Icon Generation with a Generative Adversarial Network Conditioned on class

Liviu Cotiuga
ID: 1683796

Supervisor:
Professor Uday Reddy

School of Computer Science
University of Birmingham

Contents

1 Abstract

Icon designing has always been a long and complicated process as the designers need to go through a set of sketches and iterations. Each drafted icon can take a lot of time and effort which makes it an expensive process. The goal of this project is to leverage generative algorithms to produce reasonable looking icons conditioned on fifty different random classes. This will make an enormous difference in the designing process as it can make the process shorter by skipping some key iterations. The designers will be able to use the generated icons as inspiration in the creative process or even use them as final products. The problem with the icons is that they don't have a continuous latent space and there is a limited control over shaping the output, but in the past few years, the generative models have made incredible progress and are one of the most promising approaches towards the project's goal. Therefore, I propose to use a Conditional Deep Convolutional Generative Adversarial Networks (cDCGANs) to generate icons that will offer a first glance at how Artificial Intelligence can revolutionise the design industry by assisting the designers in their creative process and how is this helpful in speeding up the process.

2 Introduction

Icon designing is referring to the designing process of a graphic symbol such as an action or an entity, but it can also represent a program or a collection of data on a computer system [1]. Usually, the icons are simple two-dimensional drawings or black silhouette. However, they can also be complex presenting a combination of graphic design elements such as three-dimensional perspective effects, shadows, or linear and radial colour gradients [1]. Designing an icon is a complex process which in most cases involves a collaboration between designers and customers. Recent advancements in generative models, specifically Generative Adversarial Networks (GANs), could assist designers in creating the icons by either providing the icons for inspiration or by reducing the number of iterations needed to create the final product. In the future, GANs will eventually become a design partner and a tool that designers can use to meet ever-evolving workplace demands [2].

Generative Adversarial Networks have proven to be successful in modelling and generating high dimensional data. They are composed of two models, a generator and a discriminator and it corresponds to a minimax two-player game. The generator aims to produce new data similar to the one in the dataset while the discriminator tries to recognise if an input data is real or fake. If an input data is real, it belongs to the original dataset while if it is fake it was generated by a forger. The Generator (forger) needs to learn how to create data in such a way that the Discriminator is not able to distinguish whether it is fake [3].

In order to generate reasonable-looking icons, a very large dataset of icons is needed. This subject has not been approached by many before and it is difficult to find a large enough dataset like MNIST digit dataset, which has around 60.000 images of hand-written digits. However, I managed to find a dataset called 'Icons-50' which contains 10.000 images belonging to 50 classes of icons (e.g. airplane, ball, clock, drinks, family, etc) collected from different technology companies (e.g. Apple, Samsung, Google).

The problem with the GANs is that the icons are generated from an unknown noise distribution. This is known as the input latent code and there is no way to control the type of images that are generated. To control the generation process, the latent input code must be shaped and this can be accomplished by implementing the class conditions using conditional Generative Adversarial Networks (cGAN). This is an architecture that was recently introduced. The novelty is that in the training process the label information y was added which is the icon class condition. Therefore, the generator model can be trained to produce data conditioned on the given class. This extra information y is being fed into both the generative and discriminative models as an additional input layer.

I propose to use a conditional Deep Convolutional Generative Adversarial Networks (CDC-GAN) to tackle the problem of generating icons based on the designer's preference of the class. I will talk about the architecture in a detailed manner in the following sections.

3 Literature Review

Reading through some articles about icon designing I have found that Artificial Intelligence (AI) could become a useful tool that designers could use in the creative process to make it less expensive and less time consuming [2]. Doing some research about AI and machine learning I have learned that they are powerful tools and it could generate images using generative models. A generative model is a method of learning any kind of data distribution using unsupervised learning[1]. Two of the most efficient and commonly used generative models are Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs). VAEs can make good generations and they are usually easier to train and get working, but they are more suitable for compressing data to lower dimensions. Although, GANs have recently shown they can be extremely powerful in generating images. The icon generation using GANs has not been tackled before, but some academic papers are trying to solve similar problems to what this project is aiming for. This section has the purpose of discussing and explaining the architectures used to reach the goal of generating icons based on a class condition.

[1]Unsupervised learning is a machine learning technique where no label is given to the learning algorithm, allowing the model to discover and work on its own.

3.1 Generative Adversarial Networks

GANs were firstly proposed by Goodfellow et al. [4] and then they were widely used for realistic image generation. The architecture as seen in Figure 1 involves two different neural networks: a generator that is used to generate fake images that look just like the ones in the dataset and a discriminator which is used to classify the generated images as real (from the dataset) or fake (generated). These neural networks learn through a loss function which is a method to analyse how well it models the data.

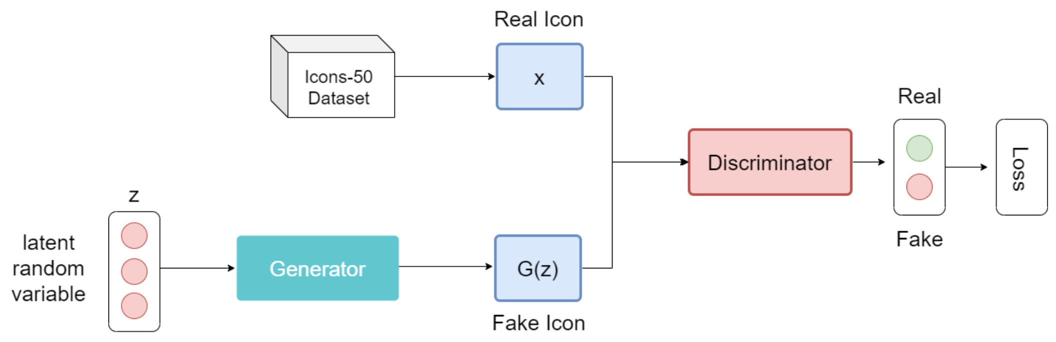


Figure 1: Generative Adversarial Networks architecture.

The Generator G(z, $\theta 1$)[2]is mapping the input noise variable z to the data space x, which means it is trained to generate data as realistic as possible by creating an RGB image with the same dimensions (i.e. 3x64x64) as the training images. The loss function for the generator is being used to maximise D(G(z)) representing the fake image generated.

The Discriminator D(x, $\theta 2$)[2] takes an image as input. This can be either the fake image from the generator (D(G(z))) or the real image from the training images (D(x)). The output is a probability that the data came from the real dataset in the range (0,1). The loss function for the discriminator is being used to maximise the function D(x) and minimise D(G(z)).

These two neural networks are trying to optimise opposite loss functions. This is a contest with each other in a two-player minimax game with the value function V (G, D) as seen in Figure 2. The generator is trying to maximise its probability of having the outputs recognised as real (log D(x)) and the discriminator is trying to minimise it (log(1-D(G(z))) [3].

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

Figure 2: The cost function V(G, D) for the Generative Adversarial Networks.

[2]In both cases, $\theta 1$ or $\theta 2$, it represents the weights or parameters that define each neural network.

3.2 Deep Convolutional Generative Adversarial Networks

GANs can be remarkable in terms of generating images and the goal is to train a stable model, but it can be challenging as they are known to have several pitfalls. Training both the generator and the discriminator simultaneously makes the process unstable. After extensive model exploration, Alec Radford et al. [5] developed a deep convolutional GAN (DCGANs) that tackles this issue and performs better image synthesis tasks. As the project is about icons, their shapes need to be reasonable-looking and have much less noise than usual around the icon. The best network design for this goal is DCGAN as it has a set of architectural constraints that makes the GANs stabilised [6]. DCGAN is composed of convolution layers[3] without max pooling [4] or fully connected layers[4] and the idea behind this design is to scale up the GANs (Figure 3). The main points that need to be followed are the following:

1. Replace pooling layers with convolutional stride and use transposed convolution for up-sampling.
2. Eliminate fully connected layers on top of convolutional layers.
3. Use Batch Normalisation except for the output layer for the generator and the input layer of the discriminator.
4. Use ReLU activation in the generator except for the output, which uses tanh.
5. Use LeakyReLU activation in the discriminator.

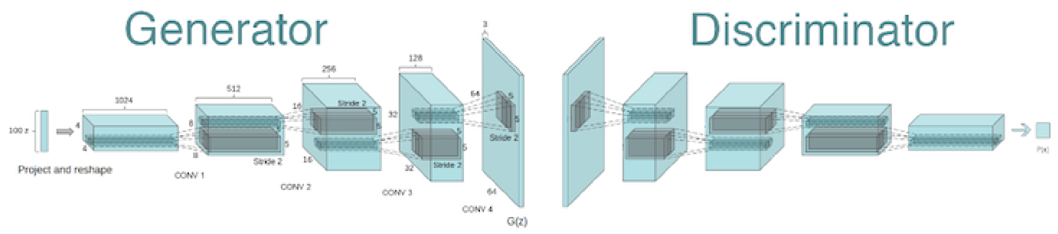


Figure 3: Deep Convolutional GANs architecture

3.3 Conditionality

To generate the icons based on a certain class (i.e. book, airplane, etc) I need to make use of conditional GANs (cGANs) which is an extension of GANs recently introduced by Mehdi

[3]A convolution layer is a layer that is applying filters over an image to extract different features.

[4]Max pooling and fully connected layers are layers that perform different tasks over an input data. Max pooling layers have the purpose of taking the maximum value of a filter region from the image and fully connected layers are using the flattened result to classify it.

Mirza et al [6]. GANs can be extended to a conditional model by adding to both the generator and the discriminator an additional information y which would be the conditioning (Figure 5). Normal GANs don't have control of the output, and it generates random images, but this problem can be addressed with cGANs. The cost function is the same and the only thing that has been added was the additional information y as follows:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x}|\boldsymbol{y})] + \mathbb{E}_{\boldsymbol{z} \sim p_z(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z}|\boldsymbol{y})))].$$

Figure 4: Loss function for conditional GANs.

3.4 GAN Applications

GANs architectures have been applied to numerous projects such as generating photographs of new human faces [7] and with different apparent ages [8], translating photos to emojis [9], generating cartoon characters [10] and many others.

Icon generation has never been investigated, but logo generation was tackled by Sage et al [11] in the paper "Logo Synthesis and Manipulation with Clustered Generative Adversarial Networks". They managed to build the Large Logo Dataset (LLD) consisting of 600.000+ logos. Also, they generated a high diversity of plausible logos and made use of synthetic labels obtained through clustering to stabilise GANs training.

Another paper that had a significant impact on the project's research is "LoGAN: Generating Logos with Generative Adversarial Networks conditioned on colour" by Mino et al. [12]. They accomplished to generate logos conditioned on twelve different colours by using an improved auxiliary classifier GAN (ACGAN). Their proposed architecture is Auxiliary Classifier Wasserstein GAN which consists of three neural networks: a generator, a discriminator and an additional classification network. The additional classification network has the purpose of assisting the discriminator in classifying the logos.

The method that I used for the icon generation project differs in several architectural choices from the above papers. I used deep convolutional GANs for training a more stable model and conditional GANs for having control over the output. This approach is considerably simpler, and I will prove that it is effective on icon generation.

4 Software Development Process

This section aims to inform what Software Engineering Principles have been used in the creation and implementation of the system.

4.1 Methodology

Artificial Intelligence systems are large and complex as they requires a lot of careful paper research to implement the right components and parameters (i.e. weights initialisation). To facilitate an easier implementation of the system's code I had to adhere to a Software Engineering Process that would allow me to develop in increments, as seen in the Agile methodologies.

Incremental Development. Taking small steps towards the goal of generating icons was very advantageous as I built the system progressively. I started with icon generation using just normal GANs, without any condition to control the output and without trying to stabilise it. The goal was to deliver improvements to the system very frequently using Kanban-style user stories; creating a to-do list every week has enabled me to analyse the workflow and it speeded up the project development and performance.

Modularity. Developing the whole system in one go would have been a problem as it could have many errors because of its size. To solve this problem, the functionality of the program was divided into independent modules such that each module contains the code necessary to execute only one part of the system. As an example, I created the Plot Images Function - which plots a sample of images from the data loader alongside their class labels.

4.2 General System Requirements

Functional Requirements

1. The user should be able to plot a sample of images from the dataset alongside their class labels.
2. The user should be able to train the model and generate icons from every wished class.
3. The user should be able to see the generated icons (with the 3x64x64 dimensions) after the training of the model is completed. The user should also see the class label of the icon generated.
4. The system should be able to show a relevant error in case of system failure and should specify the type of the error.

Non-functional Requirements

1. Plotting generated images should have the dimensions 3x64x64 and it should specify which class it belongs to.
2. The system should not take too long to plot the generated images.
3. The system's output should be clear, and everyone should be able to understand it.

5 Approach

A variety of academic papers have been explored to find the most suitable, simpler and efficient architecture for this project. This section will cover the architectural patterns and implementations that have been used with detailed descriptions of why they were integrated and what is their meaning.

5.1 System Design

The overall architecture used based on the study that was conducted is Conditional Deep Convolutional Generative Adversarial Networks. I followed the official DCGAN paper [5] as the model structure. The main differences that are introduced in the system's implementation are the conditionality and the dictionary of classes. This section is provided with more details about the architecture's design and it is organised as follows: Section 1 – presents the chosen design for the networks, Section 2 – the conditionality, Section 3 – loss function and optimisers and Section 4 – the training loop.

5.1.1 Networks Design

As mentioned in the above sections, the generator's purpose is to produce new reasonable-looking icons that ideally are indistinguishable from the real ones in the dataset and the discriminator's purpose is to classify the generated image as real or fake.

To choose the appropriate architecture for the networks, I had to consider what layer type (fully connected or convolutional) and how many layers were needed. The goal is to use a stable and simple architecture that performs effectively for the icon generation. Firstly, I experimented with normal GANs architecture consisting of fully connected layers, but I realised it was very expensive in terms of memory (as each connection has its own weights) and the output generated was not very clear and it was grainy. Therefore, I experimented with a cD-CGAN architecture and the results are promising, as it has fewer number of connections and weights which makes it cheaper and simpler than fully connected layers.

The Generator model starts by taking as input a random variable z plus the class label and they are passed through a series of strided two-dimensional convolutional transpose layers (ConvTranspose2d). Every convolutional transpose layer is paired with a two-dimensional batch normalisation layer (BatchNorm2d) and a rectified linear units (ReLU) activation. The ReLU activation allows the model to learn more quickly and the batch normalisation layer is normalising the input to each unit to have zero mean and unit variance which means that it is stabilising the learning [5]. For the classes, I used a function that returns a dictionary where the value of each class is a dictionary containing the indices and the number of datapoints. After the random variable z and the classes are being passed through the series of convolutional layers, the output is being fed into a tanh function in order to return it to the input data range of [-1, 1]. The random variable z is then concatenated with the variable y which represents the condition on the class.

The Discriminator model takes as input an RGB image concatenated with the variable y representing the class condition and it is being passed through a series of Conv2d (two-dimensional convolution), BatchNorm2d and Leaky ReLU layers. The order of the layers has been flipped and the ConvTranspose2d is replaced with a Conv2d layer. The LeakyReLU layer has been proved to work better for higher resolution modelling as specified in the DCGAN paper and it performs better for classification. In order to avoid model instability, the BatchNorm2d has not been applied to the generator's output layer and discriminator's input layer. The last layer of the network is a single node that flattens the output from the last convolution plus the class condition and outputs the probability of being true or false through a sigmoid function.

5.1.2 Conditionality

The main difference between this project and the papers that were explored is how the conditionality was done. There are two key extensions of GANs architecture that I considered in the process of choosing the right design: conditional GANs (cGANs) and Auxiliary Classifier GANs (ACGANs).

Conditional GAN changes the training of the model. The generator is provided with both the random variable z and the class label as an input and it generates an image for the specific class. Also, the discriminator is provided with the image generated and the class label and it must classify whether the image is real or fake as we mentioned in the networks design section.

CGAN is an advantageous extension of GANs as it is simple and makes the training process more stable by generating images conditioned on the class labels.

Auxiliary Classifier GAN GAN is an extension of cGANs and the main difference between them is the discriminator as it does not receive the class label as an input. However, it must predict the class label and whether the generated image is real or fake. As I have seen in the paper [13] the discriminator and the auxiliary classifier can be considered separate networks. The architecture consists of three neural networks: generator, discriminator and the classification network. Due to lack of time, this architecture is too complicated to implement and the goal is to produce reasonable-looking icons in a simpler and effective way. Therefore, the best extension is conditional GANs which has proven to satisfy the project's goal.

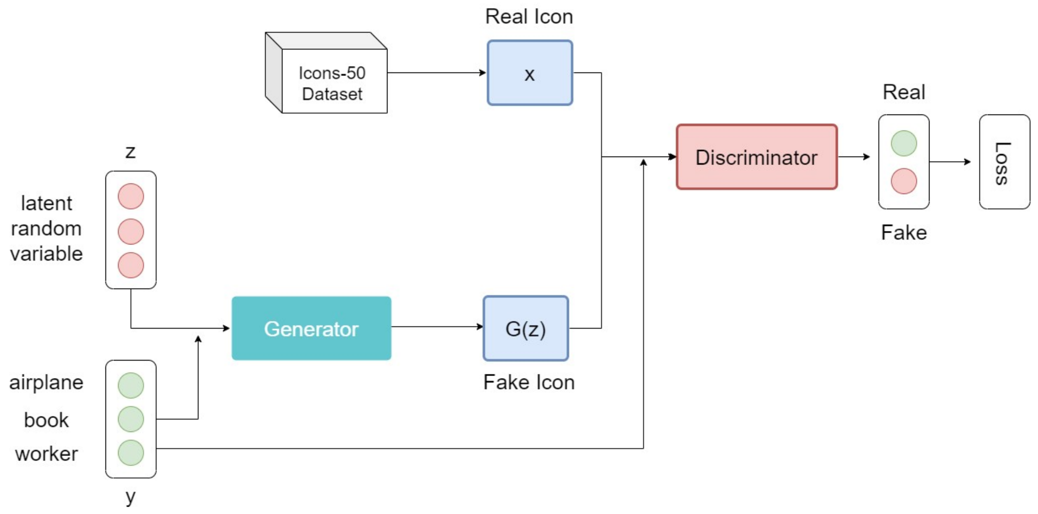


Figure 5: Conditional GAN architecture – in order to train the G and D better it adds the label to both of the networks.

5.1.3 Loss Function and Optimisers

The loss function has the purpose of measuring how well the G and D models are performing. As this project is a classification problem (yes/no decisions), the default loss function for this purpose is Binary Cross-Entropy Loss (BCELoss) which is defined as follows:

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{i=0}^{N} (y * log(\hat{y}_i) + (1 - y) * log(1 - \hat{y}_i))$$

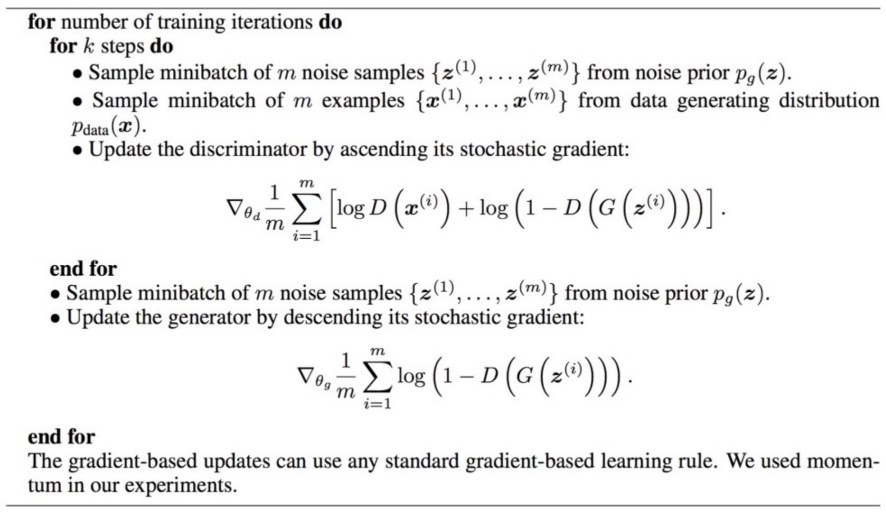
Figure 6: Binary Cross Entropy Loss function

The function is calculating both log components in the objective function and y is the predicted value. The BCELoss function is generally used to measure how far away is the prediction for each of the classes from the true value. In order to obtain the final loss for the models I needed to define the real label as value 1 and the fake label as value 0. These labels will be used to output the losses of G and D by averaging the errors.

The optimisers. In order to obtain better results and reduce the losses of the networks, optimisers are being used to change the network's attributes (weights initialisation, learning rate). I have analysed three different optimisers (Stochastic Gradient Descent (SGD), RMSProp and Adam) and discovered that Adam works very well in practice and it performs better than the other optimisers. The DCGAN paper is also using the Adam optimisers with tuned hyperparameters. The paper suggested that the learning should be at 0.0002 and the momentum term Beta1 at 0.5 to avoid training oscillation and instability.

5.1.4 The Training Loop

This section is one of the most important aspects of the project as the networks need to be trained in order to generate the desired icons. The training process is divided into two principal parts – training the D network and training the G network. The algorithm that I followed for the implementation (Figure 7) is from GANs paper by Goodfellow et al [4], but with some changes to make the models stable.



```
for number of training iterations do
  for $k$ steps do
    • Sample minibatch of $m$ noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from noise prior $p_g(z)$.
    • Sample minibatch of $m$ examples $\{x^{(1)}, \ldots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
    • Update the discriminator by ascending its stochastic gradient:
        $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right) \right].$
  end for
  • Sample minibatch of $m$ noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from noise prior $p_g(z)$.
  • Update the generator by descending its stochastic gradient:
        $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right).$
end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.
```

Figure 7: Training Loop Algorithm from GANs paper

To train the G and D models, the loop involves iterating over some steps. To get the desired output, it needs to get the results multiple times. A cycle through the loop is not an epoch, but a single update which contains specific batch updates to the networks. An epoch means that the entire dataset is passed forward and backward through the network only once. I will explain how the algorithm is being implemented in the System Implementation section and I will give more details about how the generator and the discriminator are trained.

5.2 System Implementation

Throughout the project's implementation phases, the initial assumption of the system held. The only aspect that I introduced while implementing the system's code and had significant impact on the project was the conditionality. I firstly tested the DCGAN architecture without the conditionality and the results were as expected, randomly generated with no control over the output. Therefore, I decided to take control over the output and improve the GANs performance by using the conditional GAN architecture.

The system was implemented using PyTorch, a Python-based library which provides concise and readable code, flexibility and speed. PyTorch is usually used for developing and training neural networks as it allows flexibility in building very complex architectures. The section will be divided in multiple subsections which will include – libraries, dataset, image plotting and saving functions, hyperparameters, weights initialisation, generator class, discriminator class, cDCGAN class.

5.2.1 Libraries

In this subsection I will list the important libraries used and what is their purpose:

1. Random – is a library that is implementing pseudo-random number generators which was used to set the random seed for reproducibility.
2. Torch – is a library which supports basic routines for indexing, slicing, resizing and transposing. I also used the nn package which includes forward() and backward() methods that allow to feedforward and backpropagate.
3. Torchvision – is a package that consists of model architectures, datasets and image transformations.
4. Numpy – is a library that supports large set of numerical datatypes used to construct arrays.
5. Matplotlib – is a library that is used to plot the training or the generated images.

5.2.2 Dataset

The dataset used to train the system is the Icons-50 dataset [14] which consists of 10.000 images with 50 different classes. The initial size for the icons was 120x120 pixels, but before training the images are cropped to 64x64 pixels in order to train the models better. An example of the Icons-50 dataset can be seen in Figure 8.

In order to get the class labels, I created a function called getIndex() which returns a dictionary of classes. The value of each class is a dictionary containing the indices and the datapoints. The input taken is the dataset which should be ImageFolder class – a generic data loader where the images are structured as follows:

```
/Icons-50/Icons-50/airplane/apple_0_airplane.png
/Icons-50/Icons-50/boat/apple_0_canoe.png
```

Figure 8: Icons50 Dataset. An example of images used to train our networks.

The output is a nested dictionary with the class name as the key (the folder's name) and a dictionary containing the indices representing the images in order and the length of the class as values. An example of the dictionary format would be as follows:

```
dictionary = { 'airplane':{
               'indices': [1,2,3,4,5],
               'length': 5
               },
               'book':{
               'indices': [6,7,8],
               'length': 3
               },
               'writing_utensils':{
               'indices': [9997,9998,9999,10000],
               'length': 4
             }}
```

Listing 1: Nested dictionary with the class name and dictionary containing indices and lenght of the class

To check if the algorithm is getting all the classes and the datapoints I am printing the class name and how many entries has got as follows:

```
Class airplane has 76 entries.
Class arrow_directions has 392 entries.
Class writing_utensil has 103 entries.
```

5.2.3 Input Parameters and Hyperparameters

In this section I listed all the input parameters and hyperparameters used to improve the model and what is their purpose. Getting the right hyperparameters is crucial as no model performs well without and tuning GANs takes quite a lot of time.

1. **dataroot** – is the path to the root folder for the Icons50 dataset.
2. **batch_size** – is a hyperparameter that defines the number of training examples that are propagated through the network. In this project I will use the batch size 128 as the DCGAN paper recommends.
3. **icon_size** – is the spatial size of the training images which will all be resized to 64x64 pixels before training using a transformer.
4. **nc** – is the number of channels. In this project the images are RGB which means it has 3 channels.
5. **nz** – is the size of the random variable z which it should be 100.
6. **lr** – is a hyperparameter that defines the learning rate for the Adam optimisers, and I set it to 0.0002 as I mentioned in System Design section, Loss function and optimisers subsection.
7. **beta1** – is a hyperparameter that defines the momentum term beta1 for the Adam optimiser and I set it to 0.5.
8. **ngpu** – is the number of GPUs available and I used only one.
9. **workers** – is the number of workers for the dataloader. It is used to preprocess the batches of data, so that the next batch is prepared to use when the current batch is done.
10. **device** – is the device that I want to run on – if GPU is available use it and if not use CPU.

Before doing the training, I also set the keyword arguments (kwargs) as follows:

```
kwargs = {
    'dataloader': dataloader,
    'classes': dataset.classes, # include all the classes
    'save_dir':'C:/Users/liviu/Downloads/icons50/Icons-50/results',
    'num_epochs': 500, #we trained it for 500 epochs
    'criterion': nn.BCELoss(), # Initialize BCELoss function
    'netD': netD,
    'netG': netG,
    'optimizerD': optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999)),
    'optimizerG': optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999)),
```

```python
    'device': device,
}
```

Listing 2: Keyword arguments used in our project before training.

5.2.4 Image Plotting

I defined a function called plotIcons() that plots a sample of images from the dataloader along with the class labels. It takes as inputs the dataloader to load the dataset, the classes which are array type objects containing the class labels and the number of images that should not exceed the batch size. An example of how the function works can be seen in Figure 8.

5.2.5 Weights Initialisation

Weights initialisation has the purpose of preventing the layer activation outputs from disappearing during the forward pass process. The DCGAN paper suggests randomly weights initialisation from a Normal distribution with mean = 0 and standard deviation = 0.02. I defined a function called weights_init() and I applied it to both G and D models. It takes the initialised model m as input and on the next step is reinitialising all the Convolutions and Batch Normalisation layers.

```python
def weights_init(m):

    classname = m.__class__.__name__

    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)

    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

Listing 3: The Weights Initialisation function.

5.2.6 The Networks Classes

The Generator Class. The implementation of the G model is divided into two functions – init() and forward(). In the initialiser I declared more input parameters and I also included a Sequential class which has the purpose of creating the model. Also, the forward function represents the concatenation of the label embedding and the random noise to produce the input.

The Initialiser function. The input parameters declared are: n_classes – which is the number of classes found in the dataset, ngpu – which is the number of GPUs that are going to be used in the training process and ngf – representing the size of feature maps that are propagated through the generator network. The Sequential class starts by taking in a 100x1 random variable z plus the number of classes and maps it into the G(z) output which is 3x64x64 and the class label. The network goes as follows:

`100x1` $\rightarrow$ `1024x4x4` $\rightarrow$ `512x8x8` $\rightarrow$ `256x16x16` $\rightarrow$ `128x32x32` $\rightarrow$ `3x64x64`

Firstly, the input is projected from 100x1 noise plus the number of classes into 1024x4x4. This is known as projecting and shaping as seen in Figure 3.

```
# the input plus the n_classes are going into a convolution
nn.ConvTranspose2d(nz + n_classes, ngf * 8, 4, 1, 0, bias=False),
nn.BatchNorm2d(ngf * 8),
nn.ReLU(True),

# the state size is 1024x4x4
nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
nn.BatchNorm2d(ngf * 4),
nn.ReLU(True),
```

In the next layer, the Generator is using a transposed convolutional layer (upsampling) along with Batch Normalisation and ReLU activation to produce an image from the random noise. The dimensionality of the output space is 512x8x8 as seen in the code below.

```
# the state size is 512x8x8
nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
nn.BatchNorm2d(ngf * 2),
nn.ReLU(True),
```

Therefore, the Generator is using the previous layers to produce an image. It is going through transposed convolutional layer along with Batch Normalisation and ReLU activation until it reaches the dimensionality of the output space of 3x64x64 as seen below.

```
# the state size is 256x16x16
nn.ConvTranspose2d(ngf * 2, ngf, 4, 2, 1, bias=False),
nn.BatchNorm2d(ngf),
nn.ReLU(True),

# 128x32x32
nn.ConvTranspose2d(ngf, nc, 4, 2, 1, bias=False),
nn.Tanh()
# 3x64x64
```

The Discriminator Class. The implementation of the D model is similar to the G model. The order of the layers has been flipped and the ConvTranspose2d is replaced with a Conv2d layer. It is also divided into two separate functions – init() and forward().

In the initialiser the same input parameters are declared and the Sequential class has been divided into convolution_layers and linear_layers.

The forward function is running the input through the convolutional layers and it flattens the output from the main layer. Later, it creates the label layer and concatenates the flattened output to the label layer. Therefore, the last step is to run the flattened output and the merged layer through the linear layers.

5.2.7 The CDCGAN Class

The CDGCAN class has the most complex implementation as it is divided in multiple functions with different purposes:

1. _checkDirectory() - has the purpose of checking whether the save directory exists as I need to make sure that the outputs of the training can be saved. If it doesn't exist, it is automatically creating one.

2. _init_() - in this function more parameters are being initialised. The data parameters (dataloader, classes and n_classes) are initialised along with the saving locations for the training checkpoints and generated images. The model parameters (num_epochs, start_epoch, criterion which is the loss function, real_label 1 and fake_label 0) are also initialised followed by the network's initialisation (netD, netG, optimiserD, optimiserG) and the device that we want to use (GPU/CPU). Finally, I also create a fixed noise to visualise the progression of the generator.

3. genFake() - represents the function that generates a batch of fake images using the current generator weights.

4. train() - this function represents the training loop which I am going to explain in detail. It starts by checking if the number of epochs is set for training. If the epochs are not set, it is exiting the training loop. The training loop is divided into two sections – update the D network and update the G network.

Updating the D network is maximising the log(D(x)) + log(1 - D(G(z))) which means maximising the probability of correctly classifying the image given as real or fake. This has been done by dividing the process into two separate parts.

The first part is to construct a batch of real images from the training data. Then, I forward pass it through D and calculate the loss function log(D(x)) on all real batch. Therefore, I calculate the gradients for D in a backward pass.

The second part is to construct a batch of fake images with the current generator. I accomplish this by generating two batches: one of latent vectors and one of fake labels. Afterward, I generate a batch of fake images with the G model. When I have the batch of fake images, I update the ground truth labels to fake, and classify the fake images with D model. At this point, I can calculate the loss function log(1 - D(G(z))) and the gradients for this batch and accumulate the gradients with a backward pass. The gradients for the real and fake batches are added, and the D model is updated by calling the optimiser's step.

Updating the G network is maximising the log(D(G(z))) which means to generate reasonable-looking fake images. This can be done by setting the fake labels as real for the generator cost and by performing another forward pass of all fake batch through D. Therefore, the loss function is calculated based on the forward pass of all fake batches that was done earlier. I also calculate the gradients for G model in a backward pass. After these steps are done, the G model is updated by calling the optimiser's step as it was done for the D model.

To check the loss functions and how well are the models performing I included some statistics after each epoch is done. I also check how the generator is progressing by saving a batch of fake images for all the classes every 50 epochs.

5. save() - this function has the purpose of saving the model at the end of the training. I also save the checkpoints including the G and D losses, epoch, the networks state, and the optimisers.

6. load() - this function is used to load a model's checkpoint along with the losses and optimisers.

6 Experimental Results

To check how good the implementation is, I performed some experiments and acquired enough data to analyse the networks. In this section, I am going to present the quality evaluation of the models and the results obtained from training.

6.1 Training

For the training I implemented the code in a notebook on Google Colab [16] as they offer free cloud service and free GPU's. Due to the lack of time, the CDCGAN model was trained just for 500 epochs with a Nvidia Tesla K8 GPU and it took around 10 minutes per epoch with the given environment (it lasted around four full days). Specifically, every 50 epochs I generated an image from every class I have in the dataset (fifty classes) to check how well the model is performing. I also trained a normal GAN on the same dataset to compare which one is performing better.

6.2 Quality Evaluation

A stable model will have the discriminator loss between 0.5 and 0.8. The generator loss would be higher than the discriminator and may hover around 1.0, 2.0 or sometimes higher. Also, the accuracy of both networks should be around 70% to 80%. The goal of the training is to reach the Nash equilibrium which happens when one of the networks will not change its action regardless of what the other does.

I analysed the losses of both models G and D and I have seen that the behaviour of the networks started off erratic as the values were going up and down. In Figure 10, it can be observed that the loss of the generator is higher than the discriminator as I mentioned above. However, the values still did not become stable as it was still going up and down at the end and

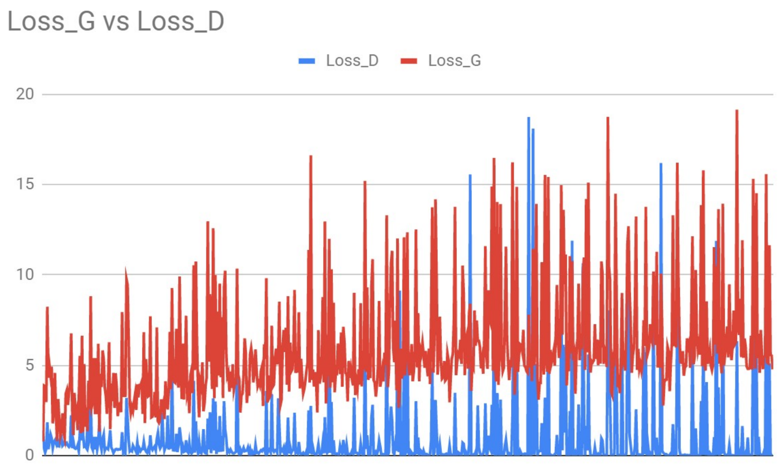


Figure 9: The Losses of Discriminator and Generator

the models did not converge. The discriminator loss is mostly closer to value 0, but sometimes it is going close to 1. The generator's loss is between values 4 and 6. If I train the model for longer it is very probable to behave the same and it is unlikely to reach convergence.

```
[445/500][0/79] Loss_D:0.11  Loss_G:5.07  D(x):0.99  D(G(z)):0.62
[445/500][50/79] Loss_D:0.05  Loss_G:4.62  D(x):0.99  D(G(z)):1.81
[446/500][0/79] Loss_D:0.01  Loss_G:5.43  D(x):1.0  D(G(z)):1.14
[446/500][50/79] Loss_D:0.15  Loss_G:4.54  D(x):0.9  D(G(z)):0.28
[447/500][0/79] Loss_D:0.16  Loss_G:5.86  D(x):1.0  D(G(z)):36.92
[447/500][50/79] Loss_D:0.01  Loss_G:5.61  D(x):1.0  D(G(z)):1.02
[448/500][0/79] Loss_D:0.13  Loss_G:4.56  D(x):0.94  D(G(z)):0.07
[448/500][50/79] Loss_D:0.04  Loss_G:4.92  D(x):0.98  D(G(z)):1.78
```

To improve the architecture and achieve model's convergence I need to do more experiments in the future (i.e. changing the loss function, try different hyperparameters). I will discuss more about it in the Further Work Section.

6.3 Results

Finally, the results generated by the model can be reviewed. As I expected, they are not reasonable looking as I wished when I started planning the project, but they are promising. Due to the time constraint I haven't had the chance to experiment with different model architectures or train for longer. A good number of epochs would be over 1000, but I still got an idea of how it performs with the current results. Also, the training dataset is not large enough as it can be seen in Sage et. Al's LLD dataset (600.000+ logos) or CelebA dataset (200.000+ faces). The images generated by FaceGAN are also high resolution and extremely detailed. I could not accomplish that as the dataset I used contains just 10.000 images.

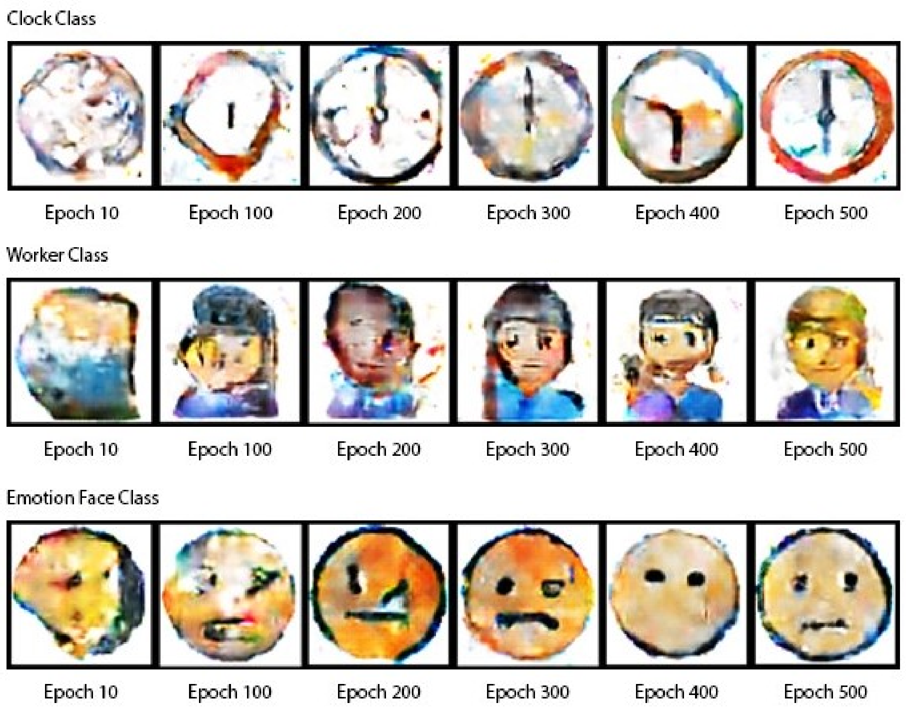


Figure 10: A few generated icons using conditional DCGANs.

In Figure 10, generated icons at different training epochs can be observed. I only included just a few classes where I have seen a better generation and progress. In some classes the shape is not recognisable, for example in the airplane class I couldn't see any airplanes. As it can be seen in the figures below, I included generated images every 100 epochs and there is a significant difference between them. A decent icon is obtained at around 200-300 epochs, but until then the results are relatively poor. For example, at epoch 10 we can just try to guess which class it belongs to as it is not visually recognisable. The generated images are mostly unclear, and they are dominated by round and square shapes. I can also see an absence of details in the generated images as it lacks high frequencies and I can observe that the quality generated can vary across the training. Consequently, the classification in some cases was not successfully completed as I found images belonging to other classes and the shapes of the icons are not clear enough yet in order to be used in the design process.

7 General Discussion and Future Work

Icon generation conditioned on class is a very complex task that requires a significant amount of research and experiments. The results of this project are promising, and it is worth making improvements and include further extensions. I plan to expand this work in a couple of directions. The first thing I would experiment with is the network architecture which can be changed to an Auxiliary Classifier GAN instead of conditional GAN. It is a much complex architecture, but it can converge and do the classification better and potentially generate icons that can be used in the designing process. I can also explore different hyperparameters and cost functions (such as Wasserstein Loss and Mean Square Error) in order to achieve better optimisation.

A really important aspect that I did not include in the project is quantitative evaluation such as Inception Score (IS), Classification Performance or Image Quality Measures (SSIM) and so on. The quantitative evaluation is used to calculate specific numerical scores to measure the quality of the generated images. The most intuitive way of evaluating GANs in general is by doing it manually [15] and in this case, it was the best solution as I did not have too many classes and I only trained our network for 500 epochs. Manual inspection is the simplest method of the network evaluation, but not the most effective though as it can be very subjective and it requires knowledge.

This project can be extended to logos which are much more complex than icons as it consists of text and different shapes and colours. I could also include a User Interface which the user can use to generate their logo. This can be accomplished by inserting the text wished and by choosing the colours, shape and even the font of the text. This is a very complex extension that requires a significant amount of work and research, but if it can be accomplished it can replace the designers of today as it will generate a new logo based on the user's specifications.

8 Evaluation

This project has been a major challenge as it was something completely new which I have never worked with. A significant amount of research has been done to build and implement the system as I had no knowledge in this area. I encountered the problem with icon designing while I was creating a new set of icons for a website and I thought I could use Artificial Intelligence to generate the icons desired or even skip some iterations in the designing process. Sometimes there is an imagination blockage and it is hard to build new icons or logos. I started by reading articles about how to use Artificial Intelligence in the graphic designing area and I discovered Generative Adversarial Networks. Everything that has been written in this dissertation was completely new for me and it was really hard and challenging to understand and apply all these concepts in a limited amount of time. Also, I had the chance to learn how to use PyTorch and Jupyter Notebook or Google Colab notebooks. Considering I had no experience in machine learning and generative models I am completely satisfied with the outcome of the project. I would love to continue the work on this project by improving the system and by acquiring more knowledge in this area as it is fascinating what it can be accomplished with these generative models.

9 Conclusion

In this project, I demonstrated that icon generation using conditional deep convolutional GANs (cDCGANs) has a significant potential in the designing process, if it is well-designed and if the right architectures are being chosen. In order to generate perfect icons that can be used by the designers it can be challenging as a lot of architecture experiments and further research needs to be done. Providing high quality and quantity input data, these GANs have been proved to be extremely powerful and can, therefore, replace the designers of today, but not the designers of tomorrow. Finally, the model was able to generate icons based on some conditions. However, the results were not very clear and in some cases the classification was not successfully completed, but I think this area of icon or logo generation needs to be explored more as it can change the design industry.

10 References

[1] Wikipedia. *Icon Design.* https://en.wikipedia.org/wiki/Icon_design

[2] Medium. *"Yes, AI will replace designers".* https://medium.com/microsoft-design/yes-ai-will-replace-designers-9d90c6e34502

[3] Medium. *"GANs from Scratch 1: A deep introduction. With code in Pytorch and TensorFlow".* https://medium.com/ai-society/gans-from-scratch-1-a-deep-introduction-with-code-in-pytorch-and-tensorflow-cb03cdcdba0f

[4] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. *"Generative adversarial nets".* https://arxiv.org/pdf/1406.2661.pdf

[5] A. Radford, L. Metz, and S. Chintala. *"Unsupervised representation learning with deep convolutional generative adversarial networks".* https://arxiv.org/pdf/1511.06434.pdf

[6] M. Mirza and S. Osindero. *"Conditional Generative Adversarial Nets".* https://arxiv.org/pdf/1411.1784.pdf

[7] T. Karras, T. Aila, S. Laine, and J. Lehtinen. *"Progressive growing of gans for improved quality, stability, and variation".* https://arxiv.org/pdf/1710.10196.pdf

[8] G. Antipov, M. Baccouche, and J. Dugelay. *"Face aging with conditional generative adversarial networks".* https://arxiv.org/pdf/1702.01983.pdf

[9] Y. Taigman, A. Polyak, and L. Wolf. *"Unsupervised cross-domain image generation".* https://arxiv.org/pdf/1611.02200.pdf

[10] Y. Jin, J. Zhang, M. Li, Y. Tian, H. Zhu, and Z. Fang. *"Towards the automatic anime characters creation with generative adversarial networks".* https://arxiv.org/pdf/1708.05509.pdf

[11] A. Sage, E. Agustsson, R. Timofte, and L. Van Gool. *"Logo synthesis and manipulation with clustered generative adversarial networks".* https://arxiv.org/pdf/1712.04407.pdf

[12] A. Mino, and G. Spanakis. *"LoGAN: Generating Logos with a Generative Adversarial Neural Network Conditioned on color".* https://arxiv.org/pdf/1810.10395.pdf

[13] A. Odena, C. Olah, and J. Shlens. *"Conditional Image synthesis with Auxiliary Classifier GANs".* https://arxiv.org/pdf/1610.09585.pdf

[14] D. Hendrycks. *"Icons-50. Image dataset of Icons".* https://www.kaggle.com/danhendrycks/icons50

[15] A. Borji. *"Pros and Cons of GAN Evaluation Measures".* https://arxiv.org/pdf/1802.03446.pdf