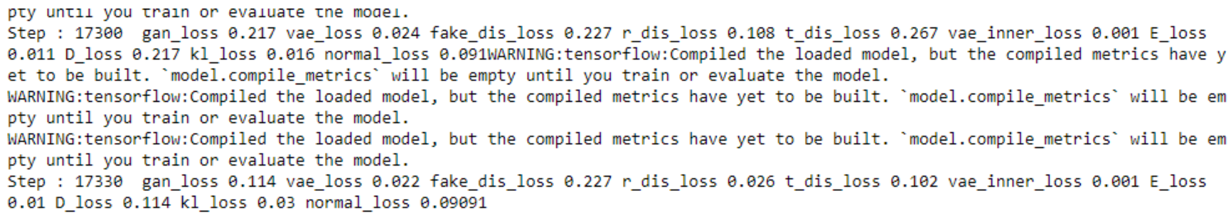
We can see that the loss is not pretty well due to the data and machine limitations. However, we got clearer data and the model is trained. We save the model in the GitHub workspace (weight.h5) and try to implement it in 03_generate_rnn_data.py. To do so, we write a new script (03_generate_rnn_VAEGAN.py) to import the VAE-GAN model instead of the VAE model.

Problems:

After we wrote the new script, we sadly found that the new VAE-GAN model is a 3 layer model and VAE model in car-racing is a 2 layer model so we can not import it. As a result, we tried to write structure for VAE-GAN model, however, we have to learn how to write it and have to find a proper environment setting, because if we try to write the same structure as VAE-GAN colab notebook, we have to set the same environment on our docker, which is also hard to deal with. In arch.py (architecture of VAE), we wrote a new model structure for VAE-GAN, but it seems like there are some environmental issues, we ran out of time on that part, really sad.

Method to solve:

We also tried to run another repo of VAE-GAN to see if that model works for the car-racing model (https://github.com/leoHeidel/vae-gan-tf2). We put the notebook on our repo(vae-gan.ipynb). In this repo, we train it through images directly, however, it takes ten more hours to train and still cannot reach an end.



```
pty until you train or evaluate the model.
Step : 17300  gan_loss 0.217 vae_loss 0.024 fake_dis_loss 0.227 r_dis_loss 0.108 t_dis_loss 0.267 vae_inner_loss 0.001 E_loss
0.011 D_loss 0.217 kl_loss 0.016 normal_loss 0.091WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have y
et to be built. `model.compile_metrics` will be empty until you train or evaluate the model.
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be em
pty until you train or evaluate the model.
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be em
pty until you train or evaluate the model.
Step : 17330  gan_loss 0.114 vae_loss 0.022 fake_dis_loss 0.227 r_dis_loss 0.026 t_dis_loss 0.102 vae_inner_loss 0.001 E_loss
0.01 D_loss 0.114 kl_loss 0.03 normal_loss 0.09091
```

Fig 8: Vae-gan notebook trained more than 10 hours with no end.

However, we can get the checkpoint of the training, and try to use them in our model. Unfortunately, the new model is with 9 layers which has the same problem as the colab model.

Following Thought and Strategy:

Although we cannot use the VAE-GAN model to train the controller, we think VAE-GAN can improve the performance of the controller. Because in the colab notebook, we can see our VAE-GAN model has a relatively low loss and the images it produced seems clearer.

Our strategy is to implement the VAE-GAN model instead of VAE model in 03_generate_rnn_data.py, so we can use the VAE-GAN model to improve the performance of rnn and controller.

If we got more time, we will try to debug our 03_generate_rnn_VAEGAN.py in order to implement the new VAE-GAN model, and the controller will get a better dataset to do the training. In that case, the final result will be better and get a higher score.