Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks

E4040.2020Fall.DCGAN.report

Christopher Reekie, CJR2198
Columbia University

Abstract

The objective of this project is to construct and train DCGAN architectures using a variety of datasets and demonstrate the quality of learned representations using the trained discriminator as a feature extractor in supervised classification tasks. The primary challenge encountered in aim of these goals was the instability of training GAN architectures. Through iterative experimentation with different architectures and refinement of the training routine, the results of the original paper were qualitatively achieved and on par quantitative representation learning results were successfully demonstrated. In addition to successful replication of the original results, the project investigates and demonstrates how to scale DCGAN architectures for image resolutions greater than 64x64. Supplementing this work, the project successfully demonstrates that a DCGAN can be used to improve classification accuracy by using the DCGAN to generate additional data when labeled data is scarce.

1. Introduction

The focus of this report is the novel class of Deep Generative Adversarial Convolutional Neural Networks (coined DCGANs) proposed by (Radford, Metz, Chintala et al. [1]) in the paper "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks"

The original paper demonstrated that the DCGAN architecture was capable of stably learning representations of the underlying training data in unsupervised learning tasks and demonstrated the quality of the unsupervised learning algorithm by using the learned representations of the DCGAN model in supervised learning tasks.

Within this report, I perform an in-depth review of the original paper, reproduce the original network architecture and validate the findings against the original target datasets. I then expand the original architecture to produce higher resolution images, and investigate whether DCGANs can be used to generate labelled data for classification tasks.

2. Summary of the Original Paper

2.1 Methodology of the Original Paper

Generative Adversarial Deep Neural Networks (GANs) are a class of neural network first proposed by Goodfellow et al.[4]. GANs consist of two adversarial networks, a generator and a discriminator, which are trained together with opposing goals. The generator learns a mapping from a latent space to the data distribution of the training set and the discriminator learns to distinguish between generated data points and data points from the training set. The network converges when the discriminator cannot determine the difference between a datapoint generated by the generator network and a true training sample.

The authors of the original paper modify this general class of network to the application of generating images.

The DCGAN paper makes a key architectural contribution to the area of Deep Generative Adversarial Networks as applied to image representation, and demonstrate uses of the architecture:

DCGAN Class Architecture

The authors propose a set of architectural constraints and guidelines for the creation of deep convolutional generative adversarial neural networks and coin this class of model architecture DCGANs.

The key purpose of the architectural constraints is to facilitate stability in training of the model in a variety of unsupervised image learning tasks.

The framework, as proposed, provides guidelines for the architecture of both the discriminator and the generator in the generative adversarial model.

The guidelines for the architecture of the **generator** are as follows:

- The model base starts with a fixed single dimension latent space with a fixed number of variables. The latent layer is then projected and reshaped to a higher dimensional dense representation.
- The remaining layers of the generative model are fully connected blocks of fractionally-strided convolutional layers, with batch normalization and Relu activation. There are no further fully connected layers.
- No pooling layers are included in the generator. Batch normalization is used at every layer except the output layer.
- The fractionally-strided layers repeat, with reducing numbers of filters at each layer by a factor of 2. The final output layer uses fractionally-strided convolution with Tanh activation to produce an image.

The guidelines for the architecture of the **discriminator** are as follows:

- Leaky Relu activation with alpha factor of 0.2 is used for all layers of the discriminator.
- Strided convolutions are used instead of pooling.
- Batch normalization is used in the discriminator except for the first layer.

The exact implementation used in the original paper is not specified, a reference generator architecture is provided and it is assumed that the discriminator architecture mirrors this architecture with a single dense unit as the discriminator head for classification in place of latent variable embedding.

Vector Arithmetic to Generate Face Samples

Treating the input vectors of latent variables to the generator as image embeddings, the original paper demonstrates similar arithmetic properties using the learned mapping of embeddings to that demonstrated for word embeddings by Mikolov et al. in Word2Vec[2].
The original paper demonstrated that by taking an average of latent variables which depict a man with glasses, then subtracting an average of latent variables of a man without glasses then adding an average of latent variables depicting a neutral woman produced images of a woman wearing glasses. This is demonstrated in the Figure 1 from the original paper below:

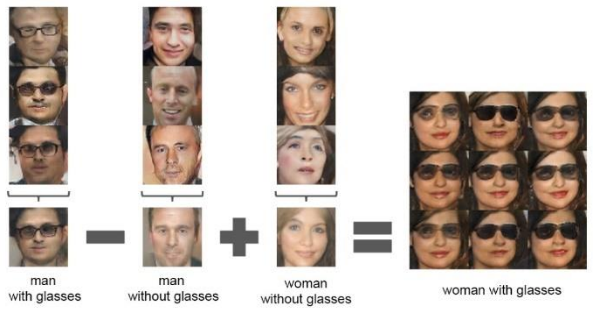


Figure 2 Vector Arithmetic

Using the Discriminator as A Feature Extractor for Supervised Learning

The original paper demonstrated that the learned feature maps in the discriminator could be used for supervised classification tasks. The authors trained a DCGAN on ImageNet-1K, then used the trained discriminator as a feature extractor on the CIFAR-10 dataset.

Features were created by extracting the convolutional features from all convolutional layers of the discriminator, applying max-pooling to each layer's representation to produce a 4x4 spatial grid.

The extracted features were then flattened and concatenated to form a feature vector. The authors then trained a regularized linear L2-SVM using the features and demonstrated competitive performance on CIFAR-10.

2.2 Key Results of the Original Paper

The paper visually demonstrates the quality of facial and bedroom images that trained DCGAN generators can produce. A sample from the original paper has been included in Figures 1, 2 & 4.

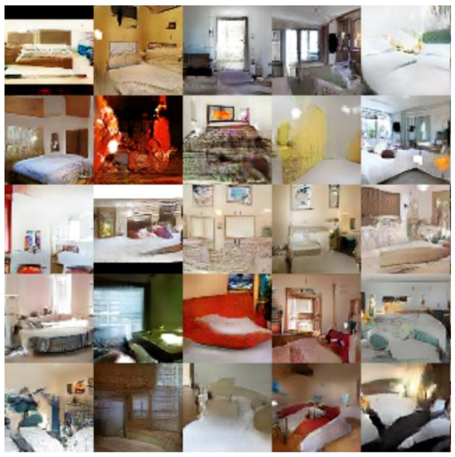


Figure 4 LSUN Bedrooms

The authors were able to demonstrate the vector arithmetic properties of the latent variable space for generated images of faces is demonstrated in Figure 1.

Additionally, the original paper evaluates the quality of the representations learned by the DCGAN by using a discriminator trained on ImageNet1k as a feature extractor for CIFAR-10 images for supervised learning with an L-2 regularized Support Vector Machine classifier. The results on the CIFAR test set, compared to other methods are shown in Figure 3 below:

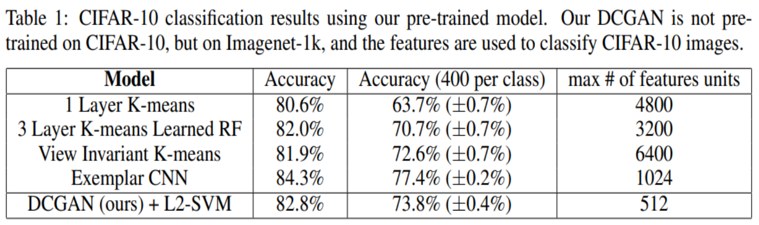


Table 1: CIFAR-10 classification results using our pre-trained model. Our DCGAN is not pre-trained on CIFAR-10, but on Imagenet-1k, and the features are used to classify CIFAR-10 images.

Model	Accuracy	Accuracy (400 per class)	max # of features units
1 Layer K-means	80.6%	63.7% (±0.7%)	4800
3 Layer K-means Learned RF	82.0%	70.7% (±0.7%)	3200
View Invariant K-means	81.9%	72.6% (±0.7%)	6400
Exemplar CNN	84.3%	77.4% (±0.2%)	1024
DCGAN (ours) + L2-SVM	82.8%	73.8% (±0.4%)	512

Figure 1 Original Paper CIFAR Results

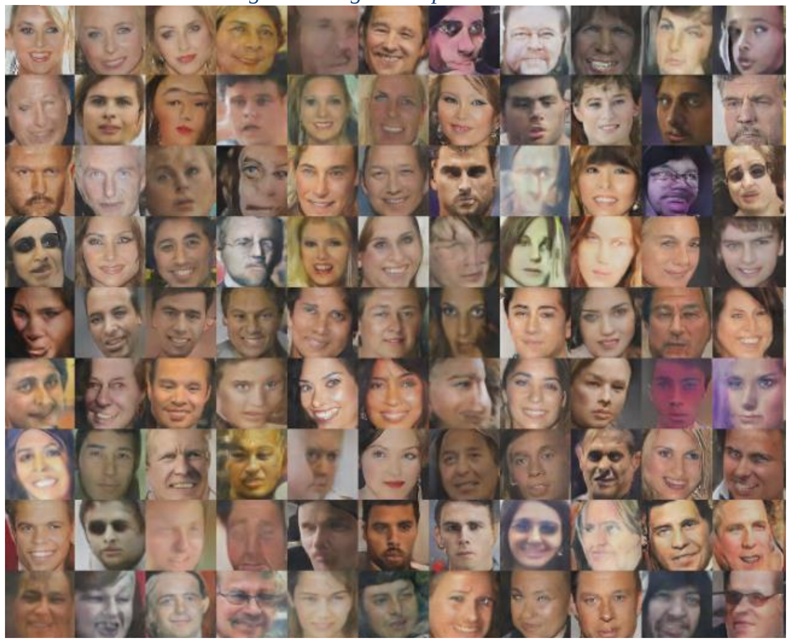


Figure 3 Generated Faces

3. Methodology (of the Students' Project)

The project has been broken down into two sections. The first was a focus on the reproduction of the results of the original paper, by iteratively investigating and evaluating different architectures adhering to the constraints of DCGANs set out in the paper.

The second section builds upon the results of the original paper by exploring deviations to the original architecture to increase image resolution and explores different applications of DCGANs not included in the original paper.

The original paper includes supplementary materials including visualization of filters and multiple examples of manipulation of the latent space. Due to the time-consuming nature of the process of repeatedly sampling the generator to determine and collect similar generated representations, I have focused on the vector arithmetic using facial images to demonstrate this result from the paper.

A summary of these objectives is provided in the section below.

Reproduction of DCGAN Paper Results

1. LSUN and Facial Generator Results

Different DCGAN architectures were iteratively investigated to reproduce the same quality of 64x64 RGB images demonstrated in the original paper.

2. Arithmetic Properties of Latent Space

A demonstration of the arithmetic properties of the latent variables mapping of the generator in producing facial images with predictable properties using the trained DCGAN models is included.

3. Evaluating DCGANs as Feature Extractors

A DCGAN architecture was trained on the ImageNet-1k dataset to reproduce 32x32 RGB images. The trained discriminator was then utilized as a feature extractor and applied to the CIFAR-10 dataset. The extracted features where then used in combination with an L2-SVM to classify the images.

Improvements to The DCGAN Architecture

4. Scaling the DCGAN Architecture to Higher Resolutions

I investigate methods and explore variations on the DCGANs architecture to produce higher resolution images than that proposed in the original paper.

Novel Applications of DCGANs

5. DCGAN for Labeled Data Generation

I demonstrate that by using a DCGAN, fewer labeled training examples can be used to achieve competitive classification performance on the MNIST dataset.

3.1. Objectives and Technical Challenges

The primary objective of the project is to successfully reproduce a DCGAN architecture and train it against a series of datasets to produce plausible output.

With a successful architecture and training routine, the models are then trained on a variety of datasets to demonstrate the generation ability of the generator and the feature extraction capabilities of the trained discriminator.

The main technical challenge in designing and training a DCGAN architecture is the inherent instability of these architectures in training. A variety of different architectures and initializations within DCGAN constraints was explored to overcome "mode collapse" and other issues which prevent the generator from training effectively to produce a range of realistic output. It was discovered that these issues increase exponentially for deeper architectures, higher resolutions and an architecture that works well on one dataset may not work well on another.

A key practical challenge in training DCGANs was the training time. In order to achieve stability in training the models needed to be trained with smaller minibatches (128 samples) and at small learning rates (Adam with learning rate = 0.0002, Momentum = 0.5). This leads to long training times on relatively small samples (64x64), to tackle this challenge the training routine was adapted to distribute training across two GPUs.

3.2. Problem Formulation and Design Description

Deep Convolutional Generative Adversarial Networks are characterized by the application of the generative adversarial approach to deep convolutional networks.

In this section I define Generative Adversarial Networks, their unique training process and the distinction of the DCGAN class of model to generic GAN.

3.2.1 Mathematical Formulation of Generative Adversarial Network Training

A mathematical formulation of the goals of the competing networks in a generative adversarial neural network are described in the following section.

The generator produces a generated sample, x_{fake} from the random latent variables:

$$G\sim Generator \quad z\sim Randomly\ Generated\ Latent\ Variables$$
$$G(z) = Generated\ Sample = x_{fake}$$

The discriminator determines if the input sample is real, or if it is fake and assigns a probability accordingly:

$$D\sim Discriminator \quad x\sim Input\ Sample$$
$$D(x) = P(y = Real \mid x) = 1 - P(y = Fake \mid x)$$

The goal of the generator G is to maximize the probability that a generated sample will be classified by the discriminator as a real image:

$$\boldsymbol{Generator\ Goal}:$$
$$\max D(G(z)) = \max P\big(y = Real \mid x_{fake}\big)$$

The goal of the discriminator D is twofold; maximize the probability that a fake sample will be labeled as fake and minimize the probability that a real sample will be labeled as fake:

$$\boldsymbol{Discriminator\ Goal}:$$
$$\min D(G(z)) = \min P\big(y = Real \mid x_{fake}\big)$$
$$\max D(x_{real}) = \max P(y = Real \mid x_{real})$$

A flowchart of the competing training routine is shown in the figure below:

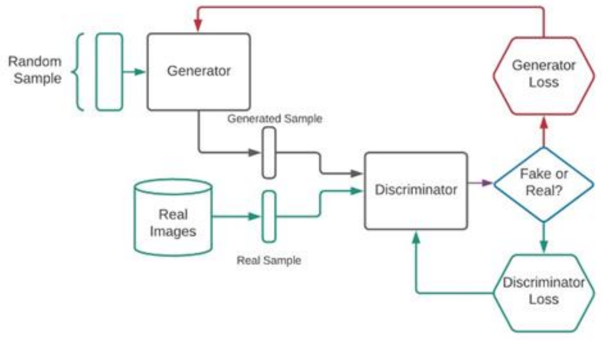


Figure 5 Training Routine Flow Chart

3.2.2 Generative Adversarial Learning

The generator's goal is to fool the discriminator into incorrectly classifying the generated sample as having come from the training data and not been generated. In this learning process, the generator learns exclusively from the feedback it receives from the discriminator in classifying the generated out. Notably the generator is never exposed to the underlying training data, only the feedback from the discriminator.

Backpropagation of the generator's gradients and loss therefore includes not only the generator's architecture, but also the discriminator's architecture, as the generator adjusts its weights to move the discriminator toward misclassifying its samples.

The discriminator's goal is to correctly recognize and classify images as originating from the training data or having been generated. The training process iterates by providing the discriminator with batches of real images and generated images, backpropagating the loss calculated against images from each class based on classification output.

In contrast to the generator, the discriminator is exposed to the underlying training data during the training process.

3.2.3 DCGAN Generator Learning

The generator in a DCGAN architecture uniquely consists of only convolutional layers beyond the projection of the random latent variables. Therefore, the architecture is primarily learning kernel feature maps to reproduce images from the latent space. The capacity of the model is therefore limited by the number of deconvolution blocks and the size and number of the learned filters.

Similarly, the discriminator's only fully connected layer is to the output node, and so is learning to extract features from the image that can be used to determine if it is real or not.

Since the discriminator is learning to use convolutional feature maps with non-linearity, rather than fully connected layers, to determine if an image is real or if it is generated it is likely then that these feature maps may be robust enough to be used to extract features from images to be applied to supervised learning tasks.

4. Implementation

I describe how I implement the original DCGAN model, with a focus on the model architecture, initialization and optimizer for both the generator and the discriminator network.

4.1. Deep Learning Network

4.1.1. DCGAN Generator Architecture

The original DCGAN paper visualized a reference generator architecture shown below:

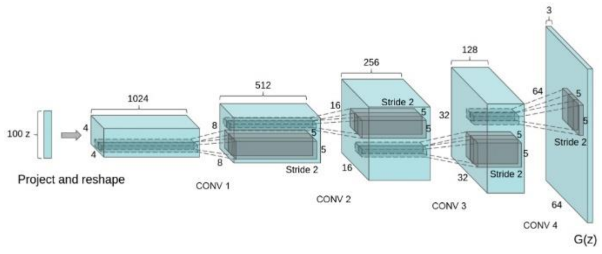


Figure 6 Original Paper Architecture

The model starts with a latent vector of randomly generated variables (z) generated from a uniform distribution on the range (-1,1). This vector is then projected and reshaped to a 4x4x1024 representation.

Repeating deconvolution blocks (fractionally strided convolutions) with batch normalization and ReLu activation are then applied to this representation. Increasing the output dimension by a factor of 2 at each block while decreasing the number of filters by a factor of 2.

The output layer of the generator omits batch normalization and uses a fractionally-strided convolutions with Tanh activation to produce the image representation with pixel values between -1 and 1.

Rather than using max-pooling and dense layers to up-sample images, DCGANs use fractionally strided convolutions. Using this approach, the model can learn its own up-sampling to increase representation resolution.

An illustration of the generator with deconvolution blocks and output layer is shown in figure 7.

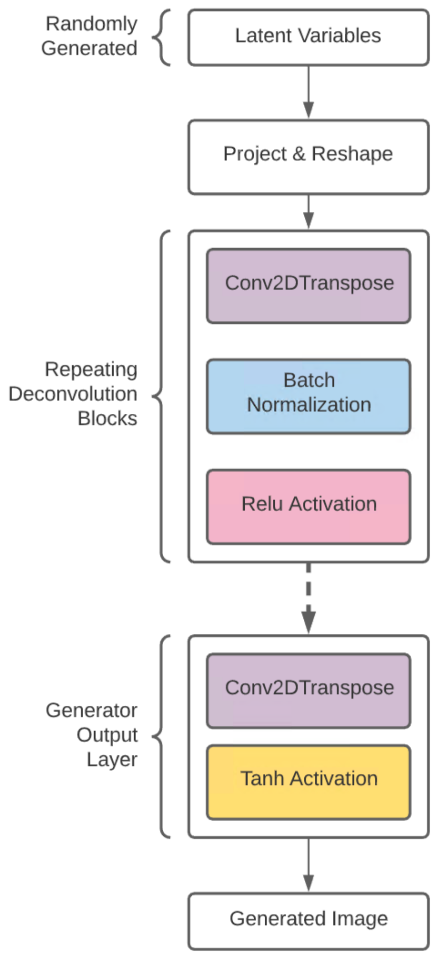


Figure 7 Generator Framework

4.1.2. DCGAN Discriminator Architecture

The authors of the original paper don’t provide the exact architecture of the discriminator used but give indications that the architecture is a mirror of the generator until the final convolutional block of dimension 8x8x512, where the project and reshape block is replaced with a flatten and final single dense output node. The convolutional blocks used in the discriminator use batch normalization and LeakyReLu activation. Batch normalization is omitted from the first convolutional layer of the discriminator.

Bias terms are omitted from all convolutional layers for which there is batch normalization applied since the batch normalization will remove this bias.

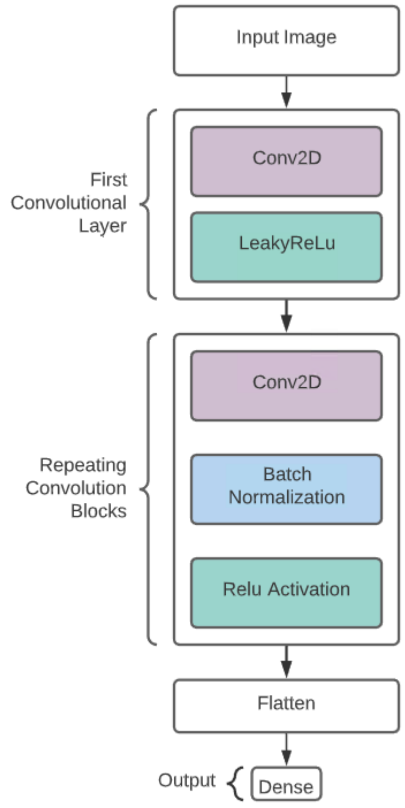


Figure 8 Discriminator Architecture

4.1.3. DCGAN Feature Extractor Architecture

The original paper utilized the trained discriminator model to create an image feature extractor which was then utilized on unseen CIFAR-10 data to train an L2-SVM model.

The feature extractor extracts the trained feature maps of the discriminator but makes manipulations to the architecture of the model to extract image features from the feature maps.

The original paper omits technical details of the pooling by layer in the feature extractor. For the purposes of this project, the feature extractor applies max pooling to each layer of the discriminators learned feature maps to produce image features. These features are then flattened and used as input features to train an L2-SVM.

Figure 9 illustrates the architecture of the feature extractor employed for this project.

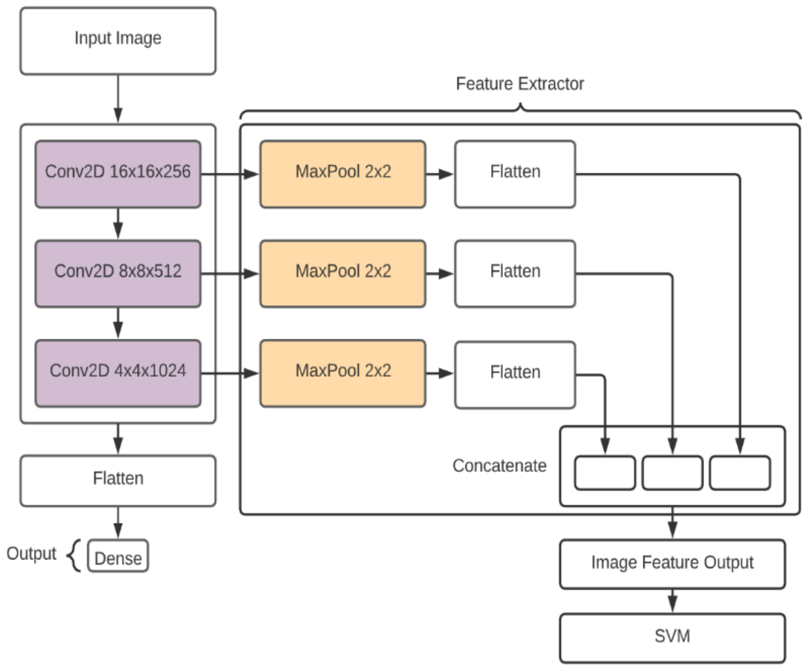


Figure 9 Feature Extractor Architecture

4.1.4. Training Routine Loop & Initialization

Due to the use of batch normalization, model training is less dependent upon initialization. For all experiments, weights were initialized with random normal distribution (mean = 0, standard dev = 0.02). Batch normalization gamma and beta parameters were initialized with a normal distribution with standard deviation = 0.02 and means 1 and 0 accordingly.

The original paper suggested that the Adam optimizer, with a low learning rate (0.0002), mini batch size of 128 samples and beta 1 term of 0.5 to prevent volatility in training be used. In order to replicate the results of the original paper, I used the same optimizer and parameters.

In training the DCGAN architecture, three gradient calculations were performed per batch. First, the gradient of the loss of the discriminator on the training data was calculated and discriminator weights were updated.

Secondly, a batch of samples were generated and the gradients of the loss of the discriminator against these samples was calculated and discriminator was updated.

Finally, a new batch of random samples was generated and gradients of the loss of the generator was calculated using the discriminator.

I found that there was less instability in training by breaking the training process down into three discrete steps rather than a single gradient calculation and update from a single batch of generated samples.

4.1.5. Distributed Training

Due to the large size of the datasets, very low recommended learning rate and the computational and memory overhead of training effectively two networks (discriminator and generator) at the same time. The training routine was distributed across two Titan RTX GPUs using a mirrored strategy.

Tensorflow's distributed mirrored training strategy stores an identical copy of the model on all GPUs used for training. Training batches are sampled from the dataset and split evenly among each GPU. GPUs independently calculate the per example loss. The user must then specify how the loss and gradient is aggregated and applied. For this project, the loss was averaged across the global batch size and model parameter gradients were calculated accordingly. The per GPU gradients are then shared across GPUs (using the Nvidia NCCL package), and identical updates are applied to all copies of the model, keeping model training synchronous. This process is illustrated in figure 10.

4.1.5. Data

All images used for training the DCGAN architecture were adjusted pixel values to the range of tanh (-1,1).

CelebA

The celeba dataset was used to as a training dataset for the purpose of generating images of faces. All images were center cropped to 70% of the original image and then resized to the target resolution (32x32, 64x64 & 128x128).

The original paper used a different dataset constructed by the authors and was not publicly shared.

LSUN

The LSUN bedroom dataset was utilized to reproduce the images of bedrooms as provided in the original paper. For 32x32 images, the bedroom images were center cropped to 50% and resized. For 64x64 the images were center cropped to 70% and resized.

ImageNet-1k

The default down sampled 32x32 Imagenet-1k images were used directly. Images were randomly horizontally flipped during the training process.

MNIST

The MNIST dataset was resized to 32x32 from 28x28 to train both the DCGAN and the classifier. Due to EfficientNet's input requirement of RGB images, the greyscale images were converted to 3 channel images to train the classifier.

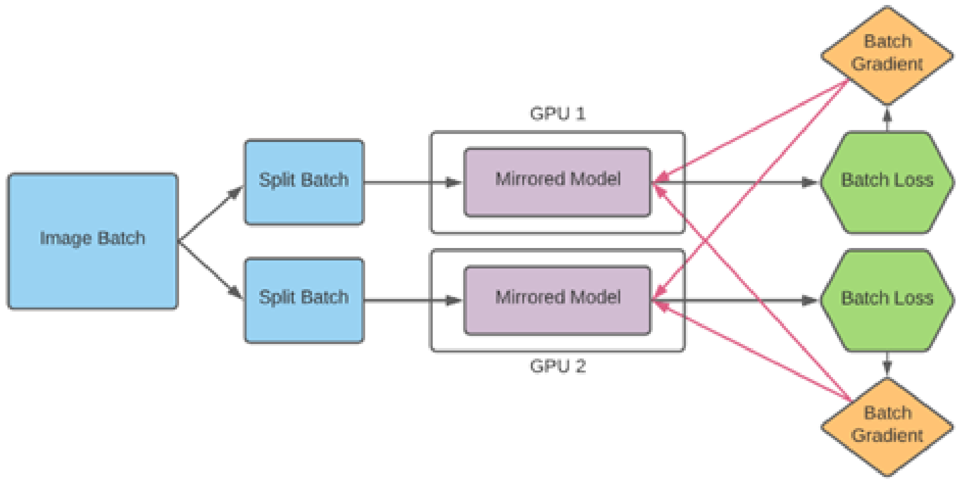


Figure 8 Distributed Training Framework

4.2. Software Design

The full DCGAN model architecture, including both discriminator and generator, is implemented as a derived class of the Keras model class with custom overrides for model training and initialization. This approach was taken to take advantage of Keras implementations for model fitting, callbacks and compatibility with Tensorflow datasets. The DCGAN class is defined in “DCGAN.py”.

Input arguments to the DCGAN class constructor define the architecture of both the generator and discriminator, these are provided as dictionaries. The user customizes the architecture of the generator and discriminator by specifying the number of filters, filter width and stride. The more layers are defined the greater the depth of the resulting network.

For the generator, the user can also customize the number of latent variables, the size of the reprojection (via dense units), and the distribution of the latent variables (normal or uniform). The activation specified will be used for all layers, the final layer with tanh activation will be added by default but the user must specify output channels (3 for RGB, 1 for Grayscale).

For the discriminator, the user can customize the top of the network by adding dropout (specifying %) and indicate whether to inject gaussian noise (N(0,0.02)) after the flatten layer.

Dedicated functions in “ModelFucntions.py” create the both the generator and the discriminator from the user provided dictionaries at initialization.

After specifying and initializing the DCGAN class, the user compile the model (as with common Keras models) by passing separate Generator and Discriminator optimizers along with loss function to the “compile” function before training.

Since the labels for training are binary (real or fake classification) Binary Cross Entropy loss is used as the loss function for training all models, for distributed training the user must set reduction to “None” as loss aggregation is calculated manually for multi-GPU training. The discriminator output doesn’t apply sigmoid to the output so the loss uses logits.

Comprehensive examples of the above process are provided and documented in the provided notebooks.

The overall training process is illustrated in figure and the pseudocode provided below:

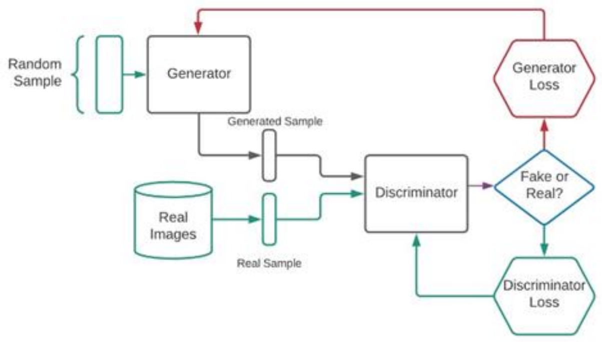


Figure 91 Training Routine Loop

Training Loop Pseudocode:

```
1. Initialize $\boldsymbol{w_{Gen}}, \boldsymbol{w_{Disc}}$
2. For epoch $e \in [1, \dots, N]$, $N =$ Total Num of Epochs:
  2.1. For every mini-batch $MB \in D$:
    2.1.1. Compute discriminator predictions and average
           loss on real images $\hat{y}_{real} \coloneqq D(MB)$, $L_{real} = \frac{1}{|MB|} L(\hat{y}_{real}, y)$
    2.1.2. Compute gradient of loss for real predictions
           with respect to Discriminator weights and update
           $\frac{\partial L_{real}}{\partial \boldsymbol{w_{Disc}}} = \Delta \boldsymbol{w_{Disc}}$; $\boldsymbol{w_{Disc}} = \boldsymbol{w_{Disc}} - \varepsilon(\Delta \boldsymbol{w_{Disc}})$
    2.1.3. Compute discriminator predictions and average
           loss on generated images $\hat{y}_{fake} \coloneqq D(G(z))$,
           $L_{fake} = \frac{1}{|MB|} L(\hat{y}_{fake}, y)$
    2.1.4. Compute gradient of loss for real predictions
           with respect to Discriminator weights and update
           $\frac{\partial L_{fake}}{\partial \boldsymbol{w_{Disc}}} = \Delta \boldsymbol{w_{Disc}}$; $\boldsymbol{w_{Disc}} = \boldsymbol{w_{Disc}} - \varepsilon(\Delta \boldsymbol{w_{Disc}})$
    2.1.5. Generate samples and compute generator loss
           from discriminator predictions and average loss
           on generated images $\hat{y}_{fake} \coloneqq D(G(z))$, $L_{fake} = \frac{1}{|MB|} L(\hat{y}_{fake}, y)$
    2.1.6. Compute gradient of loss for real predictions
           with respect to Discriminator weights and update
           $\frac{\partial L_{fake}}{\partial \boldsymbol{w_{Gen}}} = \Delta \boldsymbol{w_{Gen}}$; $\boldsymbol{w_{Gen}} = \boldsymbol{w_{Gen}} - \varepsilon(\Delta \boldsymbol{w_{Gen}})$
```

Two training loops were tested, a faster training loop where the gradients for both the generator and the discriminator were calculated in a single step and the training loop described above where gradient calculations and updates are calculated & applied in separate discrete steps. The faster training loop worked for lower resolution images but suffered from instability at image resolutions of 64x64 and above.

By splitting the steps above, the gradient for the generator is determined after the discriminator has been updated. I theorize that this results in more stable updates because doing so is similar to applying Nosterov momentum to the generator updates, facilitating more stability in learning.

Both implementations are available in the DCGAN class, the faster but unstable implementation is named “train_step_fast”. The slower, but more stable implementation detailed above is named “train_step_slow”. The training routine can be defined at model initialization and will default to the slower, more stable implementation. Both can be found within the DCGAN class.

Both distributed and single GPU training have been supported for training. The “distribute” flag should be set accordingly, both methods apply average batch gradient updates. Aggregation of gradients is calculated manually for distributed training. More detailed documentation can be found with the implementation in the “train_step_slow” function.

Model Training Evaluation

To evaluate the training of the model 5 key metrics were recorded:

1) Discriminator loss on real data
2) Discriminator loss on generated data
3) Generator loss using the discriminator against the generated data
4) Accuracy of the discriminator in correctly identifying real images
5) Accuracy of the discriminator in correctly identifying generated images

Recording and monitoring these metrics was key to determining if the generator was able to learn, or if the discriminator had become too good too early in the training process. Successfully interpreting these metrics lead to a systematic process for quickly iterating through different architectures and initializations.

In addition to the above quantitative metrics, visualizations of the generator's output after every epoch were generated and saved for inspection during the training process. This allowed fast evaluation of results early in the training process. This was done through custom callbacks using the Keras model callback API.

The evaluation metrics are included in the "train_step_slow" function within the DCGAN class.

The custom callback is defined as a derived class of the Keras Callback class. It is titled "SaveModelAndCreateImages" and can be found in "DCGAN.py", it is automatically called by the Keras derived model at the end of each epoch.

Feature Extraction Model Evaluation Process:

The DCGAN model was trained on ImageNet-1k for 60 epochs. A feature extractor model was then created from the trained discriminator. The method to create a feature extractor is part of the implemented DCGAN class. The function is a member of the DCGAN class and is named "create_feature_extractor".

Max pooling was applied to the output of each convolutional block of the discriminator. These spatial representations are flattened and concatenated and treated as image features. More details about the architecture of the feature extractor are provided in section 4.1.3. The function is generalized and can be called for any architecture of discriminator.

The model was then used to extract features from CIFAR-10 test and train datasets. An L-2 Regularized Support Vector Machine (using the Scikit-Learn library) was then fit using these features for classification using 1000 iterations. The model's performance was evaluated using the score function against the test set.

The demonstrated process and detailed documentation can be found in the "ImageNet-1K CIFAR-10 Classification" notebook.

Evaluating DCGANs As Labeled Data Generators:

To perform this investigation, the MNIST dataset was first converted to 32x32 resolution images (from 28x28).

Both the test and training datasets were converted and saved as JPG files to streamline an identical training process for training the classifier on generated images and original images.

DCGANs were trained on varying fractions of the training data (with rotation augmentation) for each labeled class (0 through 9).

Using the trained generator for each class, 30,000 labeled datapoints were generated for each label.

An EfficientNet-B0 model was then trained using SGD with learning rate 0.001 and momentum 0.3 using the generated data without augmentation and benchmarked against an identical model trained using the full original dataset using random rotation augmentation. All images were converted from grayscale to RGB to remain compatible with EfficientNet input.

The EfficientNet model used ImageNet pretrained weights with a custom top of Dropout (30%) -> Dense Layer (512 Units), ReLU activation to softmax output.

Detailed documentation and illustration of this process can be found in the MNIST Experiment notebook.

5. Results

5.1. Project Results

DCGAN results are difficult to evaluate quantitatively since the quality of generated image doesn't directly relate to loss or accuracy. Low discriminator accuracy of the generated images and low loss of the generator does not necessarily indicate good quality image generation, it may simply indicate a poor discriminator. Similarly, high discriminator accuracy may indicate a strong discriminator, but not necessarily poor quality of generated images.

While quantitative metrics were referenced for learning progress and model improvement, subjective evaluation of the quality of the generated image was used as the ultimate metric of training success.

5.1.1 Key Issues Encountered

The key challenge encountered in designing and training DCGAN class of model was in training instability.

An architecture and training routine that worked for one dataset would not necessarily work for another dataset. The DCGAN architecture that worked well for CelebA failed to learn on the MNIST dataset. For each application, an iterative approach to improving and refining different architectures was employed. During this process three issues consistently arose when attempting to train the DCGAN models.

Discriminator Too Strong Early in Training

For some model architectures and on some datasets, if the discriminator learned faster than the generator to correctly identify generated images, then the generator would be unable to learn.

Strategies could be employed to handicap the discriminator early in the training process to allow the generator time to learn, however this would often end up reducing the ultimate quality of the generated images in the later stages of training. This is illustrated in figure 12, where gaussian noise & 30% dropout has been applied to the discriminator. The generator has technically achieved convergence as the discriminator has a 50% accuracy classifying generated images, but the quality of the image is not as good as that achieved by a model trained without these discriminator handicaps.

The qualitatively best generator trained in this project (shown in section 5.1.2) images were classified as fake 73% of the time by its discriminator which had no handicap.

Generator Mode Collapse

Occasionally the generator would find a set of images that the discriminator will classify as "real" and rather than exploring and learning new mappings from the latent space to novel images, the generator will instead repeatedly generate these same images. This is illustrated in figure 10 where the same images are repeated for different random latent variable initializations.

This is referred to as "mode collapse", this appears to happen if the generator learns faster than the discriminator in the early stages of learning but is unable to adapt as the discriminator improves.

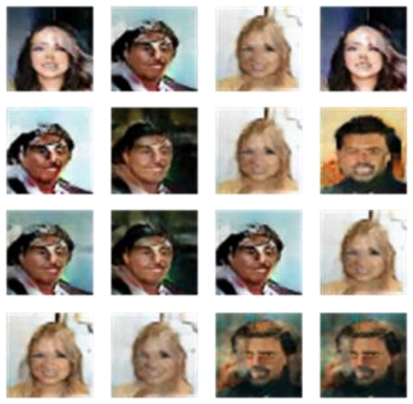


Figure 10 Mode Collapse

Generator Sensitivity to Training Data

The DCGAN architectures were particularly sensitive to backgrounds in images. Without cropping of facial images to limit the amount of background present in the training set, the DCGAN generator would struggle to correctly draw borders of the faces and the background would often blend into the detail of the generated faces. This can be observed in some of the facial images generated in figure 13.

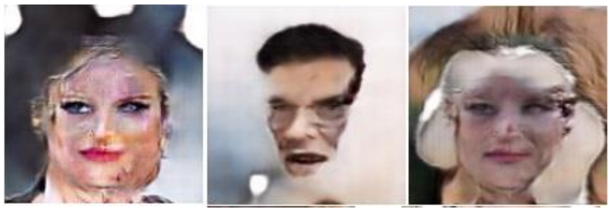


Figure 12 Underfitting

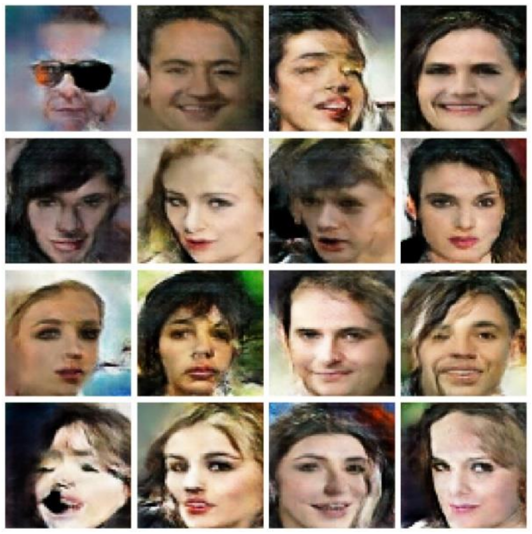


Figure 11 Discriminator with Dropout & Gaussian Noise

5.1.2 Solutions Employed to Solve Key Issues

To improve training stability and overall quality of generated images, a number of strategies, not mentioned in the original paper, were employed. Some of these suggestions came from research of online resources [3] (dropout, adding gaussian noise, label smoothing, gaussian initialization of kernel and gamma).

Other methods (variations on latent variable distribution, differing architectures, changes to training routine) were resulting from investigation arising from this project. These methods have been detailed below:

In order to prevent the Discriminator from becoming too good too early in the training process and preventing the generator from learning, the following techniques that were not included in the original paper were successful:

- Adding dropout to the discriminator to prevent the discriminator from learning too quickly.
- Adding random gaussian noise to the final layer of the discriminator improved training stability but reduced ultimate quality of generator at convergence.
- Adding label smoothing to the loss function.

DCGANs were sensitive to the dimension of the latent space. The more latent variables are used, the sparser the latent space becomes and the greater challenge it is for the generator to find a continuous mapping from the latent space distribution to the training data distribution. Reducing the number of latent variables to around 100 lead to better results. This is illustrated in figure 15 that shows the same architecture, all trained for 30 epochs, with varying numbers of latent variables. This is illustrated in figure 12.

To deal with underfitting and dataset specific issues the following methods were employed:

- Center cropping faces before resizing to minimize the amount of background in images lead to less underfitting/blending of faces with background in the generator.
- Cropping LSUN bedroom dataset images before resizing lead to more discernable features in the generated images due to less compression in the resized images and more clearly defined objects.

Training stability was further improved by experimenting with variations on the per batch gradient calculation and order of discriminator and generator updates (as mentioned in section 4.2).

Initial experimentation used a training routine where the loss and gradient of the discriminator and generator was captured in a single step. Doing so captured the loss and gradient of the discriminator and generator using the same set of generated images at the same time. This training routine worked reasonably at 32x32 resolutions but consistently resulted in generator learning collapse when applied to 64x64 RGB images.

Splitting the training routine to apply weight updates in 3 discrete steps (discriminator loss on real images, discriminator loss on generated images and generator loss on discriminator) vastly improved training stability at the expense of training speed.

I theorize that this approach is similar to applying Nosterov momentum to the generator update by calculating generator gradients after the discriminator update.

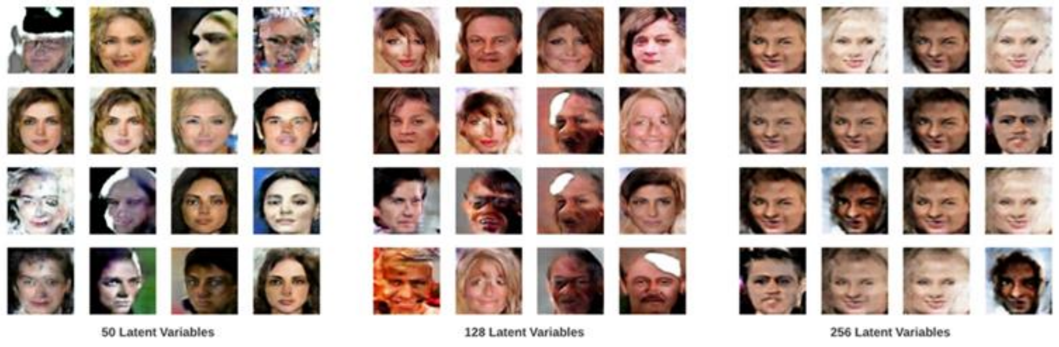


Figure 13 Latent Variable Scaling

To avoid mode collapse in the generator the following strategies were successful:

- Weight initialization for both Kernel and batch normalization Gamma parameters to Gaussian ~ (1, 0.02)
- Scale the number of filters rather than increasing the number of latent variables in the generator.

5.1.2 Facial Image Generation

Using a process of iterative improvement & experimentation, two high quality DCGAN models were trained against the CelebA dataset with image resolution of 32x32 and 64x64 for 190 epochs each. Progress of the generator's ability to generate images during the training process for images of 64x64 dimensions is shown in figure 15.

The images depict the generator's output for 9 randomly generated latent variables. The same latent vectors were used for each epoch, the evolution of generated images over time demonstrates the generator's learning process in finding a mapping from the latent space to plausible images of faces.

Cropping images to faces was necessary to minimize underfitting and a reduced number of latent variables (128) was found to achieve optimal results. The discriminator used LeakyReLU (alpha = 0.2) and 20% dropout in the head.

The specific architecture used to generate 64x64 images was 1024x4x4 dense units to deconvoluting blocks of 512, 256, 128 & 64 feature maps, stride 2. The discriminator used convolution blocks of 128, 256 and 512 feature maps. Mini-batches of size 128 were used.

A dedicated notebook has been provided demonstrating the image generation process [9] and model training process [22]. Each epoch took 4 minutes and total training time was 13 hours.

A sampling of randomly generated images from the final 64x64 generator after 150 epochs is shown in the figure 17:

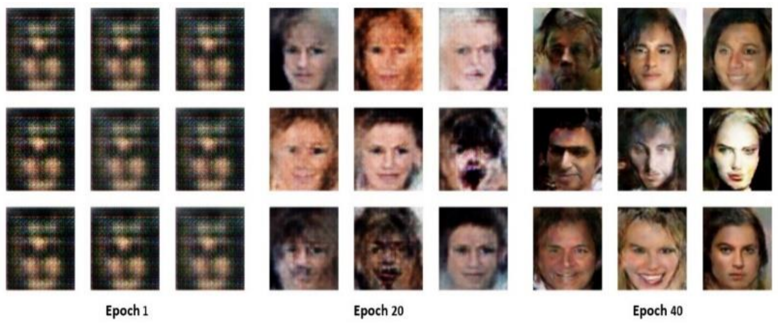


Figure 14 Training Progress

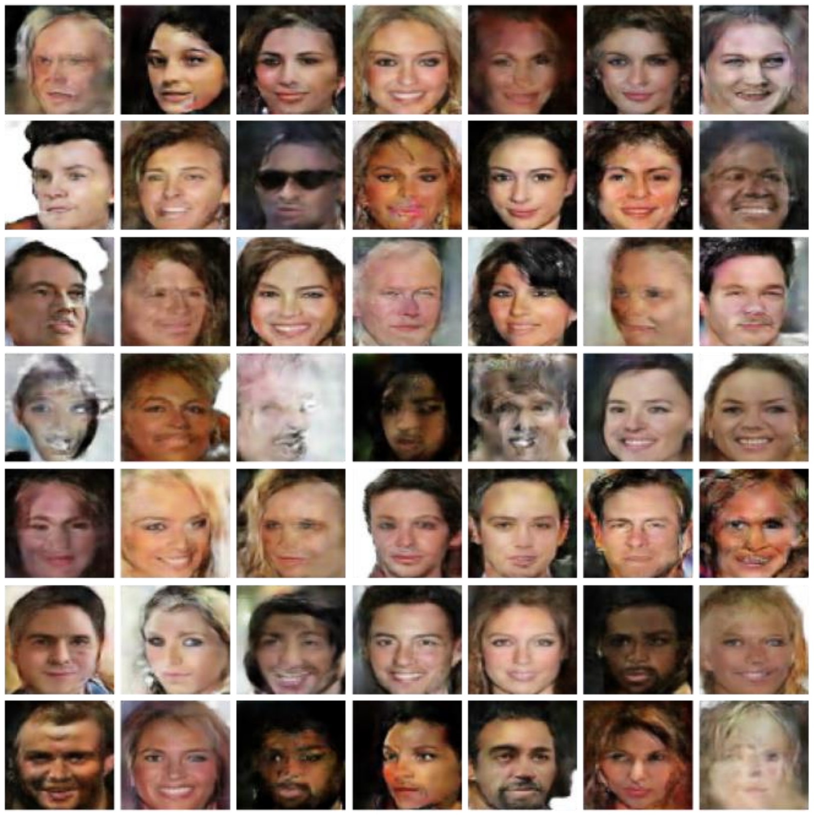


Figure 15 64x64 Images

5.1.3 Latent Space Vector Arithmetic on Facial Images

The best DCGAN model was trained on the CelebA dataset using 64x64 images for 60 epochs. Random latent vector samples were generated and grouped into vectors which represented smiling women, neutral women and neutral men. Three latent vectors were saved for each group, and their average was calculated and used to represent the group.

Using these three representations I demonstrate the vector arithmetic properties of the learned generator representations in the latent space by generating the output of the resulting vector by subtracting the neutral woman vector from the smiling woman vector and adding the neutral man vector. The resulting latent space vector represents a smiling man. The generator representation of the resulting vector of a smiling man with small random noise is displayed in figure 18 below.

This demonstration is provided in the dedicated notebook [9].

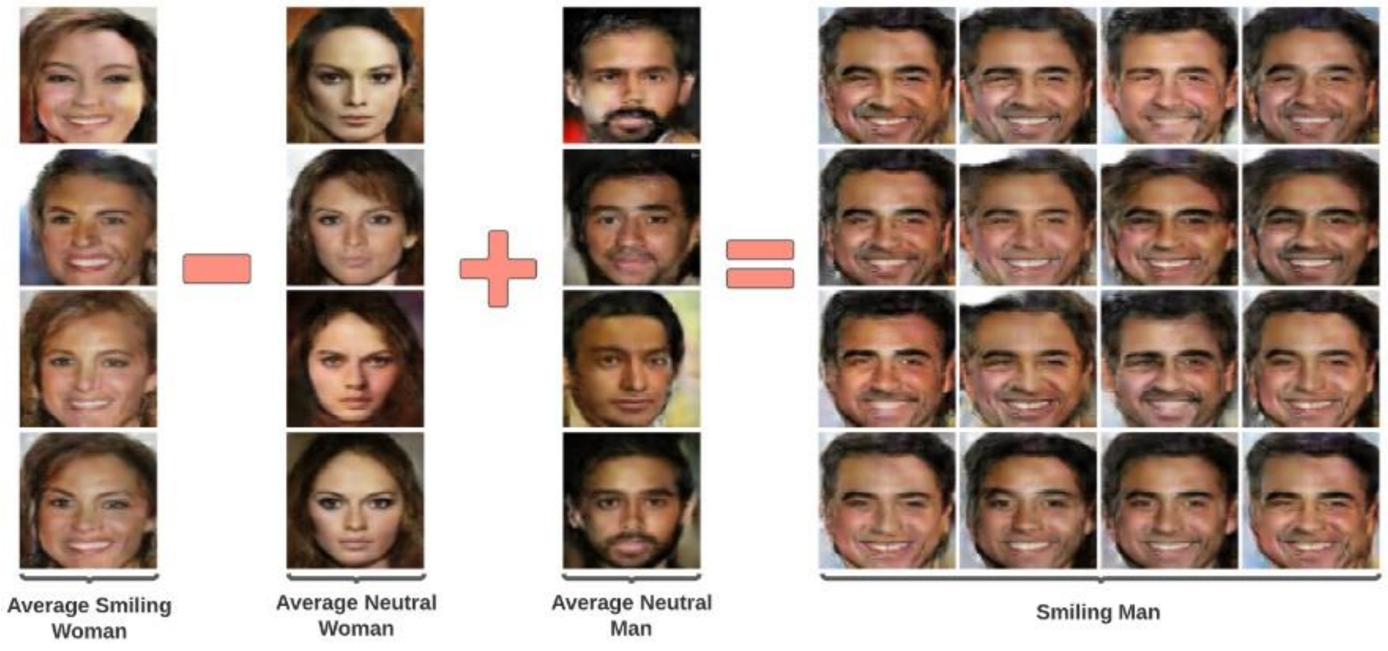


Figure 18 Demonstration of Vector Arithmetic on Faces

5.1.4 LSUN Bedroom Image Generation

Multiple DCGAN architectures were trained using the LSUN bedroom dataset. The model that achieved the best results used 5 layers of deconvolution with a larger number of filters per level than was necessary for facial images (768, 512, 384, 256) using a stride factor of 2 and 128 latent variables.

The best model was trained for 9 epochs (limited by computation time at 3 hours per epoch).

The resulting generator was used to generate the images displayed below. Despite the small number of epochs, the generator was able to recreate plausible images of bedrooms.

Visual inspection would indicate that the generator has been able to capture representations of common objects like beds, doors and windows.

In some instances, the generator has clearly learned the representation of objects but will combine them in the image in a way that doesn't make sense. Examples of this include images of beds with windows, and images of hallways imposed onto walls without doorframes.

Due to the excessive training time, it may have been possible to improve the ultimate image quality by either running for more epochs or increasing the discriminator complexity.

The LSUN tests can be found within the dedicated notebook for 64x64 LSUN Images [8].

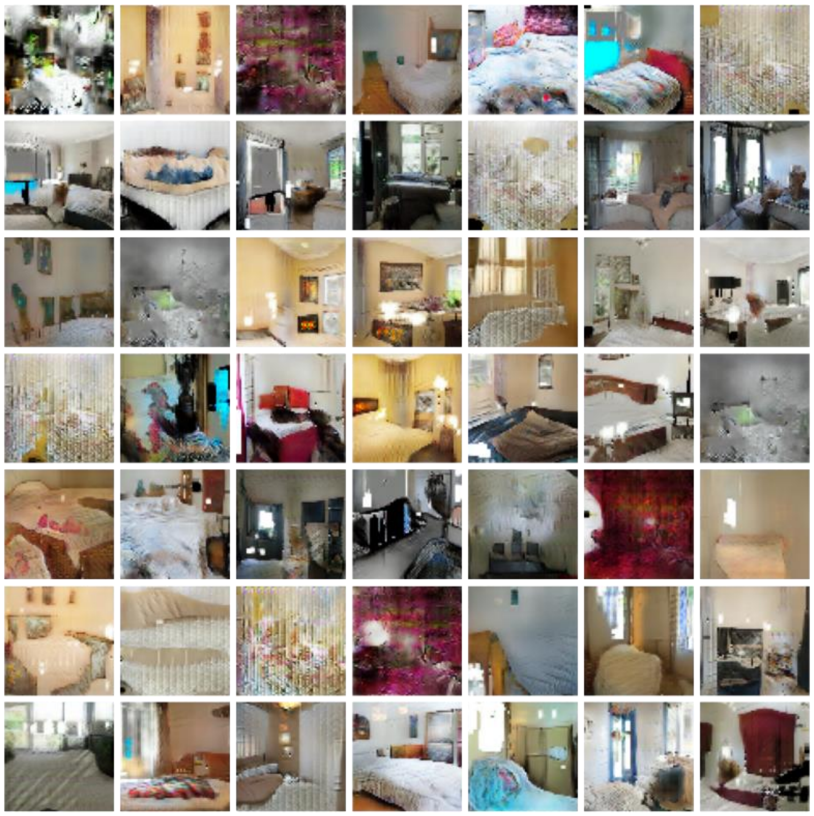


Figure 16 LSUN Bedroom

5.1.5 DCGAN Discriminator as a Feature Extractor

The most successful 32x32 RGB DCGAN architecture was trained on the ImageNet-1k dataset. After training, a feature extractor was constructed from the discriminator by applying max pooling to each convolutional layer of the discriminator.

The trained discriminator’s convolutional layers were composed of 128, 256 and 512 feature maps. Applying 2x2 max pooling to each layer of feature maps and flattening resulted in 14336 features per image.

The feature extractor was applied to CIFAR-10 images and a supervised L-2 SVM was trained for 1000 iterations each to classify using these features.

The below table illustrates the performance of the trained L-2 SVM model for greater number of training epochs of DCGAN on the ImageNet dataset:

Epochs	Test Accuracy
10	60.3%
25	64.5%
35	68.2%
45	70.8%

The results demonstrate that the discriminator has been able to learn representations that can be used to differentiate the contents of unseen images in a totally unsupervised manner.

Each Epoch took 16.5 Minutes, total training time was 12.4 hours.

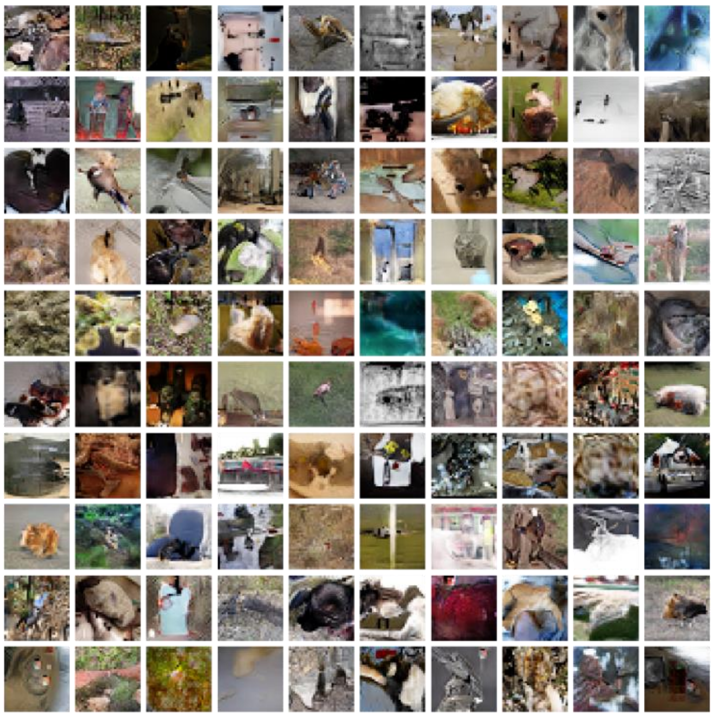


Figure 17 CIFAR Generated Images

5.1.6 Increasing DCGAN Image Resolution

It was demonstrated in the original paper and in the earlier experiments in this project that it is possible to train DCGAN architectures that adhere to the parameters and constraints set out in the original paper up to a resolution of 64x64.

Attempting to scale DCGANs beyond 64x64 to 128x128 and 256x256 proved to be very challenging. There are a few options to increase output resolution of the DCGAN; increasing the initial kernel size at the dense projection layer, increasing the depth of the generator and increasing the fractional stride factor.

All three methods of resolution upscaling were investigated (increasing fractional strides, increasing generator depth and increasing the initial kernel width).

Training instability occurred when increasing the depth of the generator and increasing the fractional stride. For deeper architectures, the discriminator would outpace the generator in learning and become too effective at classifying images causing mode collapse, this is illustrated in figure 21.

The generator was unable to learn when using larger fractional strides in place of increased depth, illustrating the limits in upscaling between representations.

Methods to “handicap” the discriminator’s learning were employed to little success (dropout and injecting gaussian noise).

In the paper “High-resolution Deep Convolutional Generative Adversarial Networks” Curtó et al. [6] the authors suggest replacing ReLU activation in the generator with SELU activation (Scaled Exponential Linear Units) for higher resolutions. This method was tested. Learning in early epochs was promising but collapsed in later epochs, this is illustrated in figures 18 and 17. Initial images demonstrates some learning after the 1st epoch but the generator loss explodes at the 4th epoch. Due to long epoch time (~45 min) time constraints modifications weren’t possible, though with reducing learning rate or increasing discriminator depth may have improved outcomes here.

The most successful method to increase resolution was in increasing the filter size of the first layer, rather than increasing the depth of the generator. The results after 20 epochs of training can be seen in figure 20. With more epochs of training the images would likely continue to improve given the trend is similar to that of 64x64 images.

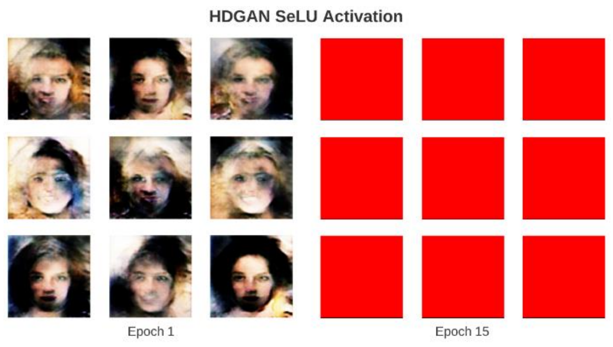


Figure 17 HDCGAN Learning Progress

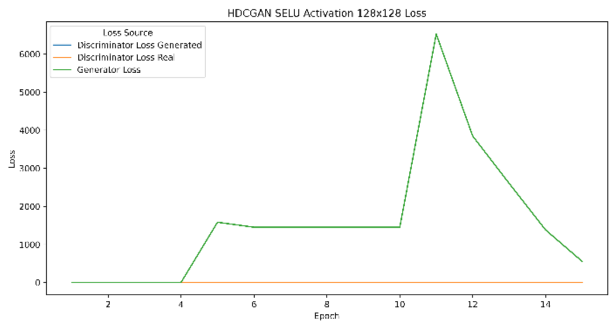


Figure 18 HDCGAN Loss

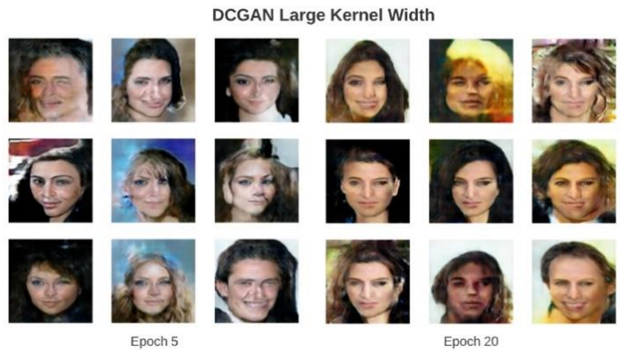


Figure 19 DCGAN Large Kernel Width

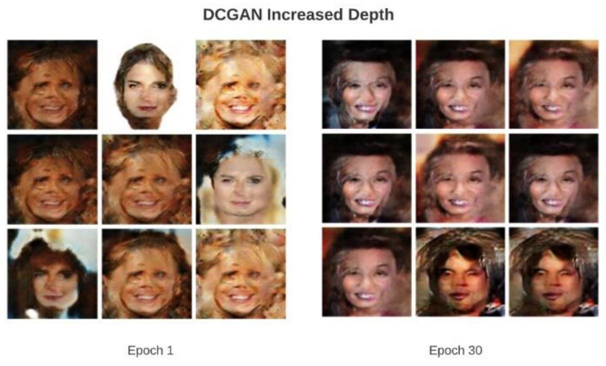


Figure 21 DCGAN Increased Depth

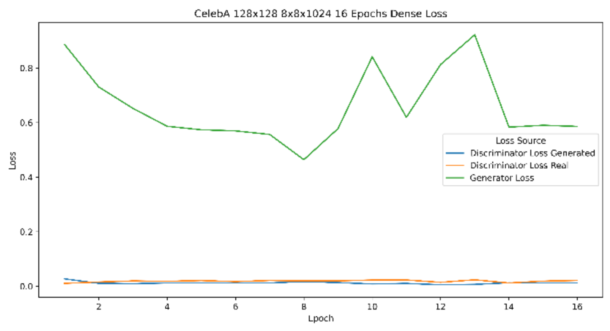


Figure 20 DCGAN Large Kernel Loss

5.1.7 DCGAN as a Labeled Data Generator

One of the main challenges in applying machine learning algorithms is in gathering, curating and labelling high quality training data which adequately captures the distribution of the target class variable.

In this project I investigate whether the generator component of the DCGAN could be trained on a fraction of MNIST training data to generate labeled examples which could then be utilized to train a classifier to achieve a similar level of performance as a classifier trained against the full dataset.

The best quality generator was determined through a process of iterative improvement and trial & error using the average discriminator loss and average generator loss to determine convergence, with qualitative evaluation of the generated images (more details in appendix).

Interestingly, unlike RGB images, the best performing DCGAN model for MNIST used Leaky ReLU [7] in the generator and the discriminator. Shallower architectures were necessary.

Training for 80 epochs allowed the generator to capture the target distribution without overfitting to the data.

For each class label (0 through 9) a DCGAN models were trained using 40%, 60% and 80% of available training data. The trained generator was then used to generate 30K random samples for each label. An example of the generator output has been provided in figure 19.

EfficientNet-B0 models with identical architectures and optimizers were then trained for 40 epochs using **only the generated data** from each training proportion without augmentation. An identical EfficientNet-B0 model was trained on the original data with augmentation (random rotation +- 30 degrees).

All models were evaluated against the test set, their respective accuracy is provided in the table below:

Model	Test Accuracy
EffNet-B0 Full Training Set	94.38%
EffNet-B0 DCGAN 40% Training Set	52.39%
EffNet-B0 DCGAN 60% Training Set	85.99%
EffNet-B0 DCGAN 80% Training Set	97.96%

The results are somewhat surprising, particularly for the 80% of training data group, it was expected that the classifier trained against the generated data would not perform as well as a model trained against the original dataset.

I theorize that the 80% group performs better than the original group because the DCGAN generator may be more adventurous with how it draws digits in ways that don't exist in the original dataset with augmentation.

It is also possible that there is a random element in training the classifier, Efficient-Net is designed for RGB images, and while it can be trained for grayscale it's possible there is volatility in convergence.

With more time, it would have been interesting to investigate this further with a simpler CNN architecture.

What the above results do demonstrate is that a DCGAN architecture can be used to augment/enhance the training set where labelled data may be scarce.

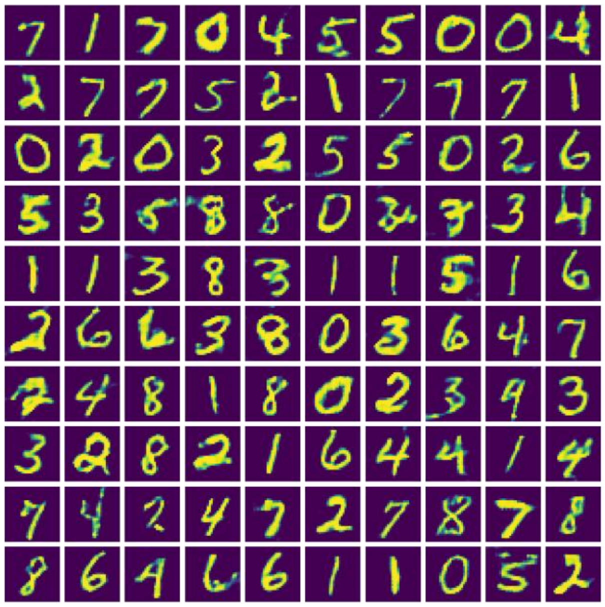


Figure 22 MNIST Generated Images

It would have been interesting to see if including the original data alongside the generated images would have resulted in better than baseline performance, but due to time constraints this could not be tested.

5.2. Comparison of the Results Between the Original Paper and Students' Project

A consistent trend throughout the attempts to replicate the results of the original paper was that the architectures that were investigated as part of this project did not train as fast as those presented by the original authors.

This remained the case even when using similar optimizer configurations and mini-batch sizes.

The factors that are driving this are likely variations in the architectures employed, differences in resolutions of output images and possibly differences in the random initializations of the weights. This level of technical detail was excluded from the original paper.

For all of the tested datasets, excluding the LSUN dataset, the same level of image was qualitatively achieved, despite more epochs required for training.

5.2.1 Facial Image Generation & Vector Arithmetic

With some adjustment to the training data and refinement of the DCGAN architecture, the generator as part of the project was able to produce similar fidelity of image as that demonstrated in the original paper. Despite using a different dataset.

Figure x demonstrates almost identical vector arithmetic when applied to the learned latent space mapping illustrated in the original paper. However, unlike the original paper, I was unable to find a consistent vector which represented sunglasses to predictably manipulate the output of the generator. I believe this is likely due to differences in the training dataset. The original dataset was trained against images of normal people in differing locations whereas the faces dataset used in this project was that of celebrities who may be less likely to include sunglasses.

An interesting differentiation between the generated images of faces in this project versus that of the original paper is in the more distinctive facial features generated by the model trained against the celebrity dataset than that generated against the original paper's dataset see figure 23:

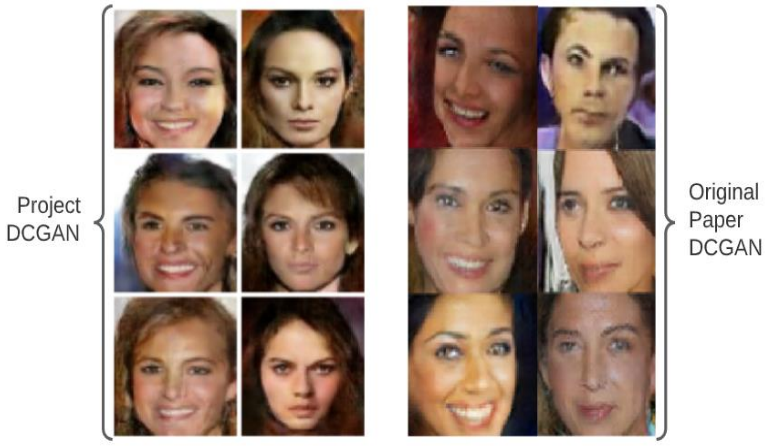


Figure 23 Comparison of Generated Image Quality

This is likely a function of the training data, celebrities tend to have distinctive facial features and generally look more unique, so the DCGAN generator trained using this data is more likely to generate distinctive facial features.

5.2.2 LSUN Dataset

The original paper demonstrated that their DCGAN architecture was able to generate plausible images after just one epoch. The DCGAN architecture trained as part of this project did not learn as quickly or as well qualitatively. This was likely due to differences in model architectures employed.

The model trained in this project did learn to represent images of bedrooms but took more epochs to learn and qualitatively produced lower quality images at the same stage of training.

A comparison of the images between the paper and the project DCGAN is illustrated in figure 24.

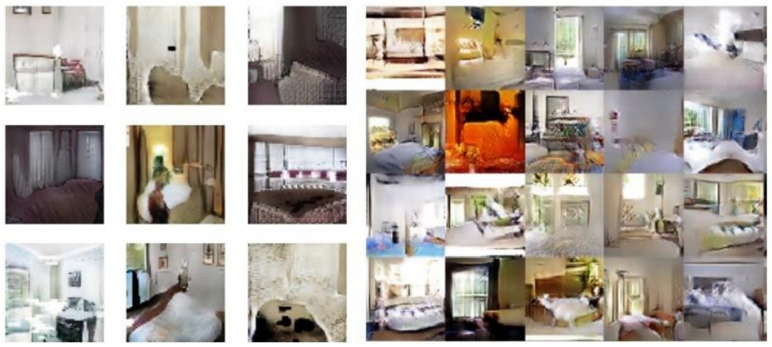


Figure 24 Comparison of Project DCGAN vs Original Paper after 1 Epoch

5.2.3 DCGAN Discriminator as a Feature Extractor

The original paper demonstrated a 73.8% accuracy vs 70.8% accuracy demonstrated in the project when using the discriminator from a DCGAN trained on ImageNet as a feature extractor for CIFAR-10 classification.

This is quantitatively close performance and demonstrates the robustness of the representation learning of the discriminator when trained in an unsupervised manner.

However, I believe the gap in performance may be due to the original paper applying differing levels of max pooling to the different levels of discriminator features.

I suspect larger kernel sizes of max pooling were applied at the lower level kernel representations and smaller/no max pooling was applied at the higher level kernels which are likely more informative than the lower level kernels.

The authors didn’t provide the exact architecture of the feature extractor and its max pooling parameters. Applying differing size max pooling would result in more preservation of the more delineated features at the higher level, perhaps improving performance.

Additionally, the original authors didn’t provide specifics with regards to the L2SVM trained (how many iterations were used in training etc).

These factors may be driving the difference in accuracy observed between my implementation and the original paper.

5.3. Discussion of Insights Gained

The most significant challenge throughout this project was dealing with the inherent instability in training DCGANs. It was determined that the key to successfully training a good quality DCGAN was in balancing the learning of the discriminator and the generator.

Failure occurs if either the generator or the discriminator becomes too strong to early in the training process. However, interventions to improve early training stability can often lead to undesirable results later in the training process.

A key breakthrough in this project was in attaining training stability by modifying the training routine to update first the discriminator then the generator (after applying discriminator updates) in discrete steps as detailed in sections 4.2 and 5.1.2. This approach was stumbled upon while trying to debug the gradient/weight updates.

I theorize that this approach improved stability for two reasons: applying updates to the discriminator using generated and real predictions before calculating the generator loss and gradients gives the generator an edge in determining the optimal gradient with which to fool the discriminator. Additionally, by not sharing the same set of generated images for both discriminator and generator updates, better coverage of the latent space is introduced into training.

The experiments performed using different architectures demonstrated that the architecture of the discriminator is equally as important to that of the generator in achieving successful training. On simpler images, an overly complicated generator will be unable to learn before the discriminator can learn to distinguish real from fake images. This was when an architecture that worked well for the CelebA dataset fell into learning collapse on MNIST dataset.

For lower resolutions, simpler and shallower generators often performed better than more complicated generators. For the discriminator, 3 convolutional blocks with leaky RELU activation was typically a good starting point. If the discriminator learns faster than the generator then regularization of the discriminator (adding gaussian noise or dropout) would typically improve training.

Constraining the number of latent variables would often lead to faster and more successful training of the generator. With higher dimension latent space, the generator must learn a mapping from a higher dimensional sparse representation space to image output. With greater number of latent variables the generator encounters the ‘curse of dimensionality’, and has a harder time learning a consistent mapping from latent space to plausible image. In a higher latent dimension, points in the latent space are farther away and the generator struggles to find a continuous mapping between latent space and data distribution which in turn encourages mode collapse.

DCGAN architectures work well up to resolutions of 64x64 images, but instability occurs for higher resolutions. Most commonly, the discriminator would learn faster to classify images than the generator would to create plausible fake images. It was found during the experimentation in section 5.1.6 that the most successful method to increase image resolution while maintaining training stability was to increase initial kernel width rather than increasing generator depth.

Subsequent research in the field has focused on scaling up architectures during while maintaining stability. Progressive GAN architectures [17] have become popular for training GANs to produce higher resolution images. Progressive GANs are capable of controlling training stability by starting with a low resolution image and progressively increasing image resolution during the training process rather than attempting to train for high resolutions directly, as was attempted in this project.

It would have been interesting to investigate methods which introduce discriminator regularization/handicaps in the earlier stages of training to allow generators with increased depth to start learning and then in reduce this regularization at later stages of learning to balance generator / discriminator learning, but time constraints limited further experimentation.

6. Conclusion

This project was able to successfully create and train a series of Deep Convolutional Generative Adversarial Network class of models on a variety of different datasets.

The results of the original paper in generated image quality were qualitatively achieved and the vector arithmetic properties of generated facial images successfully replicated. Similarly, the results of the original paper demonstrating the quality of representation learning was replicated by adapting the trained discriminator as a feature extractor for image classification on the CIFAR-10 dataset.

Building upon the work of the original paper, it was demonstrated that a trained DCGAN could be used to generate additional labeled training examples to potentially improve classification performance in instances where the availability of labeled data is limited.

In addition to this work, it was demonstrated that it was possible to scale DCGAN architectures to generate images of resolution larger than 64x64 and that increasing kernel width of the initial dense layer was the most effective method to achieve 128x128 resolutions. However, the same quality demonstrated in the HDCGAN paper Curto et al.[6] or more recent methods like Progressive GANs demonstrated by Karras et al.[17] was not achieved.

The common theme in training GANs is the inherent instability in training. DCGANs are no exception to this and iterative methods to find an architecture that would train were required for different datasets and image resolutions.

The key takeaway from this project was the recognition of the inherent instability in training GANS. While the original DCGAN constraints made success more likely, careful iterative refinements to the architecture and training data are necessary to achieve good results on a variety of datasets. Extensions to the original architecture in the form of dropout, gaussian noise and label smoothing could be used as tools to improve stability.

Advancements have been made in the field by maintaining stability in training by progressively growing both the generator and discriminator during training to produce progressively higher resolution images (Progressive GANs)[17]. These approaches have demonstrably outperformed more direct DCGAN models in terms of image quality.

Alternative non-adversarial approaches to Deep Generative models have demonstrated promising results in terms of image quality at high resolutions. Deep Hierarchical Variational Autoencoders proposed by Vahdat et al. [18] build upon Variational Autoencoders proposed by Kingma et al. [20] have demonstrated successful high quality image generation and are trained using likelihood methods rather than an adversarial approach to training.

6. Acknowledgement

Tensorflow’s DCGAN resources [7] were insightful in understanding how to train two models in the same loop via manual gradient capture/application. The machine learning mastery page on GAN stability [3] by Jason Brownlee provided ideas for experimentation (gaussian noise, label smoothing & dropout).

Lex Fridman’s podcast with Ian Goodfellow [5] lead to the inspiration to try and use a DCGAN for labeled training data generation.

7. References

[1] Alec Radford, Luke Metz, Soumith Chintala "Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks" https://arxiv.org/pdf/1511.06434.pdf

[2] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean. "Efficient Estimation of Word Representations in Vector Space", https://arxiv.org/abs/1301.3781

[3] Jason Brownlee "Tips for Training Stable Generative Adversarial Networks", https://machinelearningmastery.com/how-to-train-stable-generative-adversarial-networks/

[4] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, "Generative Adversarial Networks" https://arxiv.org/abs/1406.2661

[5] Lex Fridman, Ian Goodfellow, "Ian Goodfellow: Generative Adversarial Networks (GANs) | Lex Fridman Podcast #19" https://www.youtube.com/watch?v=Z6rxFNMGdn0

[6] Joachim D. Curtó, Irene C. Zarza, Fernando De La Torre, Irwin King, Michael R. Lyu, "High-Resolution Deep Convolutional Generative Adversarial Networks" https://arxiv.org/abs/1711.06491v12

[7] Google "Deep Convolutional Generative Adversarial Network" https://www.tensorflow.org/tutorials/generative/dcgan

[8] LSUN 64x64 Testing Notebook

[9] CelebA 64x64 Vector Arithmetic Notebook

[10] CelebA 128x128 DCGAN Notebook

[11] Liu, Ziwei and Luo, Ping and Wang, Xiaogang and Tang, Xiaoou, "Deep Learning Face Attributes in the Wild", Proceedings of International Conference on Computer Vision (ICCV), CelebA Dataset, http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

[12] Fisher Yu, Ari Seff, Yinda Zhang, Shuran Song, Thomas Funkhouser and Jianxiong Xiao
LSUN: Construction of a Large-scale Image Dataset using Deep Learning with Humans in the Loop
arXiv:1506.03365 [cs.CV], 10 Jun 2015

[13] LeCun, Yann and Cortes, Corinna and Burges, CJ, "MNIST handwritten digit database", http://yann.lecun.com/exdb/mnist

[14] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton "Learning Multiple Layers of Features from Tiny Images", 2009 CIFAR-10 Dataset https://www.cs.toronto.edu/~kriz/cifar.html

[15] Tensorflow Imagenet Dataset https://www.tensorflow.org/datasets/catalog/imagenet_resized

[16] ImageNet1k CIFAR-10 Feature Extractor Notebook

[17] Tero Karras, Timo Aila, Samuli Laine, Jaakko Lehtinen "Progressive Growing of GANs for Improved Quality, Stability, and Variation" https://arxiv.org/pdf/1710.10196.pdf

[18] Arash Vahdat, Jan Kautz, "NVAE: A Deep Hierarchical Variational Autoencoder" https://arxiv.org/abs/2007.03898

[19] Diederik P Kingma, Max Welling "Auto-Encoding Variational Bayes" https://arxiv.org/abs/1312.6114

[20] MNIST Notebook

[21] CelebA64x64 Notebook

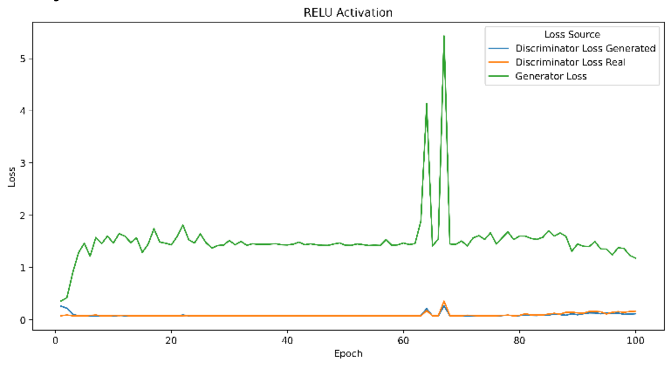
8. Appendix

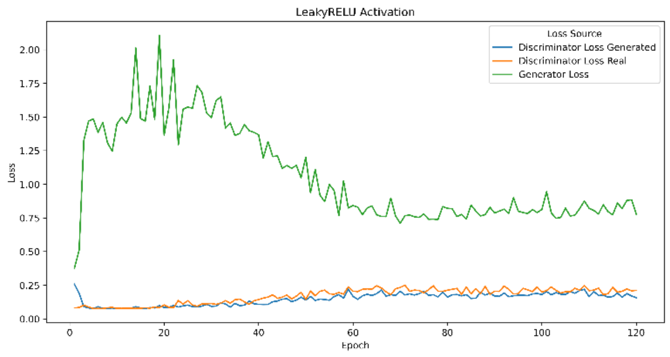
8.1 Individual Student Contributions in Fractions

This project in its entirety was product of my own work.

Comparison of MNIST Models:

LeakyReLU Activation in Generator Vs ReLU Activation


RELU Activation
Loss Source
Discriminator Loss Generated
Discriminator Loss Real
Generator Loss
Loss
Epoch


LeakyRELU Activation
Loss Source
Discriminator Loss Generated
Discriminator Loss Real
Generator Loss
Loss
Epoch