Master of Technology
Team 21 Project Report

NFTGan - Draw Masterful Art Pieces

Team Members
Mu Aohua - A0121924M
Lee Joon Hui Jeremy - A0048174A

Content

Executive Summary

Generative Adversarial Network (GAN) has proliferated in the general domain in recent years and we have seen harmless creation of deepfakes such as Barack Obama mocking Donald Trump (BuzzFeedVideo, 2018) to GAN generated videos that are used in election Delhi (India) campaign (DH Web Desk, 2020). We have chosen this technology as our project focus because we are fascinated by the potential of the technology. We hope to develop a Minimum Viable Product (MVP) to showcase the production version of our interpretation of GAN based on our knowledge acquired in this semester as well as all the available resources that we can find online and we hope to share our learnings in this report.

Business Objective

Our project is inspired by the NVIDIA's Canvas application[1] and the hype around Blockchain's non fungible tokens (NFT). Our objective is then to create a simple web canvas application called NFTGan whereby anyone can sketch their illustration on a 256x256 canvas, and then get it converted into a masterful art piece. The art piece will be stored on a blockchain provided by NFT.storage, and the user can choose to sell as a NFT on the NFT marketplaces such as OpenSea.

In our MVP, we allow our users to illustrate their interpretation of a Pokémon character on the canvas and convert it on users' local device into an artistic workpiece. The application can also swap in other GAN models for illustration other than Pokémon

Tools and Techniques

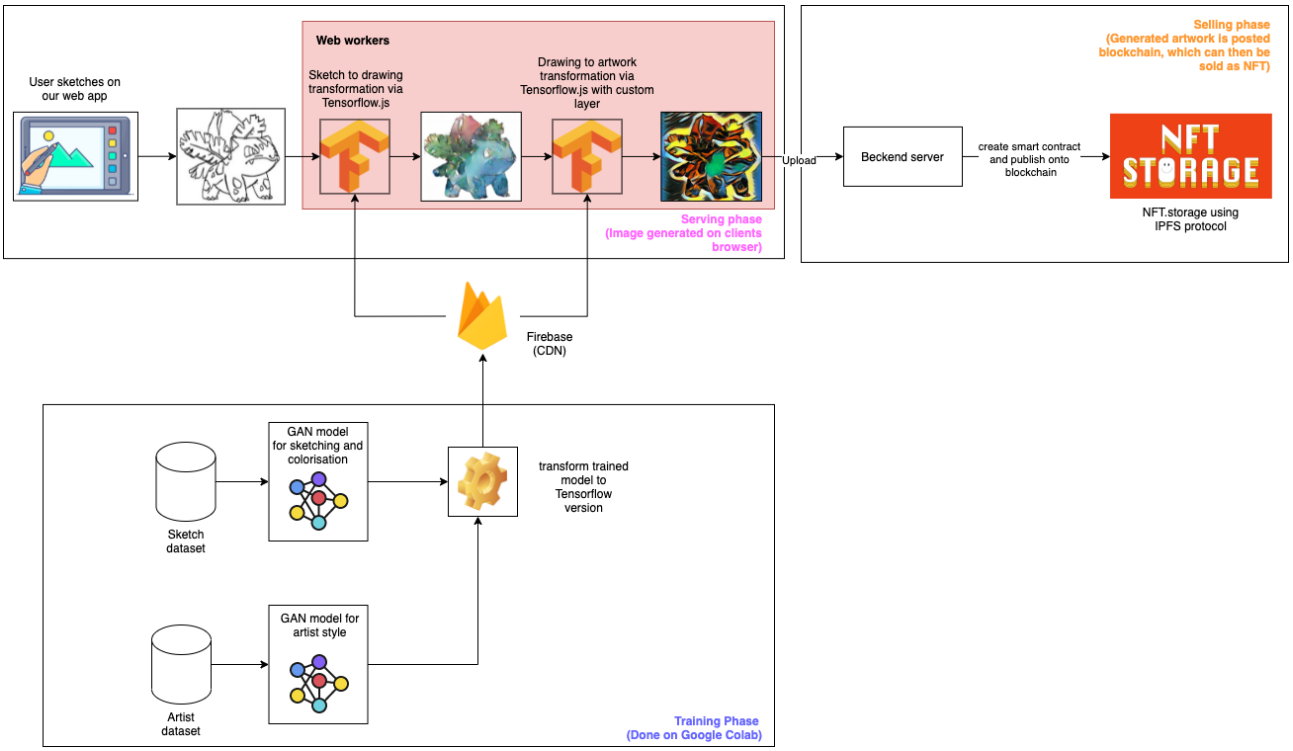
The following tools and techniques are used in the development of this project:

Tools: Colab pro, Visual studio code, and online image editing tool
Languages: python, and typescript
Packages: tensorflow, tensorboard, nodejs, react, @tensorflow/tfjs, and react-canvas-draw
Models: UNet, ResNet, PatchGAN, Batch normalization, Instance normalization, L1 loss, Cycle Consistency loss, Identity loss, and ReflectionPadding2D
Web components: Web worker, Canvas, fetch, and indexedDB

Notably, the tools and techniques highlighted in blue are new to us and we will be sharing our findings throughout the rest of this report.

[1] https://blogs.nvidia.com/blog/2021/06/23/studio-canvas-app/

Overview of System and Tools Used



System Design and Architecture

Our system comprises 3 systems; namely the training phase, serving phase and selling phase.

Training Phase

We are using 2 GANs in our application. The first GAN called **Pix2Pix (Jun et al., 2020)** allows us to colorise a sketch into a painting whereas the second GAN takes the painting as input and then produces the final artwork using **CycleGAN (Isola et al., 2018)**. More details on the model implementation will be shared in the next few sections.

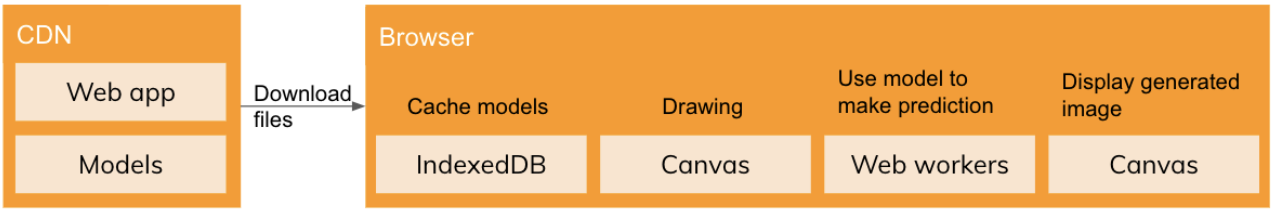
Both models are created and trained using Google Colab (Pro version) and they require at least a T4 or P100 GPUs to execute due to model complexities. The final models (only the trained generator models) will then be converted into javascript versions through TensorflowJS library and further enhanced with additional javascript code to support custom layers.

The final javascript models will be integrated into the serving phase to perform GAN transformation.

Serving Phase

In order to provide a stable and fast model inference, we opt to perform the drawing and conversion task on the user's edge device. This not only allows our users to experience the art

conversion in real-time, but it also removes our need to maintain a host of backend servers, which can be costly for this project. As such we only need to write and maintain the frontend code that was written in Typescript. All the models are served as JSON and weights files and executed in the web workers to achieve better performance.



Architecture of our serving phase

Our lightweight setup only requires us to maintain a copy of the web application on a CDN hosted on Firebase and we are able to hot deploy changes with every git commit via GitHub actions. Moreover, we added web workers to improve UI performance as well as cached models in the indexedDB to remove the need of downloading models in every page load.

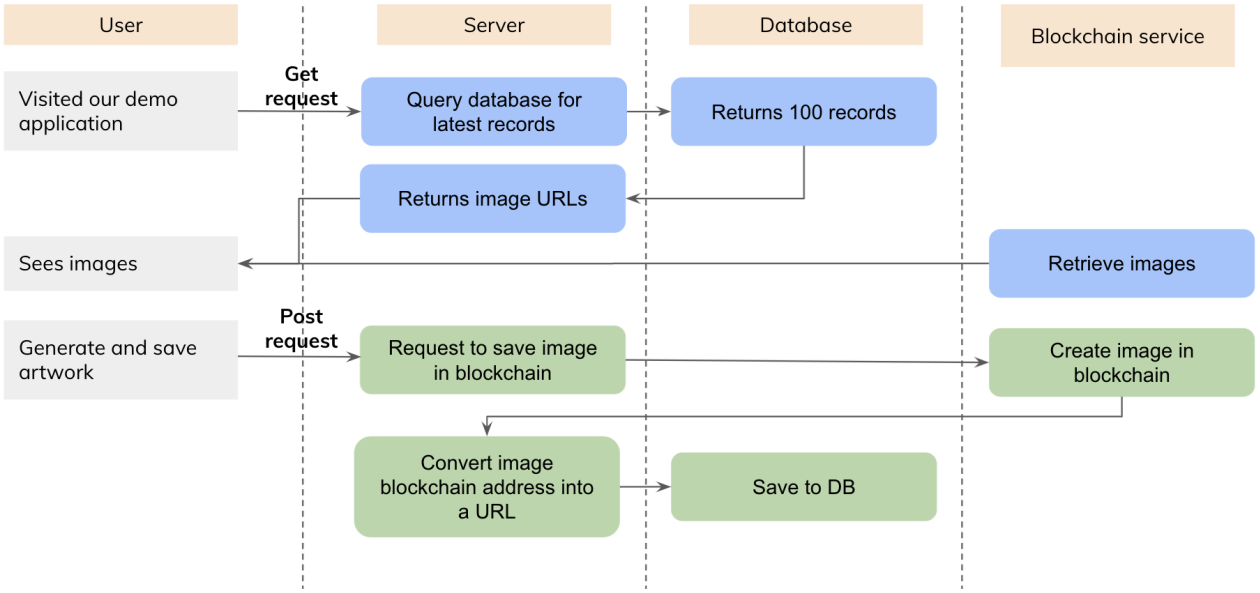
We have tested our application on a range of devices such as on a MacBook, iPad, and iPhone and found out the **latest Chrome browser** on **Windows, MAC OS,** and **Android devices** can produce satisfactory results. However, older devices with limited RAM or **IOS devices** might experience slowness, sudden termination of the app, or not even be able to load due to the high memory load required to perform GAN translation.

You can visit our hosted MVP at `https://draw-it-yourself.web.app/` for a live demonstration.

Selling Phase

In the selling phase, we have hosted a backend server on the Heroku platform. The purpose of the backend server is to provide the API endpoints to store and retrieve the artwork. The APIs will only be used when a user wants to browse artworks created by other users (GET endpoint) or wants to save her final art pieces onto the blockchain (POST endpoint). The backend is written on NodeJS and it employs a 2 tier architecture that is a server and a database.

The flow of the 2 APIs are as follow:



Retrieval of artworks

Endpoint: `https://nftgan.herokuapp.com/nft`
Method: `POST`
Arguments: `None`
Return: Latest 20 pieces of artwork contributed by the community

Description:
This API queries a Postgres database for the latest 20 artwork's blockchain proxy URL. Unlike a regular URL (i.e. HTTP) the original blockchain URL starts with IPFS to indicate the use of distributed web storage. In order to translate it back to a browser readable version, we use a IPFS publicly available gateway to translate it into a HTTP version, hence the name blockchain proxy URL.

Saving of artwork into blockchain

Endpoint: `https://nftgan.herokuapp.com/nft`
Method: `GET`
Arguments: `{nft_file: [blob], name: 'name of the image', description: 'description of the image'}`
Return: `200` code for success, `500` code for failure.

Description:
This API stores the artwork into the IPFS blockchain and then constructs the blockchain proxy URL and stores it, together with its relevant data, into the Postgres database. The actual flow is beyond the scope of this project and the details can be found on `https://nft.storage/`.

Note

For the purpose of this MVP, we have omitted key features such as authentication and WAF found in actual production systems. We have made the assumption that the project is only accessed by a handful of people and there is no risk in the event of a security incident.

Dataset

Pix2Pix

We are using the Pokémon dataset from the Kaggle challenges. It consists of 830 pokemon sketches and colored images for the training, and 20 sketches and colored images for testing. Each image is the size of 256 x 256, which fits perfectly in our unet256 generator. In order to produce a satisfactory model, we have also included image augmentation during the preprocessing (i.e. flip and random crop) phase. This helps to produce more training data for the model to generalise.

CycleGAN

We are using 3 sets of data to evaluate the performance of our CycleGAN model. The main factor to determine the “goodness” of the model purely lies on actual user poll and perception of the generated artwork. This is a similar approach taken by the authors who wrote the research paper (Jun et al., 2020). We will discuss the reasons and other metrics in detail in the **Model Evaluation** section.

The 3 datasets are the baseline dataset, augmented dataset and our own custom dataset.

Baseline Dataset (vangogh2photo)

We are using the `vangogh2photo` dataset provided by the original paper as a way for us to understand the implementation of CycleGAN because it is widely used in most online resources. We are also using the model trained by this dataset to do a baseline inference so that we can use it to quantify the other 2 datasets.

Augmented Dataset (pokemon2vangogh)

We hypothesise that our model can perform better if we use the output of our Pix2Pix GAN as the training input of CycleGAN. This augmented dataset, named `pokemon2vangogh`, uses the training data from Pix2Pix as TrainA/TestA (i.e. coloured Pokémon) and the original Van Gogh data as TrainB/TestB.

Custom Dataset (pokemon2animals)

Furthermore, we also theorised that the mapping from our training image to our generated images will yield better results if we use another set of train dataset (TrainB) in a similar domain. Inspired by the `horse2zebra` dataset where both animals share common domains, we use photos of animals, which have close resemblance to pokemons, and the pokémon data from `pokemon2vangogh` as our training pair as our this dataset. The end results will be shared more in the later section.

Data Preparation

Our main source of data comes from both Berkeley's dataset, which is used in the original paper, and from Kaggle website. Most of the datasets that we use in our project do not require further processing except for the custom set used in CycleGAN.

One of CycleGAN's strengths is to allow any two sets of input images to be the training data, which the model can then learn the relationships between them. Both input datasets can be of similar nature, such as horses and zebras, or from different domains, such as photos and Van Gogh's art pieces.

While we are creating our custom datasets `pokemon2animals`, we simply have to convert our raw images into the standardised size of 256 x 256 in order to feed into our GAN models. We wrote a custom python function just to do so and they can be found in our github[2].

Data Augmentation and Normalisation

In order to produce good results during training, we further augmented all our datasets by randomly cropping (enlarge and resize), and horizontal flipping the images in order to reduce overfitting. We also performed normalisation during both training and testing phase.

Model

In this project, we use 2 models to achieve the **sketch to artwork**. This is because it is a much more appropriate strategy given that our requirements include colorising and translating an image from one domain to the other.

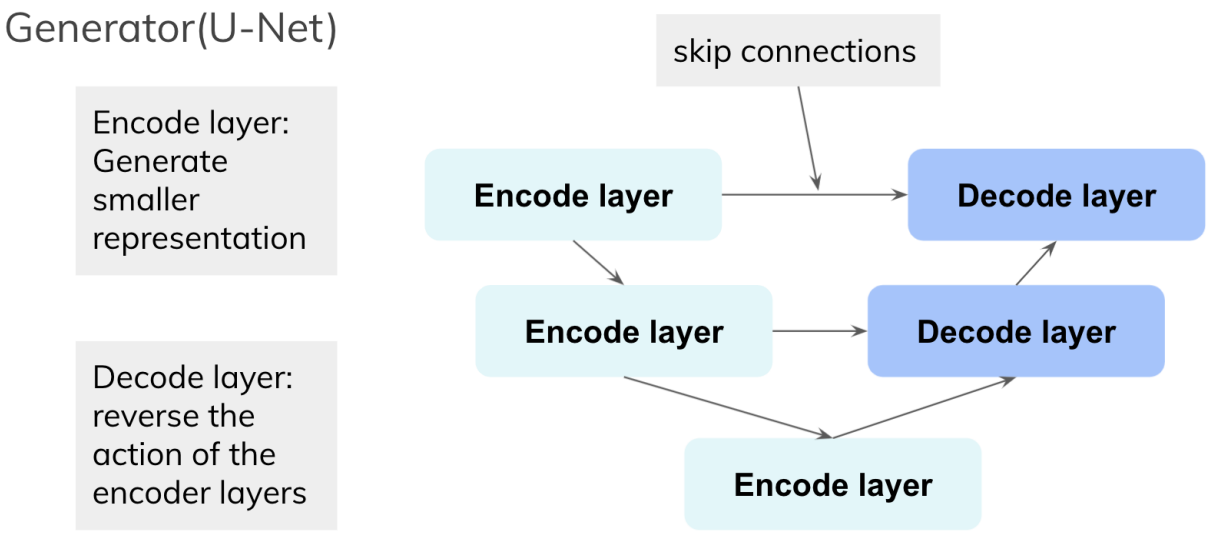
A general GAN model structure requires 3 key ingredients, which include a generator model, a discriminator model as well as a minimax loss strategy. Our models, Pix2Pix and CycleGAN are no exception and both of them share lots of similarities.

Similarities between Pix2Pix and CycleGan

Unet Generator

Both models use a common generator called Unet. The goal of Unet, which is a form of autoencoder, aims to distill the main features of the input image through downsampling and reconstruct the image using upsampling. Moreover, to preserve the location of the prominent edges of the source image, skip connections are established between encoder and decoder.

[2] https://github.com/aohua/nft-generator



Unet structure used in both Pix2Pix and CycleGAN's generator

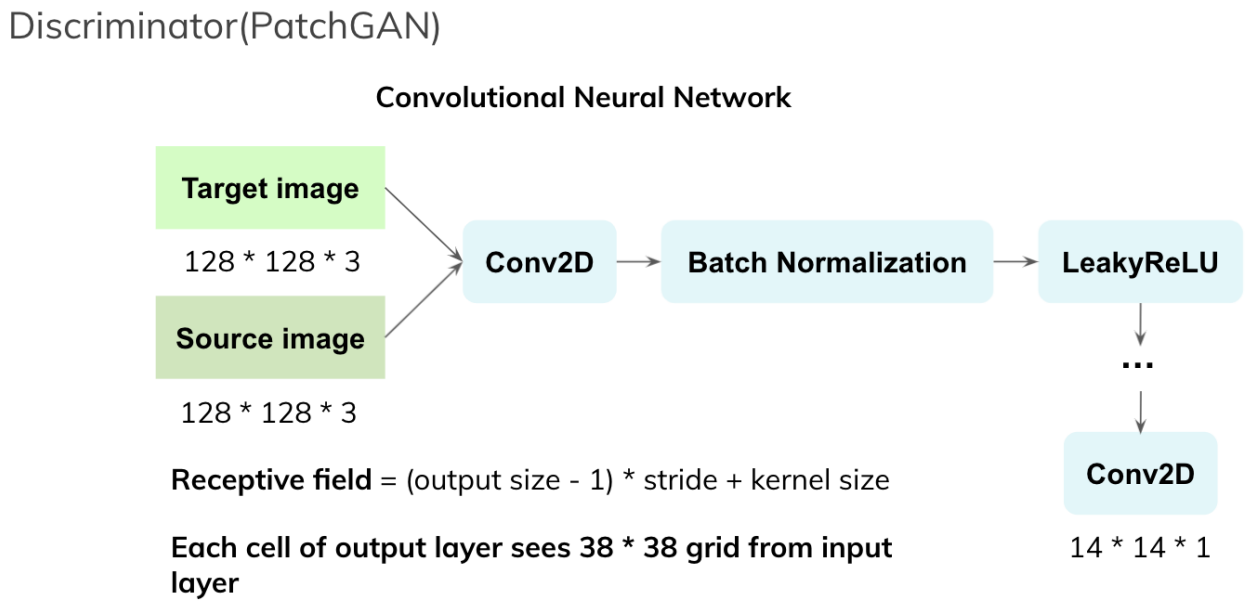
Unlike other GAN generators that use noise as the starting image, both models require two input datasets, a target image set and a real image set. The reason for this is because it allows our models to compare the input image with the real image to calculate the total loss, which in turn influences the back propagation to minimise the total loss.

PatchGAN Discriminator

When it comes to discriminator, we use PatchGAN (Demir & Unal, 2018) instead of the regular discriminator to distinguish real images from the fakes. The purpose of using PatchGAN is because:

1. It reduces the size of loss representation, and
2. It runs on fewer parameters, hence able to process faster and can be applied to arbitrarily large images.

For example, we have target image of 128 * 128 * 3 and source image 128 * 128 * 3, the total size of the input will be 128*128*6, which is quite a large input. With PatchGAN, we are able to reduce the size down to 14 * 14 * 1 after a few convolutional layers to extract the features. Through the research of the Pix2Pix team, the end result of the output layer, which represents 38 * 38 grid of the input layer was able to produce a very good result, and we will adopt the same methodology.



PatchGAN structure used in both Pix2Pix and CycleGAN's discriminator

Custom Backpropagation

Both GAN models use the generator and discriminator's loss to calculate gradients. To help keep track of the forward pass operations and then rewind the operation to calculate gradients automatically, we wrote a custom training step function using Tensorflow's GradientTape. The implementation is heavily influenced by the research paper as well as the implementation from Tensorflow main site[3].

Differences between Pix2Pix and CycleGan

Pix2Pix

Unlike CycleGAN, Pix2Pix employs the usual batch normalisation instead of instance normalisation in all of our sub models (generator and discriminator). Ideally, we could create a toggle within our sub models to choose the type of normalisation during model selection, however, we have decided to create 2 sets of similar models in each GAN's colab in order to make our code understandable.

Moreover, Pix2Pix chose to use L1 loss (Mean Square Error) to regulate the total loss during training. We expect the L1 loss to reduce overtime as an indication that our GAN is producing good results.

CycleGAN

Although CycleGAN proves to be a versatile model for style transfer, the biggest downside of it is the use of 4 sub models, specifically 2 sets of generators and discriminators. This is twice the size of a typical GAN architecture and it comes with the baggage of requiring a lot more memories and time during training. Due to this reason, it is common to see the default size used in other implementations during training to be 256 x 256 x 3 dimension. This is to balance between a presentable image translation and resource used, which we are also adopting. (As a side note,

[3] https://www.tensorflow.org/tutorials/generative/pix2pix

besides the resource constraint, another reason for the input size to be 256 x 256 is due to the use of unet256, which is also implemented in our models).

Furthermore, we have also implemented the resnet generator stated in the original research paper as a way to compare the performance between itself and the unet approach. Our results have shown that the resnet implementation took almost 50% longer time to train, and it produces slightly lower quality results. On top of that, we have to implement a custom padding layer called `ReflectionPadding2D` used in the original paper. Unfortunately, this complicates our conversion of the model into the javascript version, and pairing it with the performance hit, we picked unet as our main model in our final implementation. We will share the details of the evaluation in the **Evaluation** section later.

Another important difference from the Pix2Pix model is the use of instance normalisation (Ulyanov et al., 2017) in all the submodels instead of batch normalisation. The benefit of instance normalisation has been addressed in multiple research papers, including CycleGAN's paper and we will use the same setup. Unfortunately, we face the same conversion issue when it comes to tensorflow javascript conversion and we overcome it with a custom implementation. We will cover the details of the conversion later.

Lastly the most important feature used in CycleGAN is the Cycle Consistency loss function. In gist, the purpose of this loss is to ensure that the translated image can be as similar to the original image as possible. In this case it measures the similarity between the original image, image X (e.g. a horse) and the generated image, image X' (e.g. generated horse) that has gone through a conversion cycle between generator G and generator F.

All in all, the complete objective of CycleGAN is to minimise the loss between the generators and discriminators using Cycle Consistency loss. We also want to highlight that there is an additional loss called Identity loss function that is used to preserve the colour features in the input and output of the training data when colour accuracy is important. We tried omitting this loss in our training but found that the end results are more funky than artistic, hence we kept it in our solution.

Tensorflow Javascript Version

Our goal is to run our models in the local edge device and to do so, a javascript version of our models are required. Thankfully, we are able to convert our Python models into javascript models using TensorflowJS and 2 lines of code:

```
import tensorflowjs as tfjs
tfjs.converters.save_keras_model(generator, tfjs_target_dir)
```

We can then use the `tfjs` package to load the layers model in our web app through the following:

```
import * as tf from "@tensorflow/tfjs";
const model = await tf.loadLayersModel(<model url>)
```

However, we do face the issue of not being able to load our javascript models when using a custom layer. This is because TensorflowJS can only support predefined layers in Keras and our solution is to handcraft the javascript version of instance normalisation used in our web app. The

conversion was moderately challenging due to the lack of proper documentation, however, we managed to replicate it in our web app. In gist, our custom implementation has to load the trained weights correctly in order for the UNet generator to work normally. Our implementation can be found here.

We also faced performance issues when our web app is executing the model while we are drawing at the same time. To overcome this, we have created 2 web workers to execute the model in separate threads so that they do not block the main UI thread that is rendering the drawing. However, we did not completely eliminate the performance (i.e. lags) issue due to the fact that our models have to process a large number of parameters while displaying the end result. Our further investigation shows that we can improve our implementation by using OffscreenCanvas to render the image, but, this feature is only supported by a few browsers and we think that the current performance hit would not impact our MVP.

Evaluation and Performance

We adopt both quantitative and qualitative metrics in our evaluation. For quantitative analysis, we look at **losses over epoch** as our evaluation metrics because it gives us a sense of our model performances during training. This is especially helpful for us in the training phase because each training took a few hours and we have been experimenting with multiple model variations, as such, we can terminate the training early when the matric results are not ideal.

Moreover, we did a poll with a group of participants to ask for their opinions about our generated images. This qualitative evaluation method is an adaptation from the AMT evaluation stated in original papers, which helps to address the ‘goodness’ of the model (and the webapp). This is because GAN’s evaluation remains subjective and human interpretation is still the preferred method.

Loss Metrics

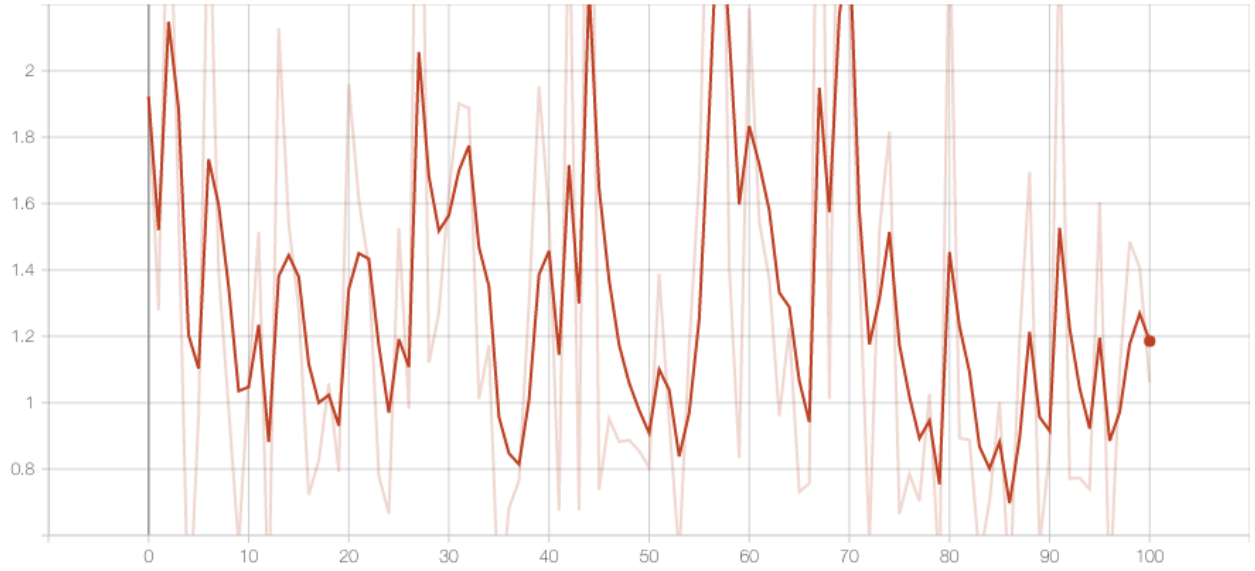
We are capturing the losses generated by the generators, discriminators as well as the objective loss functions used in each model (e.g. L1 loss for Pix2Pix). Intuitively, we should see that both generators and discriminators produce loss graphs that look random and do not converge, and this is because they are in a tug of war. This never ending challenge is expected and it gives us an indication that the model is performing as instructed.

Additionally, the object loss function should converge from high loss value to a lower value as we accumulate more epochs. This is because it tells us that the generated image is getting similar to the target image.

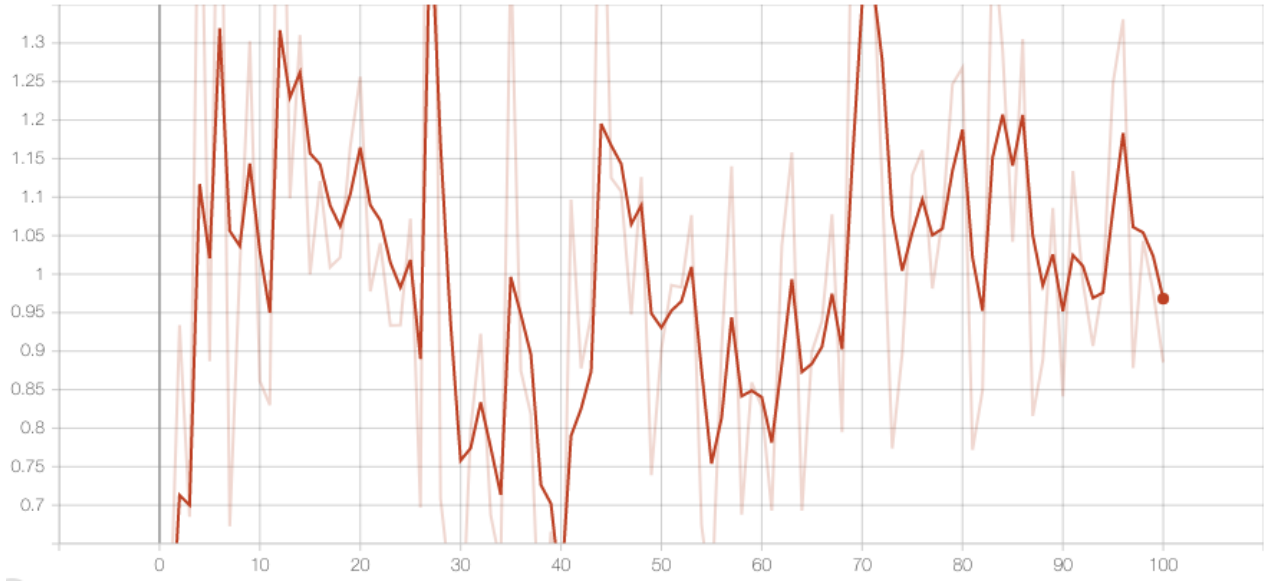
We have created a Tensorboard to visualise The full interactive logs can be found here:
`https://tensorboard.dev/experiment/V2JZmecORPO6aGoFVnn0Lg/`

Pix2Pix

The purpose of our Pix2Pix model is to convert user generated sketches into colourised versions. Our objective is then to evaluate the ‘accuracy’ of the colourisation during this phase.

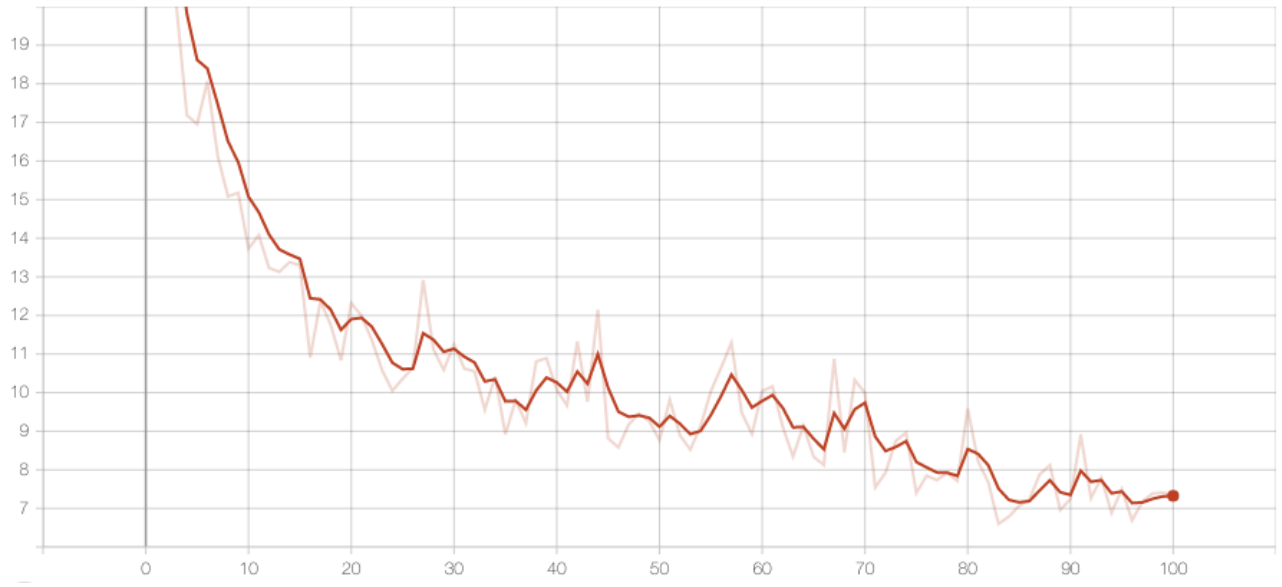


Generator loss (gen_gan_loss)



Discriminator loss (disc_loss)

We ran 100 epochs for our Pix2Pix training and we observed that neither generator nor discriminator is leading. This gives us assurance that our model is performing as expected.



Objective loss (gen_total_loss)

On the other hand, we can observe that the generator total loss (which factors in L1 norm) trend downwards and start to stabilise roughly after 85 epochs. This gives us the confidence that our training is successful.

Here are 3 snapshots of the outcome of our training:

Epoch 1

Epoch 20

Epoch 50

Epoch 80

Epoch 100

Visually, epoch 50 seems to produce respectable results while both epoch 80 and 100 are virtually indistinguishable. Unfortunately, there is no objective way to stop the training at a certain acceptable epoch due to the fact that the 'goodness' is subjective. The current best practice is to manually inspect the generated outcome and stop training as needed. This is especially important for resource intensive GANs like Pix2Pix and CycleGAN as each model takes hours.

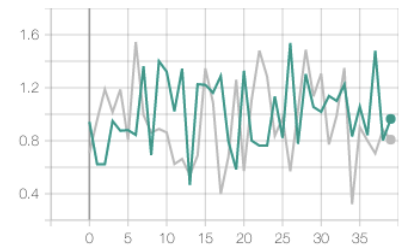
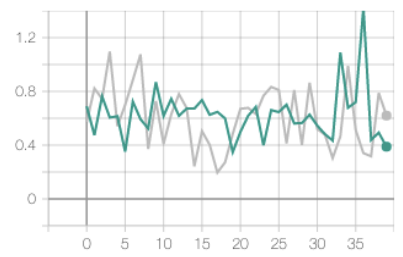
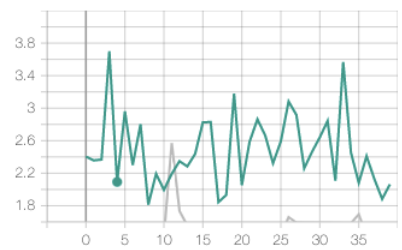
CycleGAN

The objective of our CycleGAN model is to convert the coloured drawing into an art piece. We compare our results among the 2 generators, resnet and unet, as well as the 3 datasets.

We would like to highlight that all the models are trained with the same amount of epochs (80 epochs each), however, we only managed to capture the data of the first 40 epochs for datasets `vangogh2photo` and `pokemon2vangogh`. We did not successfully capture the full 80 epochs for the affected datasets in subsequent attempts and we suspect that the long training time (about 4 hours per training) might have prevented colab in capturing the full history.

That said, it is still meaningful to display the losses here for this project purpose. (For brevity, we will only present the primary generator (generator_g) and discriminator (discriminator_x) as their counterparts have similar results.)

vangogh2photo

		
Generator loss	Discriminator loss	Cycle Consistency Loss

Grey line indicates model with unet whereas the green line represents model with resnet.

Both generator and discriminator perform as expected, however, we observe that the objective function, cycle consistency loss, has difficulty converging to a lower score. We also observe that the unet version of our model performs better than the resnet model.

Unet Model (vangogh2photo)

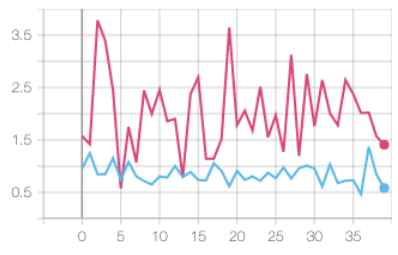
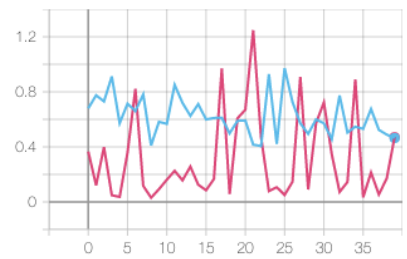
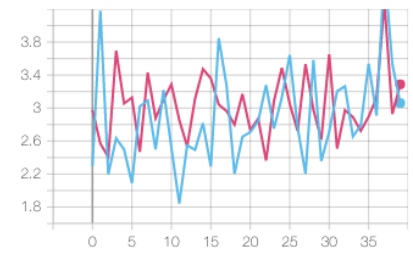
			
Epoch 1	Epoch 10	Epoch 20	Epoch 40

Resnet Model (vangogh2photo)

			
Epoch 1	Epoch 10	Epoch 20	Epoch 40

Similarly, the generated images from various epochs also show that the unet model yields better results.

pokemon2vangogh

		
Generator loss	Discriminator loss	Cycle Consistency Loss

Red line indicates model with unet whereas the light-blue line represents model with resnet.

In the `pokemon2vangogh` dataset, we see that our unet generator struggles with the discriminator more than the resnet implementation. However the cycle consistency loss also did not converge and it disprove our hypothesis that using pokémon and vangogh images as input help to generate a better model.

Unet Model (pokemon2vangogh)

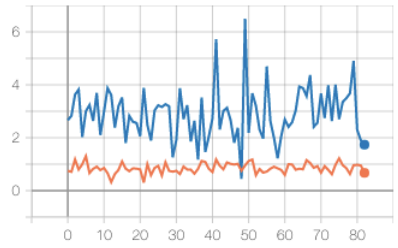
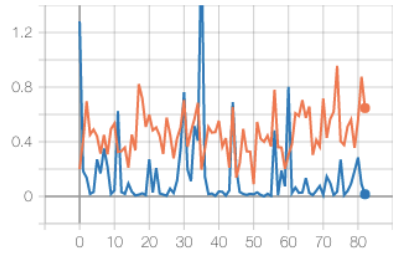
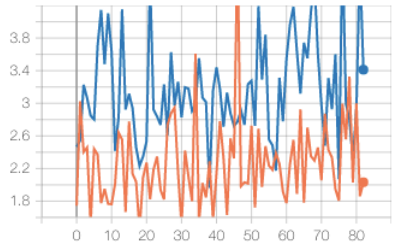
			
Epoch 1	Epoch 10	Epoch 20	Epoch 40

Resnet Model (pokemon2vangogh)

			
Epoch 1	Epoch 10	Epoch 20	Epoch 40

We can also visually see that both models did not perform any better with more epoch and it again validates the cycle consistency loss results. One theory that we suspect is that the pokemon training data has too many white spaces and the models struggle to map meaning values onto the same space. Unfortunately we did not manage to test out this hypothesis and it could be a future project topic.

pokemon2animals

		
Generator loss	Discriminator loss	Cycle Consistency Loss

Orange line indicates model with unet whereas the blue line represents model with resnet.

Similarly, we observe the same pattern from the `pokemon2animals` dataset as compared to `pokemon2vangogh` dataset.

Unet Model (pokemon2animals)

			
Epoch 1	Epoch 10	Epoch 20	Epoch 40

Resnet Model (pokemon2vangogh)

			
Epoch 1	Epoch 10	Epoch 20	Epoch 40

Visually, we can also draw the same conclusion from the `pokemon2vangogh` model evaluation and we can only conclude that appropriate datasets play an equally important value in training a better CycleGAN model. The definition of appropriate datasets remains open for debate and it could be a potential research project.

During the training and evaluation phase, we also observe that a unet based model trains at least 2 times faster than resnet based model and this is due to the simplicity of the model architecture. Given that the unet produces similar results in the last 2 datasets (`pokemon2animals` and

pokemon2vangogh), while showing better results in the baseline dataset, we pick all the unet based models as our candidate models to generate test results for our poll.

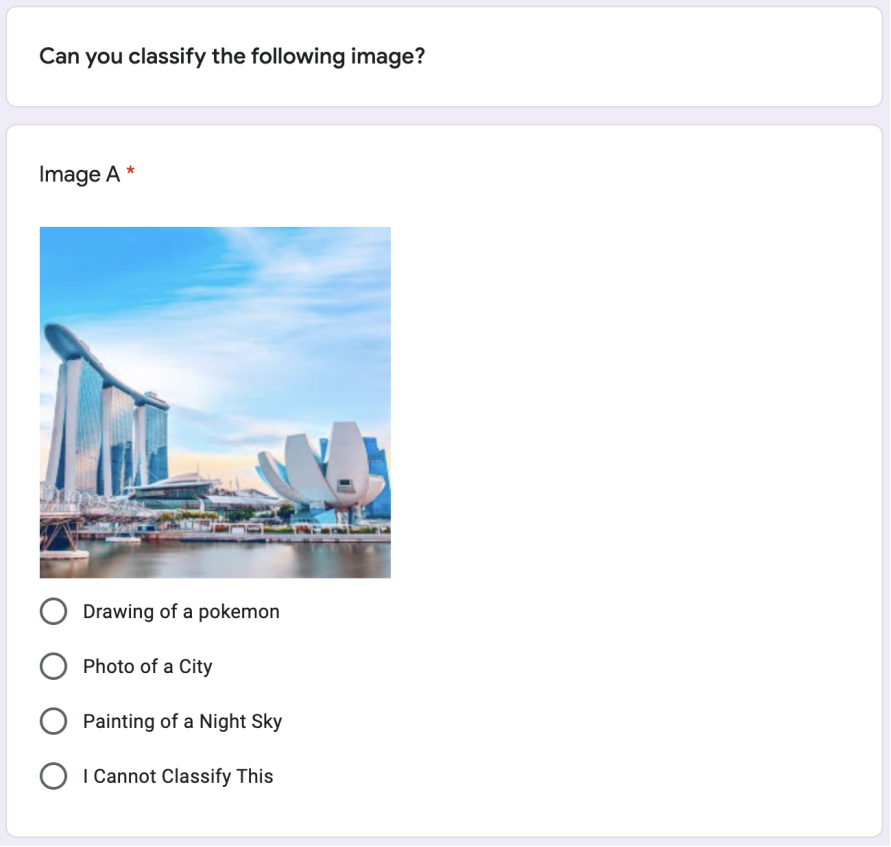
Poll Results

We structured our poll into 2 parts in order to validate the goodness of both our Pix2Pix and CycleGAN models. As of this writing, we have gathered poll results from a sample size of 38 participants and this is important because we will be using the Central Limit Theorem to project the mean of the population from the samples. As a rule of thumb, we have to collect at least 30 responses for CTL to hold. The poll questions can be found here: https://forms.gle/hzAecGS15UawsckP9

The poll results can also be found here: https://docs.google.com/spreadsheets/d/14PyUhSvSK1yOApFyBng9tZcyOf4AdwlWM-t749TyhcI/edit?usp=sharing

The first section asked our participants to classify 3 images into the right categories. What we did not tell them is that of the 3 images, one of them is generated by our Pix2Pix model.

Here is a sample of the first section:



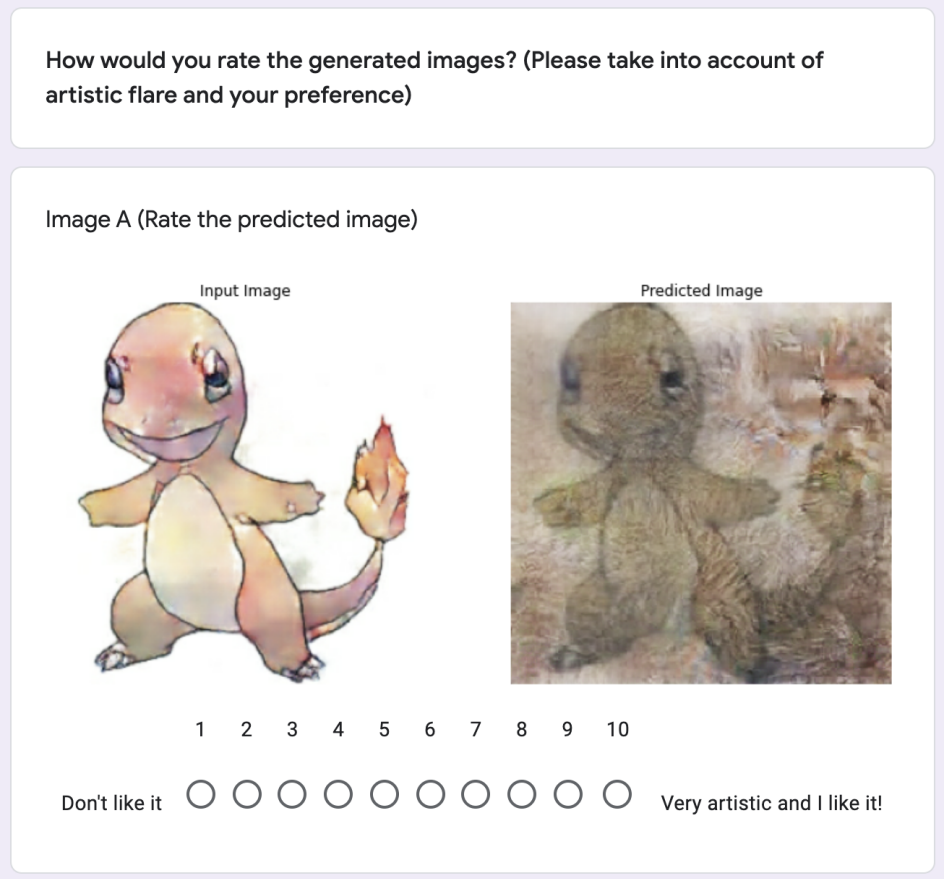
And here are our responses:

Image			
Type of photos	Actual photo	Generated by our model	Actual photo
Expected category	Photo of a City	Drawing of a pokémon	Painting of a night sky
Responses	97.4% correct	**100% correct**	100% correct

Our poll results overwhelmingly show that 100% of the participants can classify the generated image correctly (image B). This tells us that our Pix2Pix performs as expected.

In the second section, we went to test our CycleGAN performance. Because our loss metrics did not give a definitive answer, we turned to our participants for the outcome.

Here is a sample of our second section:



And here are our responses:

	Predicted Image A (pokemon2animals dataset)	Predicted Image B (pokemon2vangogh dataset)	Predicted Image C (vangogh2photo dataset)
Average score for CycleGan	4.538461538	5.692307692	**6.820512821**
S.D. for CycleGan	2.382459519	2.318732028	2.186944454

From our final poll, we can see that the model using baseline dataset (`vangogh2photo`) performs better than the 2 other models. This was not the result that we expected at the start of this project, however we can agree with the participants' responses after looking at the final generated images. We will be using this model in our actual implementation.

Conclusion and Learnings

We have learned a ton about Generative Adversarial Network in this project and it also opens up more questions than answers that are interesting to explore further. As we have come to the conclusion of this report, we would like to share our learnings here:

Learnings from Serving Phase

Here are 3 key lessons learned:

1. Generating images on a browser UI is doable but requires time for the canvas to load all 64000 inputs (400 x 400 x 4 per image) onto the screen at once, which locks up the UI thread. This is because the process of converting the generated array results from the model into tensors consumes most of the browser's resources. This means that the browser can hit the limit easily and we have to use more advanced techniques like web workers and offscreenCanvas to make an acceptable MVP.
2. Both javascript versions of Pix2pix and UNet models consume a lot of GPU and memory resources, hence making real-time transformation is very challenging. To improve the user experience further, we have implemented a debounce function that balances between real time rendering and model execution frequency.
3. TensorFlow.js Layers currently only supports Keras models using standard Keras constructs and not those with unsupported ops or layers such as custom layers, Lambda layers, custom losses, or custom metrics. This means that we cannot automatically import our GAN that uses our custom instance normalisation layer.

Learnings from Selling Phase (or Production Phase)

This is the simplest phase in our project, and it is also the most common application architecture used in today's production systems. This tells us that the AI technology (especially deep learning used here) is mature enough for business adoption.

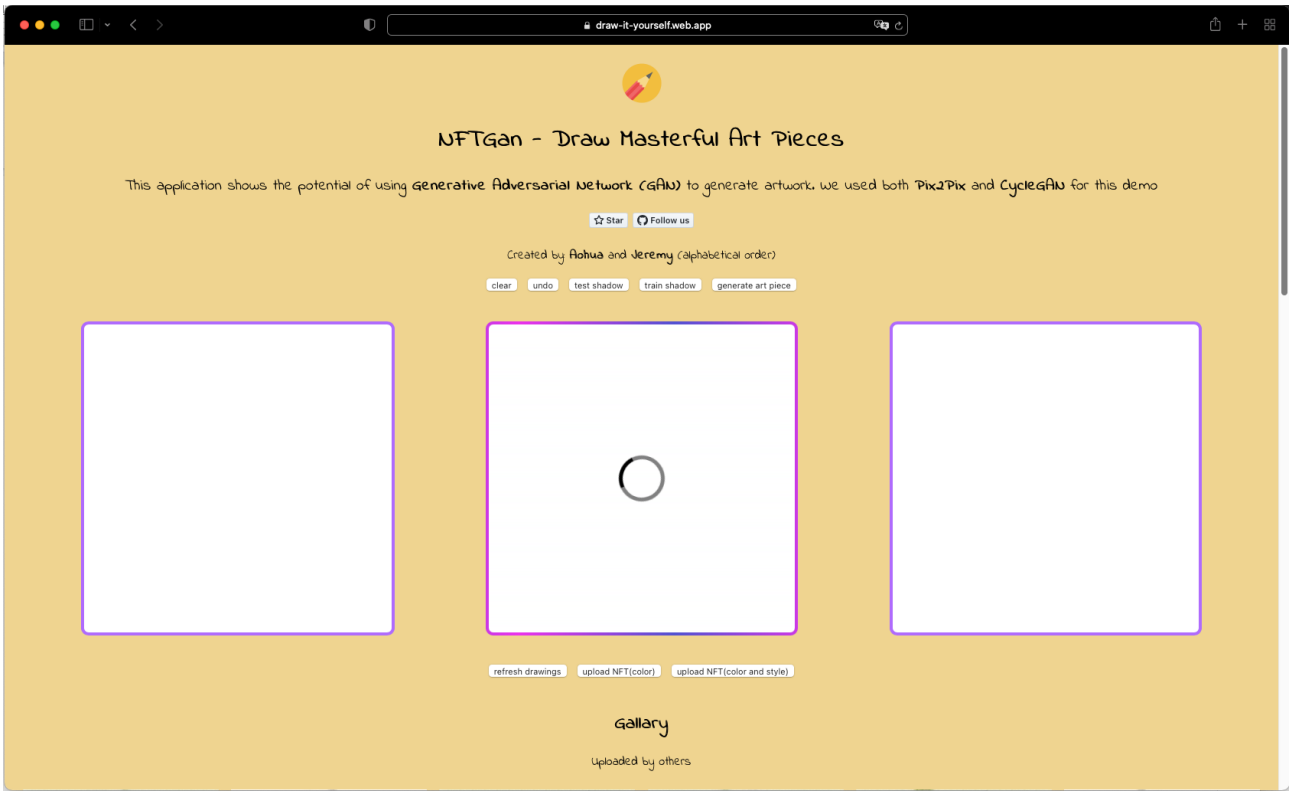
Learnings from Training Phase

1. The evaluation of GAN remains to be contextual and there are no standardised methods to quantify the results. We find that the best approach is to have human assessment on our GAN models with respect to the business or application objectives.
2. On a similar note, GAN training is resource intensive and using leading indicators like loss function can help us to stop training on bad performing models.
3. Datasets also play a huge role in the training phase, this project has shown that the dataset that we thought would train a better model did not perform any better than existing and well established datasets.

Finally we find that the concepts learned in this course help us to grasp key terminologies used in GAN easier and we are also able to experiment with various settings in our models.

References and Appendices

Our MVP can be found here: https://draw-it-yourself.web.app/



Our source code can be found in our github account:
https://github.com/aohua/nft-generator

Our references can be found here:

BuzzFeedVideo. (2018). *You Won't Believe What Obama Says In This Video!* You Won't Believe What Obama Says In This Video! https://www.youtube.com/watch?v=cQ54GDm1eL0

Demir, U., & Unal, G. (2018). Patch-Based Image Inpainting with Generative Adversarial Networks. https://arxiv.org/abs/1803.07422

DH Web Desk. (2020, Feb 21). Deepfake videos were used for the first time in India by BJP: Report Read more at: https://www.deccanherald.com/national/north-and-central/deepfake-videos-were-used-for-the-first-time-in-india-by-bjp-report-806669.html. https://www.deccanherald.com/national/north-and-central/deepfake-videos-were-used-for-the-first-time-in-india-by-bjp-report-806669.html

Isola, P., Jun, Y. Z., Zhou, T., & Efros, A. A. (2018). Image-to-Image Translation with Conditional Adversarial Networks.

Jun, Y. Z., Park, T., & Efros, A. A. (2020). Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. https://arxiv.org/abs/1703.10593

Ulyanov, D., Vedaldi, A., & Lempitsky, V. (2017). Instance Normalization: The Missing Ingredient for Fast Stylization. https://arxiv.org/abs/1607.08022