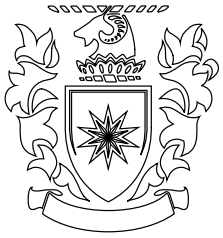


Feasibility of dashboard GUI design and optimization from machine learning

A thesis presented in partial fulfilment of the requirements for the degree of

Masters

in

Mechatronics Engineering

by
Daniel Foster

Supervisors:
Johan Potgieter
School of Food and Advanced Technologies
Massey University
Palmerston North, New Zealand

November 10, 2022

Abstract

Web dashboard Graphical User Interfaces (GUI's) are becoming ever more important for today's big data requirements. A number of industries from Internet of Things, to finance, to project management and everything in-between are producing enormous amounts of data and the need to display this data for users and internal employees in a useful way is incredibly necessary. This thesis attempts to answer the question; Is it feasible to design and optimize dashboard graphic user interfaces with machine learning?

To explore this question, we devised two experiments. One using full dashboard images, and the other using dashboard cropped components. Two datasets were built using images pulled off the internet and processed using four software scripts built for the purpose of this project. These included an image downloader, image duplicate detector, and a cropping tool designed to allow the user to crop sub-images from a single image rapidly while assigning class labels to the main, and cropped images. This resulted in two data-sets of 1024 dashboard images of size 512x512 pixels, and 5133 dashboard component images of size 256x256 pixels. We labelled these data-set images as one of eight class labels of different dashboard types. These types were IoT, agriculture, finance, analytics, miscellaneous, fitness, project management, and social.

Following this, a machine learning architecture was chosen based on an evaluation using a set of weighted criteria. The results of this evaluation settling on StyleGAN2-ADA developed by NVIDIA corporation. Chosen hardware was a purpose-built machine learning computer with two NVIDIA a-40 Graphics Processing Units (GPU's) which was able to perform at speed with the high memory and processing requirements of this architecture.

Initial training runs showed a mixing of colours and styles through all the generated images which was fixed by altering the architecture structure slightly. Subsequent parameter tuning experiments found that in built image augmentation methods of pixel blitting, geometry and colour transforms, along with a model parameter gamma value of 50 were useful in achieving better training results. The final two experiments using the two datasets that we created showed symptoms of model failure at a later stage in the training where the generated images were reduced in quality and almost identical. Image generations taken from a point in the training prior to this occurring showed patterns between the eight different dashboard class types. Although the image quality was not particularly good as training did not fully complete, we made a number of observations for full dashboard images. Bar and line graphs were extremely prevalent in most dashboards and found in similar positions (middle/top left), the algorithm favoured side menu dashboard format, and text was not readable in the generated images. Regarding dashboard components a number of classes suffered from low data numbers and generated purely text-based images, while others mirrored the components shown in the full dashboard image tests.

Acknowledgements

I would like to thank the team at Massey Agrifoods Digital Labs for offering me the opportunity to work on this project along with providing great facilities and equipment to conduct the experiments. I would also like to thank MAFDL for funding my tuition fees throughout this project. I would not have taken on a master's degree without the support provided.

Secondly, I would like to thank my supervisor Johan Potgieter for providing excellent guidance and support and pushing me to keep writing at all times as well as preventing me from disappearing down various rabbit holes throughout the project. Without this support I may not have finished this degree.

I would also like to thank Russel Wilson for taking the time to give me writing and general masters advice, Gabe Redding for providing machine learning advice and answering various questions, and Louise Hopkins for helping with anything pay and scholarship related.

I would also like to thank my family for supporting me through the degree and putting up with non-stop discussions about my master's topic. Thank you for the encouragement and support.

Finally, I would like to thank my partner Rose Duncan for supporting me in undertaking this project the past two years. I would not have been able to do this without you and I love you.

Contents

Contents iv

List of Figures ix

1 Introduction 1

1.1 Objectives . . . 2

1.2 Thesis Outline . . . 3

1.2.1 Chapter 2 — Literature Review . . . 3

1.2.2 Chapter 3 — Data Collection and Processing . . . 3

1.2.3 Chapter 4 — Material Selection . . . 3

1.2.4 Chapter 5 — Case Study . . . 4

1.2.5 Results and Discussion . . . 4

1.2.6 Conclusion . . . 4

2 Literature Review 5

2.1 Introduction . . . 5

2.2 Effective Dashboard Design Principals . . . 6

2.2.1 Data Selection . . . 6

2.2.2 Typical Dashboard Components . . . 7

2.3 Machine learning and Neural Networks . . . 8

2.3.1 Background . . . 8

2.3.1.1 Unsupervised Machine Learning . . . 8

2.3.1.2 Supervised Machine Learning . . . 8

2.3.1.3 Linear Regression . . . 9

2.3.1.4 Loss . . . 10

2.3.2 Classifiers . . . 10

2.3.2.1 Logistic Regression . . . 10
2.3.2.2 K-Nearest-Neighbour (KNN) . . . 11
2.3.2.3 Support Vector Machines (SVM) . . . 11
2.3.3 Artificial Neural Networks . . . 12
2.3.3.1 Activation Functions . . . 13
2.3.3.2 Back Propagation . . . 15
2.3.3.3 Batch Normalization . . . 15
2.3.4 Recurrent Neural Networks . . . 15
2.3.5 Convolutional Neural Networks (CNN) . . . 16
2.3.6 Residual Neural Networks (ResNET) . . . 17
2.3.7 Generative Adversarial Networks . . . 17
2.4 Existing Works Generating GUI Designs . . . 21
2.4.1 Face Off: Assisting in the Manifestation Design of Web Graphical User Interface . . . 21
2.4.2 GUIGAN . . . 22
2.5 Conclusion . . . 23

3 Data Collection and Processing 25

3.1 Image and Label Rules and Requirements . . . 25
3.1.1 Image Resolution . . . 25
3.1.2 Full Dashboard Image Content and Quality Requirements . . . 26
3.1.3 Dashboard Component Content and Quality Requirements . . . 27
3.1.4 Label Criteria . . . 27
3.1.5 Data Sources . . . 28
3.1.5.1 Google Images . . . 28
3.1.5.2 Dribbble . . . 29
3.2 Software . . . 29
3.2.1 Image Downloader . . . 30
3.2.2 Detecting and Removing Duplicate Images . . . 32
3.2.3 Multi-Crop Machine Learning Image Processing Tool . . . 33
3.3 Methodology . . . 41

3.3.1 Data Collection 41
3.3.2 Data Processing 42

4 Material Selection **45**

4.1 Hardware 45
4.2 Evaluation Metric Selection 47
4.3 Model Selection 47
4.3.1 Training Time 48
4.3.2 Conditional Training 48
4.3.3 Ease of Using Own Dataset 48
4.3.4 Performance on Small Datasets 49
4.3.5 Customization 49
4.3.6 Evaluation Metric Calculations 49
4.4 DCGAN 50
4.4.1 Architecture 50
4.4.1.1 Generator 50
4.4.1.2 Discriminator 50
4.4.2 Ranking Score 50
4.5 BigGAN 51
4.5.1 Architecture 51
4.5.2 Ranking Score 52
4.5.2.1 Training Time 52
4.5.2.2 Conditional Training 53
4.5.2.3 Ease of Using Own Dataset 53
4.5.2.4 Small Dataset Performance 53
4.5.2.5 Customization 54
4.5.2.6 Evaluation Metrics 54
4.6 FastGAN 55
4.6.1 Architecture 55
4.6.2 Ranking Score 57
4.6.2.1 Training Time 57

4.6.2.2 Conditional Training 57
4.6.2.3 Ease of Using Own Dataset 58
4.6.2.4 Small Dataset Performance 58
4.6.2.5 Customization 58
4.6.2.6 Evaluation Metrics 58
4.7 StyleGAN2-ADA 59
4.7.1 Architecture 59
4.7.1.1 Generator 59
4.7.1.2 Discriminator 60
4.7.1.3 Adaptive Discriminator Augmentation (ADA) 61
4.7.2 Ranking Score 62
4.7.2.1 Training Time 62
4.7.2.2 Conditional Training 63
4.7.2.3 Ease of Using Own Dataset 63
4.7.2.4 Small Dataset Performance 63
4.7.2.5 Customization 63
4.7.2.6 Evaluation Metrics 64

5 Case Study **66**
5.1 Data Collection 66
5.2 Data Processing 67
5.3 Training 68
5.3.1 Software Setup 69
5.3.1.1 Issues 70
5.3.2 Initial Experiments 70
5.3.3 Training 70
5.3.3.1 Code Additions 73
5.3.4 Hyper-parameter Tuning Experiments 75
5.3.5 Conditional Training 77

6 Results **81**

6.1 Initial Experiment Results 82
6.1.1 Initial Test Experiment 82
6.1.2 Hyper-parameter Tuning Experiments 83
6.2 Dashboard Components (256x256) 87
6.2.1 Training Loss and Scores 87
6.2.2 Discussion 90
6.3 Dashboards (512x512) 94
6.3.1 Training Loss and Scores 94
6.3.2 Discussion 95

7 Conclusions and Recommendations 100

Bibliography 104

a Dashboard Component Generations 110
I IoT 111
II Agriculture 112
III Finance 113
IV Analytics 114
V Misc 115
VI Fitness 116
VII Project Management 117
VIII Social 118

b Dashboard Generation Results 119
I IoT 120
II Agriculture 122
III Finance 124
IV Analytics 126
V Miscellaneous 128
VI Fitness 130
VII Project Management 132
VIII Social 134

c Image URL Extractor **136**

d Image Download Script **140**

e Image Duplicate Detection **143**

f MultiCrop Tool Source Code **146**

List of Figures

1 Design process for dashboard GUI 2

2 Examples of Typical Dashboard Widgets 7

3 Example of Linear Regression 9

4 Example of Gradient Descent in Linear Regression 10

5 Artificial Neural Network Feed Forward Network Structure. 13

6 Rectified Linear Units (ReLU) Activation Functions 14

7 Residual Block introduced in the ResNet model 18

8 High Level Overview of a Generative Adversarial Network. 19

9 GUIGAN Workflow [79] Copyright © 2021, IEEE 22

10 Structure of Google Images URL download script 30

11 Structure of Python image download script 31

12 Multi-Crop tool Graphic User Interface 34

13 BigGAN Architecture Showing a) Generator, b) Discriminator. 51

14 BigGAN Residual Blocks a) Residual Block Up Used in the Generator, b) Residual Block Down Used in the Discriminator. 52

15 FastGAN Generator a) Generator, b) SLE Block. 55

16 FastGAN Discriminator Flow 56

17 StyleGAN Generator Iterations a) Original StyleGAN generator proposed in [57], b) Revised Weight Demodulation StyleGAN2 Generator Model proposed in [65] 60

18 Residual Discriminator Model Used for StyleGAN2 61

19 **a)** Full Overview of Generated Images from the Final Training Snapshot of the First Experiment Using Cropped Dashboard Components. Examples of Generated Components Show What Appears to be a **b)** Calendar, **c)** Credit Card and **d)** a Graph 71

20 Second Initial Experiment: Style Mixing, Discriminator Residual Skip-Connections, and Path Length Regularization Disabled 75

21 (a) Generated IoT Dashboard (b,c,d) Generated IoT Dashboard Components . 79

22 Initial Experiment with Automatic Settings Using the Dashboard Component Dataset (top) Accuracy Scores of Real and Fake Data from the Discriminator (bottom) Model Loss of Generator and Discriminator Over Training Duration. 82

23 Hyper-parameter Tuning Experiment Loss (Top left) G and D loss for 50 Gamma using blit, geometry, and colour augmentations. (Top right) G and D loss for 50 Gamma using only blit and geometry augmentations. (Bottom left) Same above but with 0.5 gamma. (Bottom right) Same as above but using 0.5 gamma. 83

24 Hyper-parameter Tuning Experiment discriminator accuracy for various gamma and augmentation settings. (Top left) 50 Gamma using bgc augmentation mode. (Top right) 50 Gamma using bg augmentation mode. (Bottom left) 0.5 gamma, bgc augmentation mode. (Bottom right) 0.5 gamma, bg augmentation mode. 84

25 Generator and Discriminator loss trained on the main dashboard image dataset with different gamma values to confirm whether higher gamma values were more beneficial on the full dashboard images. (Top left) 0.5 gamma, (Top right) 50 gamma, (Bottom left) 100 gamma. All using bgc augmentation mode. 85

26 Discriminator accuracy on real and fake images using different gamma levels trained on the main dashboard image dataset. (Top left) 0.5 gamma, (Top right) 50 gamma, (Bottom left) 100 gamma. All using bgc augmentation mode. 87

27 Training Statistics for Dashboard Component Training (top) Generator Loss vs Discriminator Loss plotted over the duration of training (bottom) Discriminator Accuracy on Real dataset Images and Fake Generated Images 88

28 Training Checkpoint Output at 8628 kimg Iterations Showing Several Similar Images/Colours Indicating Possible Mode Collapse 89

29 KID score for training duration of dashboard component dataset. 89

30 Class label spread for dashboard components. Graph representation. 90

31 Training statistics for full dashboard images (top) Generator loss and Discriminator loss vs training duration (bottom) Discriminator accuracy on real dataset images and fake generated images. 94

32 Final training snapshot (6935 kimg iterations) image generation before termination due to GAN failure. 95

33 KID scores for full dashboard images for full training duration. 96

34 Class label spread for full dashboard images. Graph representation. 97

35 Twelve image generations for IoT dashboard components 111

36 Twelve image generations for finance dashboard components 112

37 Twelve image generations for finance dashboard components 113

38 Twelve image generations for analytics dashboard components 114

39 Twelve image generations for miscellaneous dashboard components 115

40 Twelve image generations for fitness dashboard components 116

41 Twelve image generations for project management dashboard components . 117

42 Twelve image generations for social media dashboard components 118

Glossary

ADA Adaptive Discriminator Augmentation. 59, 61

ANN Artificial Neural Network. 12, 13

CNN Convolutional Neural Network. 16, 17

convergence The point in GAN training where the training ceases to improve and begins to generate progressively worse images. . 48

CPU Central Processing Unit. 45

CUDA A parallel computing platform which allows software to directly utilise the graphics processing unit for general purpose processing. . 45–47, 69, 70

GAN Generative Adversarial Network. 5, 17–22, 46, 47, 49, 50

GPU Graphics Processing Unit. 45–48

hyperparameter In machine learning hyperparameters are used to control the learning process and fine tune the algorithm. . 49

kilo-image A measure of 1000 individual images. . 46, 47

NVIDIA NVIDIA corporation is a technology company which primarily focuses on the design of graphics processing units. . 45, 46, 57

overfitting In GAN training, overfitting is where a model begins to produce images that fit too closely with examples from the real set of data. . 49

px Pixels. 25, 45, 46

Pytorch An open-source machine learning library developed by Meta's AI research lab. 50

RNN Recurrent Neural Network. 15

shell script A computer program designed to run in the UNIX shell. 54

VRAM Video Random Access Memory. 46

CHAPTER 1

Introduction

A Web Dashboard is a type of graphical user interface that provides data visualization and/or control capabilities for users. Technology is rapidly advancing, it is estimated that everyday 2.5 quintillion bytes of data are created with 1200 petabytes of data shared and stored by major digital companies such as Facebook, Twitter, Google, and Amazon alone [80]. This "big data" movement has become a common part of life from the rise of the Internet of Things which sees thousands of homes and workplaces equipped with smart devices and sensors, to the finance sector which has seen 250 publications in the big data space in 2020 alone [74], and many others. With this large amount of data spread over an ever-growing number of industries, it has never been more important to find ways to easily display this to users. Typically, GUI dashboard development follows a number of steps from the initial problem definition to the actual deliverable solution which can be time consuming.

This process, shown in fig. 1 particularly the prototype design step, may be difficult for engineering companies who may have a product, but do not have the staff on hand with the necessary skills to design a dashboard. This may lead to more time spent on product development and therefore a slower transition to sales, and in a competitive big data/IoT space, time spent on product development is critical. The aim of this project is to assess whether this prototype design step can be removed altogether using machine learning. Which leads to the overall thesis question:

Is it feasible to design and optimize dashboard graphic user interfaces with machine learning?

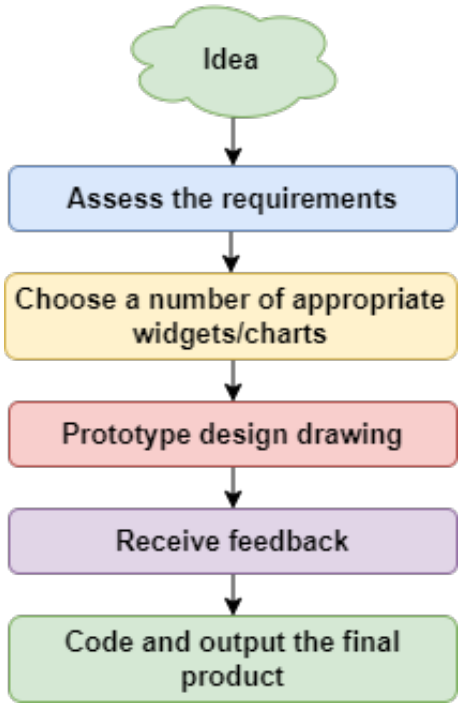


Figure 1: Design process for dashboard GUI

To answer this question, we devised two machine learning experiments. The first experiment focused on the generation of full dashboard images using a labelled data set, while the second focused on the feasibility of generating individual dashboard components of a specific dashboard type. This is a basic evaluation of the feasibility of the method.

1.1 Objectives

To evaluate this hypothesis, a number of objectives must be met:

1. Conduct research into currently used machine learning methods for image related problems and investigate if others are working in machine learning web GUI design.

2. If necessary, Build a dataset of dashboard GUI images. A set of criteria should be created to vet these images and ensure the correct ones are used.

3. Build a dataset of individual dashboard components to assess the feasibility of generating and arranging separate components.

4. As mentioned previously there are a large number of industries in need of dashboard GUI design. Dashboard and component images should be labelled to reflect the class they belong to.

5. Select a suitable machine learning architecture to generate dashboard designs.

6. Evaluate the trained results. If necessary, conduct more experiments with tuned parameters.

7. Supply findings and recommendations for future research and further development in this area.

1.2 Thesis Outline

1.2.1 Chapter 2 — Literature Review

The literature review section provides a background to current dashboard GUI design practices, and machine learning methods. We discuss Machine learning theory, and methods used to learn image data, from the basic methods to more advanced methods. We discuss Current research in machine GUI design and identify gaps in the research.

1.2.2 Chapter 3 — Data Collection and Processing

This section describes the process of collecting and processing data to build the datasets required in this project. The first section outlines the criteria used to vet images discussing image resolution requirements, content and quality requirements, image label classification requirements, and the data sources used. The second section outlines the software created to retrieve and process the images. The third section demonstrates the method used to collect and process images.

1.2.3 Chapter 4 — Material Selection

This section details the process of selecting hardware and software for this project. We select hardware from a list of available devices. The software selection is a much more involved process. We describe criteria to select an architecture in this chapter and evaluations between a number of different candidates are discussed with an eventual winner being chosen for use in the project.

1.2.4 Chapter 5 — Case Study

To prove the full workings of the solution for this project, a case study was undertaken to generate a new IoT dashboard. This section shows the method behind the experiments as well as discussing any issues that arose while conducting the experiments and how these were solved. Brief results from initial experiments are touched on in the context of showing how issues were solved but the main results are delivered in depth in the following section.

1.2.5 Results and Discussion

The results section discusses the findings from the initial test experiments followed by the main two experiments with the dashboard and dashboard component image sets. Training statistics are displayed for all experiments as well as an analysis of the generated images after training with any patterns being found between different classes of dashboard.

1.2.6 Conclusion

This is the concluding chapter and discusses the decisions made in the project, summarises the findings and presents four recommendations for moving forward.

CHAPTER 2

Literature Review

2.1 Introduction

This literature review looks to provide the reader with sufficient knowledge in two main topics. The first of which is a brief description of the current best practices in dashboard design. The purpose of the second topic is to provide a background into machine learning methods from the basics of classifiers all the way to the much more advanced generative adversarial networks typically used for image generation. Understanding of the workings of machine learning all the way from basics to the advanced are required to understand image creation in machine learning and the following methods will be discussed in relation to this project:

- Supervised and unsupervised learning types.
- Linear regression and basic cost function/loss theory.
- Classifiers including logistic regression, k-nearest-neighbourm and support vector machines.
- Artificial Neural Networks including activation functions, basic neural network structure, Recurrent Neural Networks (RNN) and Convolutional Neural Networks (CNN).
- Generative Adversarial Network (GAN)s.
- Evaluation metrics for Generative Adversarial Networks for the purpose of quantifying generated results.

The review will conclude by discussing the current research in the area of GUI generation as well as identifying potential gaps in the research.

2.2 Effective Dashboard Design Principals

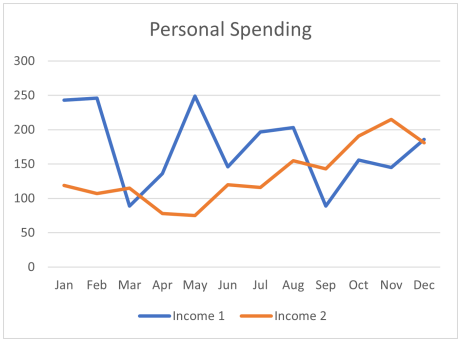
For machine learning, data-set quality is essential. If there are inadequate quality images in the dataset the output will suffer. Therefore, when selecting dashboard images, it is important to know the theory of good dashboard design. This section will cover essential dashboard design theory including data selection, typical dashboard components for different dashboard examples, and typical element colour selection.

2.2.1 Data Selection

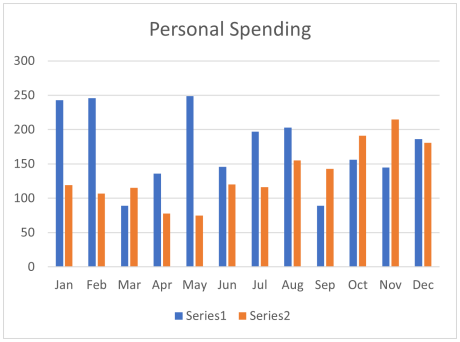
The main job of a dashboard is to visualize data for a specific purpose. A dashboard loses usability when it shows irrelevant data, meaning the dashboard becomes overcrowded and therefore the data loses meaning. [19] suggest constructing a measurement model to find what data is collected and the reason it is collected which then allows the designer to choose if it is relevant to display. Different types of dashboards will display different kinds of data for example:

- A finance dashboard may display a list of recent transactions or contain a line/bar graph of spending history or have a list of credit cards such as popular UK money management app "Money Dashboard" [84].
- A Social media dashboard may display information on subscribers and interactions on your social media accounts.
- A fitness dashboard might have data on earlier workouts or future scheduled workouts as well as other health metrics such as heart rate, calories burnt and number of steps such as the Fitbit web dashboard [83].

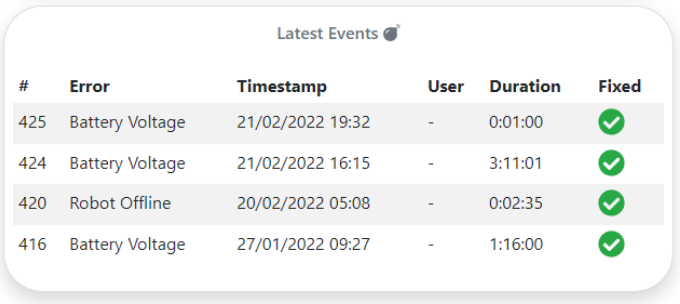
This is an important thing to consider when devising labelling criteria for dashboard images. Images chosen must display relevant data based on the type of dashboard while not crowding the display area.



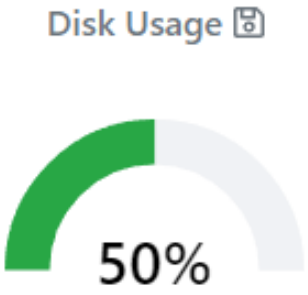
(a) Basic Line Graph



(b) Basic Bar Graph



(c) Basic Dashboard Table



(d) Basic Radial Progress Bar

Figure 2: Examples of Typical Dashboard Widgets

2.2.2 Typical Dashboard Components

To display relevant data, a dashboard must make use of various components, sometimes referred to as "widgets". Five common widgets used in dashboard development are:

- Line Graphs
- Bar Graphs
- Progress Bars and Dials
- Lists
- Tables

Line and Bar charts are effective at allowing the user to quickly compare data and show patterns [12], while progress bars and dials show real time statistics and accumulated values and are typically well suited for Internet of Things applications [77]. Tables are a well-known data presentation method for displaying substantial amounts of data, and lists are used for purposes such as a to-do list, or a list of latest events.

When finding image data to form a dataset these points should be considered. Therefore, based on this research it seems a set of rules about what denotes an acceptable dashboard should be created for this project including defining categories of dashboard based on the different data they present and the different widgets on the dashboard.

2.3 Machine learning and Neural Networks

Machine learning is an area of computer science which focuses on two things; How can a computer system be constructed which learns and self-improves through its experiences, and what statistical methods closely map the structures of natural learning in humans and organisations. Machine learning as a field, attempts to answer these questions while simultaneously providing practical software across many fields and industries [26]. Such fields and industries include finance [38], microbiology [60], network security [3] and even recently, fault diagnosis in machines [66]. In this section, theory will be discussed to provide the reader with a basic understanding of the topic, before discussing different currently used image classifiers and neural networks to find the best method for learning and generating dashboard designs.

2.3.1 Background

2.3.1.1 Unsupervised Machine Learning

Unsupervised machine learning is the process of programming a computer to learn without previously labelled examples. This method does not typically use labelled datasets as used in supervised learning [11]. Instead of labelled datasets, the algorithm learns from live data and extracts patterns. Pattern extraction uses a type of reward system for the computer where the goal is to maximize rewards by correctly finding patterns.

2.3.1.2 Supervised Machine Learning

Supervised learning is a method where the learning algorithm extracts meaningful information from a labelled data training set which a human provides [42] [26]. This training set will contain various versions of what the algorithm is programmed to search for and will contain a large number of items to achieve accuracy. Multiple areas of literature out-

line a set of steps performed in supervised machine learning systems [71] [42]. These are as follows:

1. The user must establish the type of training examples.
2. Collect training set data from various sources and build the training set.
3. Label the data on each input in the training set. The accuracy of the algorithm depends on the accuracy of the data labelling at this stage.
4. Choose a machine learning model based on the desired outputs.
5. Execute the learning algorithm on the training data set.
6. Evaluate the accuracy of the model and perform any parameter tuning required.

2.3.1.3 Linear Regression

Linear regression uses a mathematical approach to predict outcomes from the given data through a relationship between a dependent variable (y) and independent variable(s) (x). This method dates to 1894 when Sir Francis Galton first suggested it [68] and is the standard scatter plot graph with a regression line that most are familiar with. Linear regression can be thought of as the simplest artificial neural network, consisting of a single neuron [78].

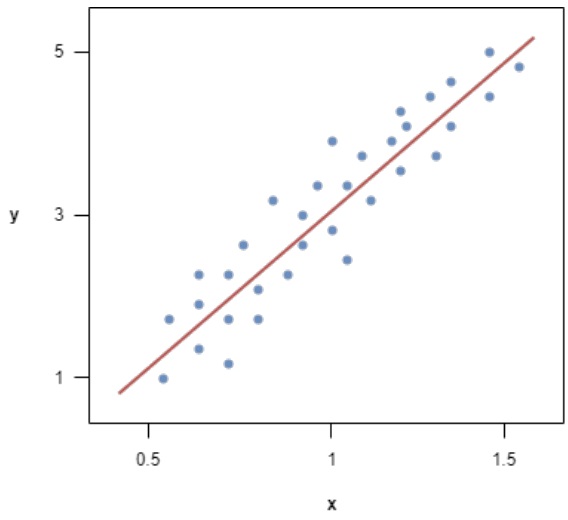


Figure 3: Example of Linear Regression

2.3.1.4 Loss

Cost functions are fundamental to machine learning and can easily be explained using the linear regression model. Learning from the data is the main objective of a machine learning algorithm. To learn correctly they must be provided with feedback on the success of the learning; this is the purpose of a cost function. For the model to perform to a high standard, the cost function must be minimised. This is achieved through gradient descent. Gradient descent is one of the most popular methods of reducing error in machine learning [33]. A basic example of gradient descent in linear regression is shown in fig. 4 where the progress of the machine learning algorithm is assessed at each stage using gradient descent. The parameters are automatically changed based on the error results before the next iteration leading to reductions in error. This allows the machine learning algorithm to progress towards convergence [41].

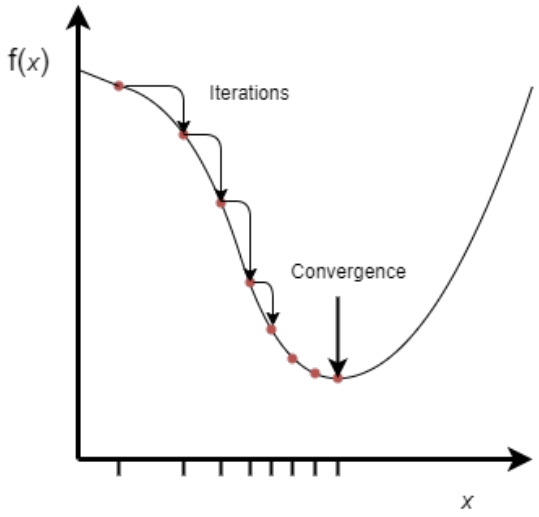


Figure 4: Example of Gradient Descent in Linear Regression

2.3.2 Classifiers

2.3.2.1 Logistic Regression

While linear regression is used for prediction and forecasting of future values, logistic regression is used for classification purposes [6]. Logistic regression in its simplest form, binary logistic regression, uses the Sigmoid function where data is re-mapped to a value between zero and one.

A probability function is used to determine if the data fits a specific requirement (i.e., is the object in the image a cat or not a cat). If the probability is approaching one, the assumption is true. If it is approaching zero, then the function deems it to be false. The Sigmoid function is as follows:

$$S(t) = \frac{1}{1 + e^{-}t}$$

Gradient descent is used as in linear regression to reduce the error of the algorithm as it trains. Multi class separation is also possible using Multinominal [13], and ordinal [9] logistic regression.

2.3.2.2 K-Nearest-Neighbour (KNN)

K-Nearest-Neighbour is a simple method of classification which classifies data points by observing the class of their nearest neighbour. To find the distance between points and identify the nearest neighbour, this method utilizes one of multiple available distance calculations, one of which being Euclidian distance which can be calculated given two vectors x and y:

$$D(x,y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Two uses of this simple classification method include, bank credit risk analysis [24], and even image classification [15].

2.3.2.3 Support Vector Machines (SVM)

Hyperplanes are one of the core components of Support Vector Machines. The definition of a hyperplane is a decision boundary between two classes of data points. Hyperplanes can be viewed on a 2d graph as a simple line, or on a 3d graph as a plane.

The purpose of support vector machines is to classify between data classes. The main task of a support vector machine is to calculate an appropriate hyperplane to separate two classes of data on a graph. Data points are mapped to high dimensional feature space in a binary classification problem where there exist two different sets of labelled data on a graph and there is an infinite number of planes that correctly separate the data. The goal of the training is to find the plane with the greatest margin on either side between

both sets of data [4]. After this optimal classification plane has been found, training is complete and future, unlabelled data can be evaluated on which side of the hyperplane it lands. Typical areas Support Vector Machines have been used in the past include face and eye detection [8], text detection in video [5], and image classification [73].

The above-mentioned methods are ideal for classifying data. For this problem however, simple classification is not enough as dashboard images have a large set of features and variability. Because of this, these methods have been deemed impractical for the task at hand. It is believed that neural networks will be required as they tend to perform better on image data and allow for a deeper learning through much more complicated multi-neuron learning algorithms.

2.3.3 Artificial Neural Networks

Artificial Neural Networks (ANN) are widely used in machine learning for image and pattern recognition [16] [46] [7] [61]. Because of this, neural networks have been determined to be of significant use to this project which requires a great deal of image and pattern recognition. There are multiple types of Neural Networks including Convolutional Neural Network (CNN), and Recurrent Neural Network (RNN). Two neural networks may also work against each one another in competition to form a GAN. These different deep learning methods will be discussed and evaluated in the following section.

Artificial Neural Network (ANN) are a popular model for classification, pattern recognition, and prediction and in terms of usefulness has become competitive to the use of regular statistical modelling and regression methods [44]. An ANN is the emulation of a biological human brain. The human brain typically consists of two main components: Neurons, and Synapses [22]. Where neurons are responsible for receiving sensory input from external sources, and synapses are the connectors between them. ANN works to target specific processes in the human brain, as emulating the entire human brain would be far too complex. According to Dave & Dutta [22] Problems where ANN perform well include the following:

- Problems which involve an association between a set of patterns.
- Problems with large and diverse datasets.

- Problems where conventional processes struggle to describe the relationship between data points in a set.
- Problems where the number of variables is quite large.

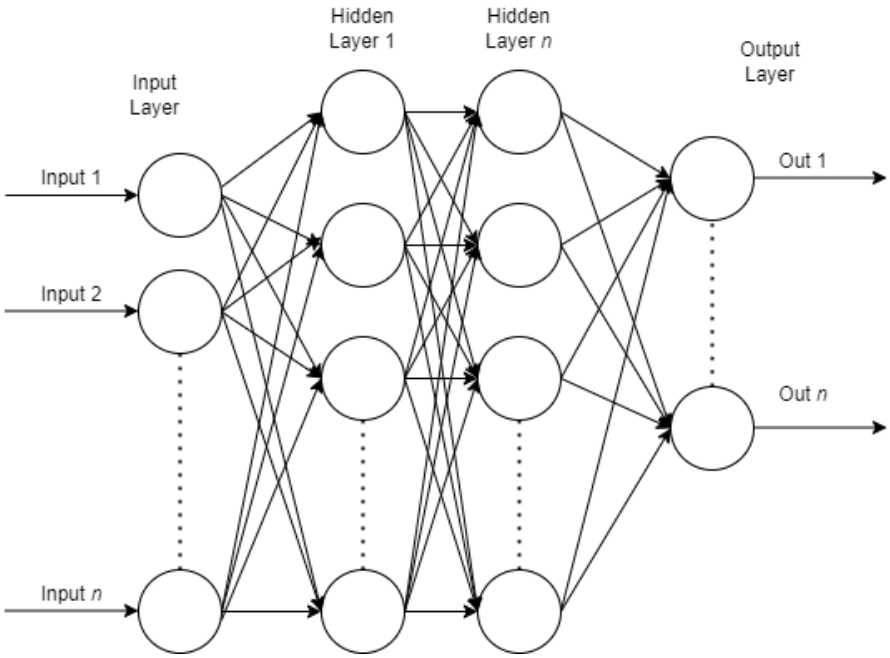


Figure 5: Artificial Neural Network Feed Forward Network Structure.

The human brain is emulated in these large computational networks by using nodes (Neurons) and weighted connections (Synapses). Each node processes the input from the previous node prior to passing it through a typically non-linear activation function to generate a scalar output with one numerical value. Nodes are arranged in layers, starting with the input layer, followed by a number of hidden layers of multiple varied sizes, and ending with an output layer where the result is passed. Nodes are aware of only the information generated in themselves or passed to them through weighted connections to other nodes [14] [43].

The sections that follow detail useful background learning on how ANN function, discussing activation functions, back propagation, and batch normalization as fundamental components of these networks.

2.3.3.1 Activation Functions

Neural networks are known for their ability to compute any function given. To accomplish this, nodes should use non-linear activation functions to process data [43]. Acti-

vation functions are responsible for deciding whether a neuron should "fire" data to the next set of neurons based on its computed output. Each neuron in a layer will typically have the same activation function. There are a number of activation functions available for nodes to compute their output, with four relevant examples being:

- Linear
- Sigmoid function
- RelU
- Leaky ReLU

Linear Activation Function:

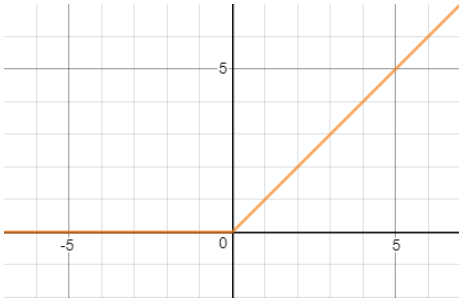
$$F(x) = ax$$

As mentioned above, it is ideal to have a non-linear activation function. However, problems which require inseparability and other simple problems can benefit from a linear activation function [43].

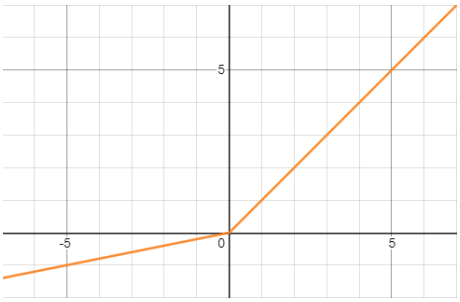
ReLU (Rectified Linear Units):

$$y = max(0, x)$$

Rectified Linear Units (ReLU) is a thresholding activation function which sees an output of zero for all values less than 0, and a linear function output when it is greater than or equal to zero [45].



(a) ReLU Activation Function



(b) Leaky ReLU Activation Function

Figure 6: Rectified Linear Units (ReLU) Activation Functions

Leaky ReLU:

$$f(x) = \begin{cases} \alpha x, & \textbf{for } x < 0 \\ x, & \textbf{for } x \geq 0 \end{cases}$$

Leaky ReLU is a slightly different version of ReLU where a small non-zero gradient exists for input values below zero [20]. The coefficient of this slope is predetermined and not learnt during training.

2.3.3.2 Back Propagation

While nodes are a particularly important part of neural networks, they only work together with the help of weighted connections. The algorithm applies weights to the input data at each node in the hidden layers of the network and these weights are responsible for the overall loss in the algorithm output. To minimise loss, weights must be adjusted, and this is achieved by comparing the real output data with the calculated model prediction data. The difference between real outputs and predicted outputs are fed back through the layered structure of the network with the goal of adjusting the connection weights to reduce the overall loss of the model; this is known as back propagation [1].

2.3.3.3 Batch Normalization

In neural networks batch normalization is used to allow for higher learning rates, accelerate training, and to improve generalization accuracy. It is a method that is applied in the deep intermediate layers of the model and is used to avoid activation explosion (a runaway gradient when training which causes training to collapse) by correcting all activations to unit standard deviation and zero-mean. The formula is as follows:

$$O_{b,c,x,y} \leftarrow \gamma_c \frac{I_{b,c,x,y} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}} + \beta_c \quad \forall\, b, c, x, y$$

Where $\mu_c = \frac{1}{|B|} \sum_{b,x,y} I_{b,c,x,y}$ equals the mean activation. Where B contains all activations in channel c, b is all features in the entire mini-batch, x, y are spatial locations, and γ_c, β_c are both learned during training [50].

2.3.4 Recurrent Neural Networks

Recurrent Neural Network (RNN) are a type of artificial neural network which is often used for tasks such as speech recognition [18], language processing and translation [21]

[17], or even image generation, which is highly relevant to this project [25]. Well known examples of Recurrent Neural Networks are Apple's Siri, and Google Voice [58]. Recurrent neural networks are used for data which involves sequences or time series data rather than independent data points. Recurrent neural networks utilize internal loops in their structure where current states of the network are affected by both their current input and their past inputs [48]. Although there have been cases of using this method for image generation. This image generation is usually sequence based such as generating images based off lines of text [72] which does not particularly fit within the scope of this project where we are trying to learn and output a set of dashboard design patterns based on the type of dashboard. Therefore, RNN have been deemed of potential future use for this project, but not entirely necessary for use in this thesis.

2.3.5 Convolutional Neural Networks (CNN)

Convolutional Neural Network (CNN) are a type of neural network that are typically used for image processing [69] [55], image detection and recognition [52], facial recognition [2], audio recognition [28] and other analysis applications. The interesting aspect of these neural networks is the image processing, detection, and recognition.

As the name suggests, convolution plays a crucial role in this type of network. In this explanation of image convolution, an image of size 512x512 pixels with three colour channels (RGB) will be used as this is the size of a small dashboard image that was show sufficient detail. If the model received an image as raw pixels, the input layer connection to a single neuron would have 786,432, (512x512x3) individual weighted connections. If an additional single neuron were added this number would increase to 1,572,864 (512x512x3x2) which is a substantial number of connections for only two nodes. The actual number of weighted connections required for the network for the image is:

$$N = (512 \times 512 \times 3) \times (512 \times 512)$$

$$N = 206,158,000,000$$

This is an exceptionally large number of connections and would take a long time to

process. To solve this issue, convolutional neural networks utilize a weighted kernel of a specific size (3x3, 5x5, etc..) to convolve the entire image by adding pixel values with their local neighbours [31] [36]. Using a kernel of 5x5 pixels, the number of connections is reduced by over two hundred billion for the full image to approximately 75 per kernel pass. The kernel is slid across the entire image and features are mapped based on this kernel, therefore features can be learnt from an image independently from their position [36].

2.3.6 Residual Neural Networks (ResNET)

While conventional CNN architectures perform well on image data, they tend to see an increase in test error and decrease in test accuracy as the network depth grows [32] [70]. Initially this was attributed to over-fitting or the exploding/vanishing gradient issue, but eventually it was proven that there was an issue with degradation in very deep neural networks causing them to be outperformed by shallower networks. To overcome this issue, Residual Neural Networks were introduced [32]. Residual networks introduce the residual block (fig. 7) which allows for a skip connection to add the input from a previous layer to the next layer. This is achieved by adding the original input to the neuron to the output function produced when the weightings and activation functions are applied to that same input. This allows networks to extend to a great depth, up to one thousand layers as recorded in [32] without suffering in test accuracy compared to shallower networks for the same problem. It is hypothesised that a deep neural network model will be required for this project and therefore ResNet is something that should be considered.

2.3.7 Generative Adversarial Networks

While the previous neural networks demonstrate the ability to detect features in images, for this project we are required to generate completely new images. First proposed in 2014 [23], GAN achieve this very well through the competition of two neural networks. Because convolution is a particularly effective way of speeding up computation of image-based datasets a number of GAN architectures incorporate it in their architectures. Three examples of these networks that utilize convolutional operations are: DCGAN [29], StyleGAN [57], and SAGAN [62]. Generative Adversarial Networks are considered as either

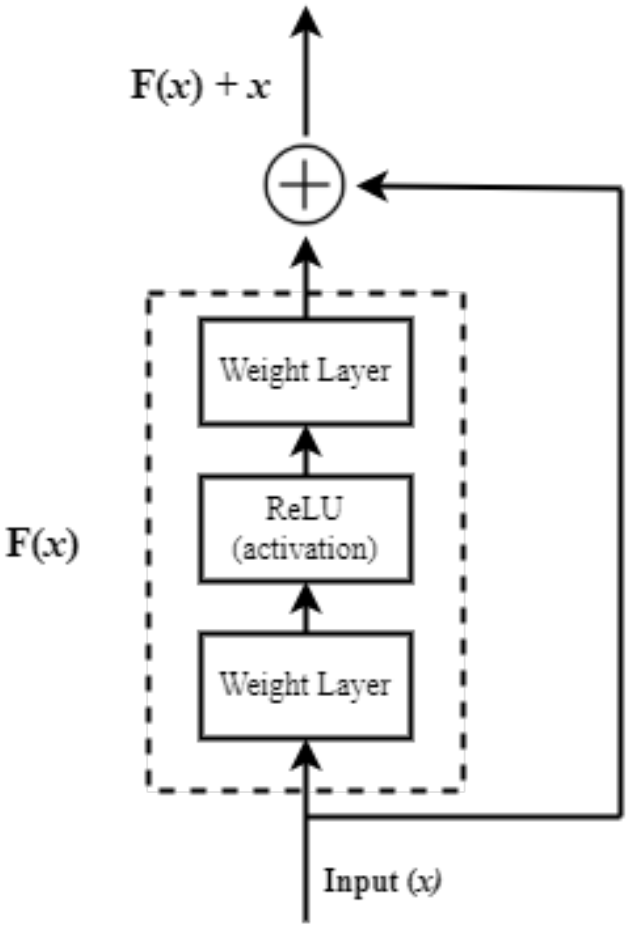


Figure 7: Residual Block introduced in the ResNet model

semi-supervised, or unsupervised machine learning and typically do not require extensive labelling on images to learn features. The network consists of a generator and a discriminator, and these can be thought of according to [53] as an art forger and an art expert. The generator is the forger and the discriminator is the art expert. During the training process, the generator attempts to make image forgeries with no knowledge of the real image data. These images are then used as an attempt to fool the discriminator; however, the discriminator is equipped with a set of real images and is therefore an expert on how images of this specific type should appear. The discriminator then informs the generator whether the images appear like they belong in the real set of images by advising on the current loss/error, leading to the generator to attempt again with this added information. The competition concludes when the generator learns to produce images that the discriminator cannot distinguish from the real ones.

Failure Points: GAN's can be notoriously hard to train, and training may result in failure due to several reasons. The most common types of failures are:

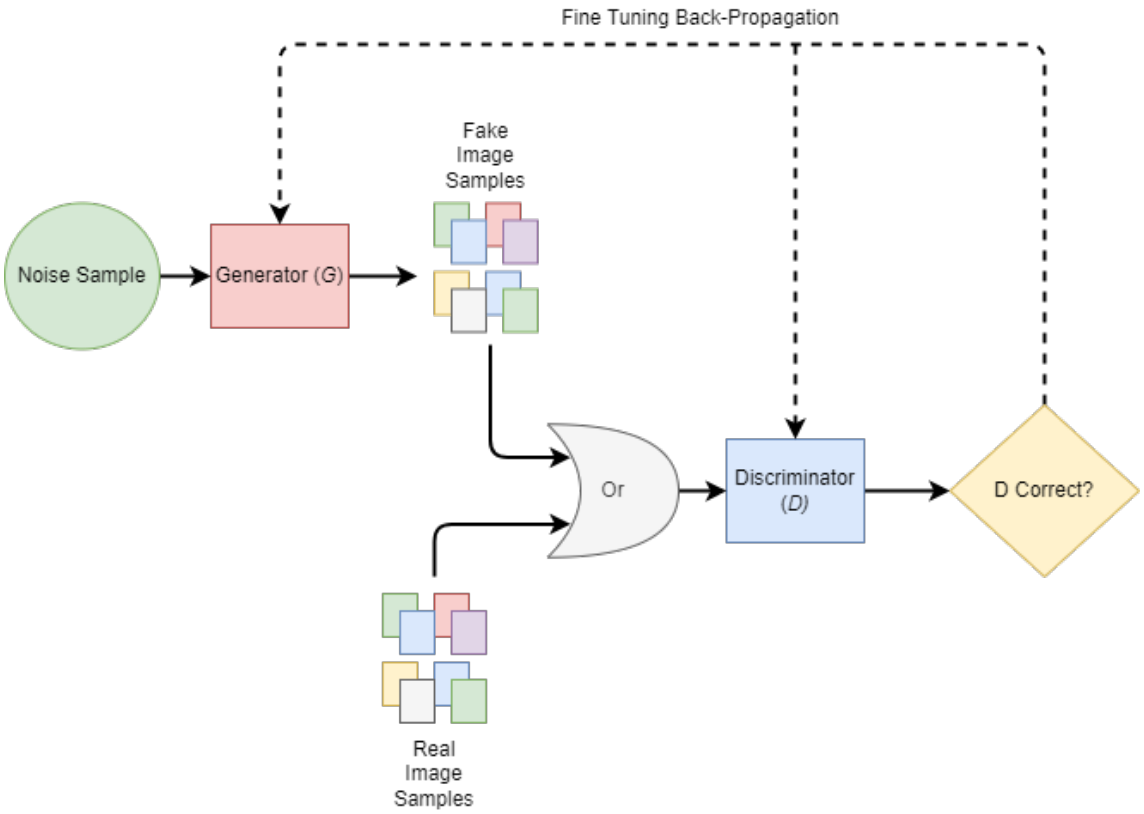


Figure 8: High Level Overview of a Generative Adversarial Network.

- Mode Collapse — Mode collapse occurs when the generator network chooses to generate images from a small number of modes [30]. This leads to many duplicate generated images and can be caused by several issues in training. Mode collapse can be diagnosed by either looking at the generated images and observing identical features in multiple places or searching for generator loss oscillations and slightly higher discriminator losses than a typical stable model of the same type.

- Vanishing Gradient — Vanishing Gradient is an issue where the discriminator does not provide adequate information to the generator and rejects fake images with high confidence, leading to the gradient of generator learning diminishing or "vanishing" to where the generator is unable to properly learn [47]. This leads to little or no improvement in generated fake images and can be avoided by not over-training the discriminator, applying Wasserstein loss [37] or a modified minimax loss as shown by the authors of the original GAN paper [23].

Evaluation Metrics: Because image quality is subjective, it can sometimes be hard to assess the quality of a GAN generated image. Multiple methods have been devised to

quantify the quality of an image including Inception Distance, Fréchet inception distance (FID), and Kernel Inception Distance (KID). These methods will be discussed in this section.

- **Inception Distance** — Inception distance is the basic method for evaluating GAN generated images proposed by [34]. It was found in their experiments that human assessors of generated images provided varied feedback with results changing dramatically when mistakes were pointed out. Every generated image has an inception model [35] applied to it to obtain the conditional label distribution based on a pre-trained ImageNet [10] model. Images that include meaningful objects in this label set should have a low entropy conditional label distribution $p(y|x)$. This, combined with a high entropy marginal $\int p(y|x = G(z))dz$ to account for the generation of varied images gives a meaningful score that resembles human judgement [34] [39]. A higher value is better with this metric.

- **Fréchet inception distance** — One of the main drawbacks of Inception Distance is that it only observes the generated data without factoring in and comparing real world samples. Fréchet inception distance seeks to achieve this comparison by using multidimensional Gaussian distribution to obtain the first two image moments: mean, and covariance [39]. Using the Fréchet distance which uses location and ordering of points to calculate a measure of similarity between curves. Fréchet inception distance is defined by the following formula:

$$d^2((m, C), (m_w, C_w)) = ||m - m_w||_2^2 + tr(C + C_w - 2(CC_w)^{\frac{1}{2}}$$

  Where:

  - (m, C) = mean and covariance obtained from generated model data
  - (m_w, C_w) = mean and covariance obtained from real world data

  For this metric, a lower score means higher image quality and more diversity in generated images.

- **Kernel Inception Distance** — Kernel Inception Distance is a metric proposed for GAN evaluation which is like FID but uses a polynomial kernel:

$$k(x, y) = (\frac{1}{d}x^T y + 1)^3$$

d = representation dimension [49].

KID compares skewness, mean, and variance and uses a simple unbiased estimator in contrast to FID which is inherently bias. This lack of bias makes KID a much better metric for use on small datasets [64] [49]. Similar to FID, a lower score indicates a better-quality image.

Generative Adversarial Networks appear to be ideal for this project as they fulfil the need for image generation through learning. Some existing methods of designing basic web pages and mobile apps have been developed recently which demonstrate the use cases for these networks. These will be evaluated in the following section.

2.4 Existing Works Generating GUI Designs

2.4.1 Face Off: Assisting in the Manifestation Design of Web Graphical User Interface

In 2019 a model known as Face Off was proposed to aid with the design of websites based on template retrieval. The styles information and screenshots of popular websites and design examples were collected into a repository and used to transform poorly developed websites into 'good looking' websites using style transfer with a Convolutional Neural Network [63]. The paper concluded that the method was a success and styles could be harmoniously embedded into existing websites to enhance their look. This research demonstrates good methods for retrieving website data using scripts which both take screenshots and download the Cascading Style Sheet (CSS) information.

While the idea of using a script to retrieve web CSS information is a promising idea, finding dashboard images may prove to be more difficult as a large number of them require signing up to visualize the main dashboard in detail. Therefore, alternative methods of data collection using scripts will have to be devised for this project.

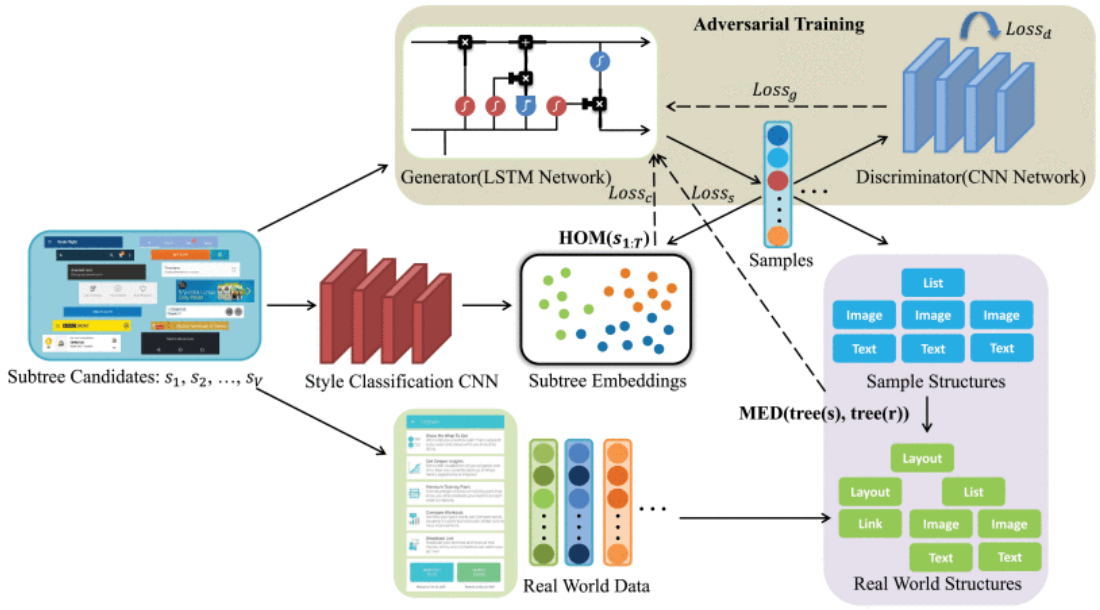


Figure 9: GUIGAN Workflow [79] Copyright © 2021, IEEE

2.4.2 GUIGAN

While the previous example provided style embedding to existing websites, the next example of previous work in this field goes a step further and seeks to output design drawings new mobile apps.

GUIGAN uses an existing SeqGAN architecture to generate new GUI designs by learning sequences of sub-tree GUI components and compatibility of styles [79]. Based on the theory that developers will reuse their components for various apps, rather than generating new UI elements; elements with compatible styles are generated in different sequences to produce new GUI designs. GUIGAN is executed using the following method:

1. Data is collected manually, and widgets and other sub-tree elements are cropped from the main GUI screenshot. Sub-tree elements are pooled in an image repository for later use.

2. Depth-first traversal is used to serialize the sub-tree elements which are then mapped to embedding space to extract vector features for input into the main GAN structure. A Siamese network with a dual channel convolutional neural network structure is used to determine if pairs of images are from the same app based on their similar styles.

3. A SeqGAN architecture using a LSTM (Long Short-Term Memory) generator and a CNN as the discriminator, is used to generate new mobile GUI designs through utilizing the pre-trained Siamese network to choose harmoniously fitting sub-tree elements from the pool and arranging them in different sequences.

While GUIGAN appears to be the latest endeavour in GUI design, there exists some gaps in the research.

- Most research discovered seems to be aimed at mobile apps or basic text and image-based websites. No evidence was discovered of design generation for complex information dashboards was found.
- The two research papers that address generation of mobile GUI's only deal in style transfer or reusing components from existing GUI's. We want to assess whether it is feasible to train an algorithm to design completely new dashboards with their own unique components.

2.5 Conclusion

The research demonstrates that a Generative Adversarial Network with some form of residual block skip-connection convolution should be used. Methods of machine learning were analysed with the following results:

- Classifiers were deemed too simple for the use case but provided a good background knowledge of the inner workings of neural networks.
- Basic Artificial Neural Networks were found to be good for data but were not the best method for image learning due to the time it takes to process and the fact there are better methods out there for feature recognition.
- Recurrent Neural Networks are suitable for language and speech processing/translation or anything else that works in sequences. To evaluate the feasibility of dashboard generation it is out of scope to work with sequences of dashboard widgets at this stage.

- Convolutional Neural Networks are especially useful for image recognition as they reduce computing time and allow for features to be discovered anywhere in the image. However, they suffer from decreased accuracy at depth compared to a shallower network. Residual Neural Networks solve this issue by introducing residual skip-connection layers which are particularly useful for GAN architectures. A GAN architecture that uses residual skip-connections is likely to perform well for this problem.
- The feasibility of generating fully new dashboard images has yet to be assessed and this report should demonstrate whether there are any learned patterns of interest generated by the chosen algorithm.

CHAPTER 3

Data Collection and Processing

This chapter outlines the processes used and developed for the purpose of collecting image data for the machine learning training process. The first topic to be discussed is the rules set out for image collection and subsequent labelling. Next to be discussed is the software written to make the job of downloading and processing the substantial amounts of data, so they meet the required standards. The final topic to be discussed is the methodology of collecting and processing the image data.

Two main types of data were required for this experiment:

1. Full dashboard images, gathered from various sources on the internet.

2. Dashboard component images, referred to as dashboard components in the rest of this chapter. These were cropped from the full dashboard images.

3.1 Image and Label Rules and Requirements

3.1.1 Image Resolution

GAN's are trained on square images at power of 2 resolutions (32x32, 64x64, 128x128, etc..). This is due to the fact most images are downscaled by powers of 2 multiple times during training. The fact that dashboards are accessed using rectangular screens means that the images must be resized and edited to fit this requirement. To ensure image details are retained through this, images sourced should be the highest possible quality. Dashboards contain a large amount of information, because of this, the training should be conducted at 1024x1024 Pixels (px) or 512x512px. The former requires a large

amount of training time and computing power, which was typically not available in this project. The latter is acceptable and seems to show adequate detail with the benefit of much lower training time therefore, the training resolution aimed for in data preparation was 512x512px.

Note: Training time is dependent on the architecture used which will be discussed in length in the next chapter.

3.1.2 Full Dashboard Image Content and Quality Requirements

It is important that all the images in the dataset follow a set of image content and quality requirements. The considerable number of images that will be downloaded from image search engines require manual and programmatic methods to be applied to ensure these images are of good enough quality, do not contain any irregularities, or are even an image of a web dashboard at all.

We devised four criteria to ensure the images are appropriate for this experiment:

1. Images must not contain any watermarks. Including watermarks in the images will misinform the algorithm and will cause irrelevant data to be processed. This will lead to larger processing times and irregularities in the image. Any image that contains watermarks must be removed from the dataset.

2. Images that are smaller than 512px in width or height should be discarded. All images must be eventually resized to 512px in width and images smaller than this will have to increase in size. Leading to a lower quality image.

3. Images that have previously been resized to be larger, while sacrificing image quality should also be discarded. These images are blurry due to the resize operation and therefore will affect the results of the training.

4. Dashboard components shown in the image should not have irregularities. It was found in prior research that design diagrams of dashboards sometimes will have one or more components rotated or placed in a position to highlight that feature. Including these images in the dataset will negatively affect the outcome of the training as the algorithm will not understand that this has been done for aesthetic pur-

poses. If possible, when evaluating an image for dataset suitability, these may be able to be cropped out while still maintaining a suitable dashboard image. If not, then the image should be discarded.

3.1.3 Dashboard Component Content and Quality Requirements

These criteria define what is considered a dashboard component that should be cropped from the main dashboard image as a dashboard component.

- Ideally, the dashboard component would be a border-defined component contained within the main section of the dashboard.
- If there are no defined borders around the component, sufficient white space between components is acceptable to distinguish separate components.
- Buttons should be included as dashboard components as they are quite prevalent on dashboards.
- Any drop downs boxes, or date selectors should be included.
- Free standing text such as headings should be included for some dashboards if it is not the same as other samples collected. Text should be readable.
- If the component suffers from low resolution or any irregularities that could affect the dataset it should not be included. This includes if it is simply too small to be sized up to a maximum of 256x256 pixels while maintaining readability.

3.1.4 Label Criteria

The following set of labels was developed for use in this experiment. These labels allow the algorithm to distinguish between the distinct types of dashboards. These labels were applied to both the main dashboard image, as well as any images that were cropped from said image. The reason for this was to ensure dashboard components were categorised by the dashboard type they belong to, rather than the type of dashboard component they represent. This list was constructed from a brief look through the main dashboard images while lightly categorizing.

1. **IOT Dashboard** — Any dashboard that is used to monitor or remotely control electronic devices or smart home setups.

2. **Agriculture Dashboard** - Dashboards that contain monitoring data specifically for an agricultural setting.

3. **Financial Dashboard** - This can include banking apps, financial market watch dashboards, crypto-currency dashboards or any other dashboard that primarily focuses on monitoring funds.

4. **Analytics Dashboard** - Dashboards that monitor a range of items such as sales figures, statistics, company overviews, and personal medical analytical data not specifically related to fitness.

5. **Misc. Dashboard** - Any dashboard that does not fit into one of the other categories. This is used for wildcards that may have a good design but due to placeholder text they are unable to be placed in a category.

6. **Fitness Dashboard** - Dashboards that display personal fitness data e.g. heart rate while exercising, exercise statistics, planned workouts etc.

7. **Project Management Dashboard** - Dashboards used for managing teams completing projects in the workplace. These dashboards consist of calendars, timelines, planners, and project group assignment functionality.

8. **Social Dashboard** - Dashboard used for displaying social media statistics of a single or multiple users.

3.1.5 Data Sources

The following websites contained images that matched the above resolution criteria as well as several other criteria that will be mentioned below. This section outlines the reasons for choosing these websites to source the data.

3.1.5.1 Google Images

Google images provides access to an extremely large number of sources all in one place. The user may also access full resolution images from the original websites. It is easy to

perform a search and retrieve a large number of satisfactory results. Especially using the advanced image search feature. The advanced image search feature allows the user to select the minimum image resolution of the search as well as providing the ability to include optional key words in the search. These features allow for searches of images that meet the resolution criteria mentioned above as well as the criteria for image quality and content.

3.1.5.2 Dribbble

Dribbble [81] is a website where top designers can highlight their work. This site is useful because it contains designs that might be perceived as good. This website also provides images at a high quality and resolution as well as provides the option to download them in assorted sizes including the largest on offer (1200x900px) which is especially useful when building a dataset of high-resolution images. Dribbble offers a large database of design images, and several searches determine there are web dashboards of distinct types available for download.

3.2 Software

The following section details the software written to process the images and extract data for this experiment.

Image URL Extractor

The following script was written based off a tutorial by Adrian Rosebrock [75]. Its purpose is to download all images currently loaded on the screen from a Google Images search.

A low-level overview of the script is as follows:

1. Iterate through the HTML document object model (DOM) and find all objects which contain the html class "isv-r".
2. For each of these objects, select the first child object that is an html anchor tag (<a>).
3. Simulate a right click on the image tag child within this object.
4. Extract the URL for the full-sized image.

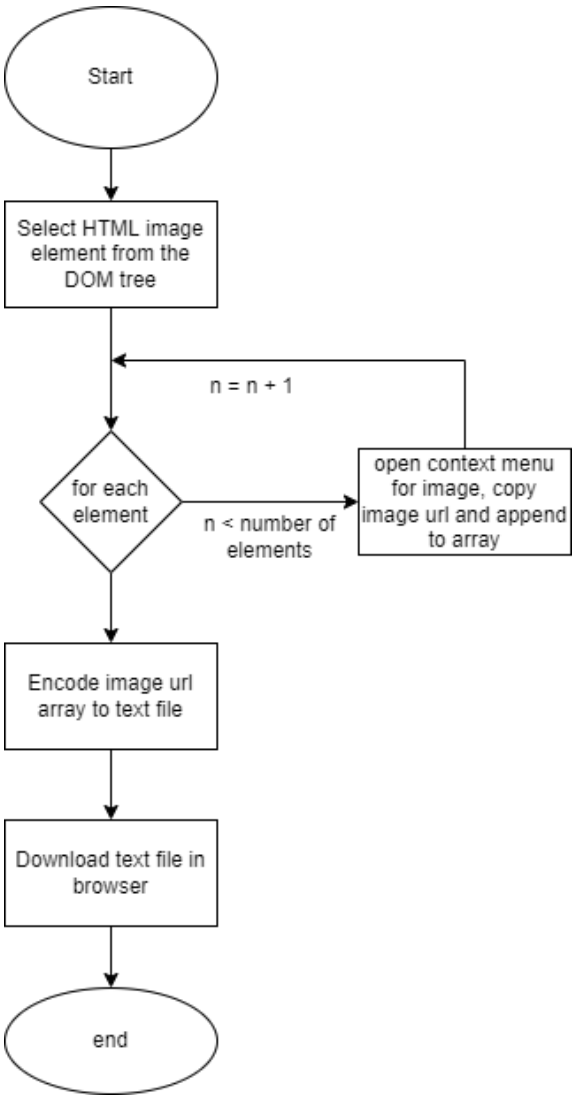


Figure 10: Structure of Google Images URL download script

5. Push this URL to an array.

6. Once finished looping through the images, initiate a browser download of a text document containing the URLs to the full-sized images.

3.2.1 Image Downloader

To download the images, a Python script was written to read the list of URLs and download them to a specified folder.

The structure of the script is as follows:

1. User inputs the file address of a text file containing image URLs to download.

2. For each URL in the text file a GET request is performed.

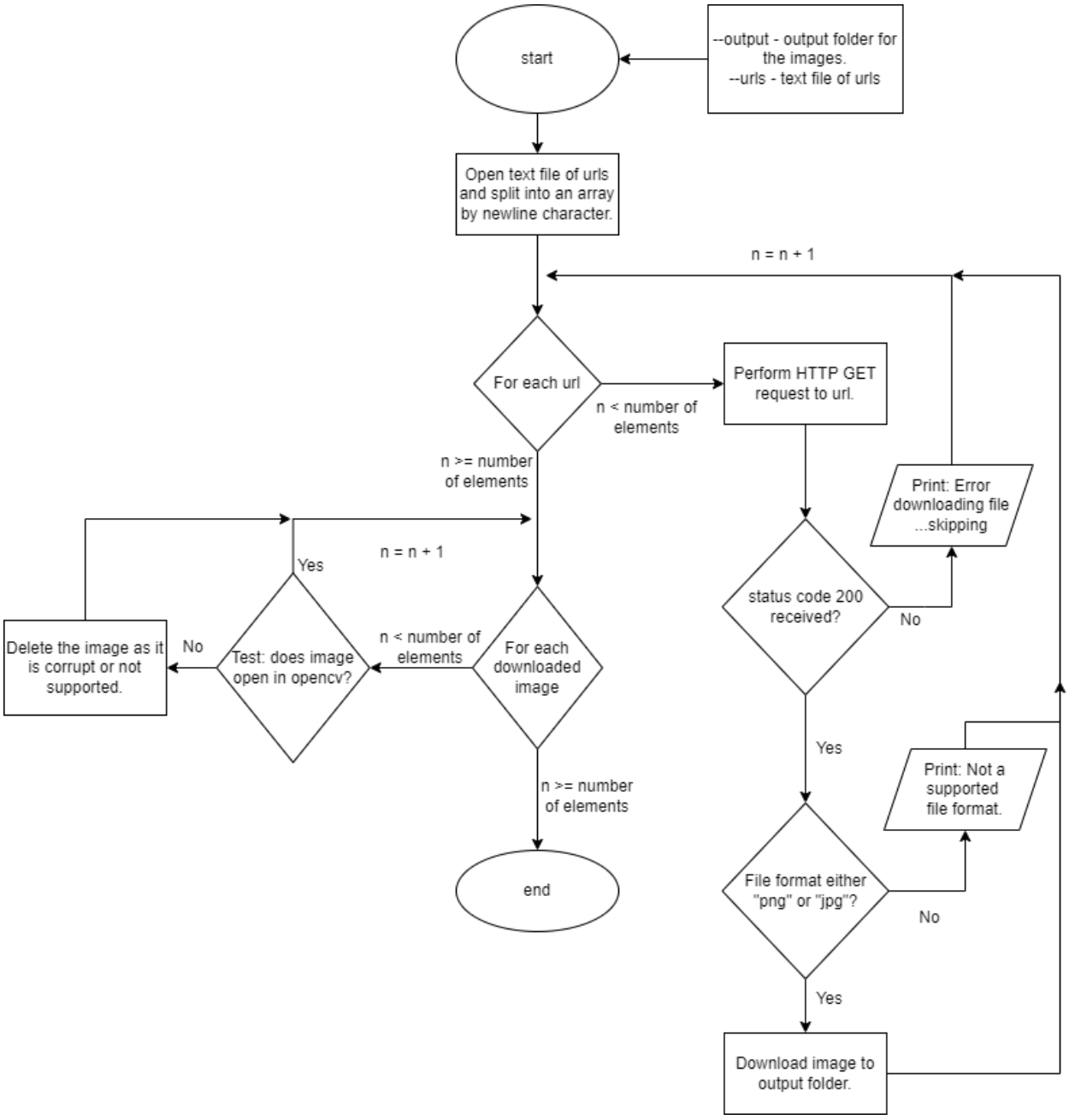


Figure 11: Structure of Python image download script

3. If the GET request returns successful (HTTP response code 200) then the image type extension is checked.

4. If the image extension is "png", "jpg" or "jpeg" then the script may proceed with the download into the given output directory on the local machine. This filters out inappropriate image types such as GIF's.

5. The script then reads the downloaded images using OpenCV to ensure the file is a proper image file and not corrupt in any way. If the image fails, this check it is deleted automatically.

3.2.2 Detecting and Removing Duplicate Images

Due to the substantial number of images required for the dataset and the methods used to collect the data in a quick manner, there is a high probability there will be duplicate images in the dataset. Duplicate images in the dataset can cause overfitting. Overfitting is where the model effectively memorizes the images in the dataset leading to reproduction of dataset images. To solve this issue, a script was written in Python to calculate the image hash using OpenCV. This was written based off a tutorial by Adrian Rosebrock [76].

The structure of the script is as follows:

1. Two initial flags are given at program run time: –dataset, the folder where the images are stored. –remove, whether to remove duplicates (false means that this is a dry run, and no images will be deleted, for testing purposes.).

2. For each image, a hash is computed. The image is converted to grayscale and the horizontal gradient between adjacent column pixels is calculated. This has value is added to an array.

3. The array of hashes is iterated over. If there are duplicates, all corresponding images are deleted except for one.

4. The image name and corresponding label will also be removed from the label file if it exists in the main image directory. This is achieved by iterating over the file contents until a match is found for the file name. This is then removed from the array using the python "pop" function.

3.2.3 Multi-Crop Machine Learning Image Processing Tool

This project required many image processing operations; from cropping/resizing main dashboard images, to extracting dashboard components, to labelling data. Based on initial experience using a number of image processing tools it was deemed necessary to create a custom tool for the needs of this project. It was found using other image processing tools that a single dashboard component could be extracted at a time, and when a machine learning data-processing tool was found with a multi-crop function, it only allowed square images to be cropped. This did not work for this project as a large majority of the dashboard components, as well as the main dashboard images themselves, are rectangular. As well as this, only certain tools were able to perform labelling for the image. The following section details the structure of this program with both high-level and low-level overviews.

Multi-Crop Tool High Level Overview

The function of the multi-crop tool is to allow the user to import a number of images, allocate a label using one of the number keys on the keyboard, and then identify regions of interest by dragging the mouse and drawing a rectangle. The regions of interest can then be saved onto the user's computer with the corresponding label data, before moving onto the next image. In order to start the program, the user must provide it with a destination folder to save the cropped dashboard components to.

The GUI for the Multi-Crop tool developed in this project contains the following features as labelled on figure: 12.

1. **Load Images** - This button allows the user to select a folder of images for processing.

2. **Save Selection** - When the user is finished cropping, this button is used to save all the user selected regions of interest on the image. The location these dashboard components will be saved should be specified when the program is run using the command line –dest option. This is required for the program to run.

3. **Undo Last** - Undo the last drawn region of interest. This is used if the user makes an error and selects the wrong region.

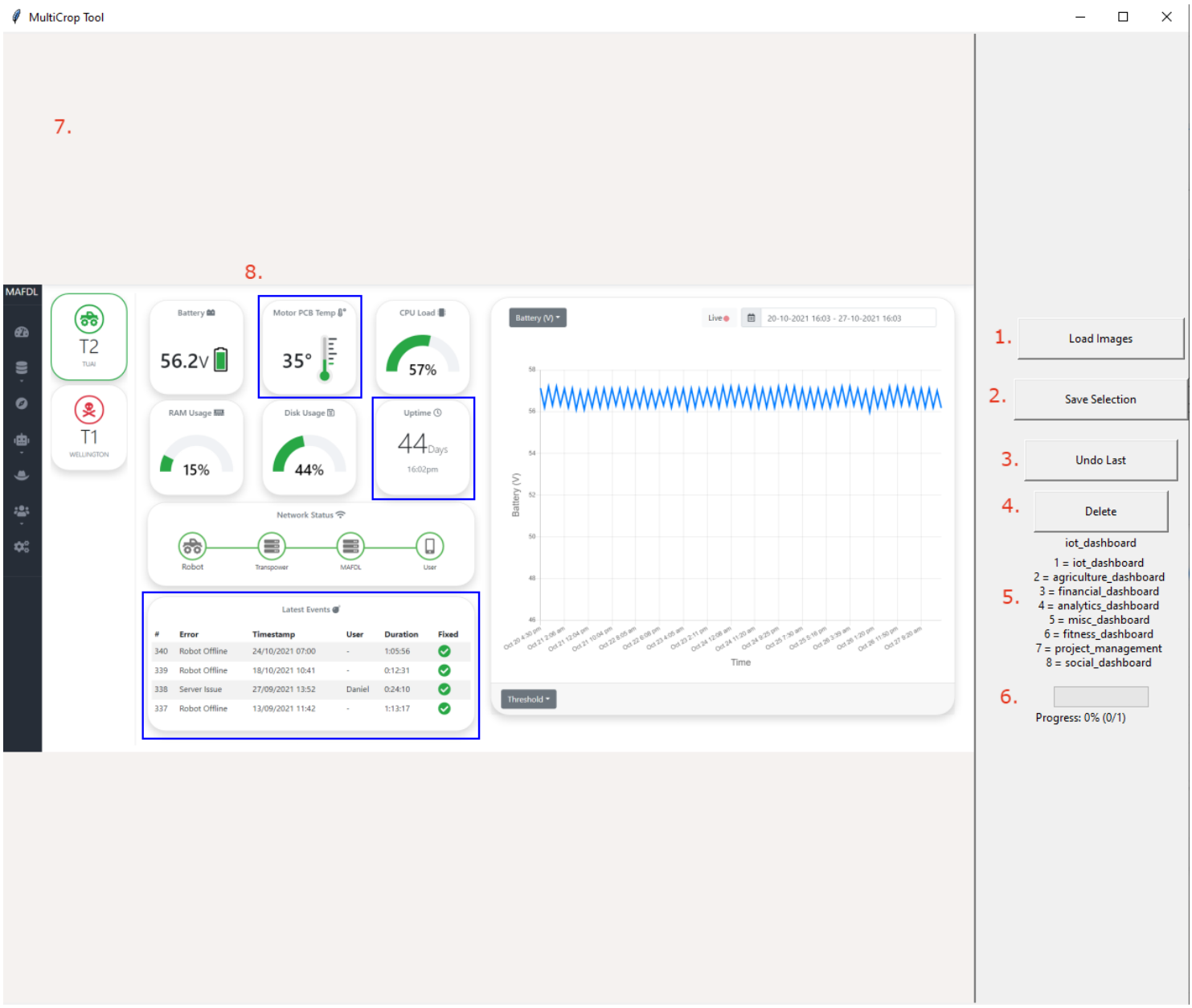


Figure 12: Multi-Crop tool Graphic User Interface

4. **Delete** - This button deletes the image permanently. Deletes the image from the file system on the user's computer. This is used if the image is found to be bad quality and should be removed from the dataset.

5. **Labels** - This is a list of the labels that have been hard coded in for this specific use case. Pressing the corresponding number key on the keyboard will identify the current image as that specific label and all subsequent dashboard components that are saved from this main dashboard image will be labelled as such. Labels are saved to two separate places:

a) The destination folder for the dashboard components as specified when the program is initialized. Labels in this file contain the name of the dashboard component that has been saved, as well as the label that has been assigned.

b) The main image folder that is selected with the load images button. Labels in this file consist of the main dashboard image name, and the corresponding label assigned to this image.

Labels are colour coded so it is easier for the user to know which label is currently active.

6. **Progress Bar** - Tracks the progress of the images processed.

7. **Image Canvas** - The area of the GUI where the main dashboard image is contained.

8. **Crop Region of Interest Example** - This blue rectangle is an example of an area to be cropped and that will be labelled as an IOT dashboard.

Low-Level Overview

The multi-crop tool is written in Python and is written using a single python class. Detailed in this section is the low-level description of how the code works and what steps are taken to produce the output cropped images. Full code for this program can be found in the appendix.

Initialization - The main function entry point is initialized when the user runs the program through the command line. The user provides the program with a number of flags:

- dest - The destination folder for the cropped images.
- target-size - Used for resizing the cropped images. This specifies size the images should be when they are saved in the destination folder.
- save-as - Specifies the image compression type. Options are limited to:
  - png
  - jpg
  - tiff

- single-crop - If this flag is included, label data will not be written to the dataset.json files. Used for initial processing of the dataset without labelling.

The MultiCrop class is then initialized with these arguments.

GUI Construction - The GUI is then built from the MultiCrop class. The GUI is written using the python tkinter library and components are placed in a grid format within two main frames on the GUI. The first frame simply contains a canvas element that houses the images. The second frame contains all the buttons and display elements such as progress bar and label information. Listeners are also defined in this function. There are three main listener groups:

- **Right and left key listeners** - These are bound to the left and right arrow keys and are responsible for iterating backwards and forwards through the images in the main image display area.
- **Number key listeners** - Each number key on the keyboard is bound to a key press event. As mentioned previously, these number keys are responsible for selecting a label for the main image and subsequent cropped dashboard components.
- **Mouse button events** - There are three main mouse button events. These are responsible for drawing the cropping region of interest on the image displayed on the canvas. The three events are left mouse button press, left mouse button release, and left mouse button motion (while pressed).

Load Images - The first button the user will press will be "load images". This button is bound to a loading function. This function prompts the user to choose a folder using a windows explorer pop-up. The folder is opened, and the image file-paths are read into an array that is stored as a class member variable. Another class member variable (pos) is used to determine the currently displayed image on the canvas based on this image path array. For example, "pos" is initialized as zero, meaning the first image to be opened and displayed is the first image in the main image array. When an arrow key is pressed, "pos" is incremented or decremented (based on the arrow key that is pressed) and the next image is opened and replaces the current image on the display canvas. The user given output directory and the selected image folder are then initialized to contain a dataset.json file

which will be populated with all the labels as the MultiCrop tool is used. This dataset.json file is initialized with the following structure:

```
{
    labels : []
}
```

When the images are loaded, they are scaled down in order to fully fit on the GUI canvas (making sure aspect ratio is maintained as to not skew the images). Before the images are scaled, a full-sized working copy of the loaded image is taken. This ensures that when users are carrying out cropping operations the scaled down image is not used. This is so image quality is not sacrificed by resizing to fit the canvas, performing the operations and then resizing the result back up to produce the final cropped image. When the image is resized to fit the canvas, a resize ratio is taken based on how much the image was resized in each direction (width and height). This ratio is used later on in the cropping operations to ensure the proper areas of the full-size copy are cropped according to the user's selections.

Resume Functionality - The program will automatically recognise if there is an existing labels file in each directory. If this file does exist and contains records from a previous use, the program will then ask the user whether they would like to continue where they left off. If the user chooses not to continue, the selected output directory for cropped dashboard components is wiped of any cropped images and both dataset.json files are reinitialized.

Drawing Regions of Interest - Once the images are loaded into the program, the user may select a label using the number keys as mentioned above. Once a label has been selected, the user may click and drag on the image in the GUI to create a bounding rectangle. This is handled by the Mouse click, release, and motion events bound to the left mouse button. Once the user clicks the left mouse button a mouse click event is fired. This triggers a function to run which notes the current coordinates of the cursor over the image. A basic rectangle is initialized and as the user holds the mouse button and drags the mouse, mouse motion events are repeatedly fired. As these events are fired the bounds of the rectangle are expanded based on the position of the cursor when the mouse motion event is fired. When the mouse button is released, a rectangle is drawn on

the GUI canvas based on the final coordinates of the cursor at the point of button release. This leaves a rectangle bounding box on the screen for the user to observe. An array containing the coordinates of this rectangle as well as the label data is appended to the main array to store user selections for future cropping operations.

$$[x_0, y_0, x_1, y_1, label]$$

Note: Because the image viewed by the user is scaled to fit the canvas, the region of interest coordinates are scaled by default. Therefore, later on when the cropping occurs these must be scaled back up in order to crop the dashboard component out of the real, full-sized image.

Undo Last Action - Since rectangle data is saved to an array in the above operation, it becomes easy to undo the last action. The last rectangle coordinate array is removed from the main array and the coordinates are used to delete the rectangle from the canvas.

Crop and Save - Once the user has drawn all regions of interest, they must be saved. The user presses the save selection button and the current array of regions selected is iterated over. For each iteration, the following actions occur:

1. The sub image is cropped. The rectangle coordinate array including label data is passed as an argument to a cropping function. This function then extracts the four rectangle coordinates and divides each one by the scaling factor given when the image was loaded and scaled to fit the GUI canvas. The image is then cropped by performing an array selection operation on the main image array.

   $$crop = [y0 : y1, x0 : x1]$$

   The colon in this operation means to obtain all the values in the array between the specified coordinates. The crop function then returns the cropped image array and corresponding image label. If there is no label selected, for example if the software is in single-crop mode, the label will simply be set to -1 as it is not needed.

2. The cropped image is then saved. To achieve this, first the image must be padded to ensure it fits with the square image requirements. This is achieved by the following steps:

a) The original height and width of the image is obtained from the image array.

b) If these values are equal, the image is already square, so it is simply returned by the function.

c) If height is greater than width, then the height is set as the target size. The width is resized.

$$\Delta w = height - width$$

$$padding\_left = \frac{\Delta w}{2}$$

$$padding\_right = \Delta w - \frac{\Delta w}{2}$$

$$padding\_left = 0$$

$$padding\_right = 0$$

d) If the width is greater than the height the same calculation occurs but with the width as the target size, and the padding is added to the top and bottom instead of left and right sides.

e) The OpenCV function cv2.copyMakeBorder() is used here to apply the borders to each side of the image. This takes the arguments; image, top, bottom, left, right, borderType, and colour of border. Border type is given as a constant border type, and colour is given as a generic grey.

3. The image is then resized based on the target size given by the user at program initialization. Since the image is square at this point, the resizing is simply achieved using the OpenCV function cv2.resize. This function takes the arguments image, dimensions, and interpolation. In this case the dimensions are given as the target size and the interpolation is given as inter-area. According to the OpenCV official documentation [82], inter-area interpolation should be the preferred method for image decimation, which is what is occurring during a crop operation. It uses a re-sampling method based on pixel area relations in the image to resize the image.

4. If the single-crop flag is not set, the label data is written to the label file in the given output directory. This is achieved through a function called write_json_file. The function performs the following actions:

a) The file is opened in read mode.

b) The file data is loaded into a variable.

c) Any existing labels data is extracted from the labels section of the data into an array.

d) The title of the cropped image and the corresponding label is added to the labels data. The resulting labels file will be as follows:

```
{
    labels : [
        ["image_name", 1],
        ["image_name2", 3],
                .
                .
    ]
}
```

e) In order to ensure a clean file update, the labels file is opened a second time in write mode. This overwrites the file entirely and allows the appended label data to be written to the file ensuring there is no dangling data points from the previous iteration of the file.

5. Finally, the cropped image is written to the output folder given by the user at program initialization. The user may also specify an image compression type. These can include JPEG (Joint Photographic Experts Group), PNG (Portable Network Graphics), or TIFF (Tagged Image File Format), with PNG being the default if none are selected. All images are compressed with the highest possible quality settings to keep image quality when the compression type is lossy, such as JPEG. The image is saved using the OpenCV function cv2.imwrite which takes the arguments; file name, image, and compression type.

After all the dashboard components have been cropped and saved, the main image is labelled using the same method as the dashboard component labelling. Where a main JSON label file located in the main image folder is appended to with the image name and the label for the full image. This is to ensure that the dashboard components retain the same label as the main image.

3.3 Methodology

This section describes the methodology of obtaining the dashboard image dataset and subsequent extraction of dashboard components.

3.3.1 Data Collection

Required software:

- imageUrlExtractor.js
- downloader.py

1. Proceed to Google Images advanced image search and enter appropriate search terms. In the section labelled "narrow your results" ensure that "image size" is set to at least 800x600px. It is preferable to find the highest possible resolution however, full dashboard images of high quality available on Google Images are extremely scarce. Click the button to proceed with the search and scroll the page until images look to be irrelevant or the page reports there are no more images to load.

2. The user must open their web browser development tools (typically by pressing f12 on the keyboard. This works in Chrome, Firefox, and Microsoft Edge). Navigate to the console section and paste the full code for the image URL extractor tool and press enter on the keyboard. This will activate the script and initiate the download of a text file containing all image URL's. Each URL will point to the highest resolution version of the referenced image.

3. The images must then be downloaded. This is achieved using the "Image Downloader" tool. The user will open the terminal of their choice (windows PowerShell, windows command prompt, Linux terminal, etc..), navigate to the directory containing the downloader tool, and enter the following command:

   ```
   python .\downloader.py --urls="PATH TO URL TEXT FILE" --output=
   ```

   The script will proceed to download all images to the given output directory. Any incompatible images will be skipped.

4. Writing scripts to automatically download images from Dribbble is specifically against terms of service. Therefore, the user must manually download the image URL from this website. To download each image URL, click on the image thumbnail to navigate to the full-sized image. Then right click and select "copy image address". This will copy the image URL which can then be pasted into the main URL's file.

3.3.2 Data Processing

After the data has been collected, it must be processed to meet the image size and resolution guidelines. The images must also be labelled. This section details the process of processing the main dashboard images, as well as extracting the dashboard components.

Required tools:

- detect_and_remove.py
- MultiCrop.py

1. The detect_and_remove.py script should now be used to remove duplicate images. This script should be run with the following command:

```
python .\detect_and_remove.py \
    --dataset="PATH TO DATASET"
```

   Duplicate images will then be deleted from the destination folder. Any images below a certain width will also be removed (default 512).

2. The next several steps required the use of the MultiCrop Tool. The software was run with the following command:

```
python .\MultiCrop.py \
    --dest="MAIN DASHBOARD DESTINATION FOLDER" \
    --target-size=512 \
    --save-as="png" \
    --single-crop
```

3. The first use of the MultiCrop software was to perform initial cropping operations and clean up the main dashboard image dataset. When the software was initial-

ized, the "Load Images" button was pressed, and the main dashboard image folder was selected in the file explorer.

4. Each image in the dataset was evaluated against the set image criteria. Images were deleted using the "Delete" button if they were deemed inappropriate for the dataset. If the images included a large border around the dashboard, this was cropped, and the image saved using the "Save Selection" button. If the image did not include a large border and simply required scaling and padding, then the "Resize Main" button was pressed. All images were automatically made square, resized, and then saved to the given output directory.

   The MultiCrop tool should be closed after this step.

5. Because the main images were being cropped in this step we can disregard the created dataset.json file in the main dashboard image folder. This was deleted before continuing.

6. The following steps detail the process of creating dashboard components by cropping the main dashboard components. The MultiCrop tool was initialized again with the following command:

```
python .\MultiCrop.py \
    --dest="SUB IMAGE DESTINATION FOLDER"
    --target-size=256 \
    --save-as="png"
```

7. The original image folder was selected for cropping. This was to ensure the dashboard components were of the highest quality.

8. A number key was pressed to select the label for the dashboard and cropped components. For the current image displayed on the screen, dashboard components were evaluated against the dashboard component criteria. Components that matched the criteria were selected using the mouse. When all the components were selected, the "Save Selections" button was pressed. This saved the images into the destina-

tion folder and saved the label data for the dashboard components and the main dashboard images. The software automatically moves onto the next image.

9. The previous step was repeated for the number of dashboard images in the dataset.

10. After all dashboard components had been extracted and saved, the dataset.json file generated in the selected folder in step 7 had to be moved to the destination folder given in step 2. This contained all the label data for the main dashboard images.

In total the data collection stage of this project yielded 1024 main dashboard images and a set of 5133 dashboard component images to be used for the model training.

CHAPTER 4

Material Selection

Based on the findings in the literature it was determined that a Generative Adversarial Network was the best way to generate new dashboard design images. It was determined that due to the period and scope of this project that it was infeasible to develop a GAN architecture from scratch and therefore, a pre-existing algorithm had to be selected. In this chapter we will discuss the algorithm selection process including the hardware available for the project, the criteria used to rank algorithms to determine the best for the project, a detailed overview and score for each algorithm candidate, and finally a detailed analysis of the different types of GAN evaluation metrics with a discussion on which metric was chosen for image generation evaluation.

4.1 Hardware

In this section the hardware available will be discussed. Generative Adversarial Networks typically require a large amount of computing power especially when dealing with images of a reasonable size such as 512x512px or 1024x1024px. The most important hardware requirement in this project is the Graphics Processing Unit (GPU), this is due to the GPU's ability to perform matrix calculations with much greater efficiency than the Central Processing Unit (CPU). There are a number of specifications that are important in the selection of GPU's:

- Brand — Due to CUDA being a platform developed by NVIDIA, only NVIDIA GPUs are compatible with it. CUDA is the only way for machine learning libraries such

as Pytorch [59] and TensorFlow [27] to interface with the GPU. Therefore, the GPU must be an NVIDIA brand GPU.

- CUDA cores — This is the amount of dedicated processing cores on the unit. This denotes how many parallel operations can be computed. The more cores available, the faster the training time.

- Video Random Access Memory (VRAM) — During the training process several items are stored in the VRAM such as: image data in batches of a certain number, the model itself, performance metrics, and other such things. If more VRAM is available, the batch size can be increased leading to faster training times. For the higher specification GAN architectures such as StyleGAN2 it is recommended that 12GB minimum VRAM be available.

- Max Power Consumption — Power consumption is important since that the units will need adequate cooling if the power consumption is too high.

GPU Candidates			
GPU	CUDA cores	VRAM	Power Consumption
NVIDIA Quadro P2200	1280	5GB	75W
NVIDIA GeForce RTX 3060	3584	12GB	170W
NVIDIA v100	5120	32GB	300W
NVIDIA GeForce RTX 3090	10,496	24GB	350W
NVIDIA A40	10,752	48GB	300W

Table 4.1: Available GPU Units for Selection with Specifications

Based on the above table we can see that the A40 seems to be the best selection for this project. The A40 has the most CUDA cores, VRAM and does not have an incredibly large power consumption. Initial tests on these GPU's using one of the most computationally intensive GAN architectures, StyleGAN2, with 512x512px test images showed that the P2200 was not usable due to having insufficient VRAM. The RTX 3060 was able to run at a speed of 367.45 seconds per kilo-image. This contrasts with the v100 which according to StyleGAN2 documentation [64] runs at approximately 72.5 seconds per kilo-image. This is bested by the a40 which computes at speeds of 50-60 seconds per kilo-image due to the larger VRAM size meaning a larger batch of images can be loaded at a single time, as well

as having over twice the CUDA cores of the v100. Power consumption for each unit seems to be at an acceptable rate to not need to consider any extravagant cooling system.

Multiple GPU's can be connected to further speed up the processing time with up to 8 GPU's able to be used at once for most GAN models. We had available a machine which holds two A40 GPU's which when tested on StyleGAN2 brought the processing time down to around 28 seconds per kilo-image.

4.2 Evaluation Metric Selection

In this project there were two main evaluation metrics to choose from to assess the quality of the images produced by the model. These were Fréchet Inception Distance (FID) and Kernel Inception Distance (KID), which are both explained in detail in the literature review chapter. Both these methods are similar except for the fact that FID carries an inherit bias when there is insufficient real image data available. This does not seem to be the case for KID as demonstrated by [64]. Therefore, due to this existing bias on small data sets, it was decided to go with the inherently non-bias option for evaluation metrics based on the real dataset which is KID.

4.3 Model Selection

When choosing an algorithm for this project, several ranking criteria were considered. This was to ensure that the overall best performing algorithm was selected. The following criteria were selected:

- Training time
- Conditional training
- Ease of using own dataset
- Performance on small datasets
- Customization
- Evaluation metrics

Each criterion was given a weighting based on how important it is from 1- 10. This value was then used to multiply the values given to each criterion for each separate model architecture to obtain the final score. The model with the highest score was chosen for use in this project.

4.3.1 Training Time

Training a GAN can take a long time. Due to the time constraints on this project, it is a great help if the algorithm can be run to convergence in a reasonable time. Training time was assessed in this ranking system as the time taken to perform a single epoch on our machine with two A40 GPU's. This is one of the most important criteria and therefore has been given a weighting of eight.

— **Weighting:** 8

4.3.2 Conditional Training

A significant part of this project is the ability to generate a dashboard by class i.e., finance dashboard, IoT dashboard, analytical dashboard etc. The chosen GAN architecture must have the ability to allow for class labels in the dataset and train conditionally in order to generate separate classes of images. This is a high-weighting criterion because it is a large part of the project. This criterion will be ranked on whether the model is able to train conditionally, as well as the difficulty to do so.

— **Weighting:** 10

4.3.3 Ease of Using Own Dataset

The chosen model should allow for the user to use a custom dataset. The reason this is included is due to the fact some models are linked to a pre-existing dataset and using a custom one involves a number of extra steps. This is not a high-ranking criterion as it is not a breaking feature if there are a few extra steps to take to get a custom dataset working.

— **Weighting:** 3

4.3.4 Performance on Small Datasets

This is one of the most important criteria. Due to the availability of viable dashboard images online, only approximately one thousand images were able to be collected. This is an exceedingly small number for GAN training and the chosen algorithm needs to account for this without overfitting. Models which allow for augmentation to combat this are preferred.

—**Weighting:** 9

4.3.5 Customization

This criterion involves the ability for the user to change hyperparameters and control the algorithm. This includes any command line features that make it easier to control or run the algorithm, as well as how easy it is to edit the code if necessary. This is a medium-to-low priority criteria as it is more of a convenience thing than anything else.

—**Weighting:** 4

4.3.6 Evaluation Metric Calculations

This criterion assesses whether the algorithm automatically calculates model performance metrics, and which metrics it can perform. Some models may not calculate the specific metrics required and therefore a metrics calculation script will have to be written if the model was the be used. This development can take valuable time away from the project which could be used for lengthy training times. This is a medium priority ranking criteria as it will have a small impact on the project completion timeline.

—**Weighting:** 5

Criteria Weighting Summary	
Criteria	Weighting
Training Time	8
Conditional Training	10
Own Dataset	3
Small Data Performance	9
Customization	4
Evaluation Metrics	5

Table 4.2: Overview of Ranking Criteria Weightings for Model Selection

4.4 DCGAN

4.4.1 Architecture

This model was introduced by [29] in 2015. It is a direct extension of the original GAN architecture, but it utilises convolutions and convolutional transpose layers.

4.4.1.1 Generator

The generator (*G*) in DCGAN maps the image data from the latent space vector to data-space. This is achieved through a number of strided two-dimensional convolutional transpose layers. Each layer is paired with a ReLU activation function and a 2d batch normalisation layer. This batch normalization layer appearing after the convolutional transpose layers is a main contribution of the paper where this algorithm was introduced. The final step is to use a tanh function to return to the input data range.

4.4.1.2 Discriminator

The discriminator (*D*) consists of a binary classification network which receives an input image and returns an output probability of whether the image is real. This uses multiple convolutional layers to process the image as well as 2d batch normalization and Leaky ReLU activation functions at each deep layer. The final layer activation function is a Sigmoid function which maps the result to a probability value.

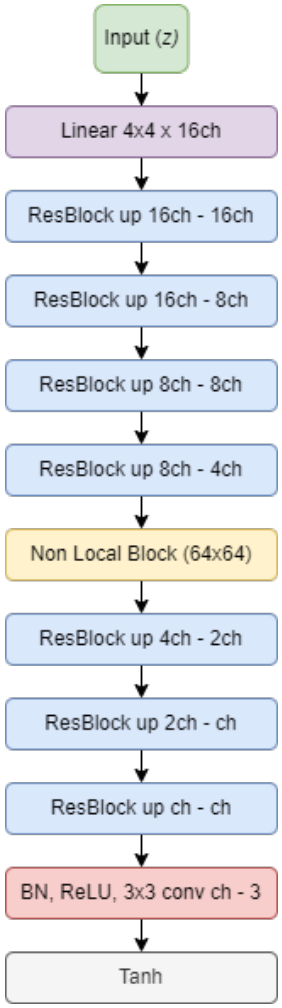
4.4.2 Ranking Score

DCGAN does not have an official code implementation available however there are a number of tutorials given in the official Pytorch documentation on how to implement this model. DCGAN is typically run on small image sizes and after initial tests and brief research it was found that it is not viable for 512x512px images without significantly altering the structure of the model. Due to the fact it was non-viable to run on the dataset for testing, no ranking score has been given for this model and it was not used in the final project but still was considered as it is a good model that is widely used.

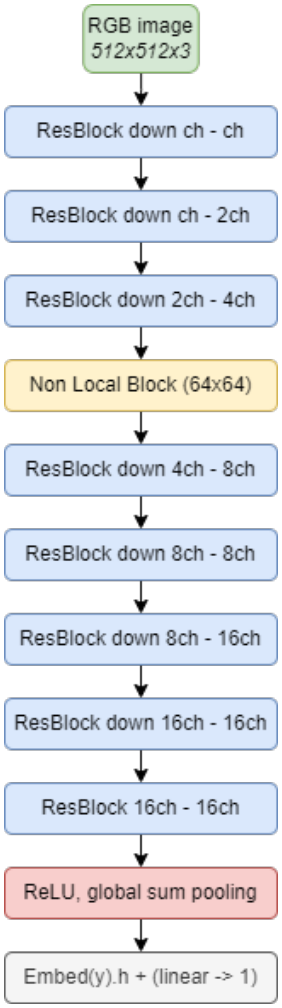
4.5 BigGAN

BigGAN was created with the intention of bridging the gap in image quality from the data generated from GAN architectures and the real data in the ImageNet [10] dataset. The contributions of this model were that GAN's benefit from scaling, the truncation trick which allows fine grained control over the trade-offs between fidelity and variety in samples, and instabilities in large scale GAN's were discovered along with some methods to reduce these at the sacrifice of performance [51].

4.5.1 Architecture



(a) BigGAN Generator for 512x512px Images

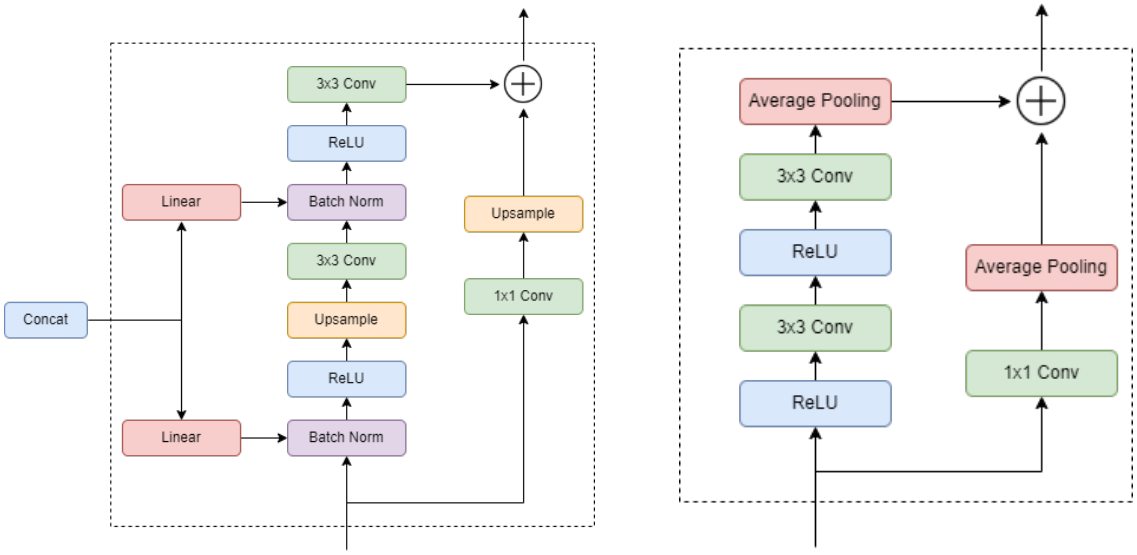


(b) BigGAN Discriminator for 512x512px Images

Figure 13: BigGAN Architecture Showing a) Generator, b) Discriminator.

The generator for the 512x512px implementation of BigGAN as described in [51] is made up of multiple residual blocks consisting of multiple instances of batch normalisation,

convolutions, and activation ReLU functions with each one shifting to various channel widths. A non-local block is used to capture long ranged dependencies [56]. As the authors mention this non-local block is moved to the 64x64 resolution to abide by memory constraints. The final output passes through a Tanh function which like the Sigmoid function, maps the value to between -1 and 1.



(a) BigGAN "Residual Block Up" Used in the Generator.

(b) BigGAN "Residual Block Down" Used in the Discriminator.

Figure 14: BigGAN Residual Blocks a) Residual Block Up Used in the Generator, b) Residual Block Down Used in the Discriminator.

The discriminator takes an RGB image as an input and then runs it through a number of residual block down functions which consist of a number of convolutions, ReLU activations and average pooling steps. The discriminator also contains a non-local block at the 64x64 resolution and finishes with a ReLU function and global sum pooling.

4.5.2 Ranking Score

4.5.2.1 Training Time

Based on the documentation for the authors "officially unofficial Pytorch BigGAN implementation" [51], the algorithm runs for approximately 15 days for 150,000 iterations on images of size 128px using eight NVIDIA v100 GPU's. That works out to be approximately 8.64 seconds per iteration. Compared to the likes of FastGAN and StyleGAN2 which run at less than one second per iteration, and 3.9 - 4.9 seconds respectively, this is slightly on

the slow side. 15 days is still a reasonable amount of time in total as it really depends on the number of iterations to convergence as well.

—**Score:** 6

4.5.2.2 Conditional Training

BigGAN offers full conditional training as it was designed around the ImageNet database which contains class labelled images. Labels are loaded into BigGAN using a text file with a JSON style format where the number image is on the left-hand side and the label or set of labels for an image are on the right side. It is not entirely clear from the documentation how to incorporate this when using a custom dataset. Therefore, the score given for this criterion is medium to high.

—**Score:** 8

4.5.2.3 Ease of Using Own Dataset

Due to the fact BigGAN was designed to work specifically with the ImageNet dataset, it can be difficult at times for the user to provide their own dataset. There are brief instructions in the main code repository, but they are quite vague and require the editing of multiple code files. This algorithm has been given a relatively low score for this criterion as it is the most difficult out of all compared models in this report.

—**Score:** 2

4.5.2.4 Small Dataset Performance

Like the previous point, since this method was designed to work with ImageNet, it was not specifically designed to manage small datasets. A typical dataset for BigGAN involves the entire ImageNet library (approximately 1.2 million images and 1000 categories) and is trained using different batch sizes of images ranging from 256 - 2048. The higher the batch size, the better the results. The dataset for this project consists of one thousand images which is considered exceedingly small. Compared to the other two methods which are specifically designed to perform well on small number datasets, this criterion scores low.

—**Score:** 2

4.5.2.5 Customization

BigGAN provides a large list of customizable options to use when initializing the program. A number of shell scripts are available to run the model with various batch sizes. Each one contains a number of flags that can be set and include everything from generator and discriminator specifications to dataset and batch size modifications. This model seems highly customizable from the command line and therefore has received a high score in this area.

—**Score:** 9

4.5.2.6 Evaluation Metrics

Two evaluation metrics are supported in the main code, Inception Score, and FID. While these are both good metric indicators, as seen in the previous section, we are interested in KID for its performance on small datasets and non-biased nature. This is mainly because BigGAN was introduced in the same year as the paper describing KID and therefore the method was not available or well-known enough to be implemented with this model. Further research into the BigGAN code would have to be completed to write a script for calculating KID which would have crossed into potential training time in the overall project timeline. The score awarded to this method reflects this fact but also considers that there were two methods available.

—**Score:** 5

BigGAN Ranking Summary	
Criteria	Weighting
Training Time(8)	6
Conditional Training(10)	8
Own Dataset(3)	2
Small Data Performance(9)	2
Customization(4)	9
Evaluation Metrics(5)	5
Total	32
Total (Weighted)	213

Table 4.3: BigGAN Ranking Overview

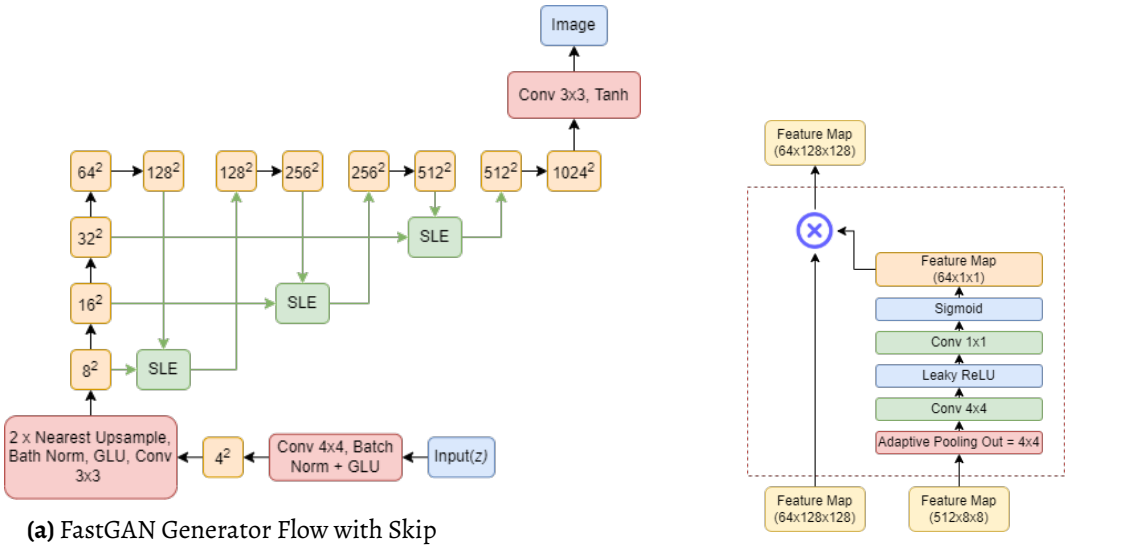
———————————————————————— **Total Score:** 213/390

4.6 FastGAN

FastGAN was designed with the goal of learning and generating high resolution images with few training samples and low computational cost. These are difficult to train due to the risk of mode collapse and overfitting meaning the model fails. The authors of this model [67] introduce three main contributions in the form of:

- The use of an SLE (Skip-Layer channel-wise Excitation) module which allows for a more robust gradient flow through the weights of the model by revising the high scale feature map channel response using low-scale activations.
- A self-supervised discriminator trained as a feature encoder which can pass more detailed signals to the generator.
- The two above contributions combined to form a fully functional network.

4.6.1 Architecture



(a) FastGAN Generator Flow with Skip Layer Excitation

(b) FastGAN Skip Layer Excitation Block

Figure 15: FastGAN Generator a) Generator, b) SLE Block.

The generator used by FastGAN uses a method called Skip-Layer Excitation. This is a reworked version of Residual Blocks due to Residual Blocks being computationally expensive. Element wise addition in Residual Blocks is swapped for a channel wise multiplication, eliminating computationally heavy convolution, and meaning one side of the

activation only needs a spatial dimension of 1x1. Skip Layer excitation can be formulated as:

$$y = \mathcal{F}(x_{low}, \{W_i\}) \times x_{high}$$

Where:

- x_{low} and x_{high} are the feature map inputs in fig.15b 8x8 being x_{low} and 128 being x_{high}.
- F is the function indicating the operations to be completed on x_{low}.
- W_i is the weights to be learned for this module.

Typically, with Residual Blocks, equal spatial dimension is required. However, this method eliminates this requirement meaning that skip-layers can be applied between different image resolution steps (e.g., between 16x16 and 256x256). This gives this method the advantages of Residual Blocks with significantly less computation [67].

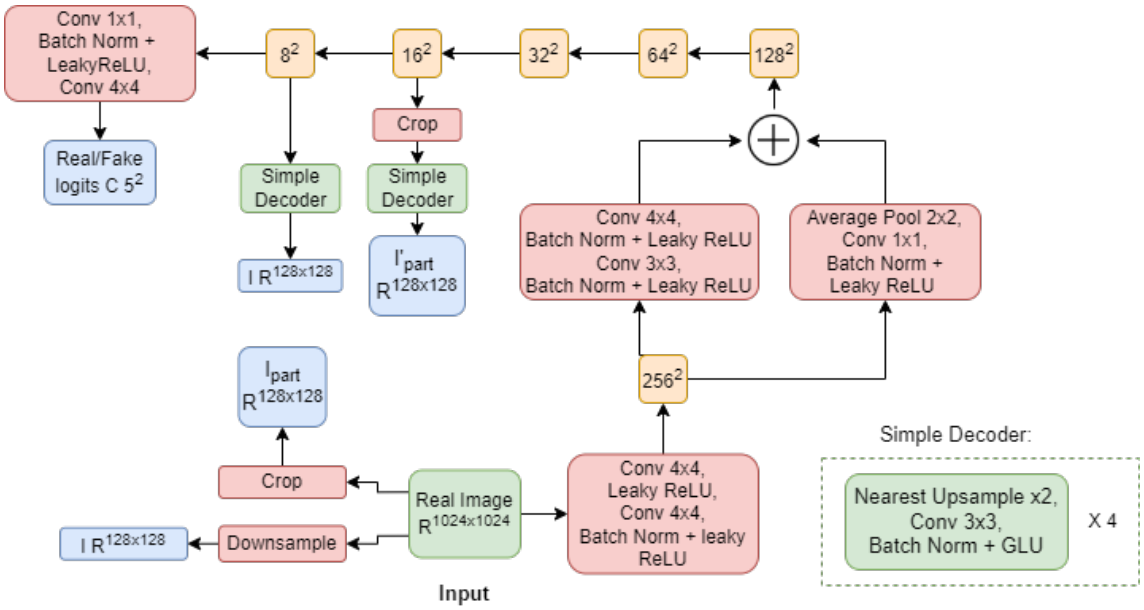


Figure 16: FastGAN Discriminator Flow

The self-supervised discriminator used by FastGAN is treated as an encoder and trained with small decoders. Auto encoded training pushes the discriminator to pull image features which can be decoded to good reconstructions. Two decoders are used on the 16x16 and 8x8 scales and these decoders are optimized together using simple reconstruction

loss trained on only real samples as demonstrated by this formula:

$$\mathcal{L}_{recons} = E_f\, D_{encode}(x), x\, I_{real}[||\mathcal{G}(f) - \mathcal{T}(x)||]$$

where:

- f equals the intermediate feature maps from the Discriminator.
- Function $\mathcal{G}$ is the processing of the decoder and f.
- $\mathcal{T}$ is the processing on sample x from the real image input.

This method introduced in [67] means that the discriminator extracts a more broad, complete data representation from all inputs. This discriminator is built to deliver high computational speeds and allow the full network to converge on high resolution data much faster than a typical GAN.

4.6.2 Ranking Score

4.6.2.1 Training Time

It is understood this method is the highest ranking with regards to training time as this is the entire purpose of the model. Testing completed on a 12GB NVIDIA GeForce RTX 3060 graphics card obtained speeds of one second per iteration on 1024x1024 sized image-data. Therefore, this method receives the highest score out of all available models with regards to training time.

—**Score:** 10

4.6.2.2 Conditional Training

Unfortunately, FastGAN does not offer class conditional training as the model is unconditional. This is a major problem due to one of the main goals of this project is to generate based on class labels. Implementing this would be extremely hard as it would require a full model re-write. Therefore, this model scores the lowest score in this category.

—**Score:** 0

4.6.2.3 Ease of Using Own Dataset

It is easy to use a custom dataset with this architecture. Images must be in a specific folder structure outlined in the documentation. This model is not designed around or tied to a specific image library like BigGAN. However, compared to StyleGAN2 it falls slightly short in ranking points in this category because styleGAN2 provides a full image processing tool to shift all images into the appropriate formats, filetype and folder structure. Therefore, it was deemed that a high score is deserved, but this does not rank the highest out of all the models.

—**Score:** 8

4.6.2.4 Small Dataset Performance

Again, this model was specifically designed to work with incredibly low dataset sizes at high speed. Therefore, the small dataset performance of this model is the best of all models compared in this chapter. Few shot image tests were conducted in [67] using image sets of 100 - 400 images achieving good results using FID metrics compared to 20-hour training on StyleGAN2 with the same datasets. This model received a high score in this criterion due to the fact it seems to outperform all other models on small datasets.

—**Score:** 10

4.6.2.5 Customization

Several command line arguments are available when running the model such as image size, batch size, number of iterations and which GPU to use. These are all standard arguments typically available to the other models. The ability to change hyper-parameters lies with editing the code directly but this is not too much of an issue as instructions on which values to change are included in the documentation. These hyper-parameters include the model depth, mode width, and augmentation options.

—Score: 8

4.6.2.6 Evaluation Metrics

Two metrics are used for evaluation in FastGAN: FID and Learned Perceptual Similarity (LPIPS). They concluded that if FID was to be inconsistent it would be inconsistent

throughout each test to the same degree, so it was appropriate for comparing their model with other models. In this case however, we are looking for models that can implement KID as it has been deemed the best for small datasets. Implementing this may be difficult and time consuming so therefore a score has been given that references this fact but also considers the two methods on offer, the same as BigGAN.

—**Score:** 5

FastGAN Ranking Summary	
Criteria	Weighting
Training Time(8)	10
Conditional Training(10)	0
Own Dataset(3)	8
Small Data Performance(9)	10
Customization(4)	8
Evaluation Metrics(5)	5
Total	40
Total (Weighted)	251

Table 4.4: FastGAN Ranking Overview

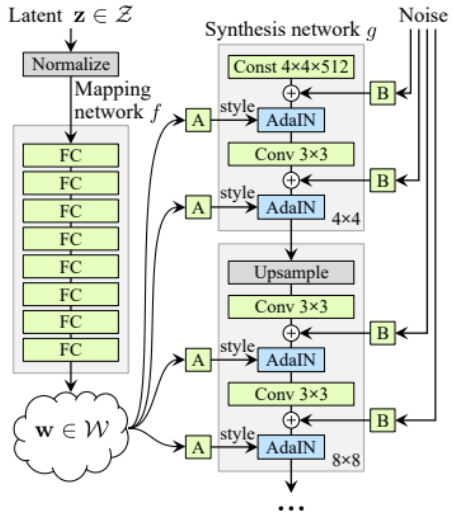
———————————————————————— **Total Score:** 251/390

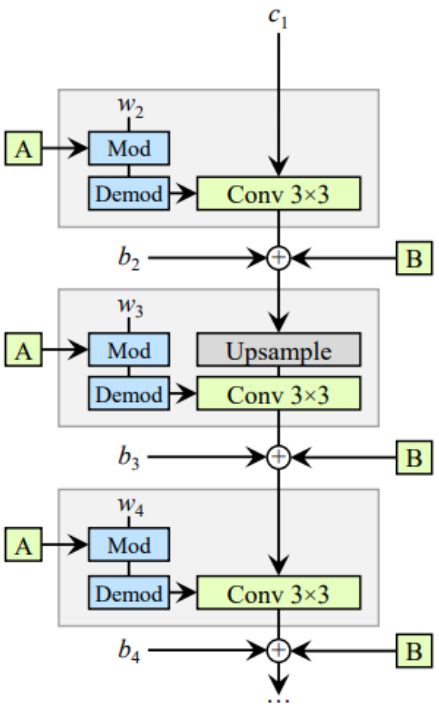
4.7 StyleGAN2-ADA

StyleGAN2 as proposed in [65] is an improved version of the famous StyleGAN, a model designed to allow control over the image synthesizing process. StyleGAN was designed with style transfer literature in mind to create an unsupervised, automatically learned model that allows for the separation of high-level features and direct control of image synthesis through its generator design [54]. Several improvements were made for StyleGAN2 including a reworked generator network, removal of normalization artifacts, and general image generation improvements. Further to this, StyleGAN2-ADA [64] is an addition to StyleGAN2 with the main goal being the ability to train on low amounts of data using Adaptive Discriminator Augmentation (ADA).

4.7.1 Architecture

4.7.1.1 Generator





(a) Copyright © 2019, IEEE

(b) Copyright © 2020, IEEE

Figure 17: StyleGAN Generator Iterations a) Original StyleGAN generator proposed in [57], b) Revised Weight Demodulation StyleGAN2 Generator Model proposed in [65]

The StyleGAN generator has undergone many revisions. The original takes the original GAN generator model where the latent code is fed through only the input layer and revises it to be mapped through an intermediate latent space ($\mathcal{W}$). The style blocks contain Adaptive Instance Normalization blocks which in [65] were found to be the cause of some unsavoury artifacts. These blocks were replaced in StyleGAN2 with a new demodulation method which is applied to the convolutional layer associated weights (W_n). The generator utilizes explicit Gaussian noise (B) to help generate stochastic detail in generated images. This noise is added to a bias variable (b_n) after each style block in the revised model before the data travels to the next convolutional layer.

4.7.1.2 Discriminator

The previous iteration of StyleGAN used progressive growing for the discriminator model, however, in [65] it was advised progressive growing should be swapped for a non-progressive growing method. The authors assessed standard feed-forward networks, skip-connection networks, and residual networks and found that a residual network discriminator provided the best results when combined with a skip-connection generator. ADA was incor-

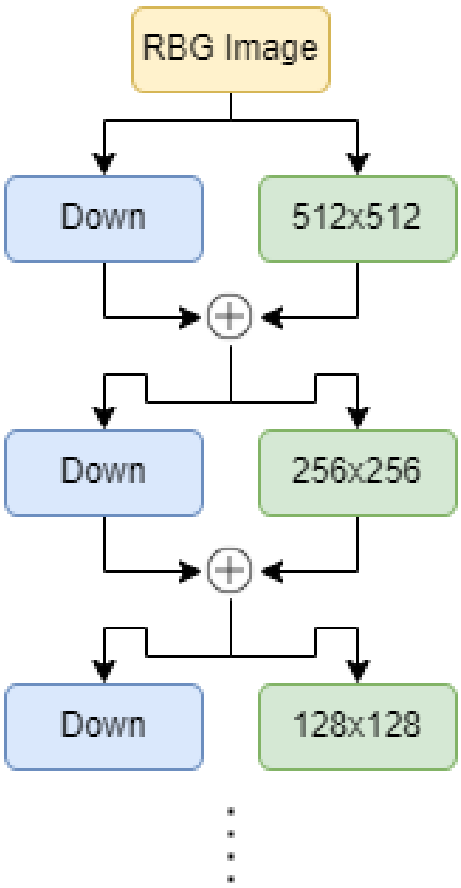


Figure 18: Residual Discriminator Model Used for StyleGAN2

porated into the discriminator model in order to work with small image sets, overfitting is an issue for small datasets and this was addressed using methods described below.

4.7.1.3 Adaptive Discriminator Augmentation (ADA)

ADA involves the manipulation of image data to effectively create more image training data. For example, a one thousand image training set is able to be trained as if there were 25,000,000 images in the dataset. StyleGAN2-ADA trains with a single epoch being made up of four minibatches of one thousand images (1kimg) each known as 1 kimg. Several manipulation techniques are used by StyleGAN2-ADA to achieve these augmentations:

- Pixel Blitting (including flipping the image in the x-direction, 90-degree rotations, and integer translation).
- Geometric transformations (including isotropic scaling, Arbitrary rotation, Anisotropic scaling, and fractional translation).

- Colour manipulation (including brightness, contrast, luma flip, hue rotation, and saturation adjustments).
- Image filtering through various frequency bands.
- Added noise
- Cutting out sections of the image.

Each transformed image is shown to the discriminator in a specific order. A scalar $p \in [0, 1]$ controls the strength of the augmentations in a way that each transformation has a probability of occurring (p) or being skipped ($1 - p$). The same value p is used for all transformations. The generator is guided by the discriminator to only produce clean images and if p remains in a 'safe zone' of $p < 0.8$ there should be no leaking of augmentations into the generated images. The augmentation strength is automatically tuned based on the level of overfitting by assessing the divergence towards zero between real and generated images as overfitting gets worse. Augmentation strength increases every four minibatches and if overfitting is detected then p is adjusted accordingly [64].

4.7.2 Ranking Score

4.7.2.1 Training Time

According to the StyleGAN2-ADA software documentation the time taken per epoch (1 kimg) is 37.7 - 40 seconds using 512x512 images on a similar 2 GPU (NVIDIA v100) setup to what is available for this project. Using similar equipment and settings as BigGAN which was trained at 8.64 seconds per epoch on 128x128 images using an 8GPU v100 setup StyleGAN2-ADA runs at 3.9 - 4.9 seconds per iteration which is twice as fast. Typically, StyleGAN2 seems to converge at 25,000 kimg runs which means at 512px resolution using 2 GPU's it should take approximately 11 days. This is not unusual for GAN training, but it also makes it difficult to adjust hyperparameters as the time tends to add up. This is significantly slower than FastGAN however, and therefore this criterion receives a score slightly better than BigGAN but less than FastGAN.

—**Score:** 7

4.7.2.2 Conditional Training

StyleGAN2-ADA offers conditional training based on image labels arranged in a JSON file. The left side of the JSON object contains the name of the image in the dataset and the right side contains its class label. Documentation provided shows the exact structure required for the class labels file and provides good instruction on how to include it. This was the best out of each method in terms of conditional labelling and therefore is deserving of the highest score.

—**Score:** 10

4.7.2.3 Ease of Using Own Dataset

As mentioned above, this model includes a dataset tool which can be used to arrange data into the correct format. The user simply provides a folder with all the images they wish to use (including a label file), and the tool will place them into folders of 1000 images each, name them appropriately, convert the file type to a .png (required) and create a new label file with the new image names. The tool is simple to use and provides many command line flags to allow for adjustments to how the data is processed. This model ranks the highest in this category because it is extremely easy to use a custom dataset.

—**Score:** 10

4.7.2.4 Small Dataset Performance

StyleGAN2-ADA was specifically designed to work with small datasets without model overfitting. Augmentation proves to be a good method of achieving this and has been tested on image sets ranging in size from 162 to 1944 when testing on small datasets. When compared to the likes of FastGAN it slightly loses out as FastGAN achieved better FID metrics on datasets of one hundred images. However, this model still performs very well on small datasets and is deserving of a high score.

—**Score:** 8

4.7.2.5 Customization

A number of customization options are available when running the training script for this model. These include; the ability to turn on/off different augmentation methods (or

turn it off completely for larger datasets), the ability to turn on/off conditional labelling, options for mirroring the data along the x axis, gamma hyper-parameter override, ADA target value settings, the number of iterations to run, resume functionality (for transfer learning or to resume a failed run), different config options for hyperparameter tuning, and many more. Custom hyper-parameter configurations can be added by editing the training code and while this is not documented, it is easy to do. This model is highly customizable, and these are documented reasonably well meaning this model receives a high score for this criterion.

—Score: 8

4.7.2.6 Evaluation Metrics

Several evaluation metrics are available and can be easily selected using a flag when running the training algorithm. The main metrics of interest are IS, FID, and KID but also included are Precision and Recall (PR) and perpetual path length (PPL). As mentioned above we are interested in KID as it is most suitable for small datasets. Metrics are calculated every n number of iteration when the model produces a training snapshot (this can be set in the command line when training the model, but the default is every fifty iterations of four kimg). A log of all metrics from each snapshot is provided throughout training as a JSON Lines file (.jsonl) which can be used for graphing at a later stage. This is the strongest metric system of all the models and therefore deserves the highest score.

—**Score:** 10

StyleGAN2-ADA Ranking Summary	
Criteria	Weighting
Training Time(8)	7
Conditional Training(10)	10
Own Dataset(3)	10
Small Data Performance(9)	8
Customization(4)	8
Evaluation Metrics(5)	10
Total	53
Total (Weighted)	340

Table 4.5: StyleGAN2-ADA Ranking Overview

———————————————————————— **Total Score:** 340/390

Overall Ranking Summary	
Model	Score
DCGAN	N/A
BigGAN	213
FastGAN	251
StyleGAN2-ADA	**340**

Table 4.6: Overall Model Ranking Summary

From this analysis it could be seen that the best architecture to test the feasibility of generating a dashboard image going forward was the StyleGAN2 model with ADA. This model ranked highly at 340/390 points followed by FastGAN which suffered in the ranking points for its lack of conditional label training ability. From now in this document any time a GAN model is used it should be noted that this is referring to the chosen StyleGAN2 model.

CHAPTER 5

Case Study

The purpose of this chapter is to demonstrate the main aspects of the previous two chapters in a case study which shows a comprehensive start-to-finish implementation of the project. This case study will show the steps taken to collect data and train the algorithm to attempt to produce results for an IoT dashboard and IoT dashboard components, as two different experiments. Results for this chapter will be demonstrated in the subsequent chapter along with full results for all tests.

5.1 Data Collection

To train the model image data must be collected. Although this section demonstrates a case study of IOT dashboard generation, all types of dashboard data were collected and labelled into categories as the StyleGAN2-ADA model we are using supports class conditional training allowing for isolated generation of specific dashboard types. Image data was obtained using the method and tools outlined in chapter 4. A quick summary of these is below:

1. Perform a Google Images or Dribbble search and scroll as far down as possible.
2. Download a number of pictures using the web scraper tool provided (or for Dribbble manually download images of interest as it is against terms of service to web scrape).
3. Download the images using the python image download script provided.

4. Repeat this using various search terms until there are a large number of images to process.

Six different searches were conducted on Google Images resulting in six separate folders of images. A preliminary search through the images revealed images that did not belong in the set such as advertising promotions for dashboard design services and front-page design presentation drawings showing parts of dashboards and many other data anomalies. These images were promptly removed as part of a pre-processing step. These six folders of images were arranged into a single folder ready for processing.

The same was completed on the website Dribbble [81]. Images that were deemed appropriate in the search were manually added to the URL list for download by method of right clicking. This was very time consuming but within the terms and conditions of the website. Approximately seven searches were conducted on Dribbble as the images from this website were of a higher quality both in resolution and content and were therefore a good fit for the training set.

5.2 Data Processing

After the images were collected, they were processed following the processing steps in chapter 3. These steps are summarized below:

1. Duplicate images were removed from the dataset using a detection script.

2. A full inspection of images was completed using the custom made MultiCrop tool and images were cropped to remove borders. Images were deleted if they did not meet a specific criterion.

3. The Multicrop tool was used to crop components from the main images. A label was selected based on the type of dashboard prior to cropping by using the number keys 1-8. This label information was appended to two different existing label files, one for the main dashboard and one for the cropped component.

After this process we were left with a set of 1024 main dashboard images and a set of 5133 dashboard component images. Of these datasets for the main dashboard set 104/1024

images were labelled as IoT dashboards, and for the component set 567/5133 images. This was a small number for each set and could possibly cause some issues due to low variability in samples during training.

Once the images were processed, the StyleGAN2 dataset tool was used to validate the files and arrange them in the appropriate folder structure for the model. Images were split into folders on 1000 images each in the main directory along with the dataset.json label file. All images were renamed in this process and all labels from the original label file were renamed accordingly and added to the new dataset.json labels file. An example is shown below of the directory structure of the main dashboard images:

```
dashboards_full_set_512px/
├── 00000/
│   ├── img00000000.png
│   ├── img00000001.png
│   ├── ...
│   └── img00000999.png
├── 00001/
│   ├── img00000000.png
│   ├── img00000001.png
│   ├── ...
│   └── img00000024.png
└── dataset.json
```

This was repeated for the dashboard components and six folders were produced for this. Once this process was complete the training could begin. Training was split into two separate sections, one set of training for the full dashboard images, and one set of training for the dashboard components.

5.3 Training

This section outlines the steps taken to train the StyleGAN2-ADA model using conditional training. Software setup steps will be discussed along with any issues during the installation, followed by initial experiments to obtain a baseline of training statistics, any code alterations that were made, and the tuning of model hyper-parameters to obtain better quality outcomes.

5.3.1 Software Setup

Four main software applications had to be installed on the machine being used to conduct the experiments. Included is a full list of software requirements to run the StyleGAN2-ADA model:

- Python version 3.7 at least
- Visual Studio 2019 (c++)
- CUDA toolkit 11.0 or later
- PyTorch 1.7.1 or later with cuda support
- As well as eight Python Libraries:
  - click
  - requests
  - tqdm
  - pyspng
  - ninja
  - imageio-ffmpeg v0.4.3
  - Python Image Library (pillow)
  - numpy

The installation began with the Python installation. This was installed and added to the windows system path environment variable. Visual Studio c++ was required as there are custom Pytorch extensions included in StyleGAN2 which rely on run-time compilation with NVCC. A Visual Studio installation was required to install NVCC capabilities, with the 2019 version being required for compatibility reasons. This had to be added to the path environment variable using instructions in the StyleGAN2 documentation. CUDA toolkit was installed and then the compatible version of Pytorch was installed along with all the python libraries.

5.3.1.1 Issues

There were a number of issues during the installation process that must be noted in case of replication.

- CUDA toolkit — There was an issue when running StyleGAN2 where it would be unable to access the GPU and would default to using the CPU. This was found to be due to a mismatch in versions between the CUDA toolkit and the NVIDIA driver installed on the machine. This was fixed by installing the appropriate version of CUDA toolkit for the existing driver software on the machine as this could not be changed due to the machine being used for other purposes.

- Custom PyTorch Extensions — StyleGAN2 as mentioned before has custom PyTorch extensions that require the NVIDIA Cuda Compiler (NVCC). We ran into an issue where these extensions could not be setup and an unexpected error was thrown upon trying to run StyleGAN2. The cause of this issue was thought to be missing system variables for CUDA_PATH, CUDA_PATH_<VERSION>, LIB and LIBPATH. These were added with both CUDA variables pointing to "C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\<VERSION>" and LIB and LIBPATH pointing to "C:\Users\<USER>\AppData\Local\Programs\Python\Python<VERSION>\libs". Once these were all added the errors did not occur again and the extensions were successfully setup.

5.3.2 Initial Experiments

5.3.3 Training

Once the software was setup, it was necessary to conduct some initial experiments to gauge the type of outputs the model would give when using automatic parameters. These experiments were run on the dashboard cropped component dataset as these images are only 256x256 resolution resulting in shorter training time for these initial tests. It was also deemed that any parameter tuning in these experiments would be valid for both datasets based on the fact the components were cropped from the main dashboard images anyway. To run the experiments StyleGAN2 requires the user to use the command line to

(a)

(b)

(c)

(d)

Figure 19: a) Full Overview of Generated Images from the Final Training Snapshot of the First Experiment Using Cropped Dashboard Components. Examples of Generated Components Show What Appears to be a **b)** Calendar, **c)** Credit Card and **d)** a Graph

run the train.py script included in the code repository. There are a number of flags and options that can be set when running this code, the options of note are shown in table 5.1.

An initial experiment was conducted using the following command line options:

- outdir — Set this to a new "results" folder created in the Stylegan2 code folder.
- data — This was set to the folder containing the processed image data from the dataset tool.
- gpus — There were two GPU devices available, so this was set to two.
- snap — The default fifty iterations was found to be too large of a spread. To prevent time lost if a failure were to occur and the model needed to be resumed from a checkpoint, this value was set to ten.

StyleGAN2 Command Line Options		
Flag	Default	Description
outdir	-	Path to where results should be saved.
data	-	Path to input training data.
gpus	1	No. GPU devices available for training.
snap	50	Interval for Metric calculation, fake image set generation and general checkpoint.
metrics	fid50k_full	Set evaluation metric (fid50k_full, kid50k_full, pr50k3_full, is50k).
cond	false	Enables class conditional training using the labels file.
subset	all	Train with this number of images.
mirror	false	Enables x direction flipping of images. Useful for symmetrical images.
cfg	auto	Config option to select (auto, stylegan2, paper256, paper512, paper1024, cifar)
gamma	-	This overrides the R1 gamma value in the chosen configuration.
kimg	25,000	The number of iterations to train.
batch	-	Overrides the config value for the number of images in each training batch.
augpipe	bgc	Augmentation method to use (blit, geom, color, filter, noise, cutout, bg (blit, geometry), bgc, bgcf, bgcfn, bgcfnc).
resume	noresume	Restart the model (if crashed or training from a pre-trained model) from the given .pkl file from the results folder.

Table 5.1: Notable training options for StyleGAN2.

- metrics — As mentioned in chapter 4 we are interested in the Kernel Inception Distance. Therefore, this option was set to kid50k_full.
- cond — This was initially left as the default "false" as this experiment was more to test hyper-parameter and augmentation settings than checking conditional aspects.
- subset — This was left as the default value.

- mirror — Mirror was set to false as dashboard images tend to not be symmetrical around the x-axis due to side menus and arrangement of different components.
- cfg — For the initial experiment auto configuration was set.
- gamma — This flag was not set as the auto config was used meaning gamma was calculated automatically.
- kimg — This flag was not set meaning the training continues until the default 25000 iterations.
- batch — This flag was not set as batch size is automatically calculated in auto configuration.
- augpipe — Initially this was set to default (bgc).
- resume — This test was conducted from scratch, so no resume flag was required.

The experiment was stopped at 19,880 kimg iterations after 5 days as the results did not seem to be improving. We can see from the full overview (19) that the images follow a very similar colour pattern. Most of the images appear to be light coloured and predominately feature a light blue colour rather than all different colours and styles. All generated images had similar features although are very clearly different overall meaning mode collapse had not occurred. Therefore, something else had to be causing the issue and this was discovered to potentially be the style-mixing regularization component of the model.

5.3.3.1 Code Additions

It was found that the file train.py in the main StyleGAN2 code an if statement which checks for a certain dataset configuration (CIFAR dataset from ImageNet). In this section of code, they show the ability to change certain parts of the model architecture. In particular; style-mixing, path length regularization, and discriminator residual skip-connections. These options can be switched off for improvements in training largely diverse datasets such as the CIFAR dataset. To switch off these options and train using our own custom parameters, two new entries were added to the configuration object

"cfg_specs" in the train.py file. There are six options of interest that can be set in this configuration object:

- ref_gpus — The number of GPUs to use. Can be overridden by the command line interface when running the training script.
- kimg — This denotes how many iterations the experiment will train for. This is overridden by the command line option.
- mb — The batch size that will be used to train the network. This is also overridden from the command line option "batch". Many values needed to be tried to evaluate the capability of the GPU. Started from a high number and worked down until the algorithm ran without an "out of memory" error.
- lrate — Learning rate. This is the step size when working towards the loss function minimum.
- gamma — Gamma is a hyper-parameter and can also be set from the command line when the program is run.
- ema — Exponential Moving Average is used to optimize the model and update the weights while training.

These new configurations were named "48-gb-256-complex" and "48-gb-512-complex" to be used with the dashboard components and full images, respectively. Initially, these configurations were set to be the same as already existing configurations consisting of similar sized images from previous experiments conducted by the authors. To allow the selection of this new config it also had to be added by name to the conditional arguments for the run-time flag "cfg". This code is located at the end of the file and dictates which values the user can enter for this command line flag. A conditional statement was then added to detect this configuration and switch off the path length regularization, style-mixing, and discriminator residual skip connections by setting the following variables in the code:

- args.loss_kwargs.pl_weight = 0 (Path length regularization)

- args.loss_kwargs.style_mixing_prob = 0 (Style Mixing)
- args.D_kwargs.architecture = 'orig' (D skip-connections)

A further experiment was conducted with these settings to determine if there was any difference in results with respect to the colour variability and variation of generated components. The model was trained for 1520 kimg iterations as this was a quick test to ensure the change in the code had worked.

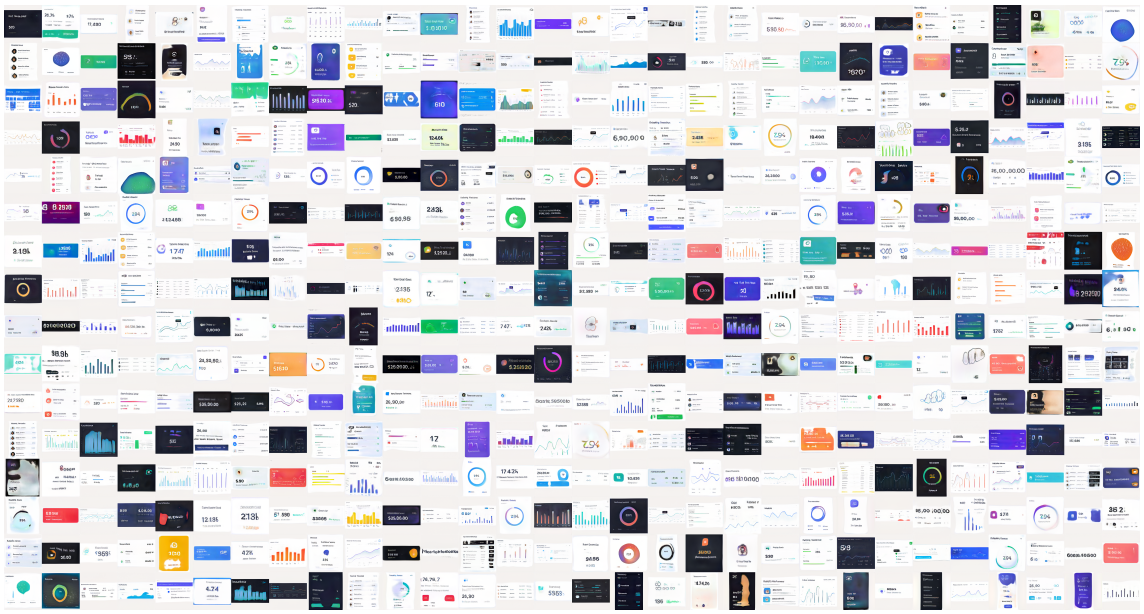


Figure 20: Second Initial Experiment: Style Mixing, Discriminator Residual Skip-Connections, and Path Length Regularization Disabled

As is demonstrated by the overall spread of generated images from the second experiment shown in fig. 20, we can see that there is a great deal more colour variation in the generated samples indicating that this change was a success.

5.3.4 Hyper-parameter Tuning Experiments

After correcting the colour and style issues from the first experiment, the model needed to be tuned to find the best settings before running the full final training experiments. Two different experiments were completed. The first being with dashboard components (256x256) testing different augmentation modes, namely blit, geometry (bg), and blit, geometry, colour (bgc). Two experiments were run for 1520 kimg iterations (12 hours), one with the bgc augmentation mode, and another with bg augmentation mode. Both experiments were run with a gamma value of 50 as this was the median value for gamma

values assessed in the following experiment. KID scores were taken for each experiment to compare results receiving scores of 0.042458 and 0.043998, respectively. As a lower KID is considered better quality, we can see in this experiment that the bgc augmentation method performed better (table 5.2). This was also evaluated with a gamma value of 0.5 to confirm and received KID scores of 0.032332 and 0.060253 respectively. This confirms that the model works better with the default augmentation using pixel blitting, geometrical transforms, and colour augmentation. Other methods such as filter, noise and cutout were not tested as they provided little change in results in the StyleGAN2-ADA paper [64] and due to time constraints it was deemed unnecessary to test these.

Experiment 2.1 Augmentation Adjustments			
Test Number	Augmentation	Gamma	KID$\times 10^3$
1	bg	50	43.998
2	**bgc**	**50**	**42.458**
3	bg	0.5	60.253
4	**bgc**	**0.5**	**32.332**

Table 5.2: Augmentation Pipe Value Experiment KID Results

Once the augmentation method was selected, more experiments were conducted to find suitable values for the gamma hyper-parameter. The dataset used for these experiments was the full sized dashboard images (512x512) set. Three experiments were conducted examining different gamma values and observing the impact on the KID score. Three different gamma values were used, these were 0.5, 50, and 100. 0.5 was chosen as the low value due to its use in the configuration "paper512" and 50, 100 were chosen to observe the effect of higher gamma values on the dataset as it is quite diverse. Other variables such as learning rate and exponential moving average were slightly altered in these experiments as well to see their effect on training time and success of the model. These models were trained for at least 3000 kimg cycles, except for the gamma 0.5 test as this experienced an early mode collapse at around 1500 kimg cycles. Each model was trained for 29 hours except the 0.5 gamma test that experienced failure. The best KID scores were noted as 0.079112 for gamma = 0.5, 0.032061 for gamma = 50, and 0.046816 for gamma = 100. This clearly shows that a gamma of 50 is the best out of the set and was to be used for future training runs. Full training graphs showing KID scores for these experiments can be found in a.

Experiment 2.2 Gamma Adjustments			
Test Number	Augmentation	Gamma	Best KID $\times 10^3$
1	bgc	0.5	79.112
2	**bgc**	**50**	**32.061**
3	bgc	100	46.816

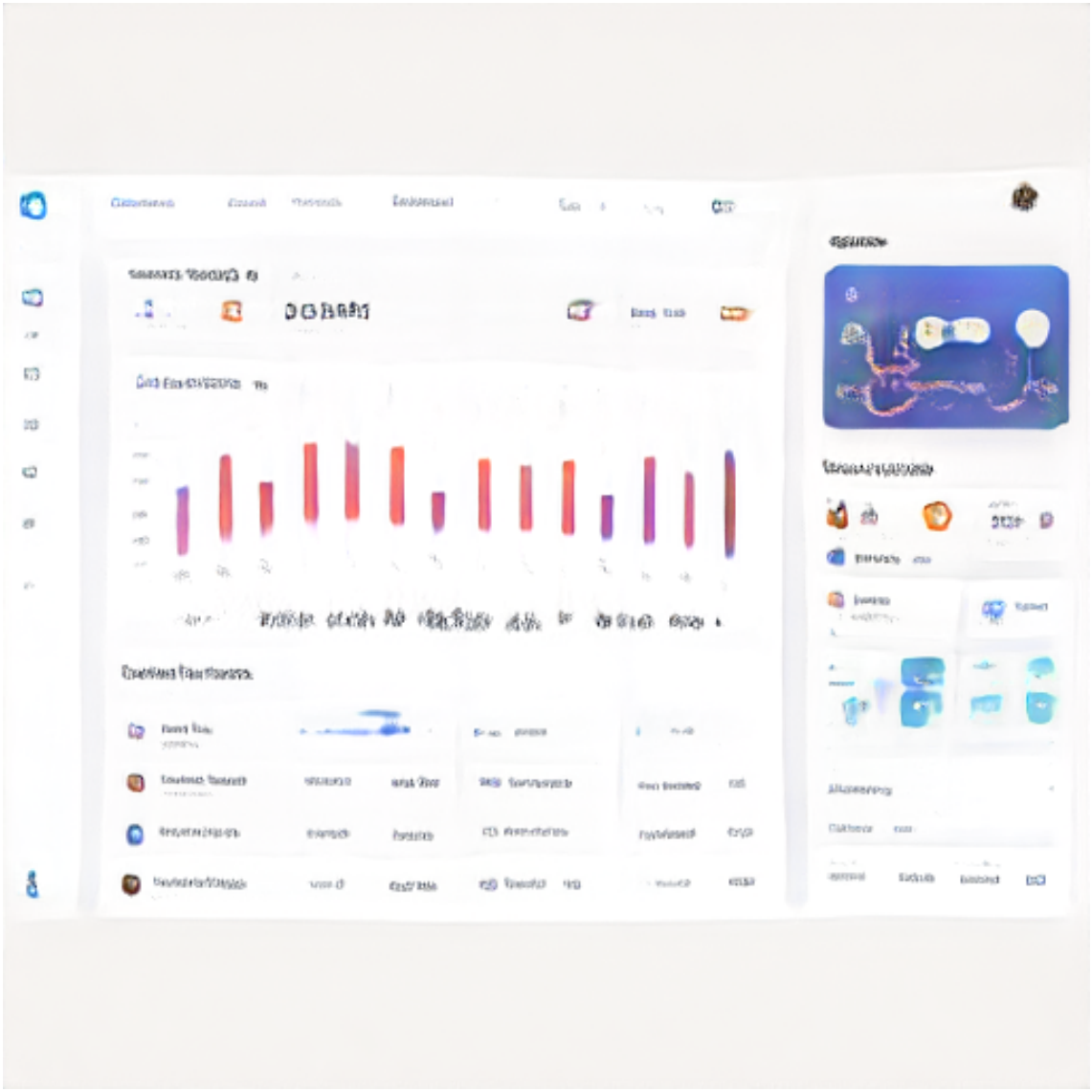
Table 5.3: Gamma Adjustment Experiment KID Results

5.3.5 Conditional Training

Once the experiments for parameter selection were complete the model was then trained using these settings with conditional labelling activated (cond=1 when running the training script). This conditional training was conducted with both dashboard components and main dashboard images and included a number of different class labels in the dataset the most important for this case study being IoT dashboards. Training was conducted for 2 days 9 hours on the dashboard components set, and 2 days 22 hours on the main dashboard image set. The resulting trained models were used to generate full sized images of IoT dashboard components and IoT dashboards. Images were generated using the StyleGAN2-ADA generate.py script which allows the following command line options:

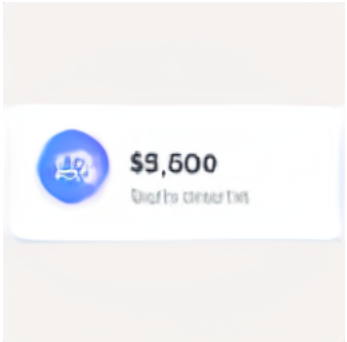
- network — The network .pkl file from the results folder that the user wishes to generate images from. This can be a .pkl file from any stage of training but is usually the latest one.
- seeds — A list of random numbers that will be used to seed the network and generate images.
- trunc — Truncation psi.
- class — The selected class label the user wishes to generate. In this case we used "1" which corresponded to IoT dashboard in our labelling file.
- noise-mode — Can choose from a list of const, random, none where const is the default value.
- projected-w — Projection results file. Not applicable in this case as we do not have a projection results file.
- outdir — The directory where the generations will be saved.

The generate.py script was altered to automatically generate random seeds between a certain range to remove the user from having to manually randomize values. When generating an IoT dashboard, a number of trunc values (0.1, 0.5, 0.7, 1) were used to get a good range of generated images. Using the .pkl file with the highest KID score, four generations were run with the various truncation values and pictured in fig. 21 is some of the best results for both the main dashboard images, and the dashboard components.

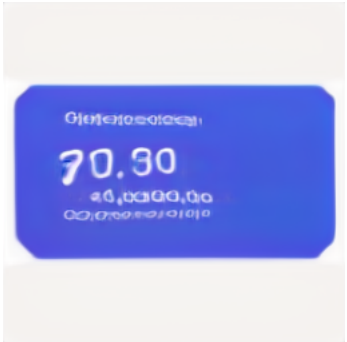


(a)

(b)



(c)



(d)

Figure 21: (a) Generated IoT Dashboard (b,c,d) Generated IoT Dashboard Components

This concludes the case study on training a model and generating an IoT dashboard image. The results from these generations will be discussed in more depth in the following chapter. It can be observed that the main dashboard image shows some form of structure with what appears to be a bar graph, list of some sort and various other com-

ponents that seem to have suffered some distortions. This could be due to the fact there were not a large number of IoT dashboard/component images in the training set and a high variability between samples.

CHAPTER 6

Results

This chapter outlines the results obtained from the final experiments as well as a discussion of these results. Initial training experiments demonstrated in the previous chapter will be discussed along with their training model loss values and discriminator accuracy in identifying fake and real images. Three main initial experiments were conducted:

- An initial test experiment with automatic settings
- A follow up experiment changing augmentation modes and testing these with different gamma values on the dashboard component dataset.
- A final tuning experiment using the chosen augmentation mode from the previous experiment and further evaluating gamma value adjustments using the main dashboard dataset.

For these initial experiments, model loss for the Generator and Discriminator networks will be discussed as well as discriminator accuracy in identifying real and fake images with discussions on why specific parameters were chosen for the main experiments. Following this, the main class conditional experiments involving the dashboard components dataset and main dashboard dataset will be discussed along with loss and accuracy graphs and full KID score graphs for the duration of training. Generated images will be shown for each class label and patterns and common widgets will be discussed with the goal of detailing if there are specific patterns that have been picked up during training for specific dashboard types.

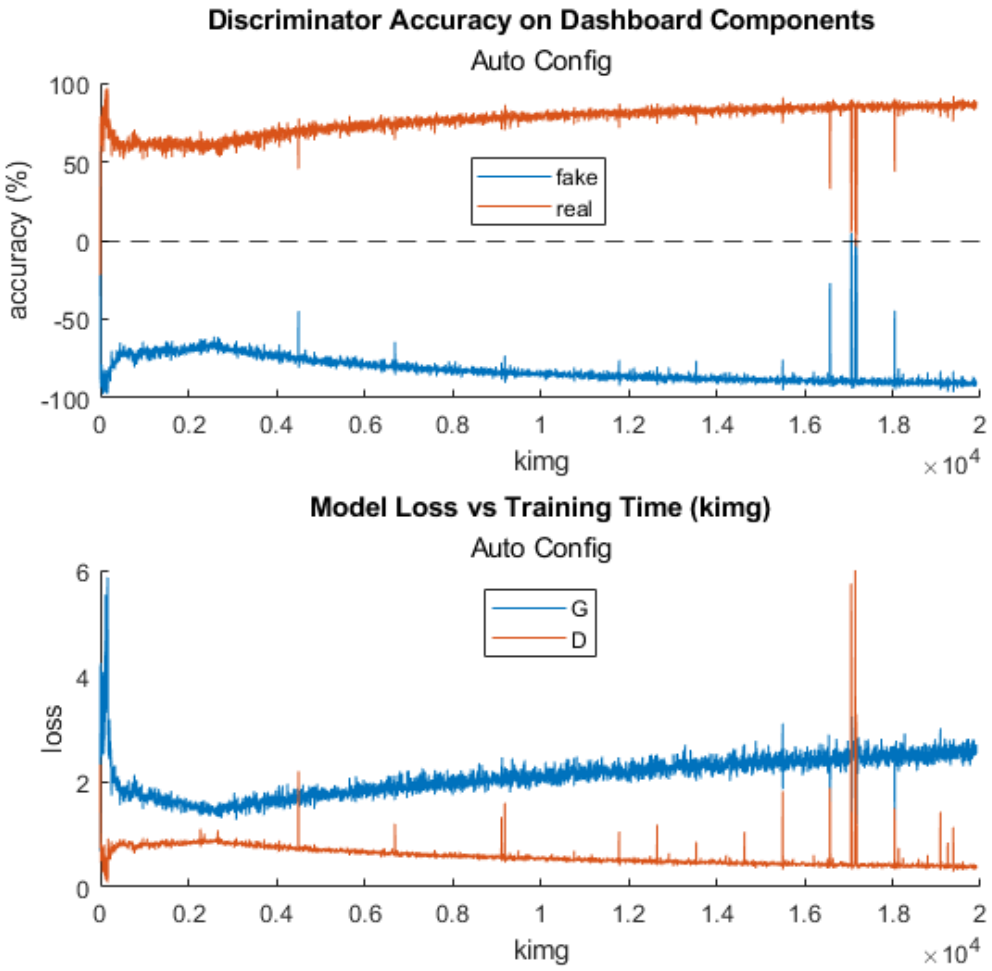


Figure 22: Initial Experiment with Automatic Settings Using the Dashboard Component Dataset (top) Accuracy Scores of Real and Fake Data from the Discriminator (bottom) Model Loss of Generator and Discriminator Over Training Duration.

6.1 Initial Experiment Results

6.1.1 Initial Test Experiment

As mentioned in the previous chapter, this experiment was conducted to evaluate the output using automatic settings and observe if any changes were needed. The dashboard component dataset was used for the first experiments as the images are all acquired from the main dashboards, and they are smaller (256x256 vs 512x512) resulting in faster training times. This experiment was run for 19,880 kimg iterations over a period of 4 days, 23 hours, 16 minutes before the image quality was found to be decreasing and the training was ceased. When plotting the loss of the generator and discriminator (fig. 22), it can be observed that there seems to be an upward trend throughout the duration of the

training, accompanied by a downward trend towards zero for the loss of the discriminator. The point these upward/downward trends occur also marks upward trends in the accuracy graphs for the discriminator shows a type of convergence failure. This tending towards 100 (100%) for real images and -100 (100%) for fake images indicates the discriminator is extremely confident in its decisions of which images are real and which are fake meaning that the discriminator has essentially overpowered the generator. This meant the generator was outputting lower quality images as training progressed and therefore, hyper-parameter tuning was required to attempt to achieve better stability between the two networks.

6.1.2 Hyper-parameter Tuning Experiments

After turning off the style-mixing, path length regularization, and discriminator residual skip-connections, different gamma parameter values, and augmentation modes were adjusted, and the results plotted to see how these changes affected training stability. These are the results from the four experiments discussed in chapter 6 where gamma

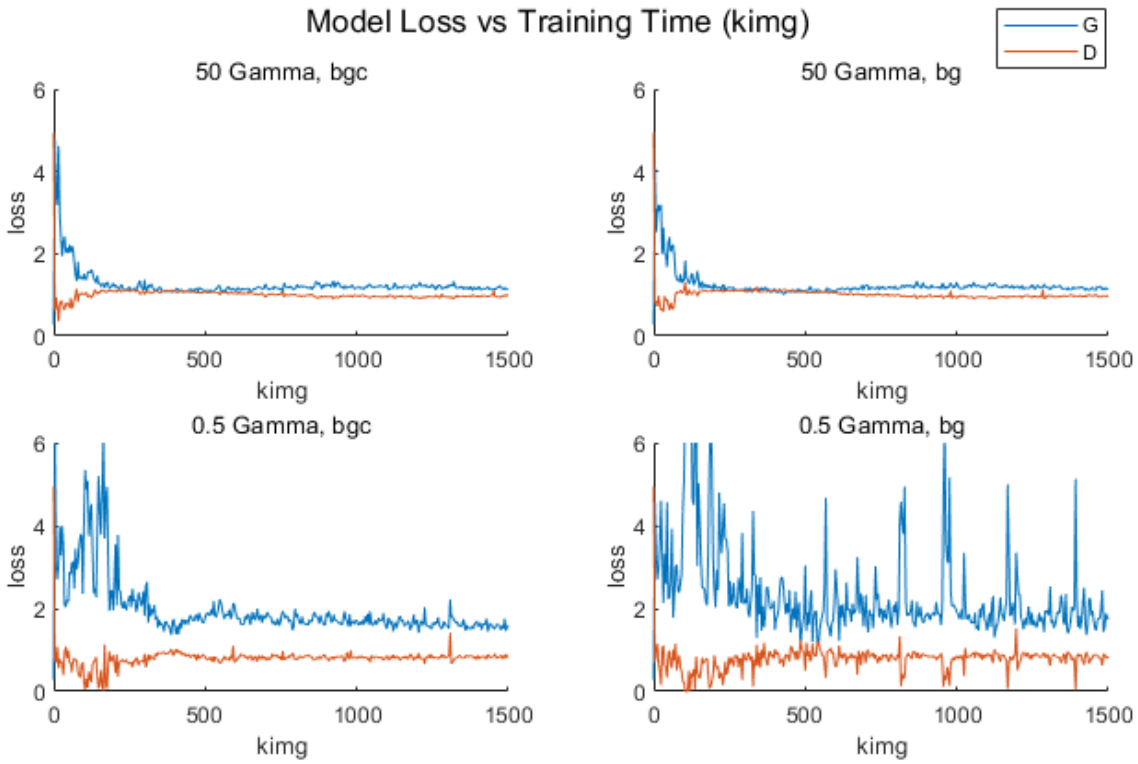


Figure 23: Hyper-parameter Tuning Experiment Loss (Top left) G and D loss for 50 Gamma using blit, geometry, and colour augmentations. (Top right) G and D loss for 50 Gamma using only blit and geometry augmentations. (Bottom left) Same above but with 0.5 gamma. (Bottom right) Same as above but using 0.5 gamma.

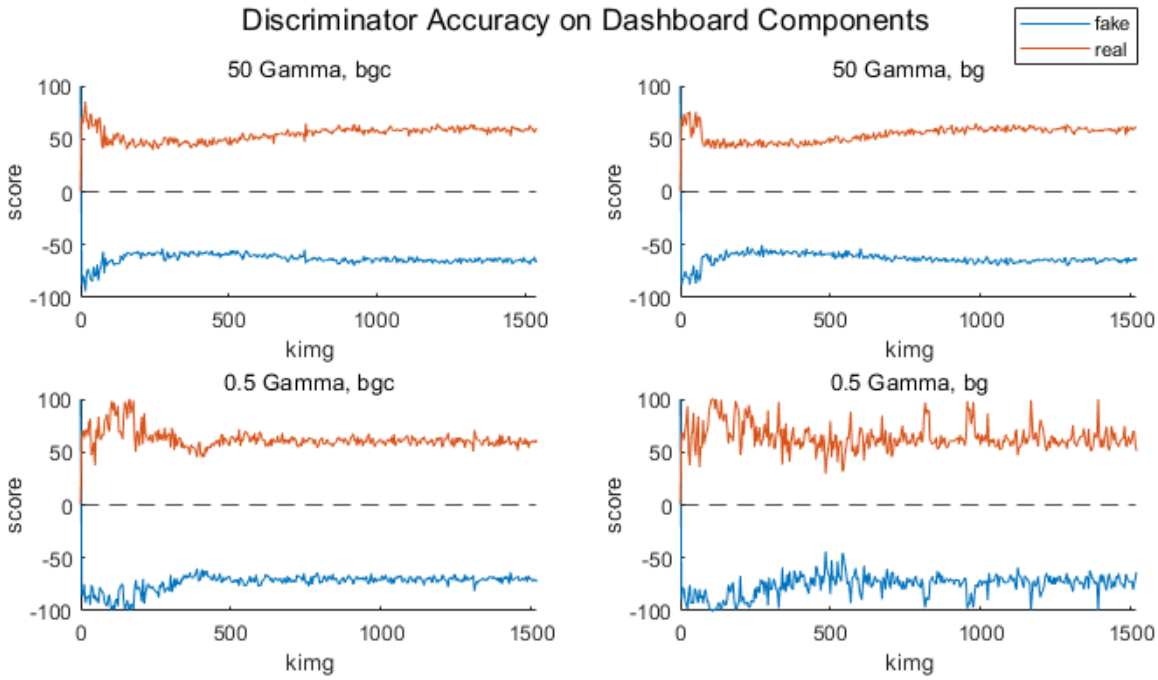


Figure 24: Hyper-parameter Tuning Experiment discriminator accuracy for various gamma and augmentation settings. (Top left) 50 Gamma using bgc augmentation mode. (Top right) 50 Gamma using bg augmentation mode. (Bottom left) 0.5 gamma, bgc augmentation mode. (Bottom right) 0.5 gamma, bg augmentation mode.

values of 50 and 0.5 were used along with two different augmentation modes, blit, geometry and colour (bgc), and blit, geometry (bg). The experiments were run for 1520 kimg iterations over a period of 11 hours each. Fig. 23 shows a stark difference between the two gamma values with the 50-gamma settings outperforming the 0.5 gamma settings in terms of stability. Failure modes do not seem to be present in the tests with the 50 gamma setting as both the generator loss and the discriminator loss begin far apart and settle to similar values with the generator being slightly higher. The discriminator is not tending towards zero rapidly, and the generator loss is not rising as we saw in fig. 22. Mode collapse does not appear to be occurring in the short duration of the experiment and this was confirmed by visual inspections of the training outputs shown in appendix a. The loss functions for the 0.5 gamma setting for both bgc and bg augmentation modes appears to be less stable, especially the bg augmentation mode which can be seen in fig. 23 which shows extremely large oscillations in generator loss.

The accuracy plot of this experiment set shown in fig. 24 shows that the 50 gamma experiments have similar accuracy beginning at 100% before settling just above 50%. A similar pattern occurs with the 0.5 gamma bgc experiment with slightly more accuracy

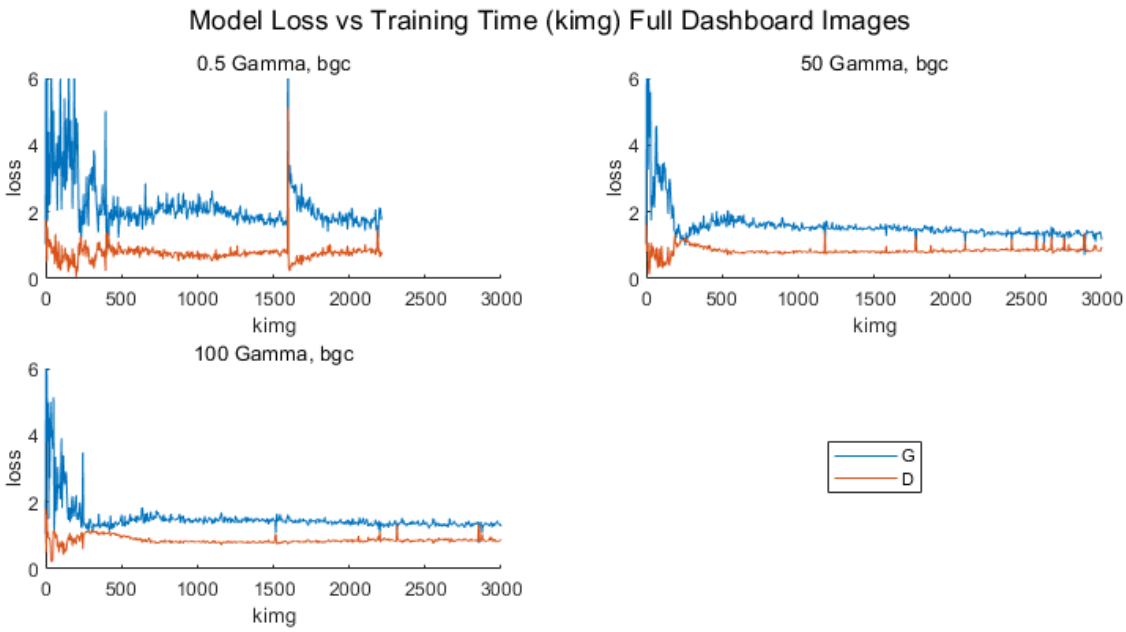


Figure 25: Generator and Discriminator loss trained on the main dashboard image dataset with different gamma values to confirm whether higher gamma values were more beneficial on the full dashboard images. (Top left) 0.5 gamma, (Top right) 50 gamma, (Bottom left) 100 gamma. All using bgc augmentation mode.

instability at the beginning but ultimately settling at approximately the same 52 - 57% values at the end of the 1520 kimg iterations. 0.5 gamma with bg augmentation shows a similar 50% accuracy but with much more variance meaning it is unstable compared to the other test runs. Overall, these tests show that stability can be increased with a higher gamma value and bgc augmentation modes. Therefore, it was decided that these higher gamma values would be assessed on the main dashboard images as a final test before the main experiment conditional training was conducted with the final selected values. The experiments on the main dashboard images were necessary to ensure the parameter tuning worked on both dashboard components and main dashboard images. It was hypothesised that this would be the case, and this needed to be confirmed.

Using the full dashboard image dataset the model was trained for at least 3000 kimg iterations for three different gamma values using bgc augmentation mode. The three different gamma values that were chosen were:

- 0.5 — This was chosen to confirm if the previous test results with this value were not just exclusive to the dashboard component dataset.
- 50 — This was the best performing gamma value in the previous tests so therefore

it was tested on the main dashboard dataset.

- 100 — This value was chosen out of curiosity for a much larger gamma value and its effect on training.

Each of the three experiments ran for 1 day, 5 hours to reach 3000 kimg iterations. The loss scores shown in fig. 25 confirm that the 0.5 gamma score is not appropriate for this image generation problem as it remains unstable on the full dashboard images. The loss for the generator was reasonably unstable at the beginning of the training for this experiment and then settled somewhat, however, there appears to be a large spike in both loss plots around 1600 kimg iterations. After this spike, the loss seems to attempt to correct itself but a visual inspection showing that the latest generated images were identical confirmed that the model had failed due to mode collapse. This is backed up by the sudden drop towards zero of the real/fake accuracy for this test in fig. 26. Similar drops in accuracy can be seen in the graphs for 50 gamma and 100 gamma as well, although these seem to recover in the next snapshot, with the 100-gamma test seemingly the most stable in this regard. The loss functions for 50 gamma and 100 gamma training cycles seem relatively similar except with slightly more discriminator loss spikes in the 50-gamma test. Training was carried on further for the 100 gamma, and 50-gamma tests past 3000 iterations to test their performance over time. The 100-gamma test showed signs of deteriorating image quality at 3300kimg iterations with KID scores dropping rapidly. Eventually, signs of mode collapse appeared with several similar images appearing in the generated results and the training was halted at 4919kimg. KID scores from the 50 gamma experiment began decreasing at 4798kimg iterations with mode collapse seemingly beginning to appear at 6200kimg iterations with the training being halted at 8588kimg iterations when image quality was reduced to extremely poor generations.

Based on these initial experiments it was determined that moving forward a gamma value of 50 with augmentation mode bgc should be used as well as ensuring style-mixing, path length regularization, and discriminator residual skip-connections remain off. The next section discusses the main experiments with these parameters using the dashboard components dataset and the main dashboard dataset.

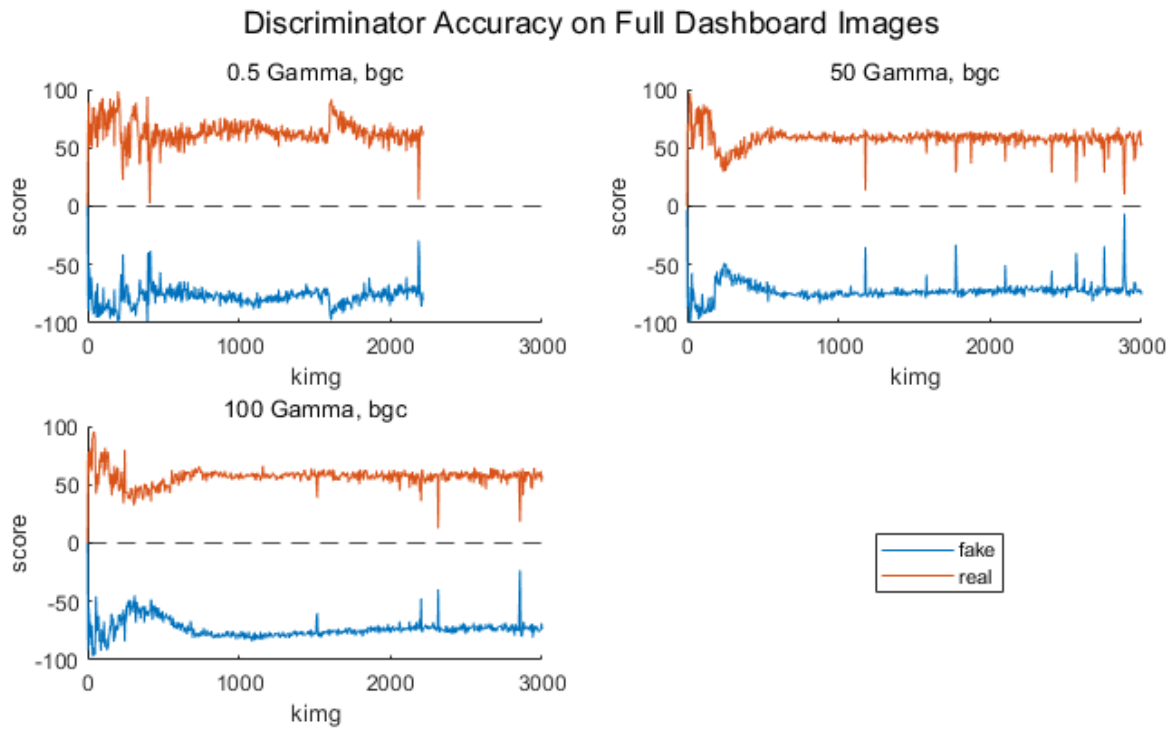


Figure 26: Discriminator accuracy on real and fake images using different gamma levels trained on the main dashboard image dataset. (Top left) 0.5 gamma, (Top right) 50 gamma, (Bottom left) 100 gamma. All using bgc augmentation mode.

6.2 Dashboard Components (256x256)

The first experiment involved training on a dataset of 5133 cropped dashboard components. This experiment was run with conditional labelling turned on with eight categories of dashboard included in the labelling. The experiment was run for a total of 2 days, 8 hours, and 38 minutes for 8656 kimg iterations or 2147 ticks averaging 18.5 seconds per kimg with added snapshot and maintenance time. 13.7-14gb of GPU memory was used during each tick of the algorithm and augmentation p value reached 0.655 by the final iteration which is still in "safe zone" in terms of augmentations leaking into the generated images. Training was halted as the KID score for the images began rapidly decreasing showing the images were progressively getting worse.

6.2.1 Training Loss and Scores

As we can see from the loss function the training remains stable for 7000 kimg cycles before relatively large spikes appear in the generator loss. These spikes can also be seen to a lesser extent in the discriminator loss as well as a slight overall increase in slope. During

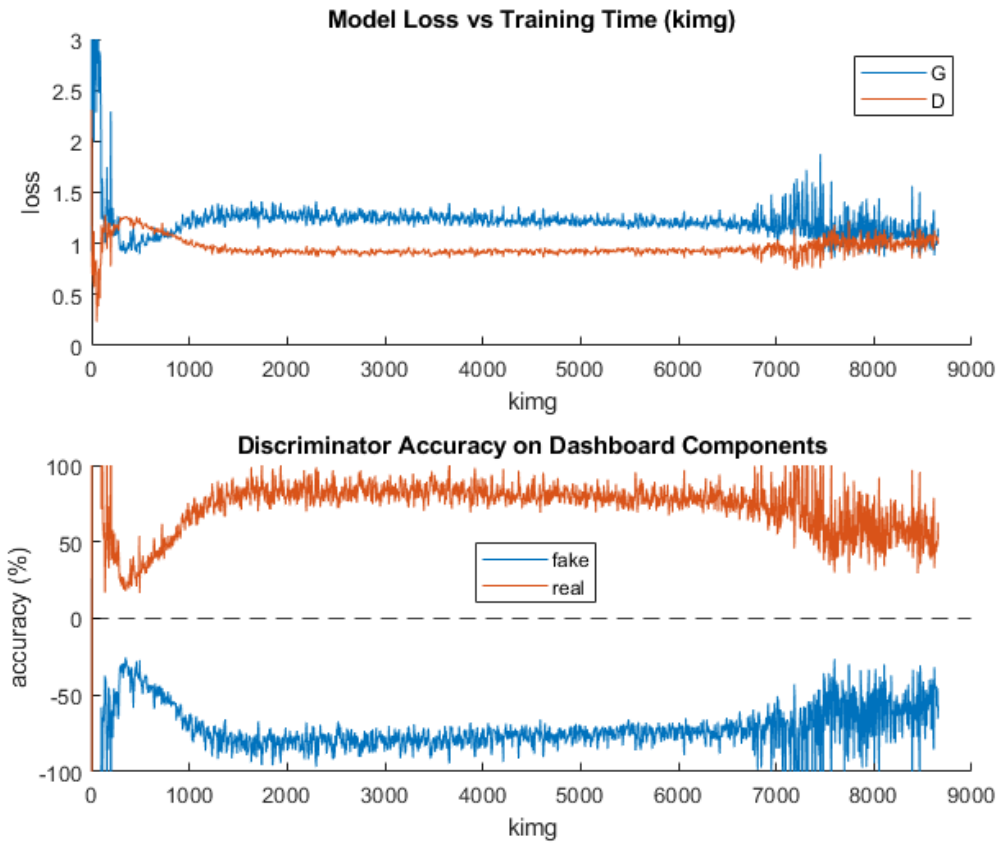


Figure 27: Training Statistics for Dashboard Component Training (top) Generator Loss vs Discriminator Loss plotted over the duration of training (bottom) Discriminator Accuracy on Real dataset Images and Fake Generated Images

GAN training spikes in the loss functions are reasonably normal and as we observe the accuracy, we can see that it is tending to 50% for whether the image is real or fake (with variance) meaning the discriminator is essentially flipping a coin. Usually when this occurs it means the GAN has reached convergence, however, in this case the images generated seem to have multiple duplicates. This potentially indicates mode collapse occurred during the training. This can be seen by fig. 28 where a blue circle is present in a number of pictures as well as a specific type of bar graph sloping to the right with three distinct bars. The image quality is not up to the standard of the real images based upon inspection and a human can easily identify that these are not real images. This means the generator has found a way to fool the discriminator easily with a specific type of dashboard component image.

Figure 28: Training Checkpoint Output at 8628 kimg Iterations Showing Several Similar Images/Colours Indicating Possible Mode Collapse

The KID score vs training duration in fig. 29 further demonstrates that this model may be suffering from mode collapse where the score troughs and then begins increasing, indicating a reduction in quality from the point where the discriminator accuracy tends to 50%.

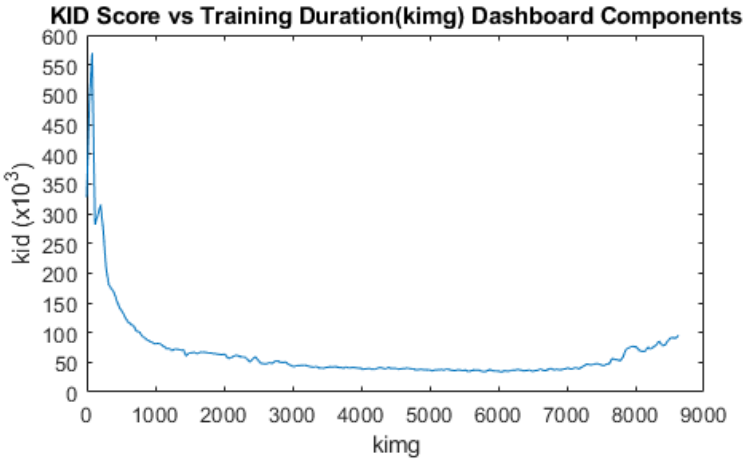


Figure 29: KID score for training duration of dashboard component dataset.

6.2.2 Discussion

Due to the appearance of mode collapse near the end of the training the model was not trained to the level it could have reached should this have not occurred. This led to image generations of reasonably low quality.

KID Score Comparison with Previous Research	
Dataset	KID ($\times 10^3$)
MetFaces	2.41
BreCaHAD	2.88
AFHQCat	0.66
AFHQDog	1.16
AFHQWild	0.45
Dashboard Components	**33.64**

Table 6.1: KID score comparison with KID score data from [64] for dashboard components.

The maximum KID score achieved in this training experiment was 33.64 (0.03364×10^3). Compared to results obtained from the StyleGAN2-ADA paper [64] this is significantly worse than the highest score of 2.88 from their experiments. This can be attributed to these issues with mode collapse as the model was unable to improve the quality of the images through further training. In future experiments it would be interesting to adjust a number of other parameters but due to time constraints, namely the large amount of time required to run experiments, this was unable to be achieved in this study. A number of other factors play a part in the differences in image quality between class label images. One of the main contributors to this is the spread of labels in the dataset.

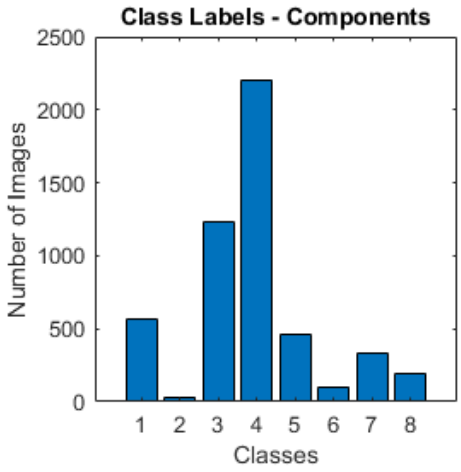


Figure 30: Class label spread for dashboard components. Graph representation.

Images/Class - Components		
Class	Label	No. Img
IoT	1	567
Agriculture	2	34
Finance	3	1229
Analytics	4	2207
Misc.	5	459
Fitness	6	105
Project	7	337
Social	8	195
	Total:	5133

Table 6.2: Class label spread for dashboard components. Table representation.

The conditional class label spread can be observed in fig. 30 and table 6.2 and definitively shows a large amount of analytical and finance dashboard components while displaying a lacking in all other areas with agriculture and fitness being the worse off. This section discusses the implications of this class label spread as well as other affecting components over each of the conditional label classes. A number of generated results can be seen in appendix A for each type of dashboard component.

IoT: Starting with the IoT dashboard components, we can see that the generated image quality is relatively low. This seems to be in part due to this low amount of labelled image data (567 images) for this particular class, as well as a large amount of image variance in cropped images. This variance can be put down to the fact that a lot of IoT dashboard components include camera feeds or other such widgets of extremely high detail that is not present in any more than a single image. This leads to a large amount of blurring with areas of the images unable to be defined at a glance. Dark coloured components featured predominantly here, showing that a number of the training images for IoT dashboards must be set to dark mode.

Agriculture: The agriculture and fitness dashboard components were dominated with images of numbers with subtext as we can observe multiple examples of this in Appendix a §a-II and §a-VI. Although the images of text may have had a high variance of text combinations as the text is non-readable. The numbers however are all readable to some extent. There was a lack of data available for the training of this class label possibly contributing to the lack of variability in the generations.

Finance: Finance dashboard components do not appear to be of good quality which is odd since they are the second most represented class in the dataset (1229 images). This could be due to a high data variability as well as simply the training time not being sufficient due to the mode collapse issues. Some of the generated components appeared to resemble credit cards, lists and bar graphs were also noted.

Analytics: The analytics components are one of the more defined and identify-able sets with a diverse spread of components. This could be because there are 2207 images labelled as this class meaning there is a great deal more learning opportunity. As well as this, the variance of images in this set could be smaller meaning more examples of similar

components for the model to learn. A number of pie and line charts as well as progress bars, circular or linear were observed for this class label set.

Miscellaneous: The miscellaneous set of dashboard components appeared to be the most defined out of all. These were dashboard components that were extracted from dashboards which did not fit into a specific category. Line and bar graphs can be observed in these generated examples as well as what appears to be a list of users.

Fitness: Fitness dashboard component generations seemed to suffer the same issues as the agricultural dashboard components due to low numbers of labelled images in the dataset. Components were predominantly made up of text. No real patterns could be seen in the components.

Project Management: The project management dashboard components generated show a clear example of a calendar and a task progress widget as well as the usual graphs which seem to be present in nearly all categories. The project management components have been learned better than several other categories despite only containing 337 images which is less than the IoT class (567 images). The reason the project management components seem to outperform the IoT components appears to be due to less variation and less complicated images in the training set.

Social: The social dashboard components are like the analytical components where they contain a number of circular progress bars. The number of images in the dataset labelled as social dashboards was reasonably low (195) not too dissimilar to the numbers for the fitness dashboards. Perhaps the presence of circular progress bars indicates that some inspiration was drawn from the analytics class when learning these components. Otherwise, the rest of the components seemed to be text based similar to the agriculture and fitness dashboards which is consistent with the class labels of low image numbers.

Throughout the training it became apparent that there was an issue with text and profile picture learning. It is assumed these occurred due to the extremely high variability in text present in the images leading to all the text almost being merged in the generated images. Numbers seemed less affected by this issue which could be placed down to the fact that there are only ten individual number characters leading to less combinations. It could also be that the designers of these dashboard in the dataset tend to use similar

filler numbers when putting placeholder data in their dashboards. With regard to profile picture learning, there is a great amount of detail in a person's face, which typically takes StyleGAN2 25,000 kimg cycles to learn using a face image dataset of 10's of thousands of images. This aspect could potentially be improved by training upon a pre-trained model using facial data. Overall, the dashboard components generations are not the best quality and would benefit from increased training time if the issue of mode collapse could be solved.

6.3 Dashboards (512x512)

The second experiment involved training the model on a dataset of 1024 full dashboard images. Conditional labelling was switched on for this test and the same labelling categories were used as for the dashboard components. The experiment was run for 2 days, 22 hours and 5 minutes for 6947 kimg iterations (1723 ticks). The average time per kimg was 29.8 seconds and each tick used 26-27 GB of GPU memory. The augmentation p value reached 0.956 by the end of the training which is above the "safe zone" of 0.8 which means that augmentations may have leaked into the generated images. Training was halted as the KID score began increasing drastically indicating the image quality was becoming worse.

6.3.1 Training Loss and Scores

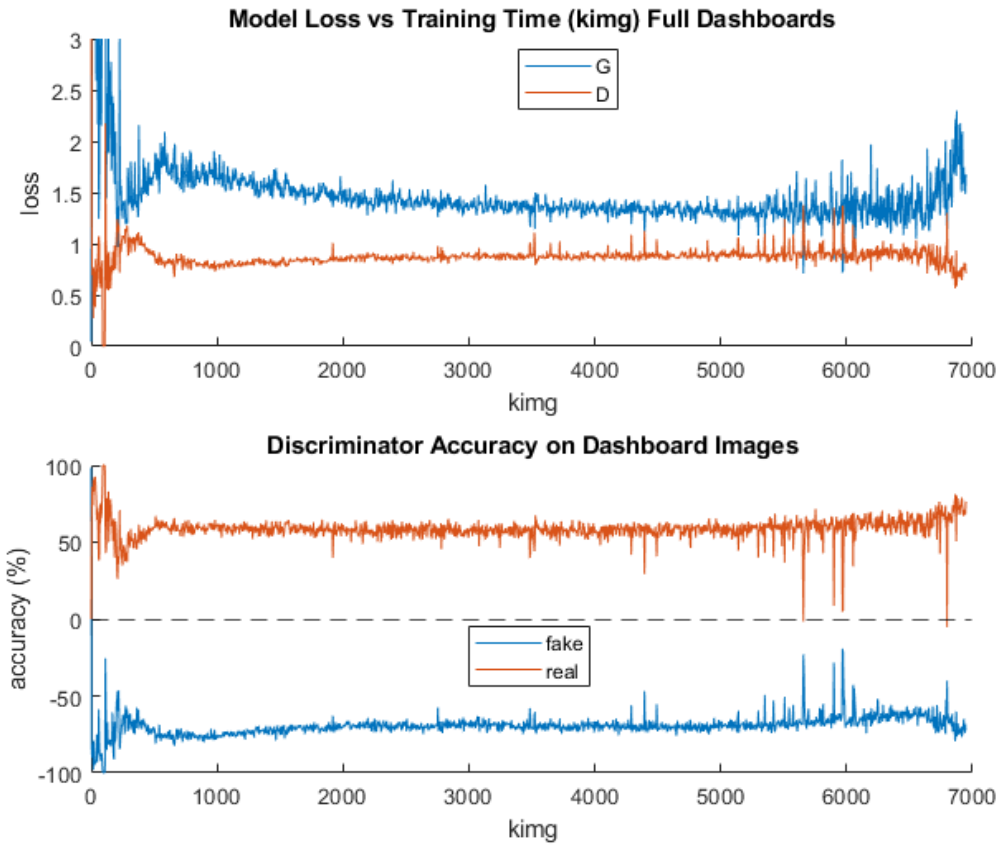


Figure 31: Training statistics for full dashboard images (top) Generator loss and Discriminator loss vs training duration (bottom) Discriminator accuracy on real dataset images and fake generated images.

Like the dashboard components training loss graph, the full dashboard image graph

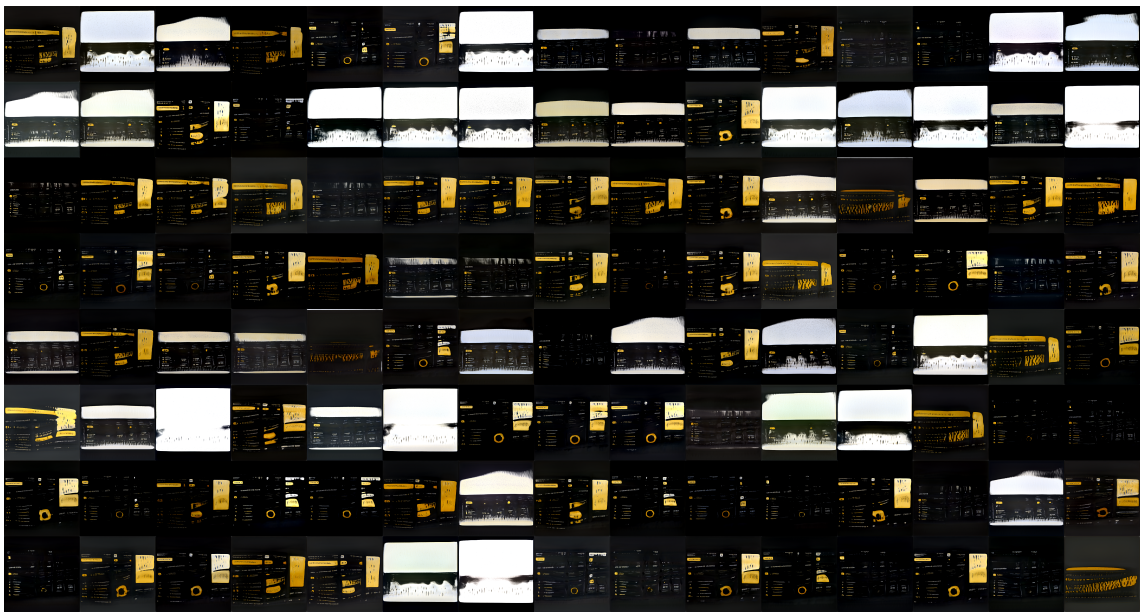


Figure 32: Final training snapshot (6935 kimg iterations) image generation before termination due to GAN failure.

shows a decrease in generator loss and increase in discriminator loss indicating that it is training correctly, however near the 7000 kimg mark the loss begins to separate with the generator loss beginning to rise and the discriminator loss dropping. At the same point, discriminator accuracy which was tending to 50% begins to rise towards 100%. This paired with a large spike in KID score shown in fig. 33 would indicate that the model has experienced a type of GAN failure. Looking at the training image output represented in fig. 32, we can see that this model is likely experiencing mode collapse as each image is nearly identical. Compounding this potential mode collapse, these separations of loss values with D tending downwards and G increasing show the discriminator potentially out training the generator without providing appropriate feedback as well. We can see this failure beginning to occur at around 6500 kimg cycles where the loss and accuracy graphs begin to separate. Image generations for this experiment can be seen in Appendix b.

6.3.2 Discussion

Similar to the dashboard components training, GAN training failure was an issue. However, prior to this failure, a passable KID score was received for one of the generations which was lower than that of the dashboard components (28.89 vs 33.64). While this is still not as good as any of the scores from the StyleGAN2-ADA paper shown in table. 6.3

it still produced some results where patterns could be recognised between the different class labels.

KID Score Comparison with Previous Research	
Dataset	KID ($\times 10^3$)
MetFaces	2.41
BreCaHAD	2.88
AFHQCat	0.66
AFHQDog	1.16
AFHQWild	0.45
Dashboards	**28.89**

Table 6.3: KID score comparison with KID score data from [64] for dashboard images.

IoT: Beginning with IoT dashboards, we can see from the generated images that there appears to be a bar graph present on the left/upper left side of 5/10 of the dashboards generated. In the top right corner, it can be observed that there is a mix of multiple different colours in a single widget on 3 of the dashboards which indicates possibly a camera widget as there were a number of these in the dataset. As previously explained in the components section a camera widget would be extremely different in each training image and carry a large amount of image data to be learnt leading to a mix of colours and patterns that does not appear to resemble anything. This is a point for future experiments whether the training of this can be improved using pre-trained models with a prior knowledge of

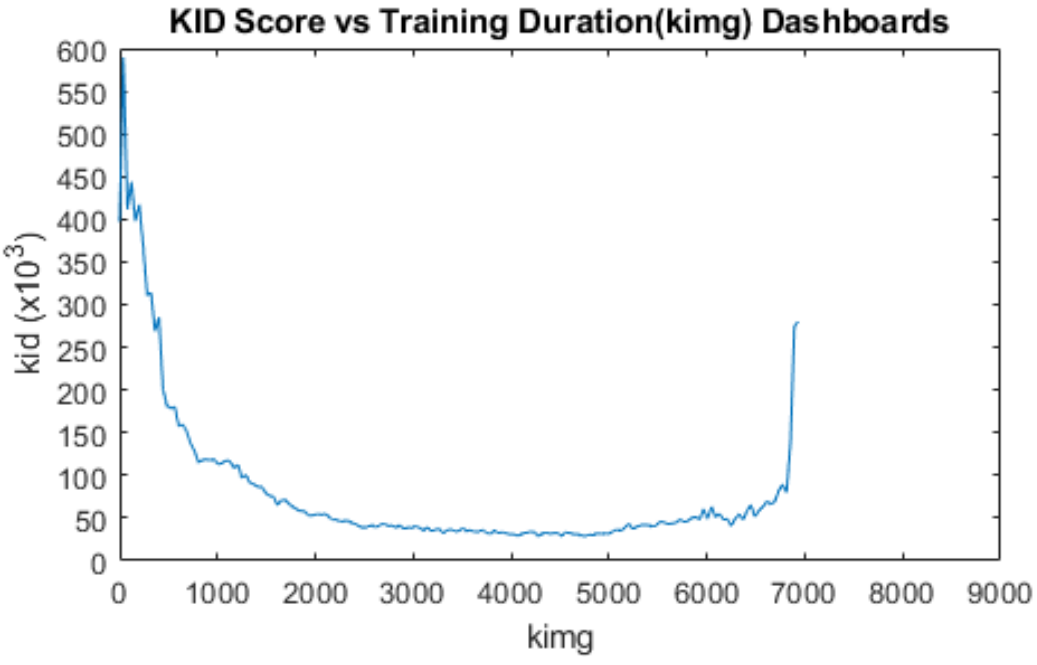


Figure 33: KID scores for full dashboard images for full training duration.

items in a home environment. Other widgets present include a line graph, and what appears to be a list.

Agriculture: The agriculture dashboards suffered from a lack of data. As seen by table. 34 there were only 5 dashboards in this dataset and upon inspection several of these were found to be the same image meaning they had by-passed the duplicate image detection script. This led to the generated agriculture dashboards being very similar, almost memorised by the model which is a symptom of repeated images in a GAN dataset. This means that the experiment failed for this type of dashboard and any patterns obtained from this would essentially be copying one or two types of dashboards, so this class label can be ignored.

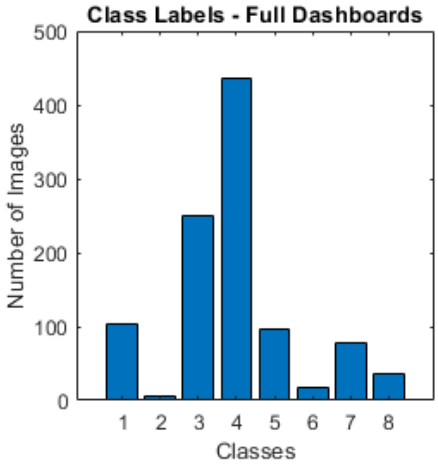


Figure 34: Class label spread for full dashboard images. Graph representation.

Images/Class - Dashboards		
Class	Label	No. Img
IoT	1	104
Agriculture	2	5
Finance	3	249
Analytical	4	436
Misc	5	97
Fitness	6	17
Project Management	7	79
Social	8	37
	Total:	1024

Table 6.4: Class label spread for full dashboard images. Table representation.

Finance: Similar to the dashboard components, the class label spread shown by fig. 34 and table 6.4 indicates that analytics and finance dashboards have the largest representation in the dataset which means any patterns found will less likely be memorised from the dataset. For the finance dashboards bar graphs can be observed again in similar positions to the IoT generations in 4/10 dashboards. A line graph can also be seen in the same position indicating it is favourable to include graphs on the upper left according to this dataset. Finance dashboards seem to include various lists, but they are too blurry to distinguish what is contained in them. This is possibly due to either augmentation leaking or the fact that they simply were not able to train enough. There seems to be attempts at generating an image of a credit card on a number of generations, however, these are very low quality and hard to distinguish. The dataset did contain a number of finance

dashboards with credit cards featured so this would make sense. These seem to be positioned on the right side of the dashboard, either middle or bottom and appear on 3/10 dashboards.

Analytics Analytics dashboards also displayed any graphs on the middle left/top left side of the dashboard and prominently featured bar and line graphs along with what appears to be circular progress bars. A number of what seems to be analytical indicators appear over the top of graphing areas consistently, with the bottom of the dashboard seemingly reserved for a list of some sort. Augmentation leakage seems quite prevalent in these images as there seems to be a number of artifacts. Image variability seems to have played a factor in the training process of these images as well as they appear quite blurry, and widgets are not so defined. This could be due to the different vertical image positioning when the dataset images were cropped and padded with borders where some images were vertically larger than others leading to the blurring we see.

Miscellaneous: The dataset section dedicated to miscellaneous dashboards was remarkably diverse as these were dashboards that did not fit into a specific category therefore, many patterns were not expected to be found in these image generations. This has proven to be true as each dashboard generated appears to be different from the last. Each one has quite blurred components indicating the high variability of component positioning in the dataset for this class label. General distinguishable widgets are still seen such as the likes of bar graphs and line graphs, but their positioning is inconsistent.

Fitness: Fitness dashboards were only represented by 17 dashboard images in the dataset similar to the agricultural dashboards. Some images are similar indicating potential memorisation in this class label set which makes sense due to the sparse number of examples.

Project Management: Project management dashboards tended to have what appears to be calendar components in the top right of the dashboard as seen in two of the generated examples. The usual graphs were also present in a number of the dashboards with a preference for both line and bar graphs. The positioning of these graphs seemed to be either top left or centre. A number of lists were included in these dashboards as well with one of the dashboards being exclusively a list without any other components. The other

dashboard displaying a list included it across the bottom of the dashboard.

Social: Social dashboards also seemed to suffer with low data numbers at 37 class labels in the dataset. These were similar to analytics dashboards in design and included line/bar graphs generally in the top left, with analytical widgets positioned above across the length of the main dashboard. Circular progress widgets were quite prevalent in these dashboards as well. Lists with what appears to be profile images were generally positioned on the right-hand side spanning the length of the dashboard in a number of generated images.

Overall the dashboard images it was noticed that a side bar menu was present in a large number of dashboards. Menu placement plays an important role in the usability of the dashboard, and this is an interesting pattern to note.

The image quality for the generated dashboard components, and generated dashboard images was relatively low compared to the scores obtained in other experiments in the StyeGAN2-ADA paper [64]. In future experiments hyper-parameters and model structure would have to be tweaked numerous times to achieve better results and training stability. There is a high variability in the data that needs to be accounted for with these parameter values and model alterations may be required to account for this.

CHAPTER 7

Conclusions and Recommendations

With a large amount of information being created every day in multiple sectors, and a large number of automation solutions being created the need for quick design dashboards has arisen. Designing a dashboard is time consuming and requires staff that certain engineering companies may not have on hand. This thesis sought to determine the feasibility of designing and optimizing a dashboard GUI using machine learning. After a literature review was performed, it was determined the best machine learning method to use for this problem was the Generative Adversarial Network (GAN) and although research had been conducted into creating mobile apps using machine learning and an existing pool of cropped components, more understanding was needed on fully generating a dashboard from a training set. A number of initial parameter testing experiments were performed followed by two main experiments, one with cropped dashboard components and full dashboard images, with each set labelled as one of eight different dashboard types. The main objective was to observe the training and find any patterns common to distinct types of labelled dashboard and receive a number of newly generated dashboard and dashboard component designs.

In order to conduct the training experiments a training dataset had to be constructed. This was achieved through the use of a Google Images, a design website called Dribbble, and a number of purpose-built program scripts. These scripts included an image download script which took a list of image URLs and downloaded them into a specific folder, an image duplicate detection and deletion script, and a multi-crop image processing GUI purpose built for this project. Images were passed through a set of criteria and processed

to produce a set of 5133 cropped dashboard components 256x256px in size, and 1024 full dashboard images 512x512px in size.

Following the construction of the datasets a GAN architecture had to be selected. It was decided to assess the feasibility of this problem an existing architecture should be selected due to time constraints. A number of architectures were considered and ranked using a weighted set of criteria, ultimately settling on the StyleGAN2-ADA architecture developed over a number of years by NVIDIA cooperation [40], [57], [64], [65].

Using the StyleGAN2-ADA model a number of initial tests were run to obtain a grasp on the various hyper-parameter and model settings. Automatic settings were tested first which showed a number of model structure settings had to be disabled to prevent colour mixing of images while training. Further initial tests showed that parameter values gamma=50 and augmentation mode bgc (blit, geometry and colour) provided the best results regarding kernel inception distance scoring. The findings from these initial experiments provided the basis for the main two experiments where the model was trained until failure for both the dashboard components and full dashboard images. It was discovered that mode collapse had occurred for both these experiments as the images produced after a number of days became terribly similar and began to suffer from reduced quality. Prior to failure a number of notes could be made about the patterns discovered in generated dashboard images:

- Side bar menus tend to be the most prevalent over all dashboard types.
- IoT —
  - Although the images were indistinguishable due to training failure, it was deduced by observing the dataset that the model was attempting to generate a camera feed, generally positioned in the top right of the dashboard.
  - Bar graphs seemed to be the most prevalent graph type followed by line and were generally located in the top left/middle left sections of the dashboard.
- Agriculture dashboards suffered from lack of data (5 images) and the model memorised the dataset. These results were discarded.
- Finance —

  - Bar graphs were noted to occur in 4/10 dashboards in the same position as IoT dashboards.
  - Items possibly resembling credit cards were noted, typically appear in the middle/bottom of the far-right side of the dashboard.

- Analytics —
  - Graphs were displayed middle left generally and featured line and bar graphs.
  - Possible analytical number indicators appeared above the graphs and spanned the width of the dashboard.

- Miscellaneous — These dashboards suffered from expected high variability in the dataset and no definitive patterns could be found.

- Fitness — Fitness dashboards suffered the same low data issues as agriculture dashboards and the generations were not good quality.

- Project Management —
  - Components resembling calendars were noted to be positioned in the top right.
  - Lists/tables with text and icons were included either at the bottom spanning the width of the page or in one dashboards case, the entire dashboard consisted of a table.

- Social — These were very similar to the analytics dashboards with a line/bar graph positioned middle left, and a number of indicators positioned above.

The image quality of generated components and full dashboards was low compared to results obtained from the StyleGAN2-ADA paper [64] and the full dashboard images began suffering from some augmentation leakage issues. Overall, dashboards and components were able to be generated however, all of these are easily distinguishable as fake by human eyes. The feasibility of generating dashboard component images is there but a number of further experiments and changes would likely have to be explored:

1. Further parameter tuning experiments should be conducted to find a combination where mode collapse does not occur. If this does not solve the issue then architecture alterations may be required.

2. Experiments should be conducted to find whether using a pre-trained model is a) possible, and b) will help the model more easily identify items such as camera feeds and photo profile pictures.

3. It was noted that text recognition and generation appears to be a problem over both the dashboard component and full dashboard image generation. Research into ways to mitigate this issue and properly recognise text during training would be highly beneficial.

4. When these issues are resolved and the model is able to train further, experiments should be conducted to train for a large amount of time and observe the results.

In conclusion, we can see that the learning and generation of dashboard images and dashboard components is feasible, however, more research is needed to improve the learning capabilities and produce vastly more realistic images. Therefore, we can answer the question posed at the beginning of this thesis:

Is it feasible to design and optimize dashboard graphic user interfaces with machine learning?

Yes it is feasible.

Bibliography

[1] R. J. Erb, "Introduction to backpropagation neural network computation," *Pharmaceutical research*, vol. 10, no. 2, pp. 165–170, 1993 (cited on page 15).

[2] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, "Face recognition: A convolutional neural-network approach," *IEEE transactions on neural networks*, vol. 8, no. 1, pp. 98–113, 1997 (cited on page 16).

[3] C. Sinclair, L. Pierce, and S. Matzner, "An application of machine learning to network intrusion detection," in *Proceedings 15th Annual Computer Security Applications Conference (ACSAC'99)*, IEEE, 1999, pp. 371–377 (cited on page 8).

[4] K. P. Bennett and C. Campbell, "Support vector machines: Hype or hallelujah?" *ACM SIGKDD explorations newsletter*, vol. 2, no. 2, pp. 1–13, 2000 (cited on page 12).

[5] C. Shin, K. Kim, M. Park, and H. J. Kim, "Support vector machine-based text detection in digital video," in *Neural Networks for Signal Processing X. Proceedings of the 2000 IEEE Signal Processing Society Workshop (Cat. No. 00TH8501)*, IEEE, vol. 2, 2000, pp. 634–641 (cited on page 12).

[6] S. Dreiseitl and L. Ohno-Machado, "Logistic regression and artificial neural network classification models: A methodology review," *Journal of biomedical informatics*, vol. 35, no. 5-6, pp. 352–359, 2002 (cited on page 10).

[7] K.-C. Fan, S.-J. Hsiao, and W.-T. Sung, "Developing a web-based pattern recognition system for the pattern search of components database by a parallel computing," in *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2003. Proceedings.*, IEEE, 2003, pp. 456–463 (cited on page 12).

[8] H. Jee, K. Lee, and S. Pan, "Eye and face detection using svm," in *Proceedings of the 2004 Intelligent Sensors, Sensor Networks and Information Processing Conference, 2004.*, IEEE, 2004, pp. 577–580 (cited on page 12).

[9] P. Warner, "Ordinal logistic regression," *Journal of Family Planning and Reproductive Health Care*, vol. 34, no. 3, p. 169, 2008 (cited on page 11).

[10] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255 (cited on pages 20, 51).

[11] T. O. Ayodele, "Types of machine learning algorithms," *New advances in machine learning*, vol. 3, pp. 19–48, 2010 (cited on page 8).

[12] L. Pappas and L. Whitman, "Riding the technology wave: Effective dashboard data visualization," in *Symposium on Human Interface*, Springer, 2011, pp. 249–258 (cited on page 7).

[13] J. Starkweather and A. K. Moske, *Multinomial logistic regression, 2011*, 2011 (cited on page 11).

[14] A. Dongare, R. Kharde, A. D. Kachare, *et al.*, "Introduction to artificial neural network," *International Journal of Engineering and Innovative Technology (IJEIT)*, vol. 2, no. 1, pp. 189–194, 2012 (cited on page 13).

[15] R. Ramteke and K. Y. Monali, "Automatic medical image classification and abnormality detection using k-nearest neighbour," *International Journal of Advanced Computer Research*, vol. 2, no. 4, p. 190, 2012 (cited on page 11).

[16] H. Wang, G. Li, Z. Ma, and X. Li, "Application of neural networks to image recognition of plant diseases," in *2012 International Conference on Systems and Informatics (ICSAI2012)*, IEEE, 2012, pp. 2159–2164 (cited on page 12).

[17] M. Auli, M. Galley, C. Quirk, and G. Zweig, "Joint language and translation modeling with recurrent neural networks," in *Proc. of EMNLP*, 2013 (cited on page 16).

[18] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*, Ieee, 2013, pp. 6645–6649 (cited on page 15).

[19] A. Janes, A. Sillitti, and G. Succi, "Effective dashboard design," *Cutter IT Journal*, vol. 26, no. 1, pp. 17–24, 2013 (cited on page 6).

[20] A. L. Maas, A. Y. Hannun, A. Y. Ng, *et al.*, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, Citeseer, vol. 30, 2013, p. 3 (cited on page 15).

[21] K. Yao, G. Zweig, M.-Y. Hwang, Y. Shi, and D. Yu, "Recurrent neural networks for language understanding.," in *Interspeech*, 2013, pp. 2524–2528 (cited on page 15).

[22] V. S. Dave and K. Dutta, "Neural network based models for software effort estimation: A review," *Artificial Intelligence Review*, vol. 42, no. 2, pp. 295–307, 2014 (cited on page 12).

[23] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014 (cited on pages 17, 19).

[24] A. K. Abdelmoula, "Bank credit risk analysis with k-nearest-neighbor classifier: Case of tunisian banks," *Accounting and Management Information Systems*, vol. 14, no. 1, p. 79, 2015 (cited on page 11).

[25] K. Gregor, I. Danihelka, A. Graves, D. Rezende, and D. Wierstra, "Draw: A recurrent neural network for image generation," in *International Conference on Machine Learning*, PMLR, 2015, pp. 1462–1471 (cited on page 16).

[26] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015 (cited on page 8).

[27] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015 (cited on page 46).

[28] K. J. Piczak, "Environmental sound classification with convolutional neural networks," in *2015 IEEE 25th international workshop on machine learning for signal processing (MLSP)*, IEEE, 2015, pp. 1–6 (cited on page 16).

[29] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015 (cited on pages 17, 50).

[30] I. Goodfellow, "Nips 2016 tutorial: Generative adversarial networks," *arXiv preprint arXiv:1701.00160*, 2016 (cited on page 19).

[31] Y. Guo, Y. Liu, A. Oerlemans, S. Lao, S. Wu, and M. S. Lew, "Deep learning for visual understanding: A review," *Neurocomputing*, vol. 187, pp. 27–48, 2016 (cited on page 17).

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778 (cited on page 17).

[33] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016 (cited on page 10).

[34] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," *Advances in neural information processing systems*, vol. 29, 2016 (cited on page 20).

[35] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826 (cited on page 20).

[36] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 international conference on engineering and technology (ICET)*, Ieee, 2017, pp. 1–6 (cited on page 17).

[37] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein generative adversarial networks," in *International conference on machine learning*, PMLR, 2017, pp. 214–223 (cited on page 19).

[38] R. Culkin and S. R. Das, "Machine learning in finance: The case of deep learning for option pricing," *Journal of Investment Management*, vol. 15, no. 4, pp. 92–100, 2017 (cited on page 8).

[39] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "Gans trained by a two time-scale update rule converge to a local nash equilibrium," *Advances in neural information processing systems*, vol. 30, 2017 (cited on page 20).

[40] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive growing of gans for improved quality, stability, and variation," *arXiv preprint arXiv:1710.10196*, 2017 (cited on page 101).

[41] C. Mc., *Machine learning fundamentals (i): Cost functions and gradient descent*, Nov. 2017 (cited on page 10).

[42] M. Praveena and V. Jaiganesh, "A literature review on supervised machine learning algorithms and boosting process," *International Journal of Computer Applications*, vol. 169, no. 8, pp. 32–35, 2017 (cited on pages 8, 9).

[43] S. Sharma, S. Sharma, and A. Athaiya, "Activation functions in neural networks," *towards data science*, vol. 6, no. 12, pp. 310–316, 2017 (cited on pages 13, 14).

[44] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, "State-of-the-art in artificial neural network applications: A survey," *Heliyon*, vol. 4, no. 11, e00938, 2018 (cited on page 12).

[45] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375*, 2018 (cited on page 14).

[46] H. Almogdady, S. Manaseer, and H. Hiary, "A flower recognition system based on image processing and neural networks," *International Journal Of Scientific & Technology Research*, vol. 7, no. 11, pp. 166–173, 2018 (cited on page 12).

[47] S. A. Barnett, "Convergence problems with generative adversarial networks (gans)," *arXiv preprint arXiv:1806.11382*, 2018 (cited on page 19).

[48] A. Bhardwaj, W. Di, and J. Wei, *Deep Learning Essentials: Your hands-on guide to the fundamentals of deep learning and neural network modeling*. Packt Publishing Ltd, 2018 (cited on page 16).

[49] M. Bińkowski, D. J. Sutherland, M. Arbel, and A. Gretton, "Demystifying mmd gans," *arXiv preprint arXiv:1801.01401*, 2018 (cited on page 21).

[50] N. Bjorck, C. P. Gomes, B. Selman, and K. Q. Weinberger, "Understanding batch normalization," *Advances in neural information processing systems*, vol. 31, 2018 (cited on page 15).

[51] A. Brock, J. Donahue, and K. Simonyan, "Large scale gan training for high fidelity natural image synthesis," *arXiv preprint arXiv:1809.11096*, 2018 (cited on pages 51, 52).

[52] R. Chauhan, K. K. Ghanshala, and R. Joshi, "Convolutional neural network (cnn) for image detection and recognition," in *2018 First International Conference on Secure Cyber Computing and Communication (ICSCCC)*, IEEE, 2018, pp. 278–282 (cited on page 16).

[53] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative adversarial networks: An overview," *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 53–65, 2018 (cited on page 18).

[54] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," *CoRR*, vol. abs/1812.04948, 2018. arXiv: 1812.04948 (cited on page 59).

[55] H. Kim, J. Kim, and H. Jung, "Convolutional neural network based image processing system," *Journal of information and communication convergence engineering*, vol. 16, no. 3, pp. 160–165, 2018 (cited on page 16).

[56] X. Wang, R. Girshick, A. Gupta, and K. He, "Non-local neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7794–7803 (cited on page 52).

[57] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 4401–4410 (cited on pages 17, 60, 101).

[58] K. Park, J. Kim, and J. Lee, "Visual field prediction using recurrent neural network," *Scientific reports*, vol. 9, no. 1, pp. 1–12, 2019 (cited on page 16).

[59] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035 (cited on page 46).

[60] K. Qu, F. Guo, X. Liu, Y. Lin, and Q. Zou, "Application of machine learning in microbiology," *Frontiers in microbiology*, vol. 10, p. 827, 2019 (cited on page 8).

[61] A. Robinson, "Sketch2code: Generating a website from a paper mockup," *arXiv preprint arXiv:1905.13750*, 2019 (cited on page 12).

[62] H. Zhang, I. Goodfellow, D. Metaxas, and A. Odena, "Self-attention generative adversarial networks," in *International conference on machine learning*, PMLR, 2019, pp. 7354–7363 (cited on page 17).

[63] S. Zheng, Z. Hu, and Y. Ma, "Faceoff: Assisting the manifestation design of web graphical user interface," in *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, 2019, pp. 774–777 (cited on page 21).

[64] T. Karras, M. Aittala, J. Hellsten, S. Laine, J. Lehtinen, and T. Aila, "Training generative adversarial networks with limited data," in *Proc. NeurIPS*, 2020 (cited on pages 21, 46, 47, 59, 62, 76, 90, 96, 99, 101, 102).

[65] T. Karras, S. Laine, M. Aittala, J. Hellsten, J. Lehtinen, and T. Aila, "Analyzing and improving the image quality of stylegan," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 8110–8119 (cited on pages 59, 60, 101).

[66] Y. Lei, B. Yang, X. Jiang, F. Jia, N. Li, and A. K. Nandi, "Applications of machine learning to machine fault diagnosis: A review and roadmap," *Mechanical Systems and Signal Processing*, vol. 138, p. 106 587, 2020 (cited on page 8).

[67] B. Liu, Y. Zhu, K. Song, and A. Elgammal, "Towards faster and stabilized gan training for high-fidelity few-shot image synthesis," in *International Conference on Learning Representations*, 2020 (cited on pages 55–58).

[68] D. Maulud and A. M. Abdulazeez, "A review on linear regression comprehensive in machine learning," *Journal of Applied Science and Technology Trends*, vol. 1, no. 4, pp. 140–147, 2020 (cited on page 9).

[69] J. Naranjo-Torres, M. Mora, R. Hernández-Garcı**********a, R. J. Barrientos, C. Fredes, and A. Valenzuela, "A review of convolutional neural network applied to fruit image processing," *Applied Sciences*, vol. 10, no. 10, p. 3443, 2020 (cited on page 16).

[70] E. Nichani, A. Radhakrishnan, and C. Uhler, "Do deeper convolutional networks perform better?," 2020 (cited on page 17).

[71] W. Yip, *Lifecycle of Machine Learning Models*. Oracle Corporation, 2020 (cited on page 9).

[72] T. Zia, S. Arif, S. Murtaza, and M. A. Ullah, "Text-to-image generation with attention based recurrent neural networks," *arXiv preprint arXiv:2001.06658*, 2020 (cited on page 16).

[73] M. A. Chandra and S. Bedi, "Survey on svm and their application in image classification," *International Journal of Information Technology*, vol. 13, no. 5, pp. 1–11, 2021 (cited on page 12).

[74] H. Nobanee, "A bibliometric review of big data in finance," *Big Data*, vol. 9, no. 2, pp. 73–78, 2021 (cited on page 1).

[75] A. Rosebrock, *How to create a deep learning dataset using google images*, Dec. 2021 (cited on page 29).

[76] ——, *How to create a deep learning dataset using google images*, Dec. 2021 (cited on page 32).

[77] B. Siddhartha, A. P. Chavan, and K. Subramanya, "Iot enabled real-time availability and condition monitoring of cnc machines," in *2020 IEEE International Conference on Internet of Things and Intelligence System (IoTaIS)*, IEEE, 2021, pp. 78–84 (cited on page 7).

[78] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, "Dive into deep learning," *arXiv preprint arXiv:2106.11342*, 2021 (cited on page 9).

[79] T. Zhao, C. Chen, Y. Liu, and X. Zhu, "Guigan: Learning to generate gui designs using generative adversarial networks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 748–760 (cited on page 22).

[80] J. Wise, *How much data is created every day in 2022? [new stats]*, Mar. 2022 (cited on page 1).

[81] *Discover the world's top designers 'i&'; creatives* (cited on pages 29, 67).

[82] *Geometric image transformations* (cited on page 39).

[83] *Innovation meets motivation* (cited on page 6).

[84] *Money dashboard: Master your money: Budgeting app uk* (cited on page 6).

APPENDIX a

Dashboard Component Generations

I IoT

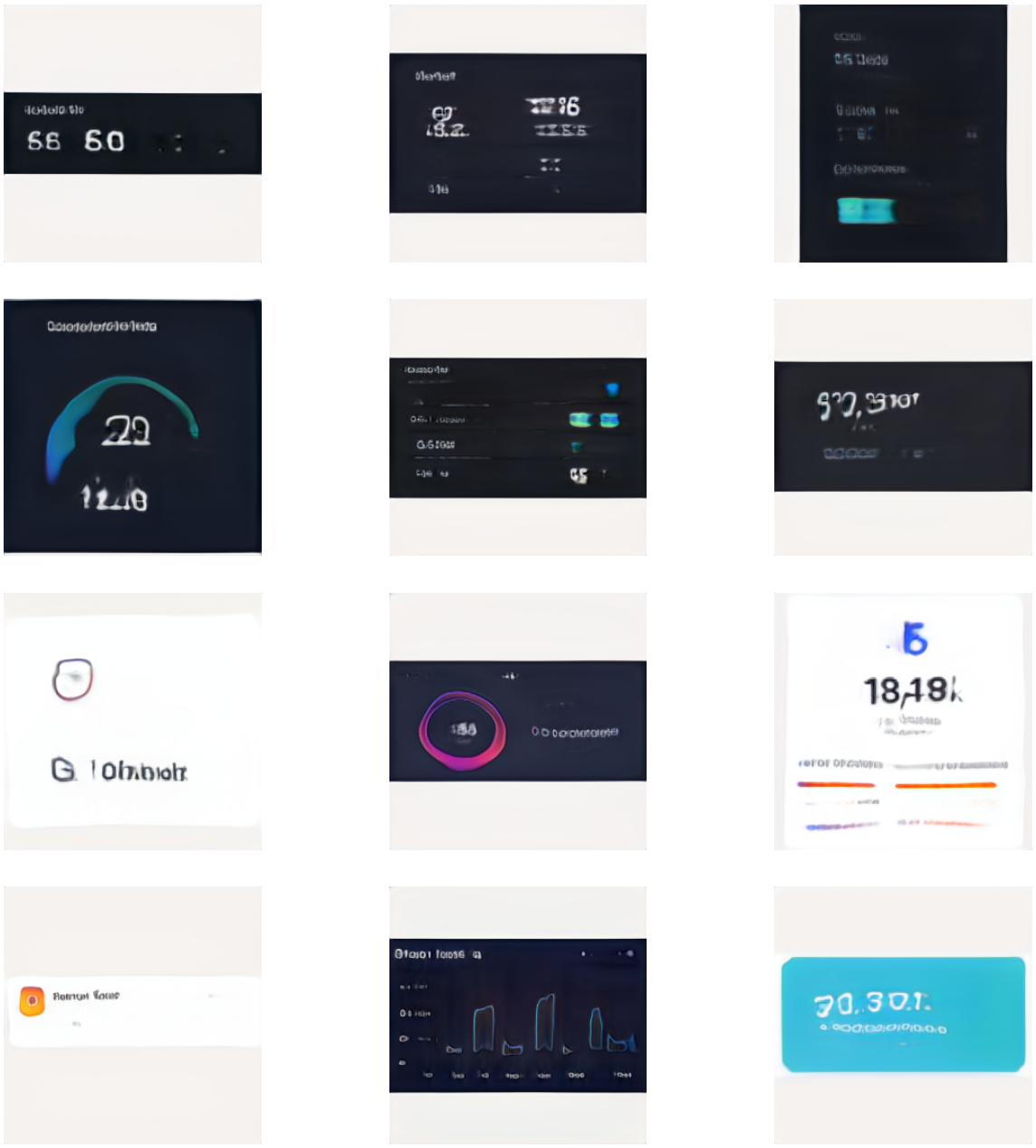


Figure 35: Twelve image generations for IoT dashboard components

II Agriculture

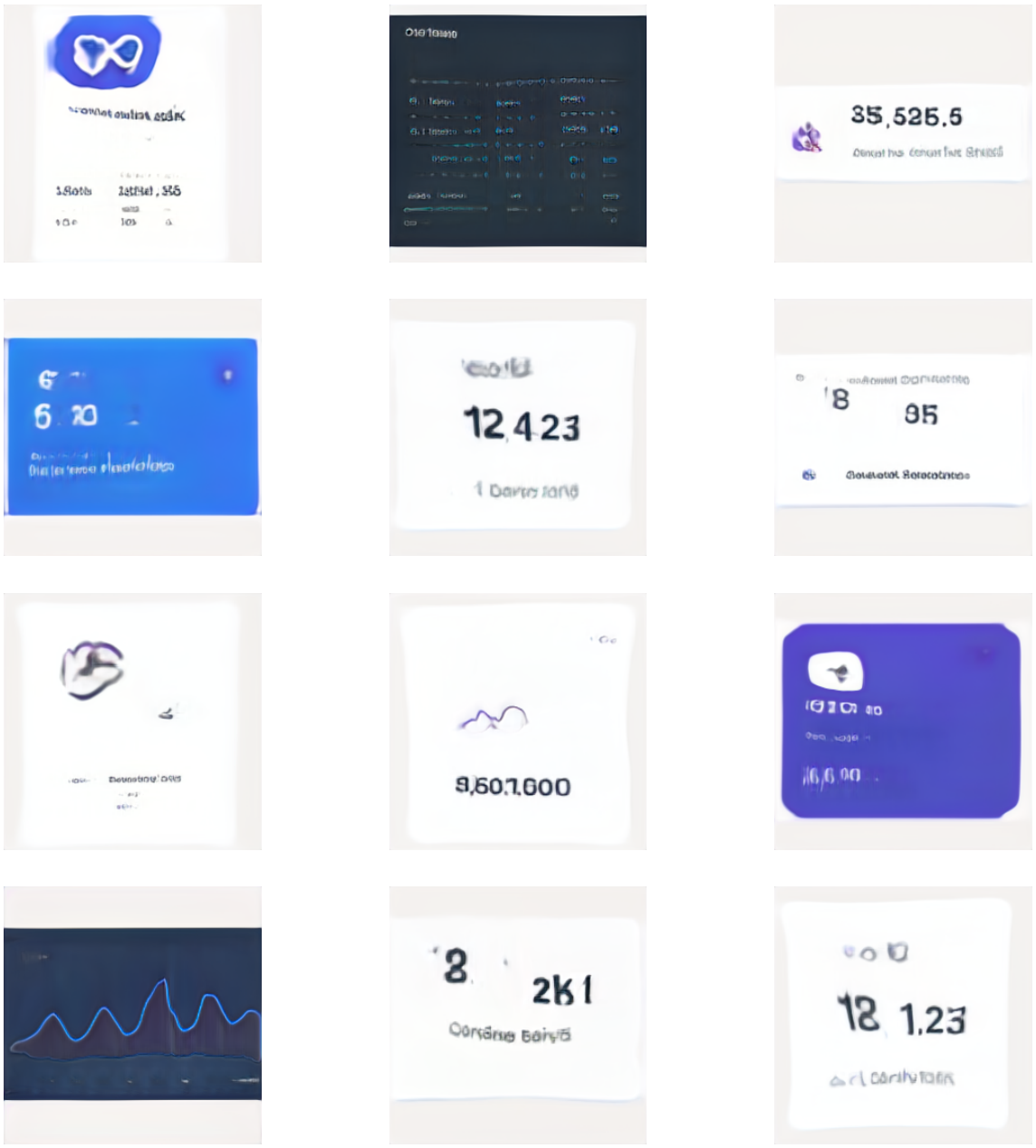


Figure 36: Twelve image generations for finance dashboard components

III Finance

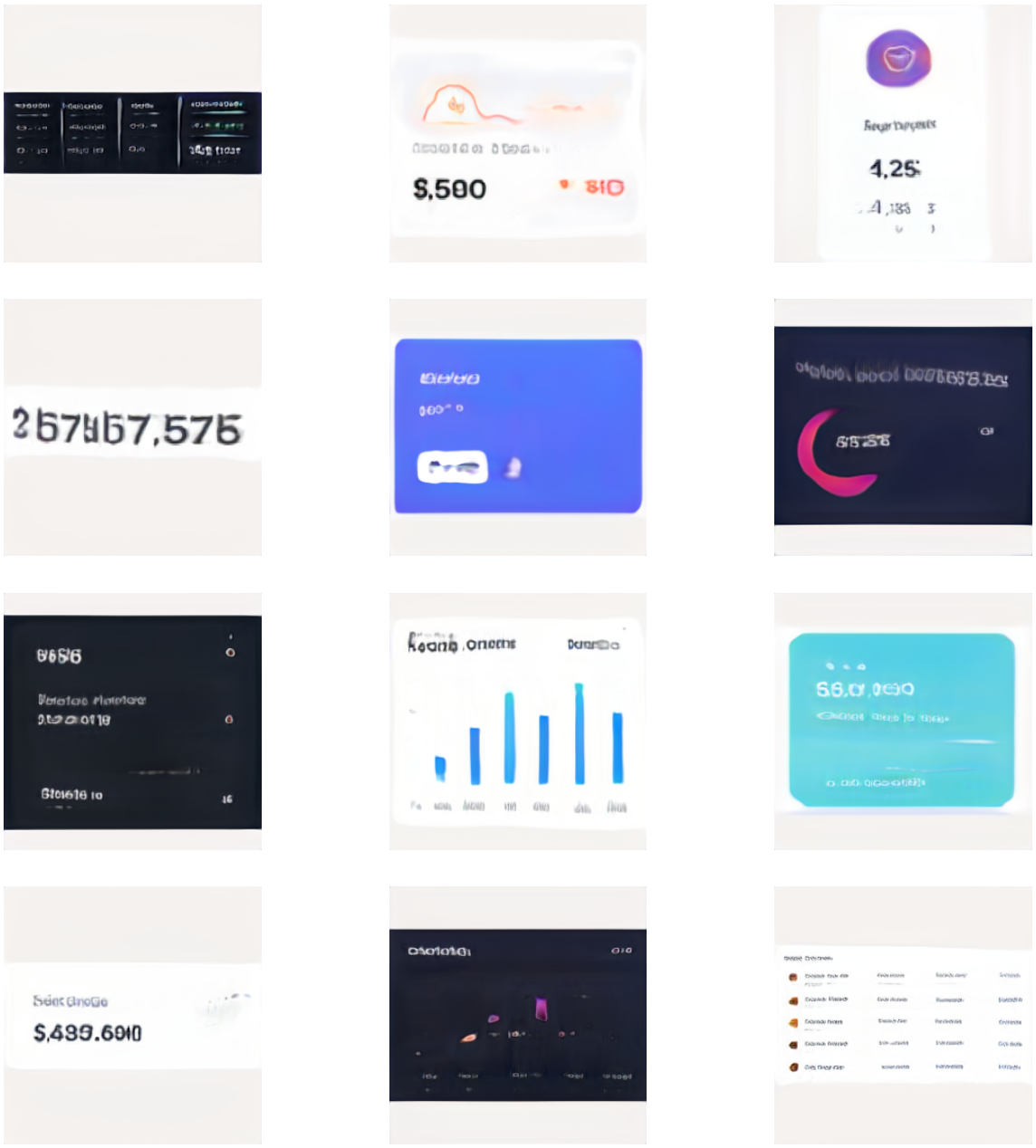


Figure 37: Twelve image generations for finance dashboard components

IV Analytics

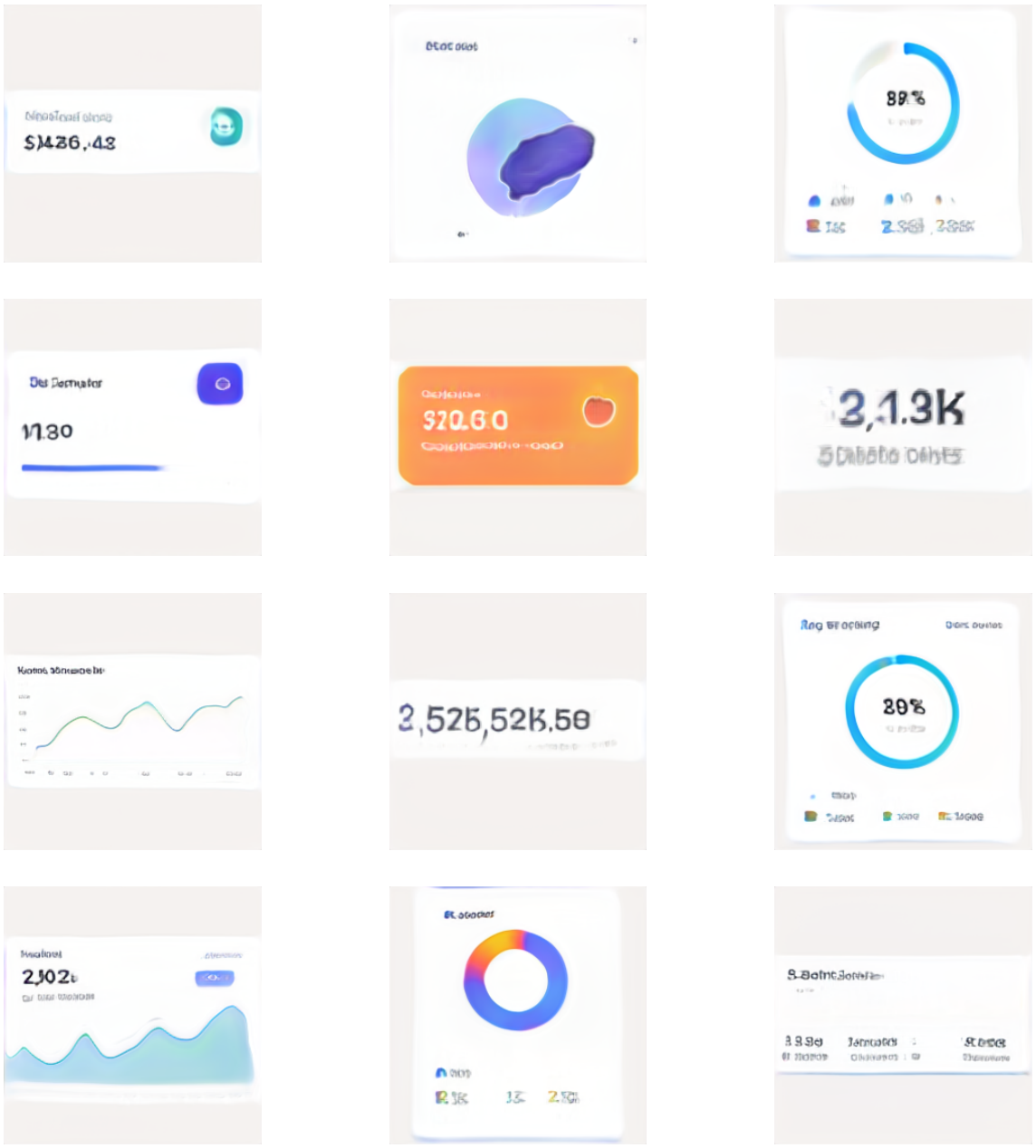


Figure 38: Twelve image generations for analytics dashboard components

V Misc

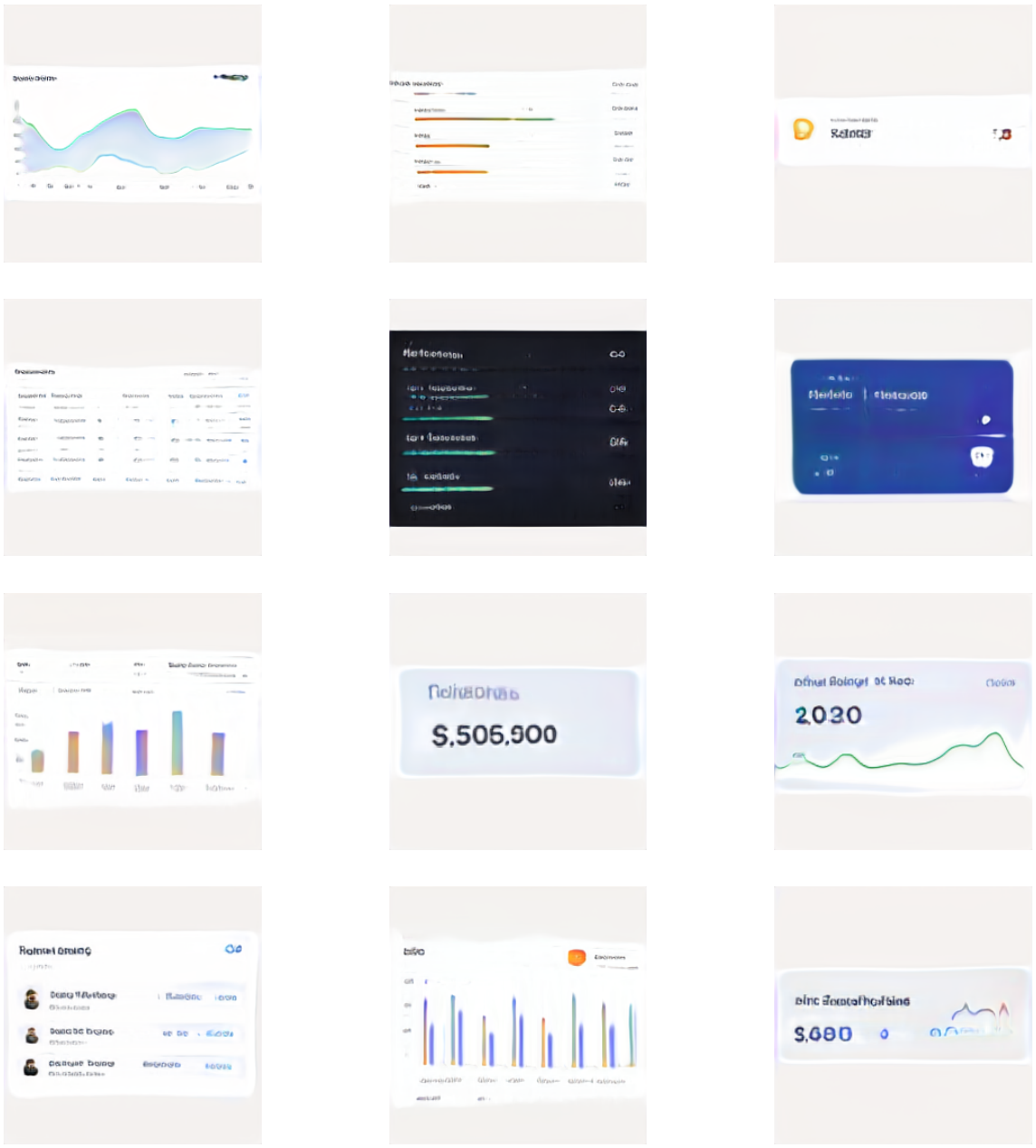


Figure 39: Twelve image generations for miscellaneous dashboard components

VI Fitness

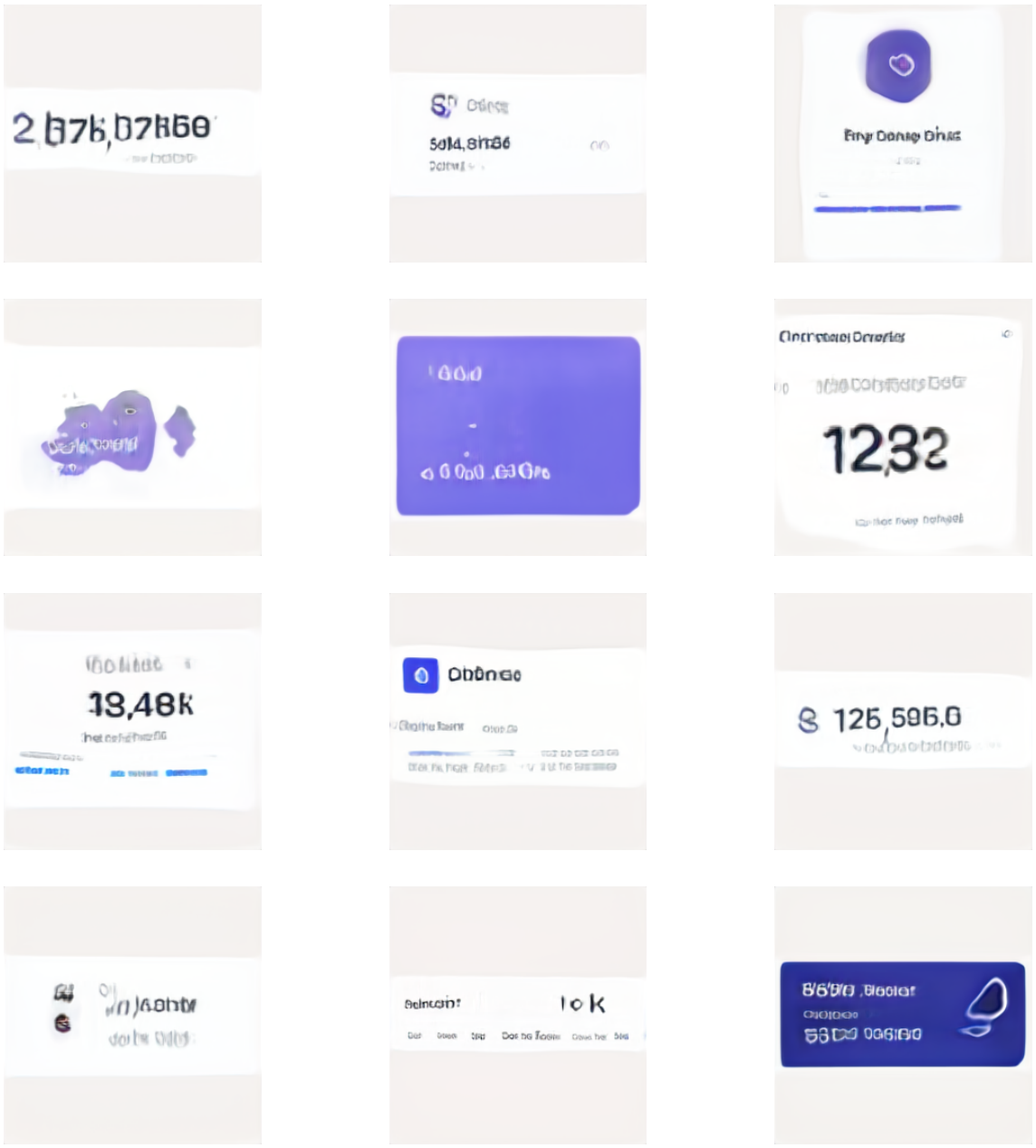


Figure 40: Twelve image generations for fitness dashboard components

VII Project Management

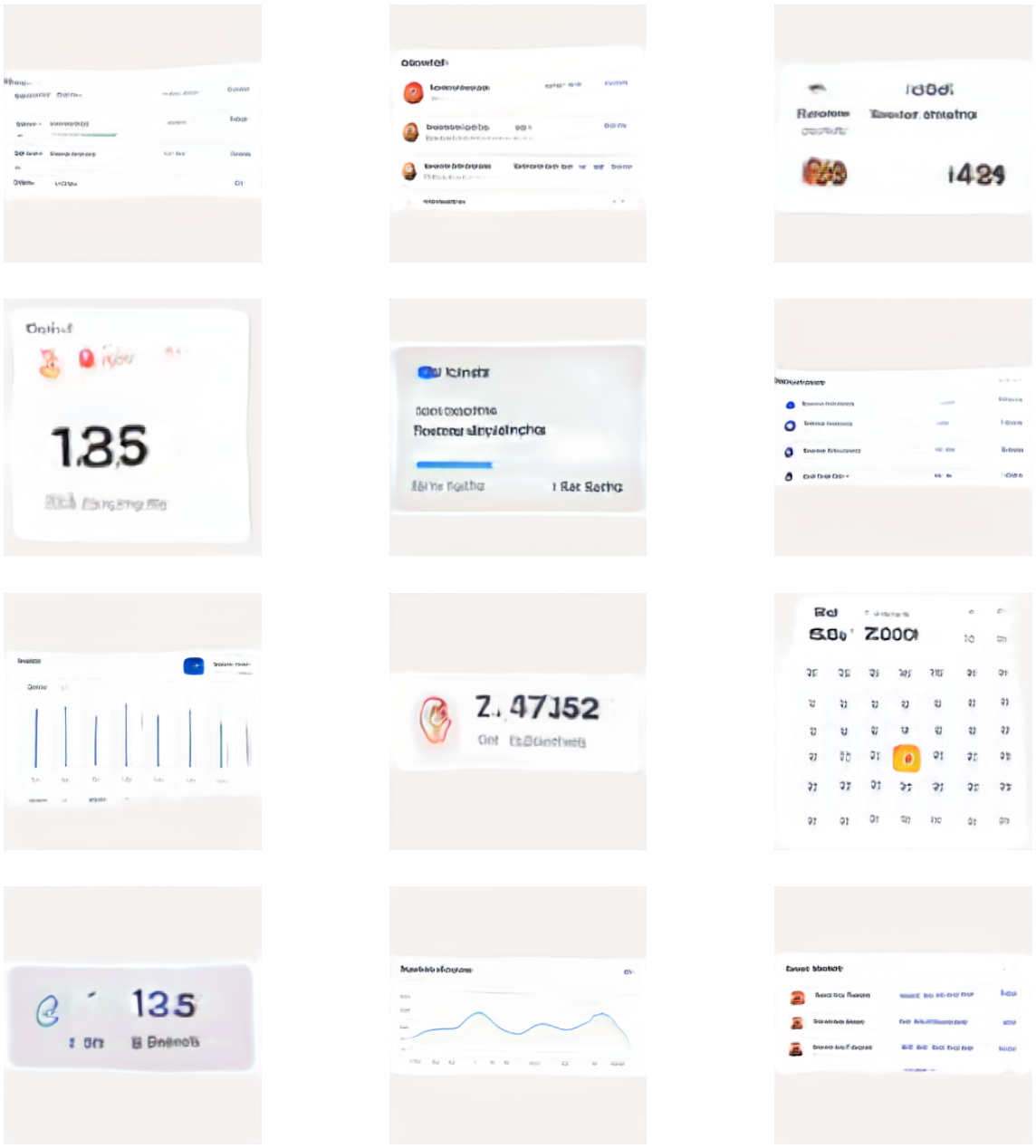


Figure 41: Twelve image generations for project management dashboard components

VIII Social

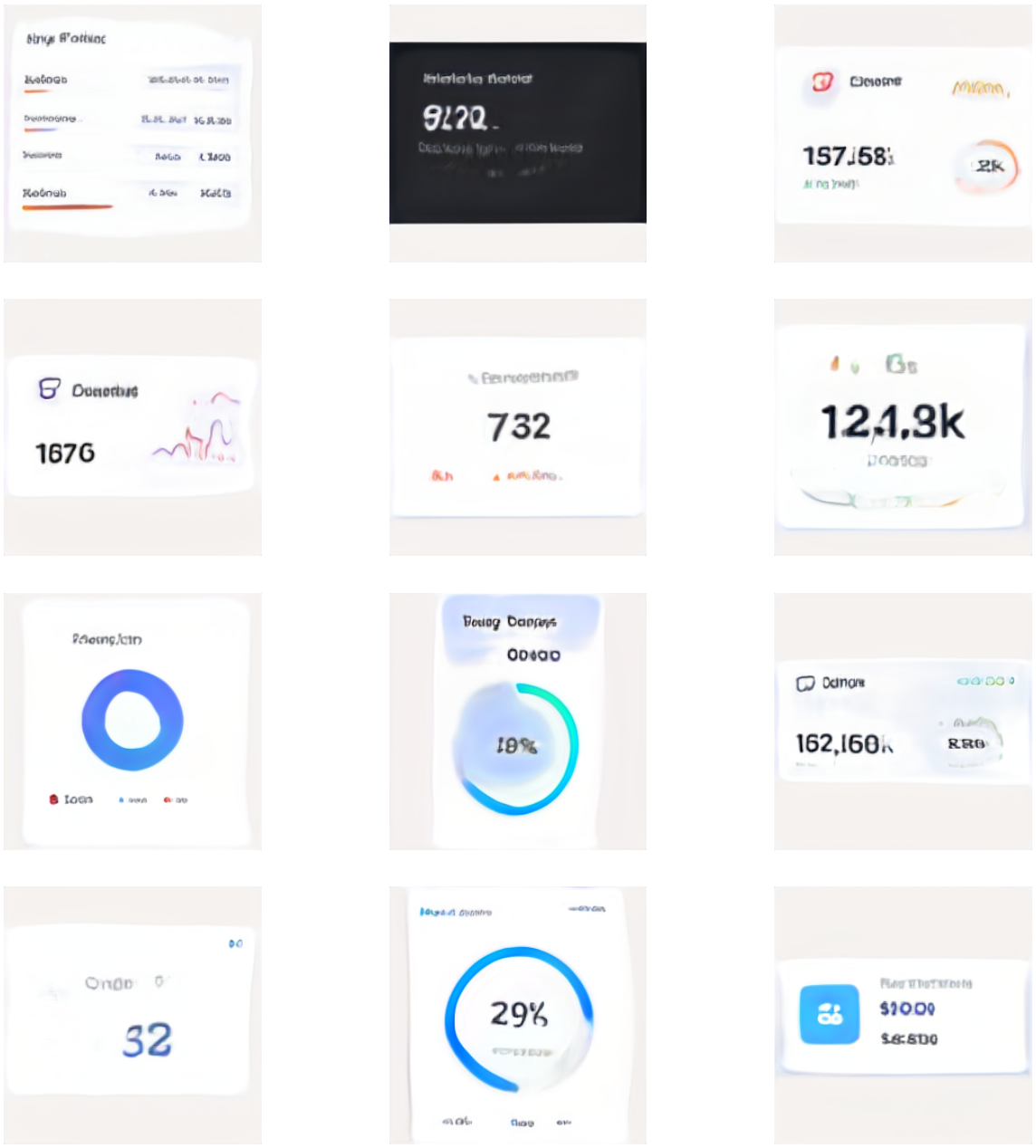
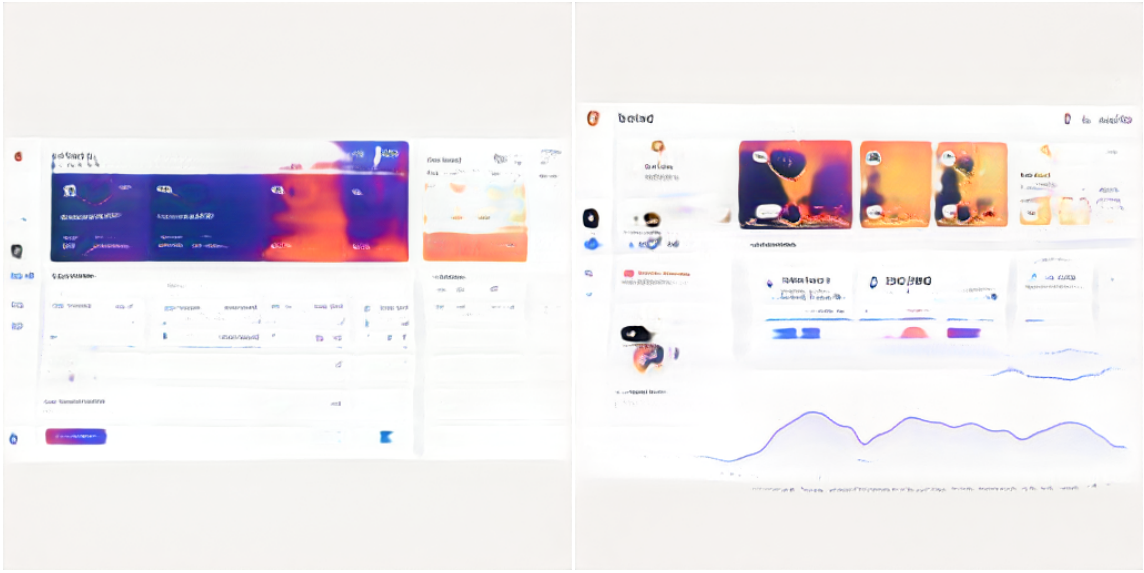


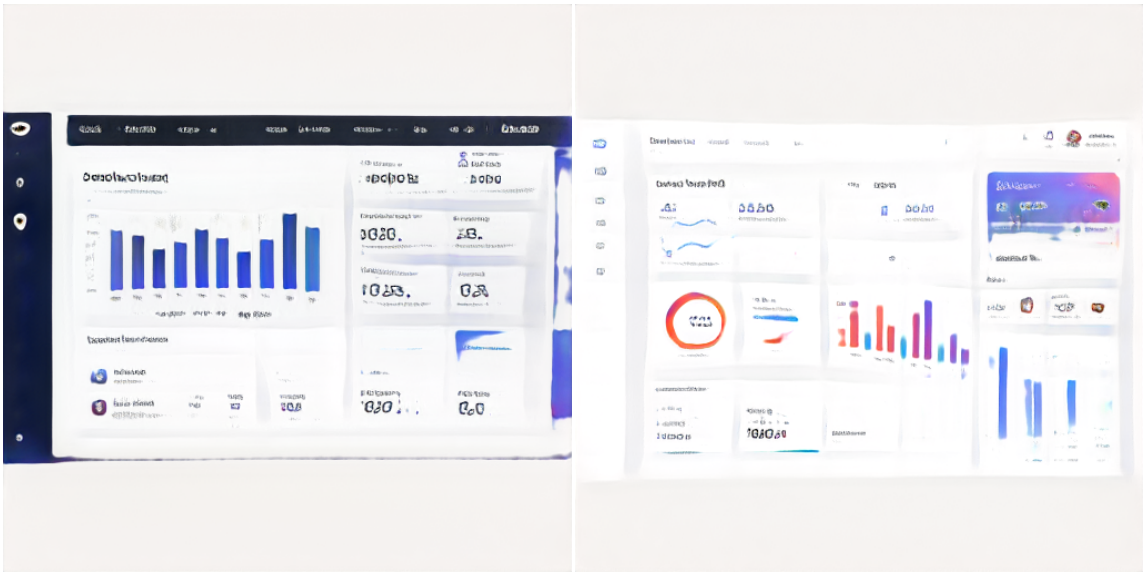
Figure 42: Twelve image generations for social media dashboard components

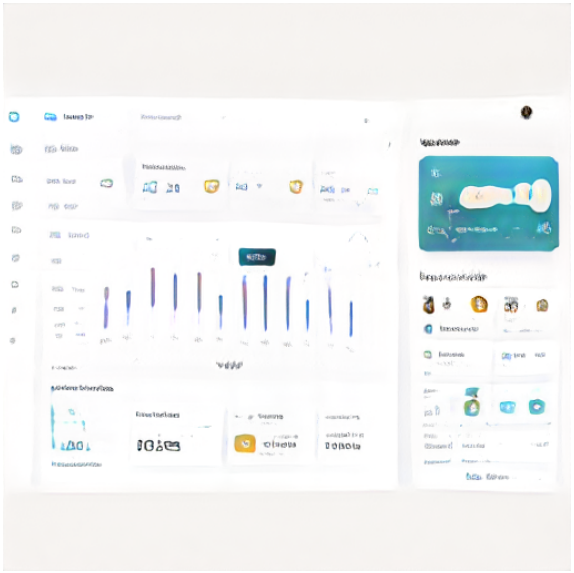
APPENDIX b

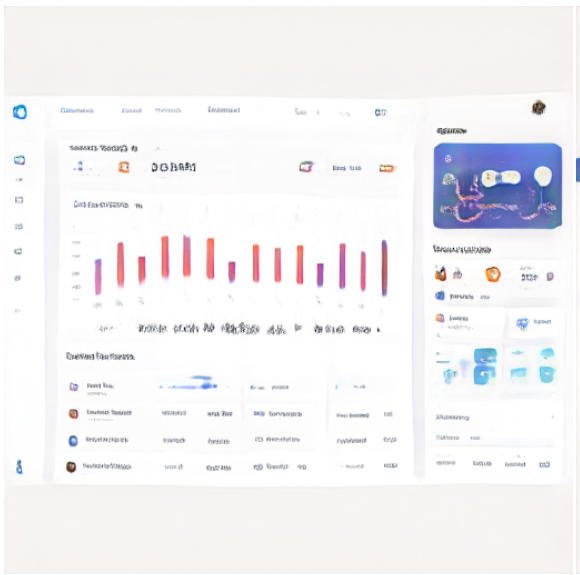
Dashboard Generation Results

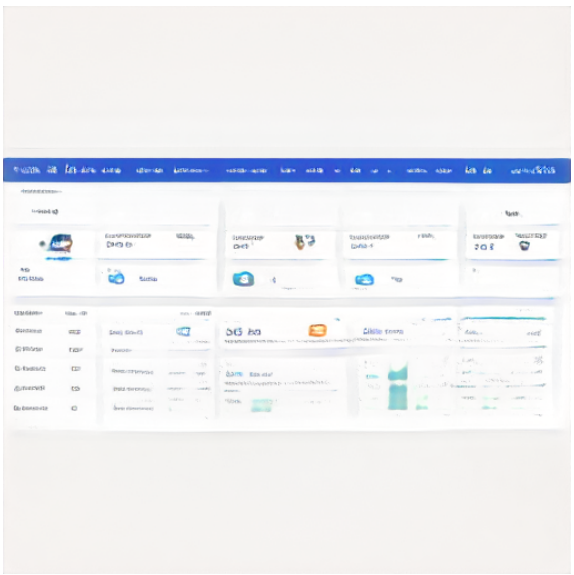
I IoT

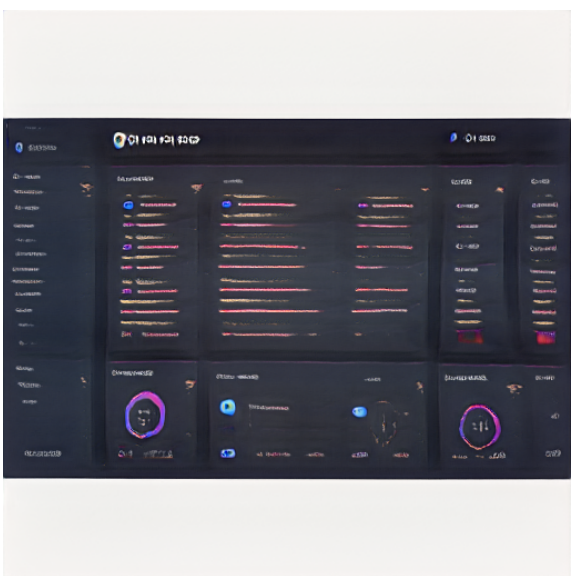




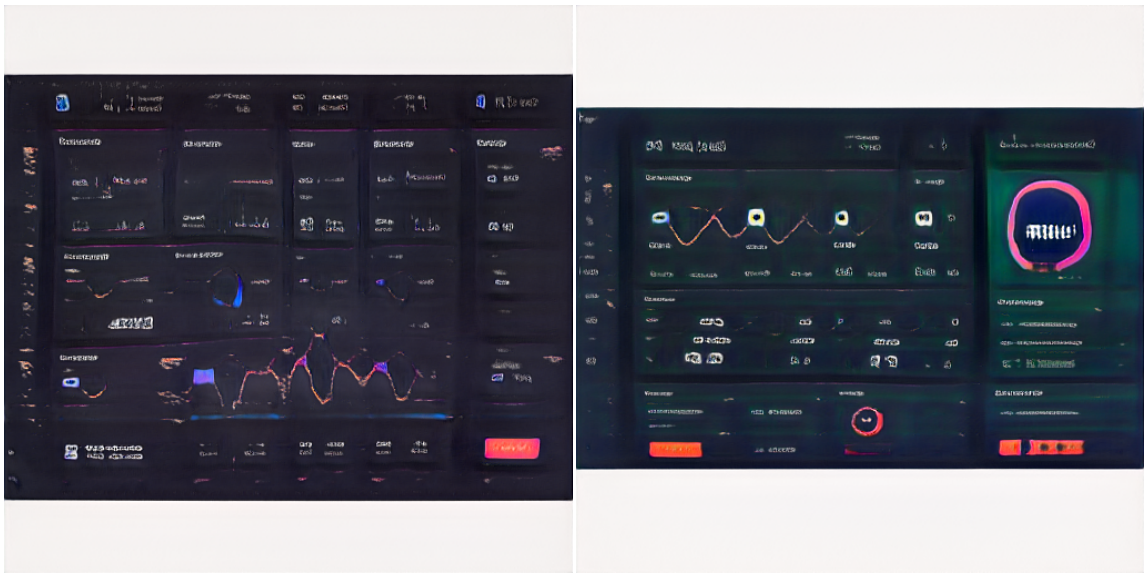


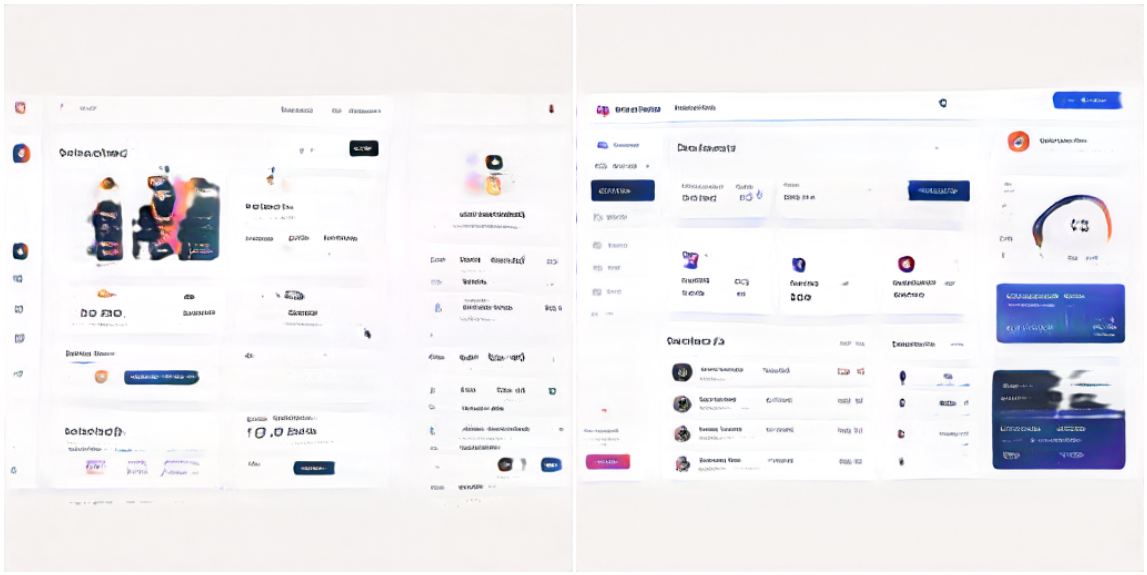


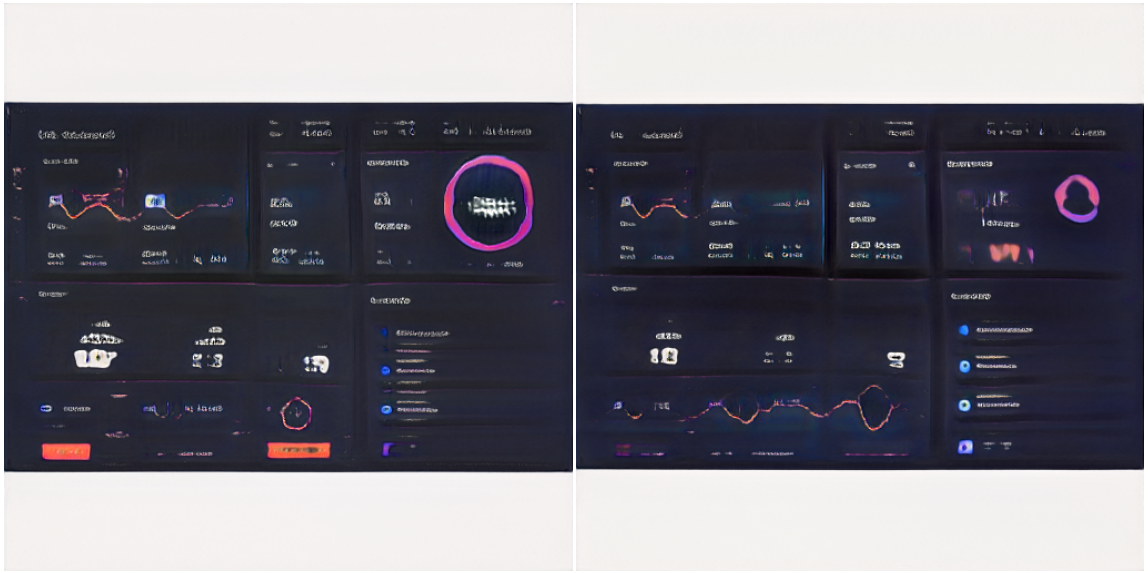




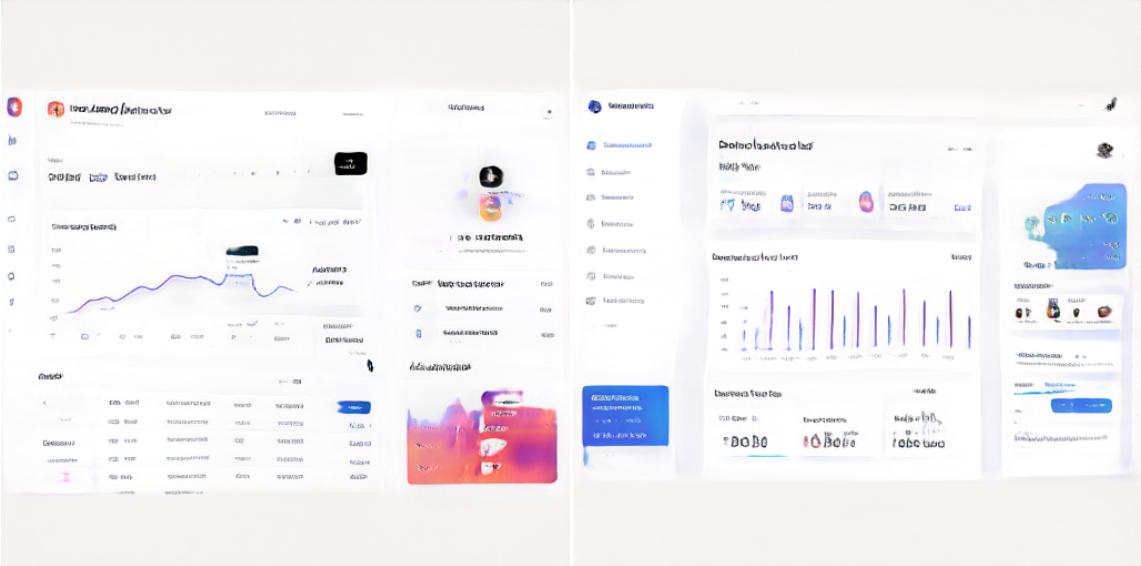
II Agriculture

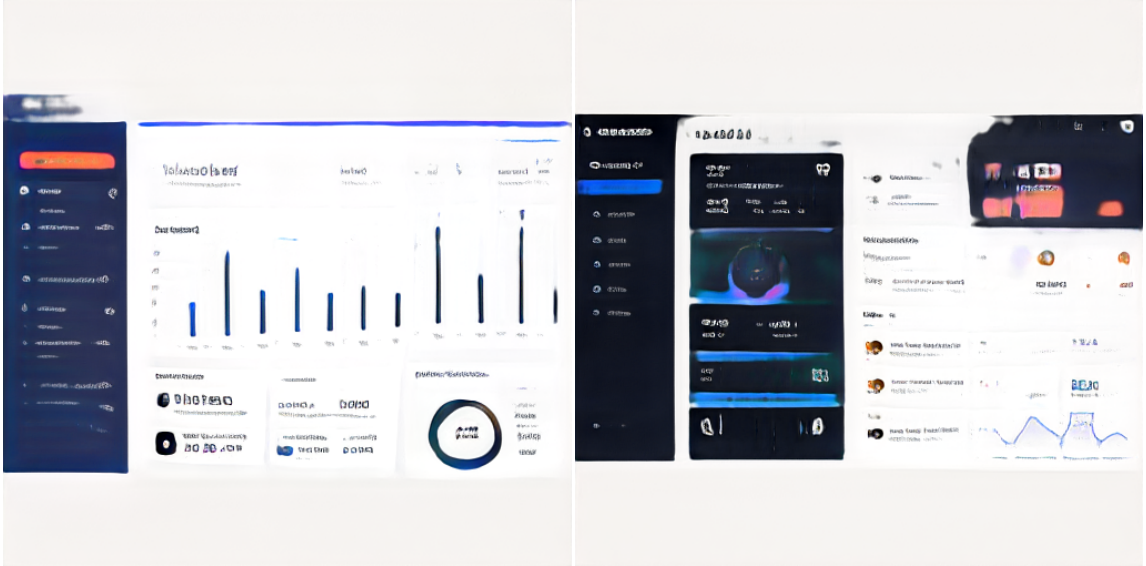


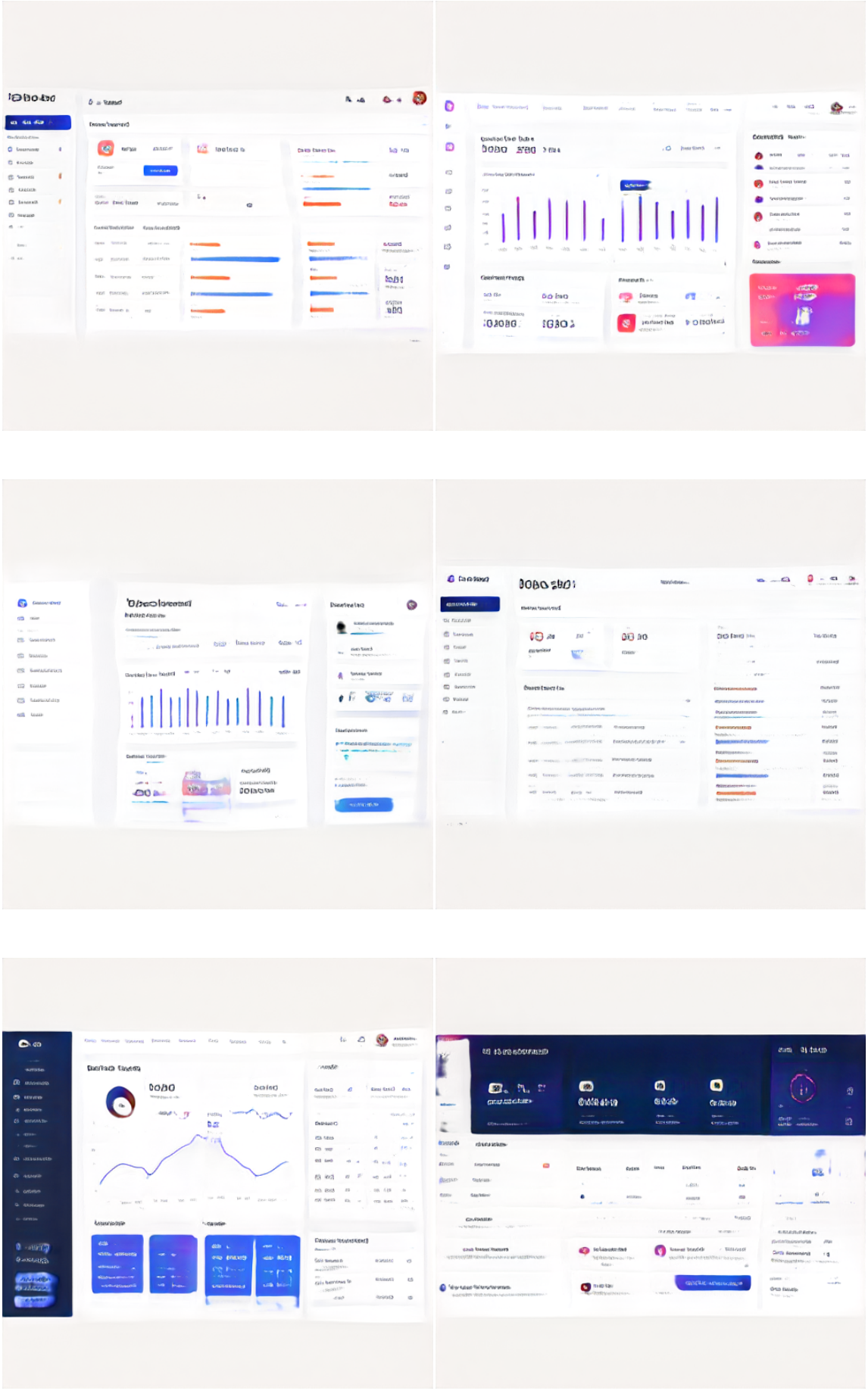




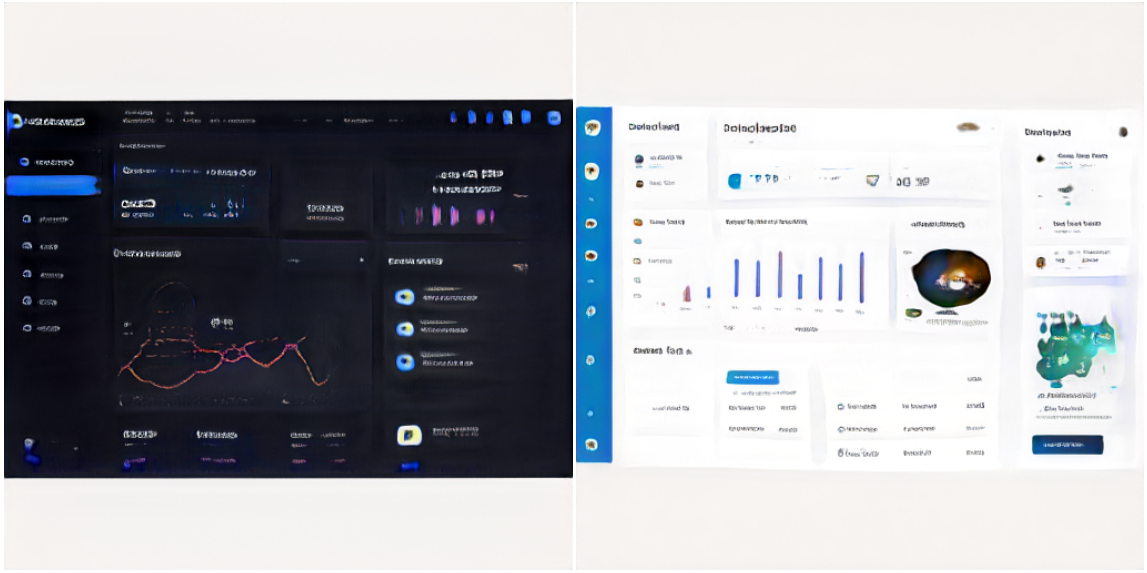
III Finance

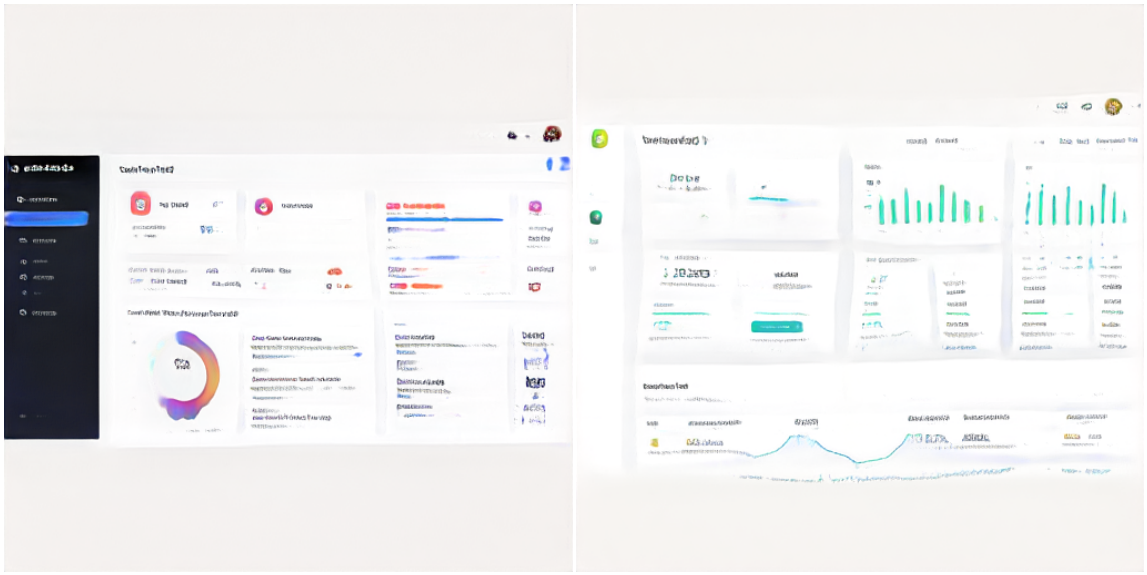


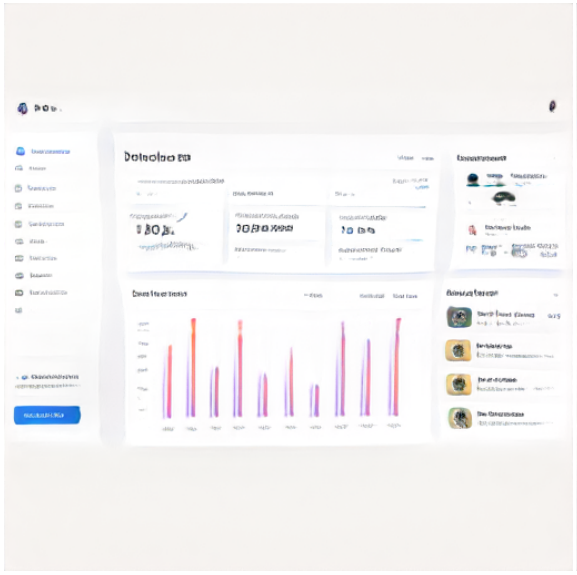


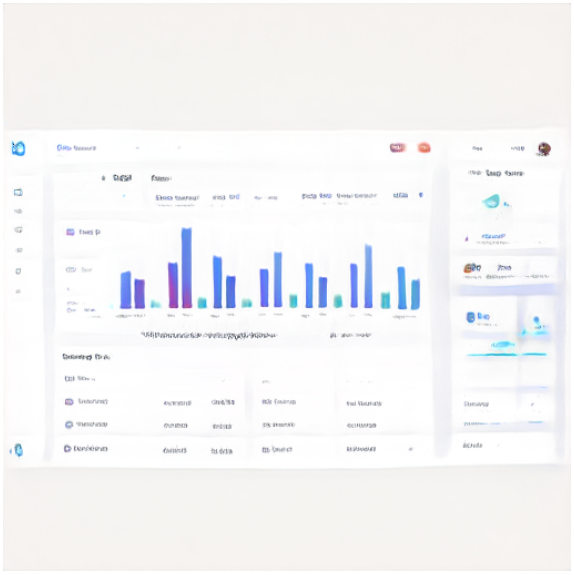


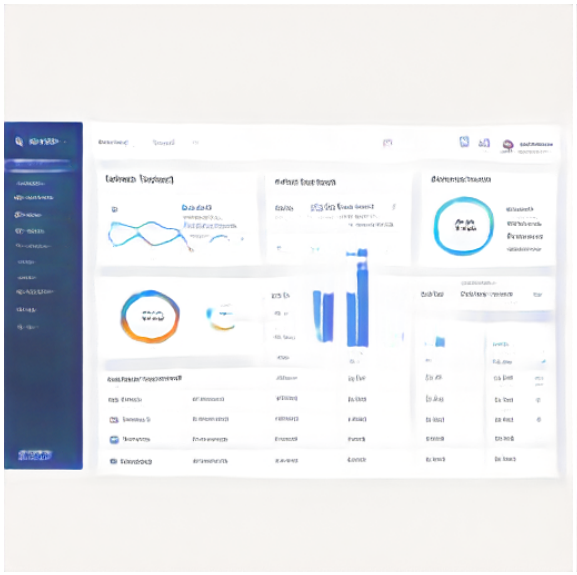
IV Analytics

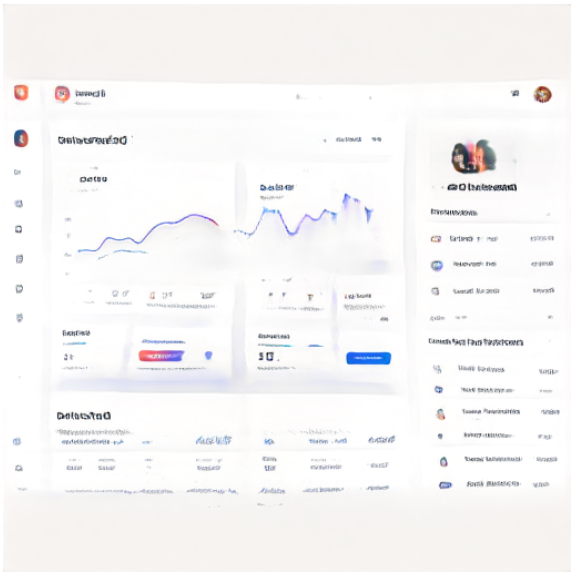


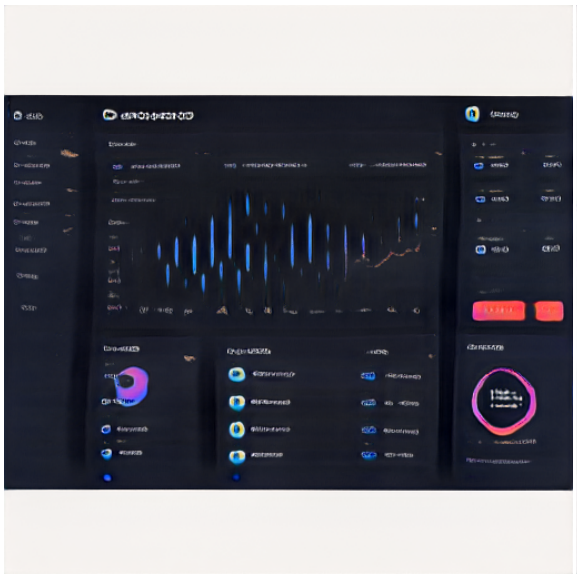


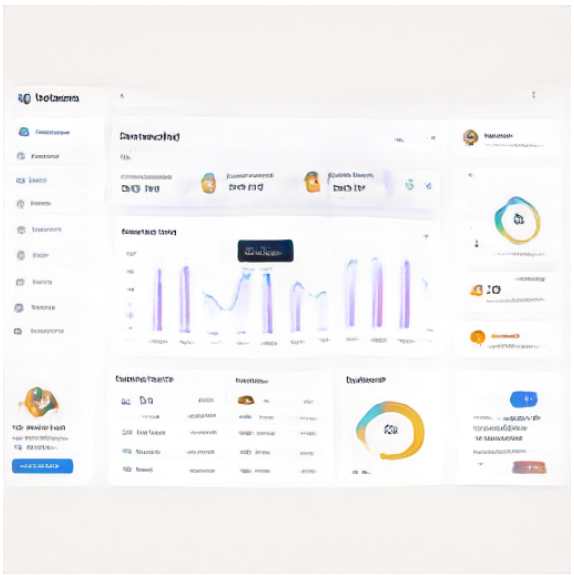




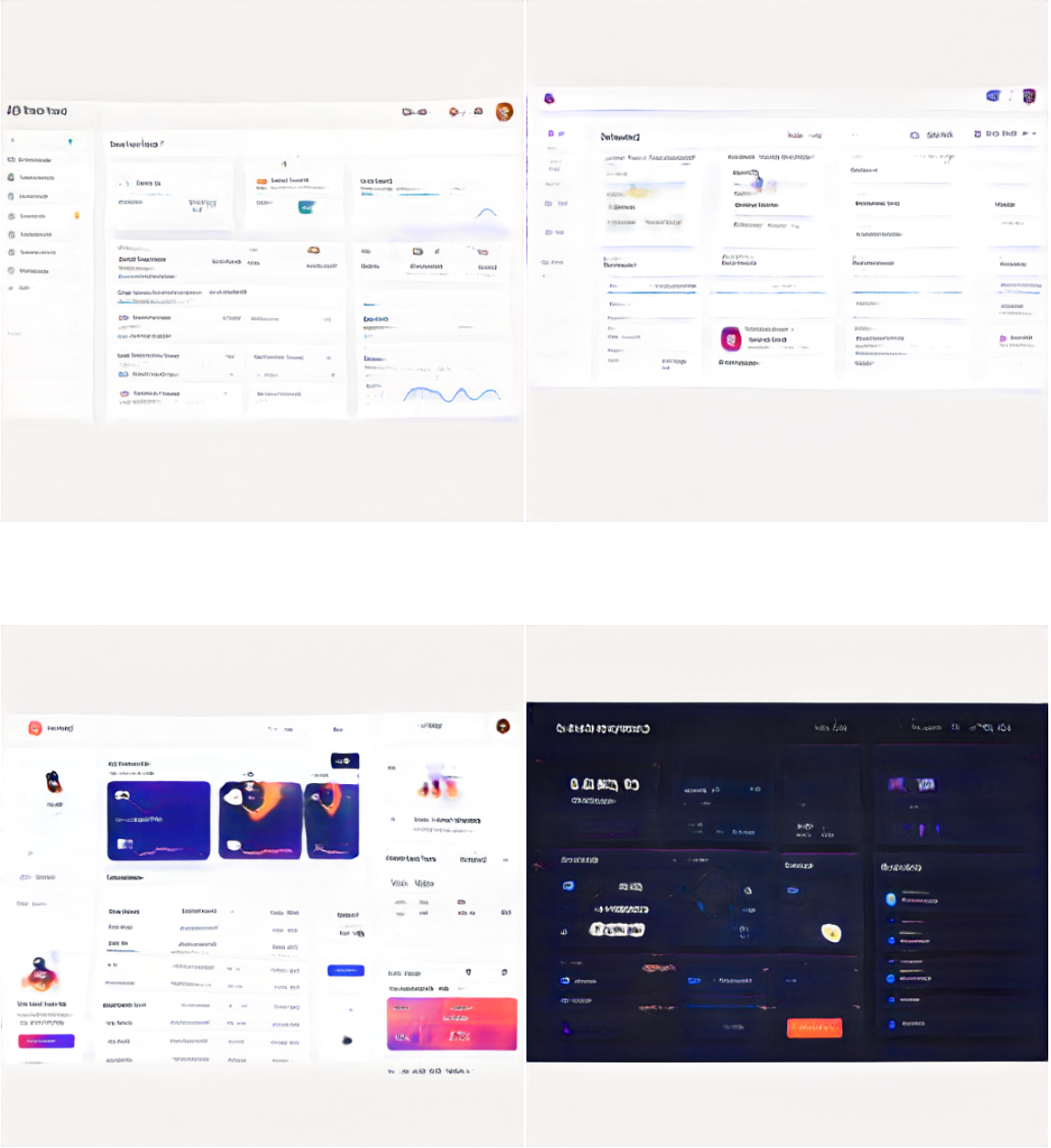


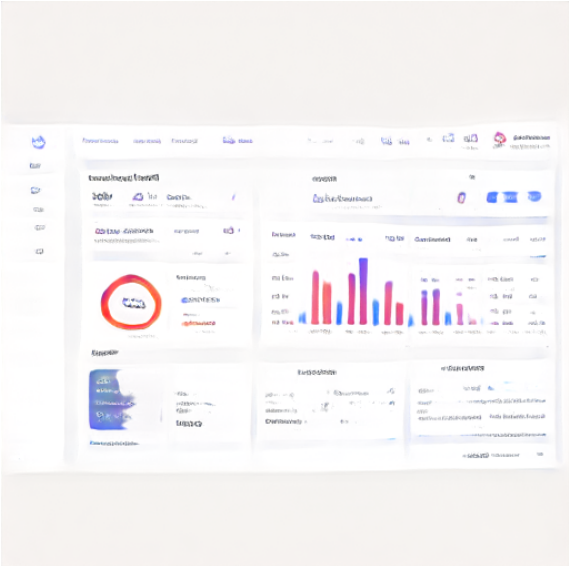


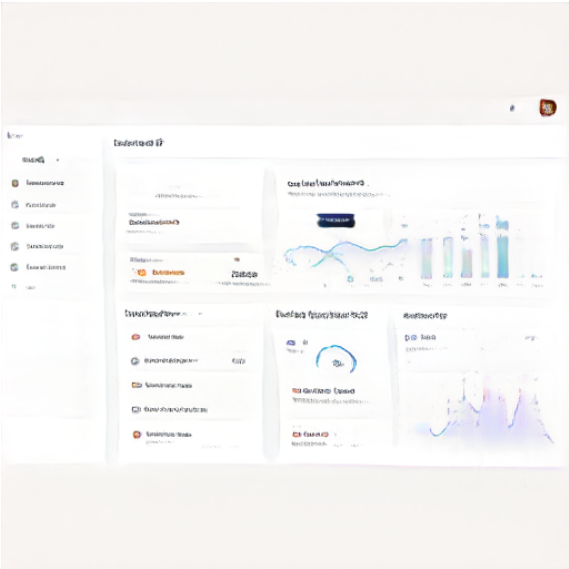


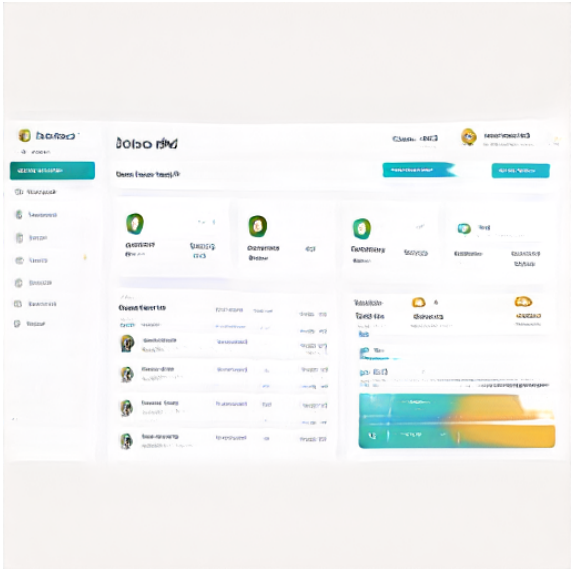


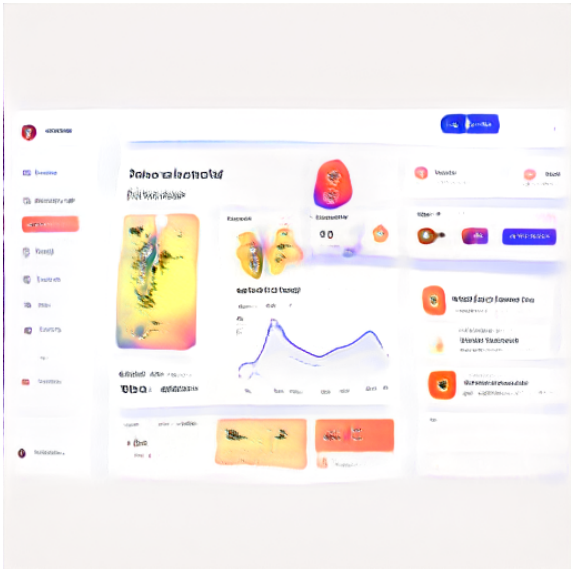
V Miscellaneous

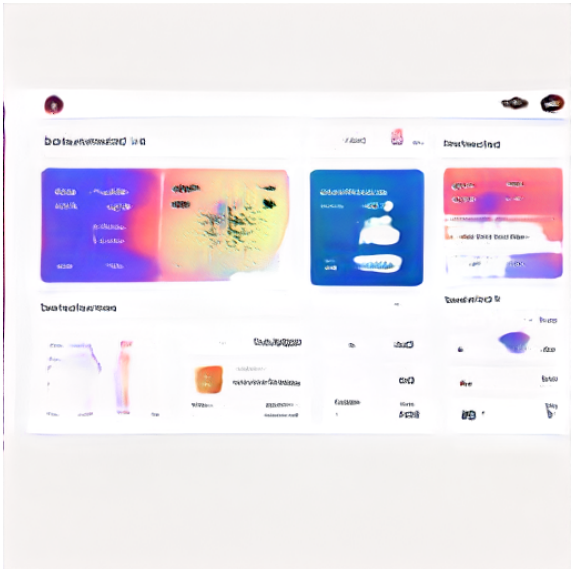




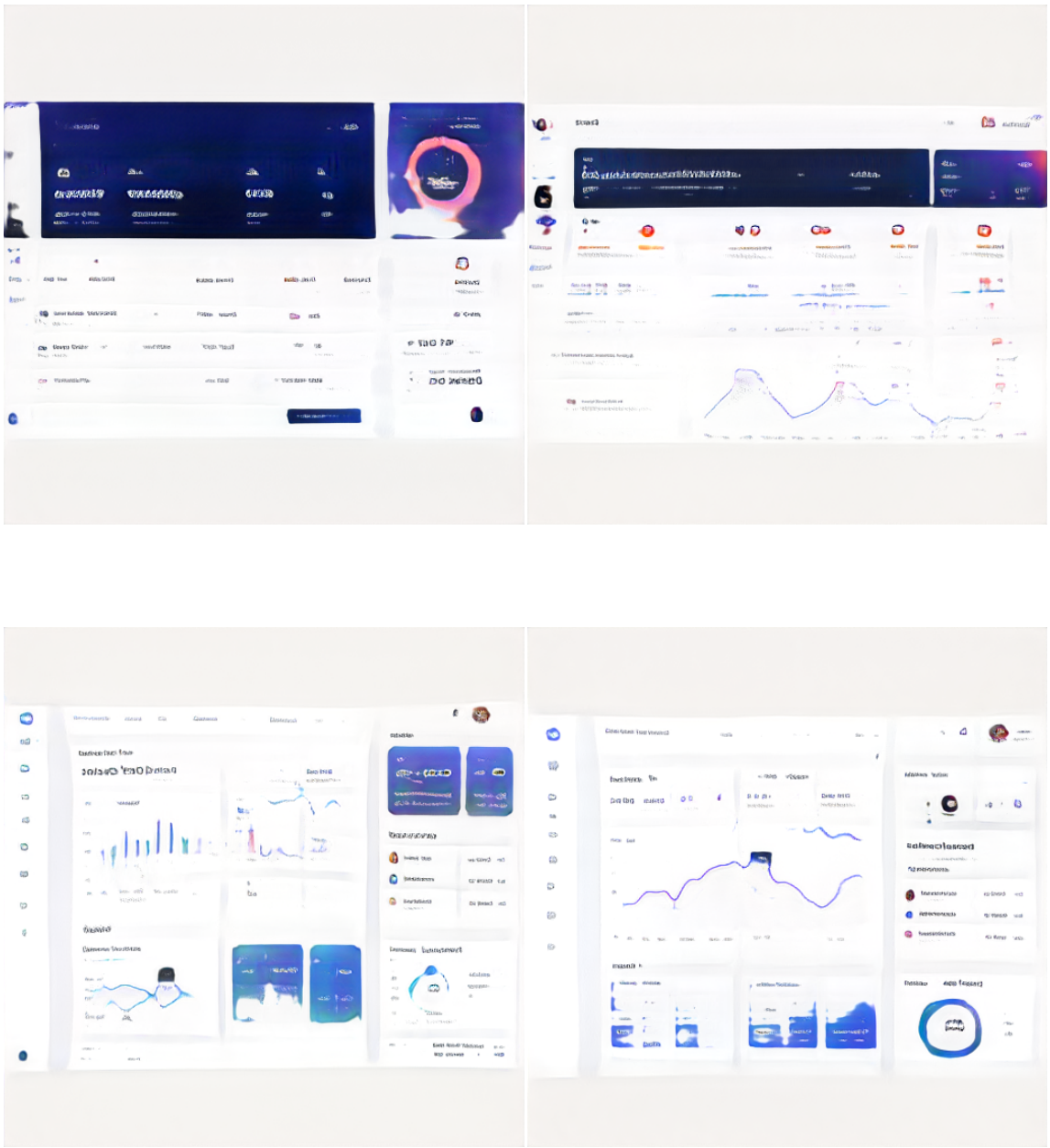


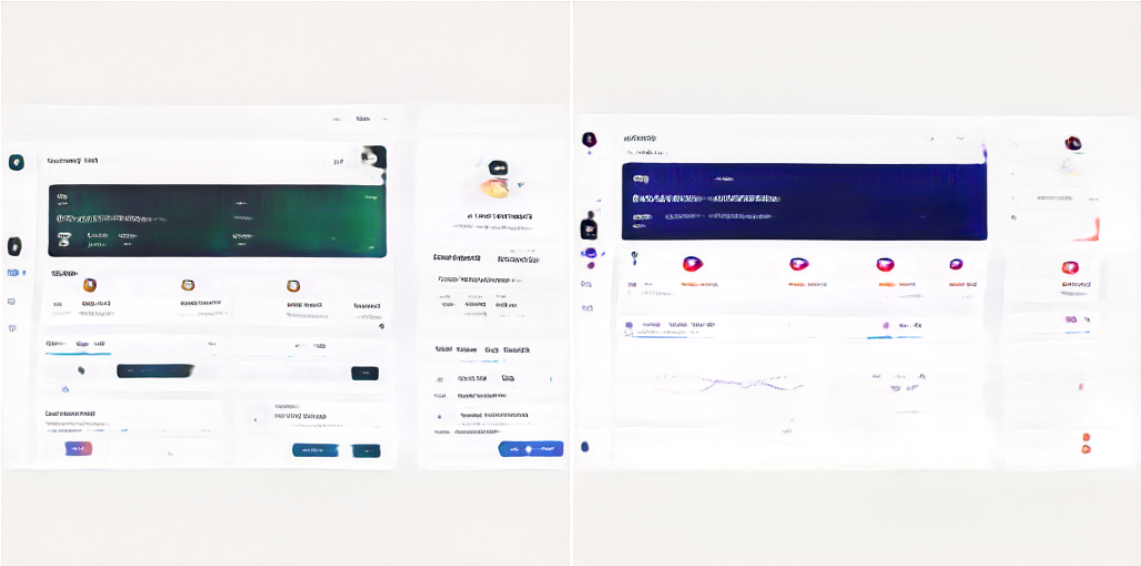


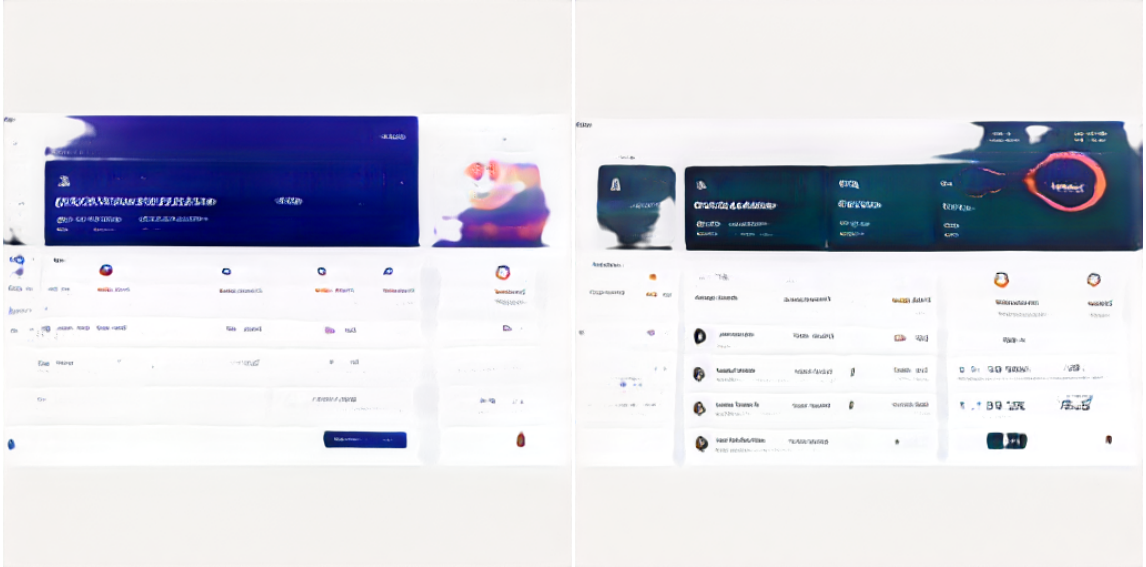


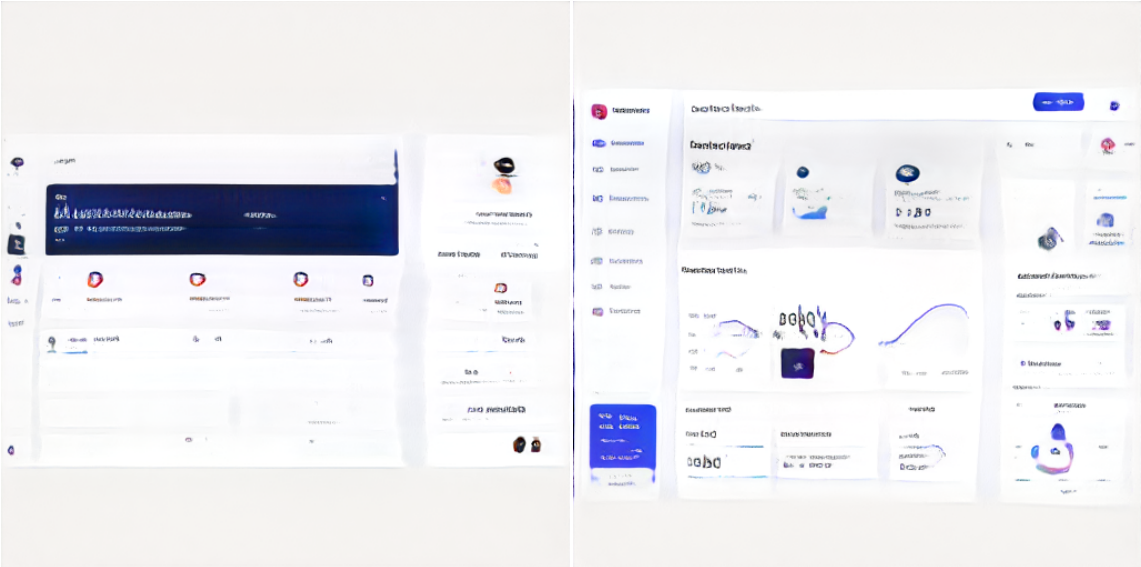


VI Fitness

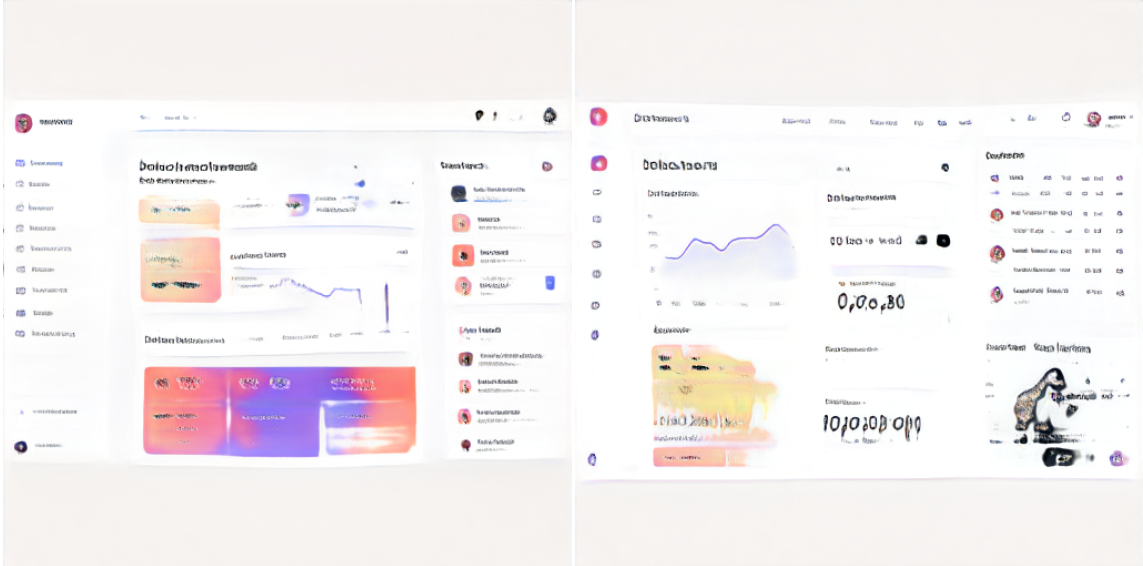


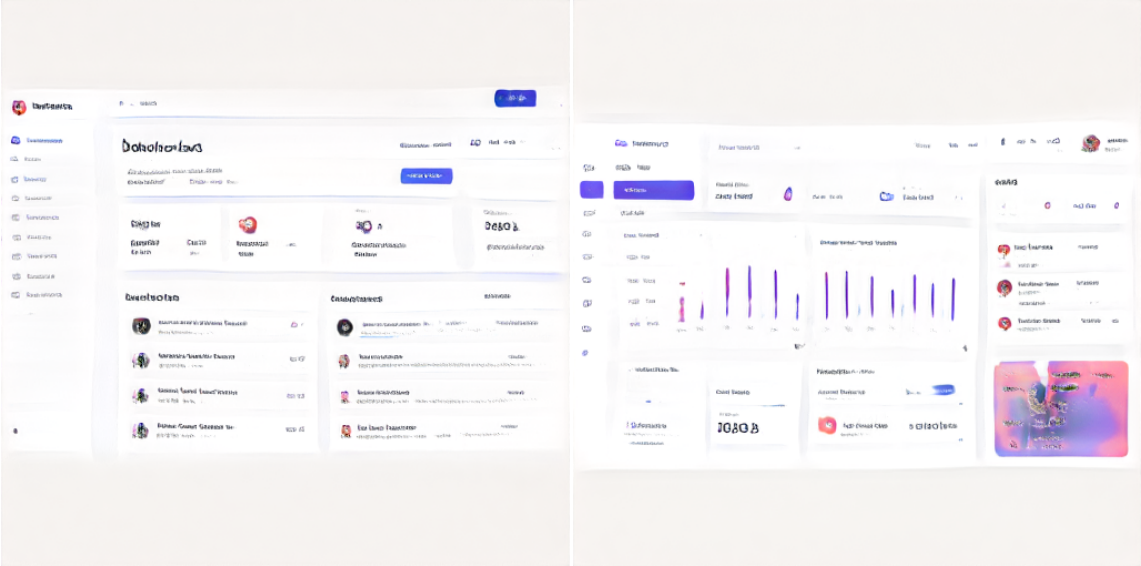


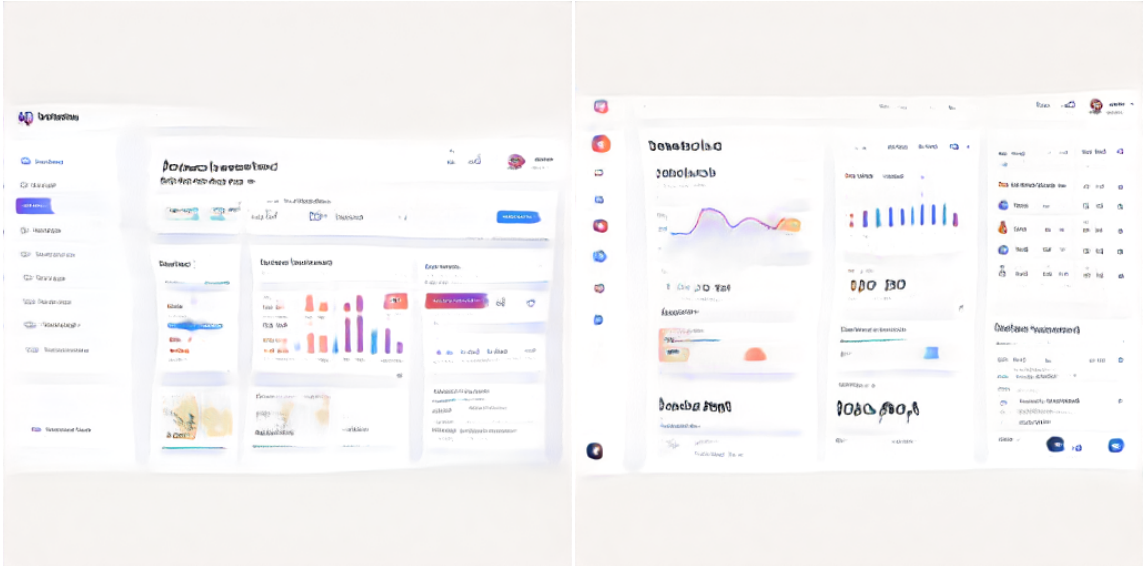




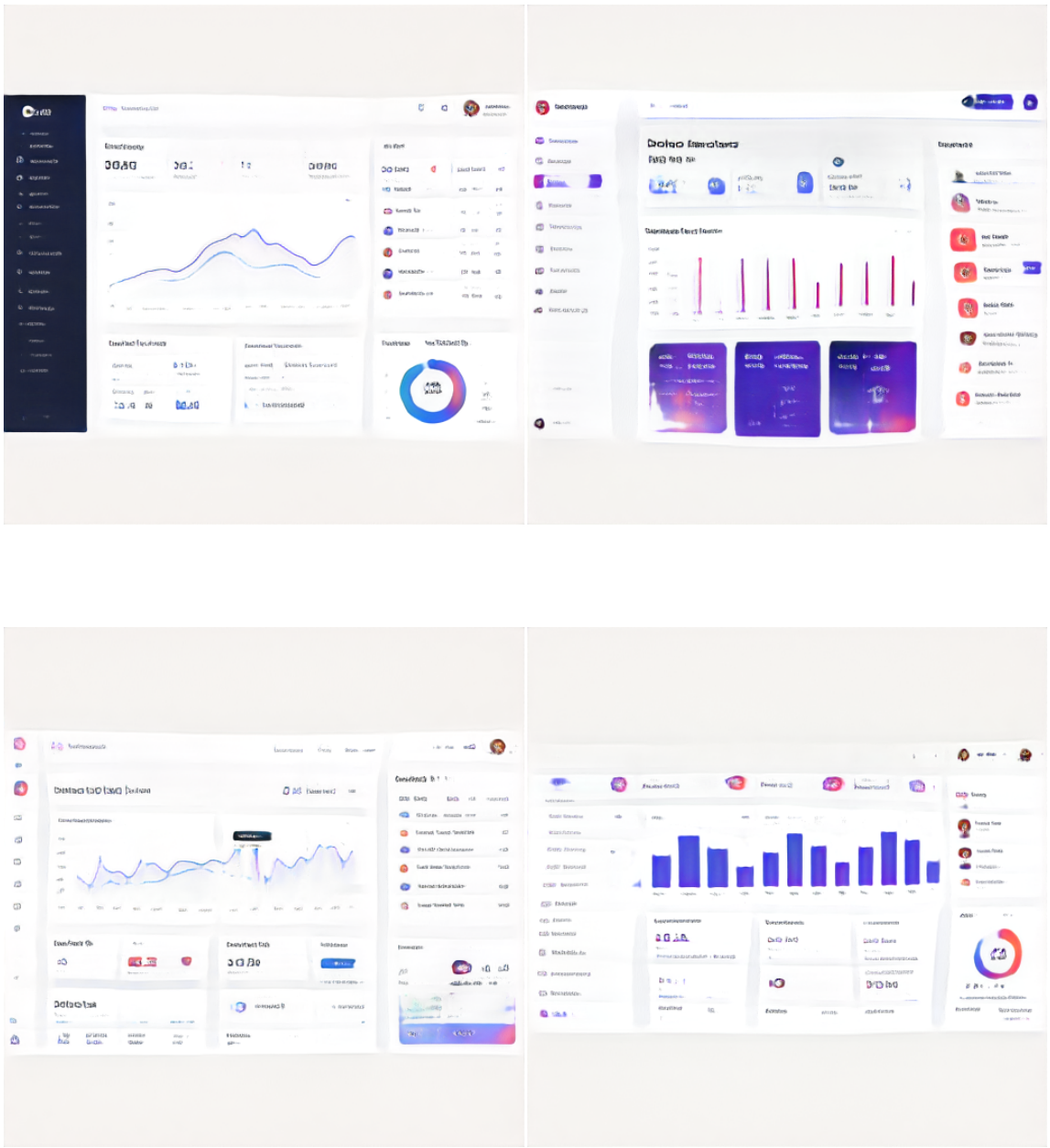
VII Project Management

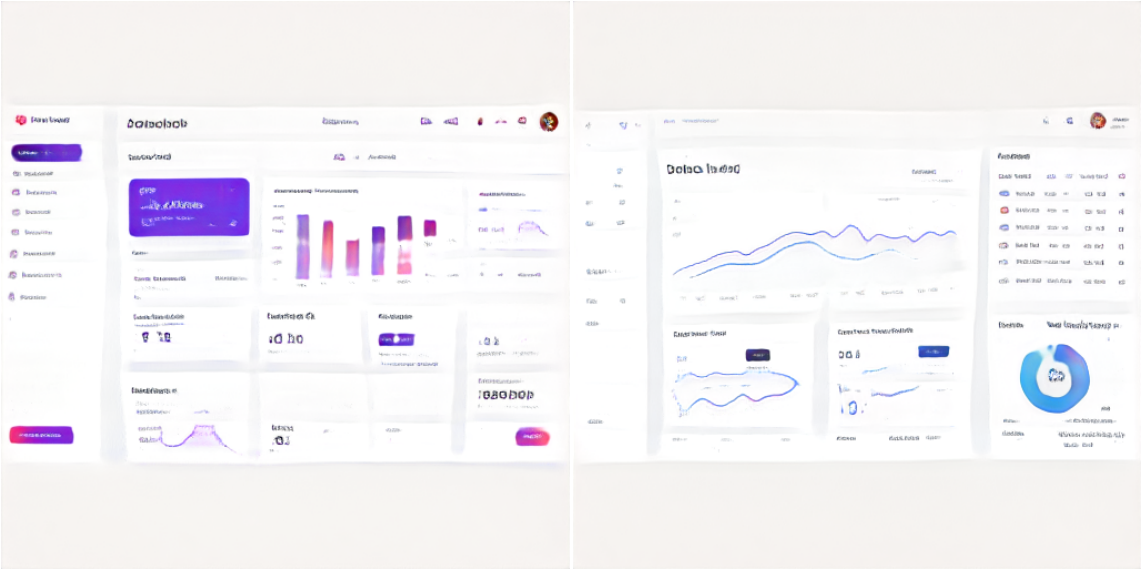


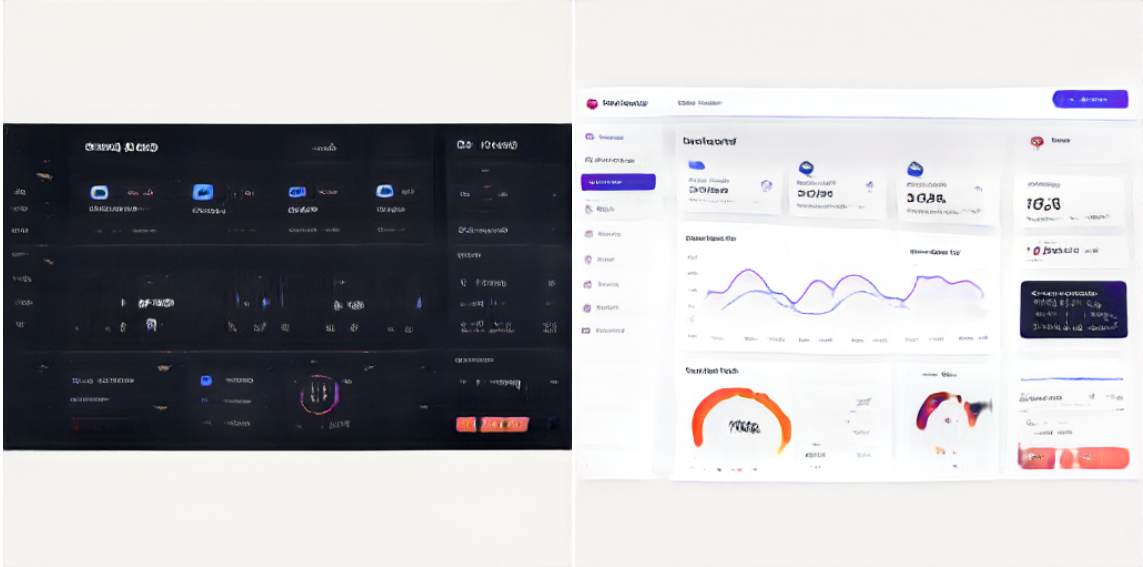


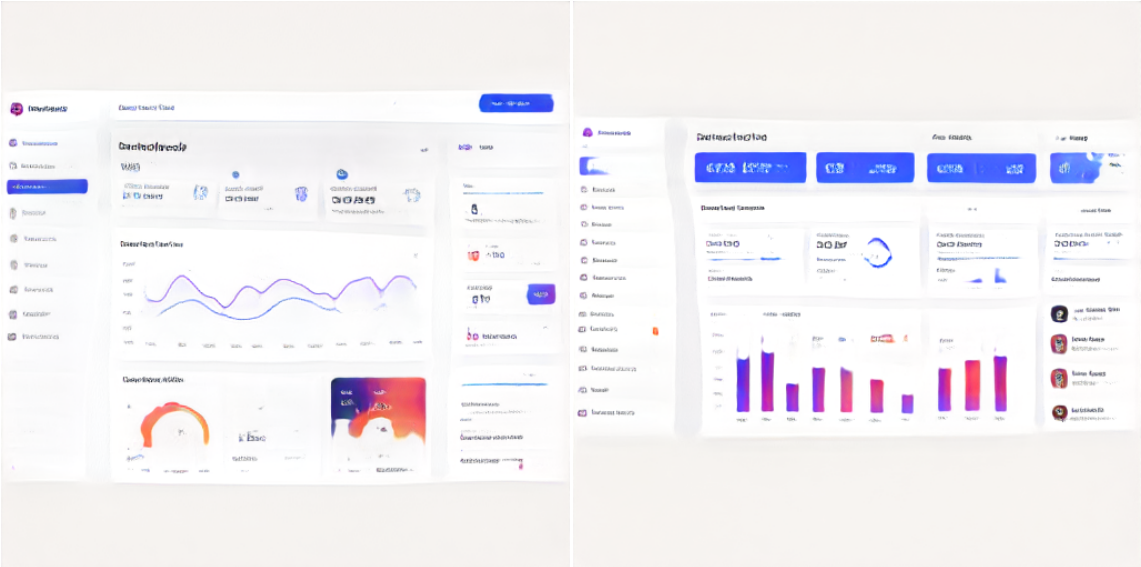


VIII Social









APPENDIX C

Image URL Extractor

```
/**
 * Simulate a right click so we can get the image url using the context menu.
 *
 *  attributed to @jmiserez: http://pyimg.co/9qe7y
 *  https://www.pyimagesearch.com/2017/12/04/how-to-create-a-deep-learning-dataset-using-google-images/
 */

function simulateRightClick(element){
    var event1 = new MouseEvent('mousedown', {
        bubbles : true,
        cancelable : false,
        view : window,
        button : 2,
        buttons : 2,
        clientX : element.getBoundingClientRect().x,
        clientY : element.getBoundingClientRect.y
    });
    element.dispatchEvent( event1 );
    var event2 = new MouseEvent( 'mouseup', {
        bubbles: true,
        cancelable: false,
        view: window,
        button: 2,
        buttons: 0,
        clientX: element.getBoundingClientRect().x,
        clientY: element.getBoundingClientRect().y
    } );
    element.dispatchEvent( event2 );
    var event3 = new MouseEvent( 'contextmenu', {
        bubbles: true,
        cancelable: false,
        view: window,
        button: 2,
        buttons: 0,
        clientX: element.getBoundingClientRect().x,
        clientY: element.getBoundingClientRect().y
    } );
    element.dispatchEvent( event3 );
}

/**
 * grabs a URL Parameter from a query string because Google Images
 * stores the full image URL in a query parameter
 *
 * @param   {string}  queryString  The Query String
 * @param   {string}  key          The key to grab a value for
 *
 * @return  {string}               value
 */
function getURLParam( queryString, key ) {
    var vars = queryString.replace( /^\?/, '' ).split( '&' );
    for ( let i = 0; i < vars.length; i++ ) {
        let pair = vars[ i ].split( '=' );
        if ( pair[0] == key ) {
            return pair[1];
        }
    }
    return false;
}

```

```
61
62 /**
63  * Generate and automatically download a txt file from the URL contents
64  *
65  * @param   {string}  contents  The contents to download
66  *
67  * @return  {void}
68  */
69 function createDownload( contents ) {
70     var hiddenElement = document.createElement( 'a' );
71     hiddenElement.href = 'data:attachment/text,' + encodeURI( contents );
72     hiddenElement.target = '_blank';
73     hiddenElement.download = 'urls.txt';
74     hiddenElement.click();
75 }
76
77 /**
78  * grab all URLs va a Promise that resolves once all URLs have been
79  * acquired
80  *
81  * @return  {object}  Promise object
82  */
83 function grabUrls() {
84     var urls = [];
85     return new Promise( function( resolve, reject ) {
86         var count = document.querySelectorAll(
87             '.isv-r a:first-of-type' ).length,
88             index = 0;
89         Array.prototype.forEach.call( document.querySelectorAll(
90             '.isv-r a:first-of-type' ), function( element ) {
91             // using the right click menu Google will generate the
92             // full-size URL; won't work in Internet Explorer
93             // (http://pyimg.co/byukr)
94             simulateRightClick( element.querySelector( ':scope img' ) );
95             // Wait for it to appear on the <a> element
96             var interval = setInterval( function() {
97                 if ( element.href.trim() !== '' ) {
98                     clearInterval( interval );
99                     // extract the full-size version of the image
100                    let googleUrl = element.href.replace( /.*(\?)/, '$1' ),
101                        fullImageUrl = decodeURIComponent(
102                            getURLParam( googleUrl, 'imgurl' ) );
103                    if ( fullImageUrl !== 'false' ) {
104                        urls.push( fullImageUrl );
105                    }
106                    // sometimes the URL returns a "false" string and
107                    // we still want to count those so our Promise
108                    // resolves
109                    index++;
110                    if ( index == ( count - 1 ) ) {
111                        resolve( urls );
112                    }
113                }
114            }, 10 );
115        } );
116    } );
117 }
118
119 /**
120  * Call the main function to grab the URLs and initiate the download
121  */
```

```
122 grabUrls().then( function( urls ) {
123     urls = urls.join( '\n' );
124     createDownload( urls );
125 } );
```

APPENDIX d

Image Download Script

```python
from imutils import paths
import argparse
import requests
import cv2
import os
import shutil

ap = argparse.ArgumentParser()
ap.add_argument("-u", "--urls", required=True, help="Path to file containging URLs")
ap.add_argument("-o", "--output", required=True, help="Path to output directory for images")
args = vars(ap.parse_args())

rows = open(args["urls"]).read().strip().split("\n")

#Download the images

total = 0
for url in rows:
    try:
        r = requests.get(url, stream=True)
        if r.status_code == 200:
            r.raw.decode_content = True
            file_format = url.split('.')
            if "?" in file_format[-1]:
                file_format = file_format[-1].split('?')
                file_format = file_format[0]
            else:
                file_format = file_format[-1]
            #file_format = file_format[-1][:3]
            if file_format == "jpg" or file_format == "png" or file_format == "jpeg":
                p = os.path.sep.join([args["output"], "{}.{}".format(str(total).zfill(8), file_format)])
                with open(p, 'wb') as f:
                    shutil.copyfileobj(r.raw, f)
                    print("[INFO] downloaded: {}".format(p))
                    total += 1
            else:
                p = os.path.sep.join([args["output"], "{}.{}".format(str(total).zfill(8), file_format)])
                raise Exception('not a supported file format')
    except Exception as e:
        print(e)
        print("[INFO] error downloading {}...skipping".format(p))

#loop over the image paths we just downloaded
for imagePath in paths.list_images(args["output"]):
    # initialize if the image should be deleted or not
    delete = False
    # try to load the image
    try:
        image = cv2.imread(imagePath)
        # if the image is `None` then we could not properly load it
        # from disk, so delete it
        if image is None:
            delete = True
    # if OpenCV cannot load the image then the image is likely
    # corrupt so we should delete it
    except:
        print("Except")
        delete = True
    # check to see if the image should be deleted
```

```
60      if delete:
61          print("[INFO] deleting {}".format(imagePath))
62          os.remove(imagePath)
```

APPENDIX e

Image Duplicate Detection

```python
# Made by following this tutorial https://www.pyimagesearch.com/2020/04/20/detect-and-remove-duplicate-images-from-a-dataset-for-deep-learning/#pyis-cta-modal
# Uses image hashing to detect duplicate images

from imutils import paths
import numpy as np
import argparse
import cv2
import os
import json

def dhash(image, hashsize=8):
    # Convert to greyscale adding a width column so we can compute the horizontal gradient
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    resized = cv2.resize(gray, (hashsize + 1, hashsize))

    #compute the horizontal gradient between adjacent column pixels
    diff = resized[:, 1:] > resized[:, :-1]

    return sum([2 ** i for (i, v) in enumerate(diff.flatten()) if v])

def detect_and_remove_from_dataset(filename):
    #Open the json file
    folder = os.path.join(args["dataset"], "dataset" + "." + "json")
    labels = []
    with open(folder, "r+") as file:
        filedata = json.load(file)
        labels = filedata["labels"]
        for i, label in enumerate(labels):
            if label[0] == filename:
                labels.pop(i)

    # Open the file again to do a clean overwrite to avoid any dangling data when writing smaller data.
    with open(folder, 'wt') as file:
        full_labels = {
            "labels" : labels
        }
        json.dump(full_labels, file, indent=4)


ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", help="Path to input dataset.", required=True)
ap.add_argument("-t", "--target", type=int, default=512, help="Remove any images who have a width below this number.")
ap.add_argument("-r", "--remove", type=int, default=1, help="whether or not duplicates should be removed (i.e., dry run).")
args = vars(ap.parse_args())

print("[INFO] computing image hashes")

imagePaths = list(paths.list_images(args["dataset"]))
hashes = {}
count = 0

for imagePath in imagePaths:
    image = cv2.imread(imagePath)
    (h, w) = image.shape[:2]
    if w < args["target"]:                # Remove any images below a specific target
        os.remove(imagePath)
        count+=1
```

```
        print("Removed an image that was too small ({}x{}).".format(w, h))
        continue
    h = dhash(image)

    p = hashes.get(h, [])
    p.append(imagePath)
    hashes[h] = p


# loop over the image hashes
for (h, hashedPaths) in hashes.items():
# check to see if there is more than one image with the same hash

    if len(hashedPaths) > 1:
        # check to see if this is a dry run
        if args["remove"] <= 0:
            # initialize a montage to store all images with the same
            # hash
            montage = None
            # loop over all image paths with the same hash
            for p in hashedPaths:
                # load the input image and resize it to a fixed width
                # and heightG
                image = cv2.imread(p)
                image = cv2.resize(image, (150, 150))
                # if our montage is None, initialize it
                if montage is None:
                    montage = image
                # otherwise, horizontally stack the images
                else:
                    montage = np.hstack([montage, image])

            # show the montage for the hash
            print("[INFO] hash: {}".format(h))
            cv2.imshow("Montage", montage)
            cv2.waitKey(0)

        else:
            count += 1
            for p in hashedPaths[1:]:
                print(p)
                filename = p.split('\\')[-1]
                detect_and_remove_from_dataset(filename)
                os.remove(p)

print("Removed " + str(count) + ' files')
```

APPENDIX f

MultiCrop Tool Source Code

```python
from bz2 import compress
import shutil
from tabnanny import check
import tkinter
from tkinter import messagebox
from tkinter.ttk import Progressbar
import cv2
from tkinter import *
from PIL import Image
from PIL import ImageTk
import tkinter.filedialog as fd
from imutils import paths
import numpy as np
import json
import os
import argparse


GENERAL_WIDGET = 0
IOT_WIDGET = 1
GRAPH_WIDGET = 2
INDICATOR_WIDGET = 3
MISC_IMAGE = 4
MISC_WIDGET = 5
FINANCIAL_WIDGET = 6

label_dict = {
    1 : "iot_dashboard",
    2 : "agriculture_dashboard",
    3 : "financial_dashboard",
    4 : "analytics_dashboard",
    5 : "misc_dashboard",
    6 : "fitness_dashboard",
    7 : "project_management",
    8 : "social_dashboard"
}

colour_mapping = {
    1 : "blue",
    2 : "black",
    3 : "cyan",
    4 : "yellow",
    5 : "magenta",
    6 : "green",
    7 : "#FF7D33",
    8 : "#33A5FF"
}

WIDGET_LABELS = [
    GENERAL_WIDGET,
    IOT_WIDGET,
    GRAPH_WIDGET,
    INDICATOR_WIDGET,
    MISC_IMAGE,
    MISC_WIDGET,
    FINANCIAL_WIDGET
]



BASE_STRUCTURE_LABELS_FILE = {
```

```
    "labels" : []
}

class MultiCrop():
    def __init__(self, args):
        self.pos = 0
        self.prog = 0
        self.imgs = []
        self.current_open_img = None
        self.current_img_name = ""
        self.current_img_ext = ""
        self.selections = np.empty((0, 5), dtype=float)
        self.images_loaded = False
        # Drawing
        self.rect = None
        self.x = self.y = 0
        self.start_y = None
        self.start_x = None
        self.curX = None
        self.curY = None
        self.latest_rect = None
        # Key press
        self.key_held = False
        self.number_key_active = None
        # Folder and files
        self.resume = False
        self.dest_folder = args["dest"]
        self.path = None
        self.main_json = None
        self.output_json = None
        self.labels_on = args["single_crop"]
        # Image resizing
        self.target_size = args["target_size"]
        self.scaling_factor = 1
        self.save_as = args["save_as"]

    '''
        Function for resetting canvas after clicking to the next image.
        Prevents possible memory leaks due to drawn objects remaining in the memory
        because they stayed on the canvas in the background
    '''
    def clear_canvas(self):
        # Reset all canvas related variables
        self.rect = None
        self.x = self.y = 0
        self.start_x = None
        self.start_y = None
        self.curX = None
        self.curY = None
        self.latest_rect = None
        self.scaling_factor = 1
        self.selections = np.empty((0, 5), dtype=float)
        # Clear the canvas and delete all drawn items
        self.img_panel.delete("all")

    def reset(self):
        self.clear_canvas()
        self.imgs = []
        self.pos = 0
        self.current_open_img = None
        self.number_key_active = None
```

```
123         self.current_img_name = ""
124         self.current_img_ext = ""
125         self.path = None
126 
127 
128     def set_output(self, output):
129         self.dest_folder = output
130 
131 
132     '''
133         Loads a folder of images using the windows file explorer
134     '''
135     def load_folder(self):
136         self.reset()                                # Make sure everything is re-initialised
137         self.path = fd.askdirectory()
138 
139         self.main_json = os.path.join(self.path, 'dataset' + '.' + 'json')
140         self.output_json = os.path.join(self.dest_folder, 'dataset' + '.' + 'json')
141 
142         # Re-init the labels json folder
143         images = list(paths.list_images(self.path))
144         if len(images) > 0:
145             self.imgs = images
146             self.images_loaded = True
147 
148             if self.check_label_data():
149                 ans = messagebox.askquestion("Resume", "Detected previously loaded data in
    dataset files, would you like to resume?")
150 
151                 if ans == 'yes':
152                     indx = self.read_latest_image_from_json()
153                     self.pos = indx + 1
154                 else:
155                     self.reset_folders()
156             else:
157                 self.reset_folders()
158 
159             #init the progress bar
160             self.init_progress()
161             #Load the first image in the folder
162             self.load_image(self.imgs[self.pos])
163 
164         else:
165             self.images_loaded = False
166             self.imgs = []
167 
168 
169     def check_label_data(self):
170         check_pass = False
171         paths = [
172             self.main_json,
173             self.output_json
174         ]
175 
176         for i in paths:
177             try:
178                 with open(i, "r") as file:
179                     file_data = json.load(file)
180                     if len(file_data["labels"]) == 0:
181                         check_pass = False
182                         break
```

```
                    else:
                        check_pass = True
            except:
                check_pass = False
                break

        return check_pass



    def read_latest_image_from_json(self):

        with open(self.main_json, "r+") as file:
            file_data = json.load(file)
            labels = file_data["labels"]
            last_accessed = labels[len(labels) - 1][0]

            filepath = os.path.join(self.path, last_accessed)

            try:
                indx = self.imgs.index(filepath)
            except:
                print("Error finding image in folder, resuming from start")
                indx = 0
            finally:
                return indx
    '''
        Resets the folders.
        -Empties the entire contents of the given output directory
        -Re-initializes the dataset.json files in the image directory and the output directory
    '''
    def reset_folders(self):
        self.empty_folder_contents(self.dest_folder)
        self.reinit_dataset_json(self.main_json)
        self.reinit_dataset_json(self.output_json)

    '''
        Empties the contents of a given folder
    '''
    def empty_folder_contents(self, folder):
        list_dir = os.listdir(folder)
        for file in list_dir:
            file_path = os.path.join(self.dest_folder, file)
            self.delete_file(file_path)

    '''
        This function allows the deletion of a specific file or folder.
    '''
    def delete_file(self, file_path):
        if os.path.isfile(file_path) or os.path.islink(file_path):
            print("Deleting: {}".format(file_path))
            os.unlink(file_path)
        elif os.path.isdir(file_path):
            shutil.rmtree(file_path)

    def remove_file_from_img_array(self, file_path):
        try:
            indx = self.imgs.index(file_path)
            self.imgs.pop(indx)
        except:
```

```
243             print("Error removing file.")
244 
245 
246     def reinit_dataset_json(self, filepath):
247 
248         try:
249             with open(filepath, "w+") as file:
250                 file_data = {
251                     "labels" : []
252                 }
253 
254                 file.seek(0)
255                 json.dump(file_data, file, indent=4)
256         except:
257             print("Error initializing json")
258 
259 
260     def load_image(self, file):
261 
262         img = cv2.imread(file)                                  # Read in the image
263         self.current_open_img = img.copy()                      # Take a copy of the image
before editing it for display. This raw image will be the one that is worked on.
264         dim = self.get_resolution(img)                          # Get the image resolution
265         resized = self.resize_full(img, 1024)                   # Resize the image to fit on
the canvas. Width should be 1024. Aspect ratio is maintained.
266         img = cv2.cvtColor(resized, cv2.COLOR_BGR2RGB)          # Convert the colour to rgb
for python image library
267         img = Image.fromarray(img)
268         pimg = ImageTk.PhotoImage(img)
269         # self.img_panel.configure(image=pimg)
270         # self.img_panel.image = pimg
271         self.img_panel.create_image(0, 0, anchor='nw', image=pimg)
272         self.img_panel.image = pimg
273         self.current_img_name, self.current_img_ext = os.path.basename(file).split('.')
274         # Display the resolution of the image
275         self.resolution_label['text'] = "{}w x {}h".format(dim[1], dim[0])
276 
277 # ================================================================================
278 #                               SAVE FUNCTIONS
279 # ================================================================================
280     '''
281         Saves all of the selected image crops into the output folder.
282         Also updates the main label file in the main image folder that was selected by the
user.
283     '''
284     def save_selection(self):
285 
286         if len(self.selections) > 0:
287             i = 0
288             for roi in self.selections:
289                 crop_img, label = self.crop(roi)
290                 self.save_image(crop_img, i, label)
291                 i += 1
292 
293             if self.labels_on:
294                 self.write_json_file(self.main_json, [self.current_img_name + '.' +
self.current_img_ext, self.number_key_active])  # Write to the user selected folders
dataset.json
295         else:
296             print("Nothing selected")
297 
298         self.save_success(len(self.selections)) # Say how many images have been saved
```

```
299
300     '''
301         Save an image and its label data to the output directory
302         img - The cropped image to be saved
303         img_num - The number of the image to be cropped based on the number that have been
    cropped from the main image. To be included in the name.
304         label - The label assigned to the image.
305     '''
306     def save_image(self, img, img_num, label):
307
308         compression = cv2.IMWRITE_PNG_COMPRESSION
309
310         if label == -1:
311             filename = self.current_img_name + "_img" + str(img_num) + "_label" +
    str(label) + '.' + self.save_as
312         else:
313             filename = self.current_img_name + '_processed' + '.' + self.save_as
314         filename_full = os.path.join(self.dest_folder, filename)
315
316         square = self.make_square(img)                                  # Add padding to make
    the image square
317         resized = self.resize_square(square, self.target_size)      # Resize the image to
    the specified target size
318
319         if self.labels_on:
320             self.write_json_file(self.output_json, [filename, str(label)])      # Save the
    label data
321
322         if self.save_as == "png":
323             compression = [cv2.IMWRITE_PNG_COMPRESSION, 0]          # png compression with
    the highest quality.
324         elif self.save_as == "jpg":
325             compression = [cv2.IMWRITE_JPEG_QUALITY, 100]           # Jpg compression with
    the highest quality.
326         elif self.save_as == "tiff":
327             compression = [cv2.IMWRITE_TIFF_COMPRESSION]
328
329         cv2.imwrite(filename_full, resized, compression)    # Save the image
330
331
332     def save_success(self, num):
333         messagebox.showinfo("Saved", "{} images saved".format(num))
334         # Move to the next image
335         self.next_img()
336         # Update the progress bar
337         self.update_progress()
338
339
340     def crop(self, roi: np.ndarray):
341         #Open the original image in opencv
342         img = self.current_open_img
343         print(roi)
344         roi = roi.astype(float).astype(int)
345         for i, x in enumerate(roi):
346             if i < len(roi) - 1:
347                 roi[i] = int(x / self.scaling_factor)
348
349         x0, y0, x1, y1, label = roi
350
351         crop_img = img[y0:y1, x0:x1]
352         return crop_img, label
353
```

```
354     def write_json_file(self, filepath, new_data):
355 
356         # with open(filepath, 'r+') as file:
357         #     # Add the new label, file pair to the labels json file
358         #     file_data = json.load(file)
359         #     file_data["labels"].append(new_data)
360         #     file.seek(0)
361 
362         #     #Save the file with the added json entry
363         #     json.dump(file_data, file, indent=4)
364         labels = []
365 
366         with open(filepath, 'r+') as file:
367              # Add the new label, file pair to the labels json file
368              file_data = json.load(file)
369              labels = file_data["labels"]
370              labels.append(new_data)
371 
372         with open(filepath, 'wt') as file:
373             full_labels = {
374                 "labels" : labels
375             }
376             json.dump(full_labels, file, indent=4)
377 
378 
379 # ========================================================================
380 
381     def undo_last_selection(self):
382         print("ORig")
383         print(self.selections)
384         self.selections = self.selections[:-1]
385         self.img_panel.delete(self.latest_rect)
386         self.img_panel.delete(self.rect)
387         self.rect = None
388         print("New")
389         print(self.selections)
390 
391 
392 
393     '''
394         Make the cropped image square for training on StyleGAN2 or 3
395     '''
396     def make_square(self, img):
397         GRAY = [240, 242, 245]
398         target = 0
399         # Get the current img width and height
400         h, w = img.shape[0], img.shape[1]
401 
402         if h == w:
403             # Already square
404             return img
405         elif h > w:
406             # Height is the target because it is the largest
407             target = h
408             delta_w = target - img.shape[1]                    # Calculate change in width
409             left, right = delta_w//2, delta_w-(delta_w//2)      # Calculate left and right
410             top = bottom = 0                                    # don't need to change top
    and bottom
411         else:
412             # Width is the target
413             target = w
```

```
414             delta_h = target - img.shape[0]
415             top, bottom = delta_h//2, delta_h-(delta_h//2)
416             left = right = 0
417 
418         print("Adding border - bottom: {}, top: {}, left: {}, right: {}".format(top,
    bottom, left, right))
419         img = cv2.copyMakeBorder(img, top, bottom, left, right, 0, cv2.BORDER_CONSTANT,
    value=GRAY)
420 
421         return img
422 
423     '''
424         Resize the cropped image to a specified size
425     '''
426     def resize_square(self, img, size):
427         dim = None
428         (h, w) = img.shape[:2]
429 
430         # if width is None:
431         #     r = size / float(h)
432         #     dim = (int(w * r), size)
433 
434         # else:
435         #     r = size / float(w)
436         #     dim = (size, int(h*r))
437 
438         dim = (size,size)
439         resized = cv2.resize(img, dim, interpolation=cv2.INTER_AREA)
440 
441         new_height = resized.shape[0]
442         new_width = resized.shape[1]
443         print("Image: , Original Size: ({}, {}). Image size : ({},{})".format(w, h,
    new_width, new_height))
444 
445         return resized
446 
447     def resize_full(self, img, width=None, height=None, inter=cv2.INTER_AREA):
448         dim = None
449         r = None
450         (h, w) = img.shape[:2]
451 
452         if w <= width:                  # If the width is already the resize width or
    smaller we don't need to do anything as it will already fit on the screen. The scaling
    should be set to 1.
453             self.scaling_factor = 1
454             return img
455 
456         if width is None:
457             r = height / float(h)
458             dim = (int(w * r), height)
459 
460         else:
461             r = width / float(w)
462             dim = (width, int(h*r))
463 
464         resized = cv2.resize(img, dim, interpolation=inter)
465         self.scaling_factor = r
466 
467         new_height = resized.shape[0]
468         new_width = resized.shape[1]
469         print("Original Size: ({}, {}). Image size : ({},{})".format(w, h, new_width,
    new_height))
```

```
470
471         return resized
472
473     def get_resolution(self, img):
474         return img.shape
475
476     def previous_img(self):
477
478         #Reset all the drawing variables
479         self.clear_canvas()
480
481         start = self.pos
482
483         if self.pos - 1 < 0:
484             self.pos = len(self.imgs) - 1
485         else:
486             self.pos -= 1
487
488         if(self.pos != start):
489             self.load_image(self.imgs[self.pos])
490
491     def next_img(self):
492         #Reset all the drawing variables
493         self.clear_canvas()
494
495         start = self.pos
496
497         if self.pos + 1 > len(self.imgs) - 1:
498             self.pos = 0
499         else:
500             self.pos += 1
501
502         if(self.pos != start):
503             self.load_image(self.imgs[self.pos])
504
505     def next_img_no_incr(self):
506         self.clear_canvas()
507         self.load_image(self.imgs[self.pos])
508
509     def update_progress(self):
510         self.prog += 1
511         val = self.linear_map(0, 100, 0, len(self.imgs), self.prog)
512         self.progress['value'] = val
513         self.progress_label['text'] = self.update_progress_label()
514
515     def update_progress_label(self):
516         return f"Progress: {round(self.progress['value'])}% ({self.prog}/{len(self.imgs)})"
517
518     def linear_map(self, out_start, out_end, in_start, in_end, x):
519         return out_start + ((out_end - out_start) / (in_end - in_start)) * (x - in_start)
520
521
522     def init_progress(self):
523         self.prog = self.pos
524         val = self.linear_map(0,100, 0, len(self.imgs), self.pos)
525         self.progress['value'] = val
526         self.progress_label['text'] = self.update_progress_label()
527
528
529
530     def on_right_key(self, event):
```

```
531         self.next_img()
532 
533     def on_left_key(self, event):
534         self.previous_img()
535 
536     def on_mouse_press(self, event):
537         print("Mouse pressed")
538         self.start_x = self.img_panel.canvasx(event.x)
539         self.start_y = self.img_panel.canvasy(event.y)
540 
541         if not self.rect:
542             self.rect = self.img_panel.create_rectangle(self.x, self.y, 1, 1,
outline=colour_mapping[int(self.number_key_active)], width=2)
543 
544 
545 
546     def on_move_press(self, event):
547         self.curX = self.img_panel.canvasx(event.x)
548         self.curY = self.img_panel.canvasy(event.y)
549 
550         self.img_panel.coords(self.rect, self.start_x, self.start_y, self.curX, self.curY)
551 
552 
553 
554     def on_mouse_release(self, event):
555 
556         #print(self.img_panel.coords(self.rect))
557 
558         if self.img_panel.coords:
559 
560             coords = np.array(self.img_panel.coords(self.rect))                 #Get the
coordinates of the user selection
561 
562             # x0, y0, x1, y1 = coords                                             #Split
out the coordinates of the rectangle
563 
564             # self.latest_rect = self.img_panel.create_rectangle(x0, y0, x1, y1,
outline="red")      #Create a rectangle on the canvas for the user to see
565             # self.selections = np.append(self.selections, np.array([coords]), axis=0)
566             if self.number_key_active is not None:
567                 self.draw_rect(label=self.number_key_active,
colour=colour_mapping[int(self.number_key_active)], coords=coords)
568                 pass
569             else:
570                 self.draw_rect(colour="red", coords=coords)
571                 pass
572 
573             print(self.selections)
574 
575     def on_number_key_pressed(self, event):
576 
577         if self.key_held == False:
578             print("Key pressed " + event.char)
579             self.number_key_active = event.char
580             self.key_held = True
581             self.label_indicator['text'] = label_dict[int(self.number_key_active)]
582             self.save_button['state'] = NORMAL
583             self.resize_main_button['state'] = NORMAL
584 
585     def on_delete(self):
586         filepath = os.path.join(self.path, self.imgs[self.pos])
587         ans = messagebox.askyesno("Delete", "Are you sure you want to permanently Delete
```

```
this image?")
588         if ans == True:
589             self.delete_file(filepath)                  # Delete the file from the
directory
590             self.remove_file_from_img_array(filepath)   # Remove the file from the image
array
591             self.next_img_no_incr()
592             self.update_progress()
593         else:
594             pass
595     '''
596         For when the user wants to resize the main image to the target size without
cropping any sub images.
597     '''
598     def on_resize_main(self):
599         if self.number_key_active is not None:
600             self.save_image(self.current_open_img, 0, self.number_key_active)
601             self.write_json_file(self.main_json, [self.current_img_name + '.' +
self.current_img_ext, self.number_key_active])
602             self.save_success(1)
603         else:
604             messagebox.showerror("No Label", "Please select a label and try again.")
605
606
607
608     def on_number_key_released(self, event):
609
610         #print("Key released " + event.char)
611         #self.number_key_active = None
612         self.key_held = False
613         #self.label_indicator['text'] = "No Label Selected"
614
615     def on_s_pressed(self, event):
616         self.save_selection()
617
618     def on_r_pressed(self, event):
619         self.on_resize_main()
620
621     def on_d_pressed(self, event):
622         self.on_delete()
623
624     def draw_rect(self, colour, coords, label=-1):
625         x0, y0, x1, y1 = coords
626         self.latest_rect = self.img_panel.create_rectangle(x0, y0, x1, y1, outline=colour,
width=2)
627         if abs(x0 - x1) > 10 and abs(y0 - y1) > 10:
628             coords = np.append(coords, label)
629             self.selections = np.append(self.selections, np.array([coords]), axis=0)
630         else:
631             print("Rectangle too small")
632
633
634     def build_gui(self):
635
636         self.root = Tk()
637         self.root.title("MultiCrop Tool")
638         self.root.geometry('1250x1024')
639
640         self.left_frame = Frame(self.root, height=1024, width=1024)
641         self.left_frame.pack(side=LEFT)
642
643         # self.img_panel = Label(self.left_frame)
```

```
644         # self.img_panel.image = None
645         # self.img_panel.pack(side=LEFT)
646         self.img_panel = Canvas(self.left_frame, height=1024, width=1024, bg="gray")
647         self.img_panel.pack(side=LEFT)
648 
649         right_frame = Frame(self.root, height=1024, width=176)
650         right_frame.pack(side=RIGHT)
651 
652         load_button = Button(right_frame, text="Load Images", command=self.load_folder,
    padx=50, pady=10)
653         load_button.grid(row=0, column=0, columnspan=2, pady=10)
654 
655         self.save_button = Button(right_frame, text="Save Selection",
    command=self.save_selection, padx=50, pady=10, state=DISABLED)
656         self.save_button.grid(row=1, column=0, columnspan=2, pady=10)
657 
658         self.resize_main_button = Button(right_frame, text="Resize Main",
    command=self.on_resize_main, padx=50, pady=10, state=DISABLED)
659         self.resize_main_button.grid(row=3, column=0, columnspan=2, pady=10)
660 
661         self.undo_button = Button(right_frame, text="Undo Last",
    command=self.undo_last_selection, padx=50, pady=10)
662         self.undo_button.grid(row=4, column=0, columnspan=2, pady=10)
663 
664         delete_button = Button(right_frame, text="Delete", command=self.on_delete, padx=50,
    pady=10)
665         delete_button.grid(row=5, column=0, columnspan=2)
666 
667         # back_button = Button(right_frame, text="\u2190", command=self.previous_img,
    padx=30, pady=10, )
668         # back_button.grid(row=3, column=0)
669 
670         # forward_button = Button(right_frame, text="\u2192", command=self.next_img,
    padx=30, pady=10)
671         # forward_button.grid(row=3, column=1)
672 
673         self.label_indicator = Label(right_frame, text="No Label Selected")
674         self.label_indicator.grid(row=6, column=0, columnspan=2)
675 
676         self.progress = Progressbar(right_frame, orient='horizontal', length=100,
    mode='determinate')
677         self.progress.grid(row=14, column=0, columnspan=2)
678 
679         self.progress_label = Label(right_frame, text=self.update_progress_label())
680         self.progress_label.grid(row=15, column=0)
681 
682         self.resolution_label = Label(right_frame, text="")
683         self.resolution_label.grid(row=16, column=0)
684 
685         info_string = ""
686 
687         for key, value in label_dict.items():
688             info_string = info_string + "%i = %s \n"%(key, value)
689 
690         label_info = Label(right_frame, text=info_string)
691         label_info.grid(row=7, column=0, columnspan=2, rowspan=5)
692 
693 
694 
695         # Bind the arrow keys to go through pictures easier
696         self.root.bind('<Right>', self.on_right_key)
697         self.root.bind('<Left>', self.on_left_key)
698 
```

```
699         # Bind some hot keys for saving and resizing main image
700         self.root.bind('s', self.on_s_pressed)
701         self.root.bind('r', self.on_r_pressed)
702         self.root.bind('d', self.on_d_pressed)
703 
704         #Bind the number keys for different labels
705         for i in range(10):
706             self.root.bind(str(i), self.on_number_key_pressed)
707 
708         #Bind the number key release event
709         for i in range(10):
710             self.root.bind("<KeyRelease-%i>"%(i), self.on_number_key_released)
711 
712         #Bind the mouse click events to the canvas
713         self.img_panel.bind("<ButtonPress-1>", self.on_mouse_press)
714         self.img_panel.bind("<ButtonRelease-1>", self.on_mouse_release)
715         self.img_panel.bind("<B1-Motion>", self.on_move_press)
716 
717 
718         self.root.mainloop()
719 
720 
721 def main():
722     ap = argparse.ArgumentParser()
723     ap.add_argument("--dest", required=True, help="Output folder for the cropped files")
724     ap.add_argument("--target-size", required=True, type=int, help="The target size the
    images should be resized to at the end of the cropping.")
725     ap.add_argument("--save-as", type=str, default="png", help="The file type to save the
    output images as. Default is png.", choices=["png", "jpg", "tif"])
726     ap.add_argument("--single-crop", action="store_false", help="For cleaning up the
    dataset. No label data will be saved.")
727     args = vars(ap.parse_args())
728 
729     mc = MultiCrop(args)
730 
731     mc.build_gui()
732 
733 
734 if __name__ == "__main__":
735     main()
```