**Generative Adversarial Nets**

**Abstract**

**Introduction.**

GANs as a Two Player Game

Player-1). **Generator**: Generates fake images

The generator model takes a fixed-length random vector as input and generates a sample in the domain, such as an image. A vector is drawn randomly from a Gaussian distribution and is used to seed or source of noise for the generative process. To be clear, the input is a vector of random numbers. It is not an image or a flattened image and has no meaning other than the meaning applied by the generator model. After training, points in this multidimensional vector space will correspond to points in the problem domain, forming a compressed representation of the data distribution. This vector space is referred to as a latent space, or a vector space comprised of latent variables. Latent variables, or hidden variables, are those variables that are important for a domain but are not directly observable.

We often refer to latent variables, or a latent space, as a projection or compression of a data distribution. That is, a latent space provides a compression or high-level concepts of the observed raw data such as the input data distribution. In the case of GANs, the generator model applies meaning to points in a chosen latent space, such that new points drawn from the latent space can be provided to the generator model as input and used to generate new and different output examples.

Player-2). **Discriminator**: Identifies fake and real images

The discriminator model takes an example from the problem domain as input (real or generated) and predicts a binary class label of real or fake (generated). The real example comes from the training dataset. The generated examples are output by the generator model. The discriminator is a normal classification model. After the training process, the discriminator model is discarded as we are interested in the generator.

Generative modeling is an unsupervised learning problem, although a clever property of the GAN architecture is that the training of the generative model is framed as a supervised learning problem.

The two models, the generator and discriminator, are trained together. The generator generates a batch of samples, and these, along with real examples from the domain, are provided to the discriminator and classified as real or fake. The discriminator is then updated to get better at discriminating real and fake samples in the next round, and importantly, the generator is updated based on how well, or not, the generated samples fooled the discriminator.

We can think of the generator as being like a counterfeiter, trying to make fake money, and the discriminator as being like police, trying to allow legitimate money and catch counterfeit money. To succeed in this game, the counterfeiter must learn to make money that is indistinguishable from genuine money, and the generator network must learn to create samples that are drawn from the same distribution as the training data.

In this way, the two models are competing against each other. They are adversarial in the game theory sense and are playing a zero-sum game. In this case, zero-sum means that when the discriminator successfully identifies real and fake samples, it is rewarded and no change is needed to the model parameters, whereas the generator is penalized with large updates to model parameters. Alternately, when the generator fools the discriminator, it is rewarded and no change is needed to the model parameters, but the discriminator is penalized and its model parameters are updated..

At a limit, the generator generates perfect replicas from the input domain every time, and the discriminator cannot tell the difference and predicts unsure (e.g. 50% for real and fake) in every case. This is just an example of an idealized case; we do not need to get to this point to arrive at a useful generator model.

**Architecture**

Random input vector

Generator model

Generated example Real example

Discriminator model

***Update model Update model***

Binary classification (real/fake)

Both models are multilayer perceptrons

**ALGORITHM**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Algorithm 1** Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used k = 1, the least expensive option, in our experiments.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**for** number of training iterations **do**

**for** k steps **do**

• Sample minibatch of m noise samples {z (1) , . . . , z (m)} from noise prior pg(z).

• Sample minibatch of m examples {x (1) , . . . , x (m)} from data generating distribution pdata(x).

• Update the discriminator by ascending its stochastic gradient:

∇θd 1/m Σ [ log D ( x (i)) + log ( 1 − D ( G ( z (i)))) ]

**end for**

• Sample minibatch of m noise samples {z (1) , . . . , z (m)} from noise prior pg(z).

• Update the generator by descending its stochastic gradient:

∇θg 1/m Σ[ log ( 1 − D (G ( z (i)))) ] .

**end for**

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**x** –--- real sample

**z**---- input of the generator and it is usually a random noise

**D**- ---discriminator function or D is a differentiable function represented by a multilayer perceptron with parameters θd

**G**----generator function or G is a differentiable function represented by a multilayer perceptron with parameters θg

**D(x , θd )** –---output of the discriminator when the input is x , θd is a discriminator parameter

**G(z , θg)** –---fake sample generated by the generator when input is z , θg  is a generator parameter

**Pg** –---probability distribution of the data generated by the generator over data x

**Px =pdata(x)**----probability distribution of the real data

**Pz** –---probability distribution of the latent space which is usually a random Gaussian distribution.

**value function V (G, D)**

minG maxD V (D, G) = Ex∼pdata(x) [log D(x)] + Ez∼pz(z) [log(1 − D(G(z)))] 1

x is real data

D(x) = 1

G(z) ,is generated data

D(G(z)) = 1 ( detected as real )

D(G(z)) = 0 ( detected as fake )

Generator tries to minimise the value of **value function** by generating the data which is very close to real data where the discriminator fails to detect it as fake , means detector detects it as real and provide D(G(z)) = 1 hence **value function** becomes minimum .

Similarly discriminator tries to maximise the value of **value function** by detecting generator’s generated data as fake and provides D(G(z)) = 0 .

The generator G implicitly defines a probability distribution pg as the distribution of the samples G(z) obtained when z ∼ pz. Therefore, we would like Algorithm 1 to converge to a good estimator of pdata, if given enough capacity and training time. The results of this section are done in a nonparametric setting, e.g. we represent a model with infinite capacity by studying convergence in the space of probability density functions

Optimizing D to completion in the inner loop of training is computationally prohibitive, and on finite datasets would result in overfitting. Instead, we alternate between k steps of optimizing D and one step of optimizing G. This results in D being maintained near its optimal solution, so long as G changes slowly enough.

In practice, equation 1 may not provide sufficient gradient for G to learn well. Early in learning, when G is poor, D can reject samples with high confidence because they are clearly different from the training data. In this case, log(1 − D(G(z))) saturates. Rather than training G to minimize log(1 − D(G(z))) we can train G to maximize log D(G(z)). This objective function results in the same fixed point of the dynamics of G and D but provides much stronger gradients early in learning

**Suppose if discriminator identifies fake image more accurately ,then D(G(z)) value will be very close to zero**

**Example:**

D(G(z)) = 0.01 then log(1 – D(G(z))) = - 0.004364 , logD(G(z)) = - 2.00

= 0.02 = - 0.008773 , = - 1.69

**Gradient\_12 = - 0.4409 = 31**

= 0.03 = - 0.013228 = - 1.5228

**Gradient\_23 = -0.445 = 16.72**

**Figure 1**

Generative adversarial nets are trained by simultaneously updating the discriminative distribution (D, blue, dashed line) so that it discriminates between samples from the data

generating distribution (black, dotted line) px from those of the generative distribution pg (G) (green, solid line).

The lower horizontal line is the domain from which z is sampled, in this case uniformly. The horizontal line above is part of the domain of x.

The upward arrows show how the mapping x = G(z) imposes the non-uniform distribution pg on transformed samples.

**G contracts in regions of high density and expands in regions of low density of pg ,so that pg becomes equals to pdata.( pdata = pg)**

**(a) Consider an adversarial pair near convergence: pg is similar to pdata and D is a partially accurate classifier. Because discriminator is not trained , which shown by fluctuated blue line**

(b) In the inner loop of the algorithm D is trained to discriminate samples from data, converging to

D∗(x) = pdata(x)/ pdata(x)+pg(x) .

**Here discriminator identifies real and fake perfectly because it is trained**. Clear transformation of blue line shows it.

**(c) After an update to G, gradient of D has guided G(z) to flow to regions that are more likely to be classified as data. Shift of green line towards black dotted line.**

**(d) After several steps of training, if G and D have enough capacity, they will reach a point at which both cannot improve because pg = pdata. The discriminator is unable to differentiate between the two distributions, i.e. D(x) = 1 /2 . Because green line and black lines completely merged means both the data’s have similar distribution**.

**Kullback–Leibler divergence(KL):** it is also known as relative entropy . It is equal to Binary-cross entropy.

Binary-cross entropy is a function that is used in binary classification task ( like: ’yes’ or ’no’ , ‘0’ or ‘1’ , ‘left’ or ‘right’ )

KL measures the how much a given arbitrary distribution is away the true distribution

If two distributions perfectly match then KL(pdata /pg) = 0 , otherwise it can take value between 0 and infinity.

Lower the KL divergence value , the better we have matched the true distribution with our approximation.

KL divergence is asymmetric , means KL(pdata / pg ) is not equal to (pg /pdata )

KL(pdata // pg ) = sum x in X Pdata(x) \* log(pdata(x) / pg(x) )

**Jensen– Shannon divergence(JS):** JS divergence is another way to quantify the difference (or similarity) between two probability distributions.

It uses the KL divergence to calculate a normalised score that is symmetrical. JS(pdata / pg ) = JS(pg / pdata)

**It is more useful as a** measure as it provides a smoothed and normalised version of KL divergence with score between 0 ( identical ) and 1 ( maximally different )

JS = KL ( pdata //pdata + pg /2 ) + KL ( pg//pdata + pg/2 )

**Global Optimality of pg = pdata**

We first consider the optimal discriminator D for any given generator G.

**Proposition 1**. For G fixed, the optimal discriminator D is

DG\*(x) = pdata(x)/( pdata(x) + pg(x)) (2)

Proof. The training criterion for the discriminator D, given any generator G, is to maximize the quantity V (G, D)

V (G, D) = ʃx pdata(x) log(D(x))dx + ʃz pz(z) log(1 − D(g(z)))dz

= ʃx pdata(x) log(D(x)) + pg(x) log(1 − D(x))dx

For any (a, b) ∈ R 2 \ {0, 0}, the function y → a log(y) + b log(1 − y) achieves its maximum in [0, 1] at a /a+b . The discriminator does not need to be defined outside of Supp(pdata) ∪ Supp(pg), concluding the proof.

**Supp = support , ( If the object is a measure , like probability examples , then the support is typically, the smallest closed set whose complement has measure zero )**

Note that the training objective for D can be interpreted as maximizing the log-likelihood for estimating the conditional probability P(Y = y|x), where Y indicates whether x comes from pdata (with y = 1) or from pg (with y = 0). The minimax game in Eq. 1 can now be reformulated as:

C(G) = maxD V (G, D)

=Ex∼pdata [log D∗ G(x)] + Ez∼pz [log(1 − D∗ G(G(z)))] (4)

=Ex∼pdata [log D∗ G(x)] + Ex∼pg [log(1 − D∗ G(x))]

=Ex∼pdata [ log pdata(x)/( Pdata(x) + pg(x))] + Ex∼pg [ log pg(x)/( pdata(x) + pg(x) )]

**Theorem 1**. The global minimum of the virtual training criterion C(G) is achieved if and only if pg = pdata. At that point, C(G) achieves the value − log 4.

Proof. For pg = pdata , D∗ G(x) = 1/ 2 , (consider Eq. 2). Hence, by inspecting Eq. 4 at D∗ G(x) = 1/ 2 , we find C(G) = log 1/ 2 + log 1/ 2 = − log 4. To see that this is the best possible value of C(G), reached only for pg = pdata, observe that

Ex∼pdata [− log 2] + Ex∼pg [− log 2] = − log 4

and that by subtracting this expression from C(G) = V (DG\*, G), we obtain:

C(G) = − log(4) + KL( pdata //(pdata + pg )/ 2 + KL ( pg //( pdata + pg )/2 (5)

where KL is the Kullback–Leibler divergence. We recognize in the previous expression the Jensen– Shannon divergence between the model’s distribution and the data generating process:

C(G) = − log(4) + 2 · JSD (pdata // pg ) (6)

Since the Jensen–Shannon divergence between two distributions is always non-negative and zero only when they are equa l, we have shown that C∗ = − log(4) is the global minimum of C(G) and that the only solution is pg = pdata, i.e., the generative model perfectly replicating the data generating process.

**Convergence of Algorithm 1**

**Proposition 2**. If G and D have enough capacity, and at each step of Algorithm 1, the discriminator is allowed to reach its optimum given G, and pg is updated so as to improve the criterion

Ex∼pdata [log D∗ G(x)] + Ex∼pg [log(1 − D∗ G(x))]

then pg converges to pdata

**Proof**. Consider V (G, D) = U(pg, D) as a function of pg as done in the above criterion. Note that U(pg, D) is convex in pg. The sub-derivatives of a supremum of convex functions include the derivative of the function at the point where the maximum is attained. In other words, if f(x) = supα∈A fα(x) and fα(x) is convex in x for every α, then ∂fβ(x) ∈ ∂f if β = arg supα∈A fα(x). This is equivalent to computing a gradient descent update for pg at the optimal D given the corresponding G. supD U(pg, D) is convex in pg with a unique global optima as proven in Thm 1, therefore with sufficiently small updates of pg, pg converges to px, concluding the proof.

In practice, adversarial nets represent a limited family of pg distributions via the function G(z; θg), and we optimize θg rather than pg itself. Using a multilayer perceptron to define G introduces multiple critical points in parameter space. However, the excellent performance of multilayer perceptrons in practice suggests that they are a reasonable model to use despite their lack of theoretical guarantees.

**Advantages and disadvantages**

The disadvantages are primarily that there is no explicit representation of pg(x), and that D must be synchronized well with G during training (in particular, G must not be trained too much without updating D, in order to avoid “the Helvetica scenario” in which G collapses too many values of z to the same value of x to have enough diversity to model pdata).

The advantages are that Markov chains are never needed, only backprop is used to obtain gradients, no inference is needed during learning, and a wide variety of functions can be incorporated into the model.

The aforementioned advantages are primarily computational. Adversarial models may also gain some statistical advantage from the generator network not being updated directly with data examples, but only with gradients flowing through the discriminator. This means that components of the input are not copied directly into the generator’s parameters.

Another advantage of adversarial networks is that they can represent very sharp, even degenerate distributions, while methods based on Markov chains require that the distribution be somewhat blurry in order for the chains to be able to mix between modes.

Successful generative modeling provides an alternative and potentially more domain-specific approach for data augmentation.

In complex domains or domains with a limited amount of data, generative modeling provides a path towards more training for modeling. GANs have seen much success in this use case in domains such as deep reinforcement learning.

. Among these reasons, he highlights GANs’ successful ability to model high-dimensional data, handle missing data, and the capacity of GANs to provide multi-modal outputs or multiple plausible answers. Perhaps the most compelling application of GANs is in conditional GANs for tasks that require the generation of new examples.

**>** Image Super-Resolution. The ability to generate high-resolution versions of input images.

**>** Creating Art. The ability to create new and artistic images, sketches, painting, and more.

**>**  Image-to-Image Translation. The ability to translate photographs across domains, such as day to night, summer to winter, and more.

Perhaps the most compelling reason that GANs are widely studied, developed, and used is because of their success. GANs have been able to generate photos so realistic that humans are unable to tell that they are of objects, scenes, and people that do not exist in real life. Astonishing is not a sufficient adjective for their capability and success.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**DCGAN**

UNSUPERVISED REPRESENTATION LEARNING WITH DEEP CONVOLUTIONAL

GENERATIVE ADVERSARIAL NETWORKS

In this paper, we make the following contributions

• We propose and evaluate a set of constraints on the architectural topology of Convolutional GANs that make them stable to train in most settings. We name this class of architectures Deep Convolutional GANs (DCGAN)

• We use the trained discriminators for image classification tasks, showing competitive performance with other unsupervised algorithms.

• We visualize the filters learnt by GANs and empirically show that specific filters have learned to draw specific objects.

• We show that the generators have interesting vector arithmetic properties allowing for easy manipulation of many semantic qualities of generated samples.

*We also encountered difficulties attempting to scale GANs using CNN architectures commonly used in the supervised literature. However, after extensive model exploration* ***we identified a family of architectures that resulted in stable training across a range of datasets and allowed for training higher resolution and deeper generative models.***

**ARCHITECTURE**

**The first** is the all convolutional net (Springenberg et al., 2014) which replaces deterministic spatial pooling functions (such as maxpooling) with strided convolutions, allowing the network to learn its own spatial downsampling. We use this approach in our generator, allowing it to learn its own spatial upsampling, and discriminator.

**Second** is the trend towards eliminating fully connected layers on top of convolutional features. The strongest example of this is global average pooling which has been utilized in state of the art image classification models (Mordvintsev et al.). We found global average pooling increased model stability but hurt convergence speed. A middle ground of directly connecting the highest convolutional features to the input and output respectively of the generator and discriminator worked well. The first layer of the GAN, which takes a uniform noise distribution Z as input, could be called fully connected as it is just a matrix multiplication, but the result is reshaped into a 4-dimensional tensor and used as the start of the convolution stack. For the discriminator, the last convolution layer is flattened and then fed into a single sigmoid output. See Fig. 1 for a visualization of an example model architecture.

**Third** is Batch Normalization (Ioffe & Szegedy, 2015) which stabilizes learning by normalizing the input to each unit to have zero mean and unit variance. This helps deal with training problems that arise due to poor initialization and helps gradient flow in deeper models. This proved critical to get deep generators to begin learning, preventing the generator from collapsing all samples to a single point which is a common failure mode observed in GANs. Directly applying batchnorm to all layers however, resulted in sample oscillation and model instability. This was avoided by not applying batchnorm to the generator output layer and the discriminator input layer.

The ReLU activation (Nair & Hinton, 2010) is used in the generator with the exception of the output layer which uses the Tanh function. We observed that using a bounded activation allowed the model to learn more quickly to saturate and cover the color space of the training distribution. Within the discriminator we found the leaky rectified activation (Maas et al., 2013) (Xu et al., 2015) to work well, especially for higher resolution modeling. This is in contrast to the original GAN paper, which used the maxout activation (Goodfellow et al., 2013).

**Architecture guidelines for stable Deep Convolutional GANs**

• Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).

• Use batchnorm in both the generator and the discriminator.

• Remove fully connected hidden layers for deeper architectures.

• Use ReLU activation in generator for all layers except for the output, which uses Tanh.

• Use LeakyReLU activation in the discriminator for all layers.

**ACTIVATION FUNCTIONS**

1). **SIGMOID FUNCTION:** Sigmoid function is the logistic function , which maps any real value to the range (0 to 1)

A sigmoid function placed as the last layer of a DL model can serve to convert the model’s output into a probability score.

Sigmoid(x) = 1 / ( 1 + e- x )

Input value = x Sigmoid (x)

1. 0.5

+ infinity 1.0

- infinity 0.0

Graph

2). **tanh function** (Hyper tangent function )

tanh (x) = (ex – e-x) / ( ex + e-x )

Range of tanh function is -1 to +1

Input value = x tanh (x)

0 0.0

+ infinity 1.0

- infinity -1.0

Graph

The ReLU function has a constant gradient of 1 whereas a sigmoid/tanh function has a gradient that rapidly converges towards zero. This property makes Neural Network with sigmoid activation function slow to train .This phenomenon is known as the **vanishing gradient problem** .The choice of ReLU as activation function alleviates the problem because the gradient of the ReLU is always ‘1’ for positive x and so the learning process will not be slow down by gradient becoming small. Hence sigmoid/tanh function is used in the last layer and ReLU is used in hidden layers .

3). **ReLU** (Rectified Linear Units)

f(x) = max (x , 0)

f(x) = 0 for x<0

x for x>=0

f(x) range is , (0 , infinity)

first derivative of f(x) is f’(x) = 0 for x<0

1 for x>=0

It is non-zero centred

It is non differentiable at zero

It has dying ReLU problem ( because for x<0 f(x) =0 ) where some ReLU neurons essentially die for all inputs and remain inactive no matter what input is supplied , here no gradient flows and if large number of dead neurons are there in a Neural Network it’s performance is affected . This dying ReLU problem can be corrected by LeakyReLU

graph

4).**LeakyReLU**

In LeakyReLU for all inputs x<0 , the outputs are slightly descending (this is obtained by introducing small slope value for negative input value) , these small numbers reduce the death of ReLU activated neurons

f(x) = ax for x<o

x for x>=0

first derivative of f(x) is f’(x) = a for x<0

1 for x>=0

LeakyReLU range is ( - infinity to infinity )

Graph

**OPTIMISATION**

**1**).**Stochastic Gradient Distance(SGD)**

Ɵt = Ɵt-1 - ƞ.ƟJ ( Ɵ ; x(i) , y(i) )

Ɵ ---- weight or bias parameter

Ƞ----- learning rate

J( Ɵ ; x(i) ,y(i) ) -------objective function

ƟJ ( Ɵ ; x(i) , y(i) ) --------- first derivative of objective function w.r.t. Ɵ

**2). Momentum**

νt = ϒνt-1 + ƞ. ƟJ ( Ɵ )

Ɵ = Ɵ – νt

ϒ ------ momentum , it’s value is < 1 , usually 0.9

νt-1 ----- update vector of (t-1) step

**3).RMS prop**

E[g2]t = ϒ E[g2]t-1 + (1 – ϒ ) gt2

E(g2)t -------- running average at time step t

ϒ = 0.9 , ƞ = 0.001

gt,i = ƟJ ( Ɵt , i)

Ɵt+1 = Ɵt – ƞ / (E[g2]t + ε )1/2 . gt

= Ɵt – ƞ /( RMS[g]t) . gt

Instead of accumulating all past squared gradients , RMSprop / Adadelta restricts the window of accumulated past gradient to some size w

**4).Adam (**Adaptive Moment Estimation )

Adam computes adaptive learning rates for each parameter . In addition to storing an exponentially decaying average of past squared gradients νt like Adadelta and RMSprop , Adam also keeps an exponentially decaying average of past gradients mt ,similar to momentum .

1) Decaying averages of past gradient

mt = β1 mt-1 + (1 – β1 ) gt

mt ------ estimates of first moment (the mean ) of the gradients

2) decaying averages of past squared gradients

νt = β2νt-1 + ( 1 – β2 ) gt2

νt -------- estimates of the second moment ( the un-centred variance ) of the gradients

Ɵt+1 = Ɵt – ƞ / ( νt + ε )1/2 . mt

Default values of β1 = 0.9 , β2 = 0.999 , ε = 10-8

**BatchNormalization:**

BatchNormalization normalizes the output of each layer before passing it through the next layer . This speeds up training and appears to lead to better results .

**Dropout:**

**DETAILS OF ADVERSARIAL TRAINING**

We trained DCGANs on three datasets, Large-scale Scene Understanding (LSUN) (Yu et al., 2015), Imagenet-1k and a newly assembled Faces dataset. Details on the usage of each of these datasets are given below.

No pre-processing was applied to training images besides scaling to the range of the tanh activation function [-1, 1].

All models were trained with mini-batch stochastic gradient descent (SGD) with a mini-batch size of 128.

All weights were initialized from a zero-centred Normal distribution with standard deviation 0.02.

In the LeakyReLU, the slope of the leak was set to 0.2 in all models.

While previous GAN work has used momentum to accelerate training, we used the Adam optimizer (Kingma & Ba, 2014) with tuned hyperparameters.

We found the suggested learning rate of 0.001, to be too high, using 0.0002 instead.

Additionally, we found leaving the momentum term β1 at the suggested value of 0.9 resulted in training oscillation and instability while reducing it to 0.5 helped stabilize training.

**EMPIRICAL VALIDATION OF DCGANS CAPABILITIES**(not yet completed ,need to read some stuff to understand)

**WALKING IN THE LATENT SPACE**

The first experiment we did was to understand the landscape of the latent space. Walking on the manifold that is learnt can usually tell us about signs of memorization (if there are sharp transitions) and about the way in which the space is hierarchically collapsed. If walking in this latent space results in semantic changes to the image generations (such as objects being added and removed), we can reason that the model has learned relevant and interesting representations. The results are shown in Fig.4.

**VISUALIZING THE DISCRIMINATOR FEATURES**

Previous work has demonstrated that supervised training of CNNs on large image datasets results in very powerful learned features (Zeiler & Fergus, 2014). Additionally, supervised CNNs trained on scene classification learn object detectors (Oquab et al., 2014). We demonstrate that an unsupervised DCGAN trained on a large image dataset can also learn a hierarchy of features that are interesting. Using guided backpropagation as proposed by (Springenberg et al., 2014), we show in Fig.5 that the features learnt by the discriminator activate on typical parts of a bedroom, like beds and windows. For comparison, in the same figure, we give a baseline for randomly initialized features that are not activated on anything that is semantically relevant or interesting.

**MANIPULATING THE GENERATOR REPRESENTATION**

**FORGETTING TO DRAW CERTAIN OBJECTS**

In addition to the representations learnt by a discriminator, there is the question of what representations the generator learns. The quality of samples suggest that the generator learns specific object representations for major scene components such as beds, windows, lamps, doors, and miscellaneous furniture. In order to explore the form that these representations take, we conducted an experiment to attempt to remove windows from the generator completely.

On 150 samples, 52 window bounding boxes were drawn manually. On the second highest convolution layer features, logistic regression was fit to predict whether a feature activation was on a window (or not), by using the criterion that activations inside the drawn bounding boxes are positives and random samples from the same images are negatives. Using this simple model, all feature maps with weights greater than zero ( 200 in total) were dropped from all spatial locations. Then, random new samples were generated with and without the feature map removal.

The generated images with and without the window dropout are shown in Fig.6, and interestingly, the network mostly forgets to draw windows in the bedrooms, replacing them with other objects.

**VECTOR ARITHMETIC ON FACE SAMPLES**

In the context of evaluating learned representations of words (Mikolov et al., 2013) demonstrated that simple arithmetic operations revealed rich linear structure in representation space. One canonical example demonstrated that the vector(”King”) - vector(”Man”) + vector(”Woman”) resulted in a vector whose nearest neighbor was the vector for Queen. We investigated whether similar structure emerges in the Z representation of our generators. We performed similar arithmetic on the Z vectors of sets of exemplar samples for visual concepts. Experiments working on only single samples per concept were unstable, but averaging the Z vector for three examplars showed consistent and stable generations that semantically obeyed the arithmetic. In addition to the object manipulation shown in (Fig. 7), we demonstrate that face pose is also modeled linearly in Z space (Fig. 8).

These demonstrations suggest interesting applications can be developed using Z representations learned by our models. It has been previously demonstrated that conditional generative models can learn to convincingly model object attributes like scale, rotation, and position (Dosovitskiy et al., 2014). This is to our knowledge the first demonstration of this occurring in purely unsupervised

models. Further exploring and developing the above mentioned vector arithmetic could dramatically reduce the amount of data needed for conditional generative modeling of complex image distributions.

**GAN Failure Modes**

1. **Stable GAN**

1. Discriminator loss on real and fake images is expected to sit around 0.5.

2. Generator loss on fake images is expected to sit between 0.5 and perhaps 2.0.

3. Discriminator accuracy on real and fake images is expected to sit around 80%.

4. Variance of generator and discriminator loss is expected to remain modest.

5. The generator is expected to produce its highest quality images during a period of stability. 6.Training stability may degenerate into periods of high-variance loss and corresponding lower quality generated images.

1. **Mode COLLAPSE**

Mode collapse, also known as the scenario, is a problem that occurs when the generator learns to map several different input z values to the same output point.

A mode collapse can be identified when reviewing a large sample of generated images. The images will show low diversity, with the same identical image or same small subset of identical images repeating many times.

A mode collapse can also be identified by reviewing the line plot of model loss. The line plot will show oscillations in the loss over time, most notably in the generator model, as the generator model is updated and jumps from generating one mode to another model that has different loss.

We can impair our stable GAN to suffer mode collapse a number of ways. Perhaps the most reliable is to restrict the size of the latent dimension directly, forcing the model to only generate a small subset of plausible outputs.

1. The loss for the generator, and probably the discriminator, is expected to oscillate over time.

2. The generator model is expected to generate identical output images from different points in the latent space.

A mode collapse is less common during training given the findings from the **DCGAN** model architecture and training configuration.

1. **Convergence Failure**

Perhaps the most common failure when training a GAN is a failure to converge. Typically, a neural network fails to converge when the model loss does not settle down during the training process. In the case of a GAN, a failure to converge refers to not finding an equilibrium between the discriminator and the generator. The likely way that you will identify this type of failure is that the loss for the discriminator has gone to zero or close to zero. In some cases, the loss of the generator may also rise and continue to rise over the same period.

This type of loss is most commonly caused by the generator outputting garbage images that the discriminator can easily identify. This type of failure might happen at the beginning of the

run and continue throughout training, at which point you should halt the training process. For some unstable GANs, it is possible for the GAN to fall into this failure mode for a number of batch updates, or even a number of epochs, and then recover.

There are many ways to impair our stable GAN to achieve a convergence failure, such as changing one or both models to have insufficient capacity, changing the Adam optimization algorithm to be too aggressive, and using very large or very small kernel sizes in the models.

The properties of a convergence failure

1. The loss for the discriminator is expected to rapidly decrease to a value close to zero where it remains during training.

2.The loss for the generator is expected to either decrease to zero or continually increase during training.

3.The generator is expected to produce extremely low-quality images that are easily identified as fake by the discriminator.

**GAN Evaluation**

1. Manual GAN Generator Evaluation

Many GAN practitioners fall back to the evaluation of GAN generators via the manual assessment of images synthesized by a generator model. This involves using the generator model to create a batch of synthetic images, then evaluating the quality and diversity of the images in relation to the target domain

1. It is subjective, including biases of the reviewer about the model, its configuration, and the project objective.

2. It requires knowledge of what is realistic and what is not for the target domain.

3. It is limited to the number of images that can be reviewed in a reasonable time. …..evaluating the quality of generated images with human vision is expensive and cumbersome, biased [...] difficult to reproduce, and does not fully reflect the capacity of models.

For a thorough survey, see the 2018 paper titled ***Pros and Cons of GAN Evaluation Measures.***

This paper divides GAN generator model evaluation into qualitative and quantitative measures

2.Qualitative GAN Generator Evaluation

This is where human judges are asked to rank or compare examples of real and generated images from the domain.

The Rapid Scene Categorization method is generally the same, although images are presented to human judges for a very limited amount of time, such as a fraction of a second, and classified as real or fake. Images are often presented in pairs and the human judge is asked which image they prefer, e.g. which image is more realistic. A score or rating is determined based on the number of times a specific model generated images that won such tournaments. Variance in the judging is reduced by averaging the ratings across multiple different human judges. This is a labor-intensive exercise, although costs can be lowered by using a crowdsourcing platform like Amazon’s Mechanical Turk, and efficiency can be increased by using a web interface.

One intuitive metric of performance can be obtained by having human annotators judge the visual quality of samples. We automate this process using Amazon Mechanical Turk [...] using the web interface [...] which we use to ask annotators to distinguish between generated data and real data.

A major downside of the approach is that the performance of human judges is not fixed and can improve over time. This is especially the case if they are given feedback, such as clues on how to detect generated images.

By learning from such feedback, annotators are better able to point out the flaws in generated images, giving a more pessimistic quality assessment.

Another popular approach for subjectively summarizing generator performance is Nearest Neighbors. This involves selecting examples of real images from the domain and locating one or more most similar generated images for comparison. Distance measures, such as Euclidean distance between the image pixel data, is often used for selecting the most similar generated images. The nearest neighbor approach is useful to give context for evaluating how realistic the generated images happen to be.

3.Quantitative GAN Generator Evaluation

Quantitative GAN generator evaluation refers to the calculation of specific numerical scores used to summarize the quality of generated images.

**1.Inception Score.(IS)**

High score-----good quality image

Low score-----bad quality image.

**2.Frechet Inception Distance(FID)**

The score summarizes how similar the two groups are in terms of statistics on computer vision features of the raw images calculated using the inception v3 model used for image classification.

Lower scores indicate the two groups of images are more similar, or have more similar statistics, with a perfect score being 0.0 indicating that the two groups of images are identical.

The Frechet Inception Distance summarizes the distance between the Inception feature vectors for real and generated images in the same domain.

**CODE**