Project #1
By: Adam Lieberman

Please visit https://ad1m.github.io/GANs/ to view the blog. This will take you to index.html where you can find Part 1 and Part 2 of the Generative Adversarial Networks tutorial series. If you would like to view part 1 you can go to https://ad1m.github.io/GANs/blog_html_files/main.html . If you would like to view part 2 you can go to https://ad1m.github.io/GANs/blog_html_files/main2.html . The following pages showcase the main page that links you to Part 1 and Part 2, then Part 1 of the tutorial, and then Part 2 of the tutorial. Please note that when converting HTML to PDF the margins and styling may not be preserved. Additionally, some code might be cut out as I have a scrolling code feature on my website. Please view the index.html in the localhost or visit my website (links above) to view my work if possible. If so google chrome works best for my site. For more information please see my README.md. Thank you very much.

Posts

Mar 28, 2017

Part 1: Introduction to Generative Adversarial Networks?

In this lesson you will learn about the history of generative adversarial networks, why they are important, how they work, and some applications they are useful for.

Read More

Mar 30, 2017

Part 2: Building a DCGAN

In this lesson we will build a deep convolutional generative adversarial network in python.

Read More

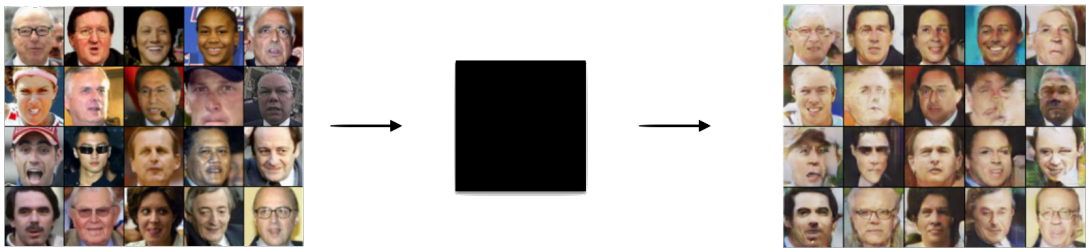
Generative Adversarial Networks

Part 1: Motivation, History, Theory, & Application

By: Adam Lieberman

Motivation

Imagine we have an unlabeled dataset, a collection of facial images, and a black box algorithm. We pass these images into the algorithm and the algorithm is able to learn a probability distribution of the dataset.



Once this probability distribution is learned , the algorithm is able to generate synthetic samples from the dataset that were not part of the original training data. This algorithm will be able to input the facial images and generate synthetic facial images like the real ones used as training data.

We could pass in pictures of all employees in a company and generate the facial image of a new employee. We could pass in images of homes and create new houses. We could even pass in short videos clips and generate synthetic videos. To generate this synthetic data, we can use what is called a Generative Adversarial Network.

> "There are many interesting recent development in deep learning...The most important one, in my opinion, is adversarial training (also called GAN for Generative Adversarial Networks). This, and the variations that are now being proposed is the

> most interesting idea in the last 10 years in ML, in my opinion."
>
> – Yann LeCun

History

In 1992 Artificial Intelligence researcher Jürgen Schmidhuber from the University of Colorado proposed a principle, based on two opposing forces, for unsupervised learning of distributed non-redundant internal representations in his paper Learning Factorial Codes By Predictability Minimization. Here, each hidden unit in a neural network is trained to be different from the output of a second network, which predicts the value of that hidden unit given the value of all of the other hidden units.

In 2014 Ian Goodfellow and his tem proposed a new framework for estimating generative models via an adversarial process in their paper Generative Adversarial Nets. Here, Goodfellow describe a generative model that captures the data distribution which is trained simulatneously with a discriminative model that estimates the probability that a sample came from the training data rather than the the generative model, hence giving rise to Generative Adversarial Networks (GANs).

Goodfellow submitted this paper at the 2014 Neural Information Processing Systems (NIPS) conference. The crowd was very interested, but the following was questioned in the Export Reviews, Discussions, Author Feedback and Meta-Reviews:

> Finally, how is the submission related to the first work on "adversarial" MLPs for modeling data distributions through estimating conditional probabilities, which was called "predictability minimisation" or PM (Schmidhuber, NECO 1992)? The new approach seems similar in many ways. Both approaches use "adversarial" MLPs to estimate certain probabilities and to learn to encode distributions. A difference is that the new system learns to generate a non-trivial distribution in response to statistically independent, random inputs, while good old PM learns to generate statistically independent, random outputs in response to a non-trivial distribution (by extracting mutually independent, factorial features encoding the distribution). Hence the new system essentially inverts the direction of PM - is this the main difference? Should it perhaps be called "inverse PM"?
>
> - Assigned_Reviewer_19

Goodfellow responded with:

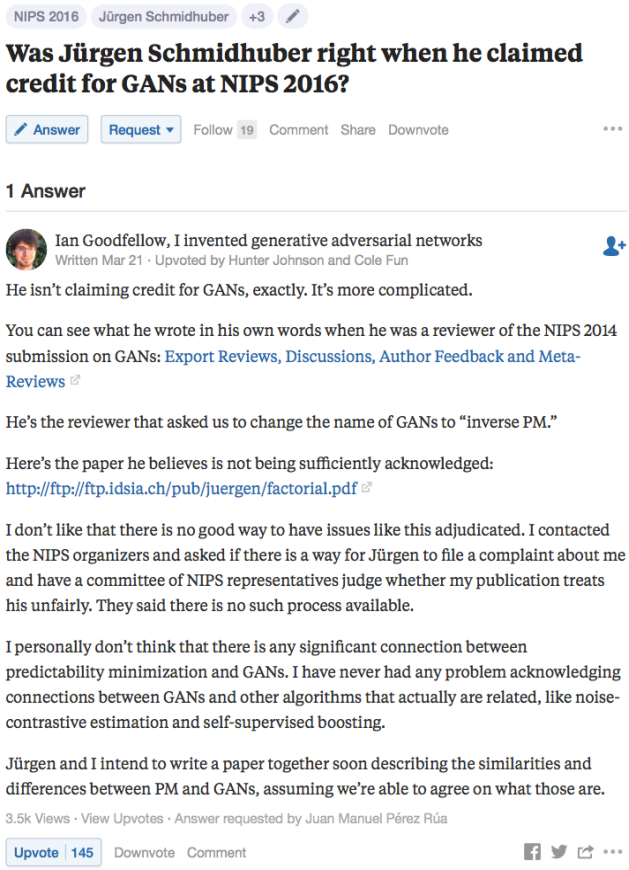
> PM regularizes the hidden units of a generative model to be statistically independent from each other. GANs express no preference about how the hidden units are distributed. We used independent priors in the experiments for this paper, but that's not an essential part of the approach. PM involves competition between two MLPs but the idea of competition is more qualitative than formal--each network has its own objective function qualitatively designed to make them compete, while GANs play a minimax game on a single value function. Nearly all forms of engineering involve competition between two forces of some kind, whether it is mechanical systems exploiting Newton's 3rd law or Boltzmann machines inducing competition between the positive and negative phase of learning.
>
> - Ian Goodfellow

Assigned reviewer 19 was Jürgen Schmidhuber. Schmidhuber believed that his paper and Goodfellow's paper were quite similar, claiming that Generative Adversarial Networks were the inverse of his predictability minimisation. It was clear that Goodfellow did not think the two had much in common. Goodfellow noted that the competition between two forces was not unique to the predictability minimisation paper, that nearly all forms of engineering involve this competition between two forces. Hence he believed that this was not a major factor in claiming the similarity between predictability minimisation and general adversarial networks.

Two years later, at the 2016 NIPS conference, Schmidhuber interrupted the lecture on Generative Adversarial networks being given by Ian Goodfellow. He walked up to the microphone while Goodfellow was speaking and exclaimed that he had previously done very similar work, which was overlooked and not given credit. Goodfellow handled this interruption well, and proceeded with his talk; however, this situation sparked social media interest outside the conference.

Some people took to Goodfellow's side, claiming that Schmidhuber had no right to interupt a conference talk and disturb viewers. Others took to Schmidhuber's side seeing similarity in the two works, believing that Schmidhuber did inspire GANs. Others thought the entire situation was comical and jested about what was going on.

Eventually, a Quora post popped up and Goodfellow responded to it. Here, he tried to explain the complicated nature of the Schmidhuber situation. He noted that Schmidhuber did not claim credit for the invention of GANs, but wanted the name to be changed to inverse PM. He explained that there is no good way to have situations like this mediated and had asked NIPS if Schmidhuber could file a complaint as to whether his publication was unfair to the preceeding Learning Factorial Codes By Predictability Minimization paper. He noted that he did not see any significant connection between the two papers as Schmidhuber's paper was about predictability minimization and his was on generative adversarial networks.



NIPS 2016 Jürgen Schmidhuber +3

Was Jürgen Schmidhuber right when he claimed credit for GANs at NIPS 2016?

Answer Request Follow 19 Comment Share Downvote

1 Answer

Ian Goodfellow, I invented generative adversarial networks
Written Mar 21 · Upvoted by Hunter Johnson and Cole Fun

He isn't claiming credit for GANs, exactly. It's more complicated.

You can see what he wrote in his own words when he was a reviewer of the NIPS 2014 submission on GANs: Export Reviews, Discussions, Author Feedback and Meta-Reviews

He's the reviewer that asked us to change the name of GANs to "inverse PM."

Here's the paper he believes is not being sufficiently acknowledged: http://ftp://ftp.idsia.ch/pub/juergen/factorial.pdf

I don't like that there is no good way to have issues like this adjudicated. I contacted the NIPS organizers and asked if there is a way for Jürgen to file a complaint about me and have a committee of NIPS representatives judge whether my publication treats his unfairly. They said there is no such process available.

I personally don't think that there is any significant connection between predictability minimization and GANs. I have never had any problem acknowledging connections between GANs and other algorithms that actually are related, like noise-contrastive estimation and self-supervised boosting.

Jürgen and I intend to write a paper together soon describing the similarities and differences between PM and GANs, assuming we're able to agree on what those are.

3.5k Views · View Upvotes · Answer requested by Juan Manuel Pérez Rúa

Upvote 145 Downvote Comment

Soon after, Goodfellow revised his paper citing Schmidhuber and explicitly denoting three key differences between the two works:

- With Goodfellow's GANs, the competition between the networks is the sole training criterion, and is sufficient on its own to train the network. Predictability minimization is only a regularizer that encourages the hidden units of a neural network to be sta- tistically independent while they accomplish some other task; it is not a primary training criterion.
- The nature of the competition is different. In predictability minimization, two networks' outputs are compared, with one network trying to make the outputs similar and the other trying to make the 2 outputs different. The output in question is a single scalar. In GANs, one network produces a rich, high dimensional vector that is used as the input to another network, and attempts to choose an input that the other network does not know how to process.
- The specification of the learning process is different. Predictability minimization is described as an optimization problem with an objective function to be minimized, and learning approaches the minimum of the objective function. GANs are based on a minimax game rather than an optimization problem, and have a value function that one agent seeks to maximize and the other seeks to minimize. The game terminates at a saddle point that is a minimum with respect to one player's strategy and a maximum with respect to the other

player's strategy.

Since then, things have cooled off between Goodfellow and Schmidhuber and the two plan to write a paper together. Goodfellow is still researching and presenting on GANs and is planning to showcase his recent work on GANs at the 2017 GPU Technology Conference. Ian Goodfellow is still credited for inventing the General Adversarial Network. Currently some professionals in academia see the work as dissimilar while others find the two works quite similar.

Prerequisites

Before discussing the technicalities of Generative Adversarial Networks, let us briefly cover some prerequisites. If you are comfortable with the field of machine learning, please feel free to skip this section and move onto the section titled "What are Generative Adversarial Networks (GANs)".

Machine Learning:

Machine Learning is a type of Artificial Intelligence that provides computers with the ability to learn without being explicitly programmed. It is a method of teaching computers to make and improve predictions or behaviors based on some data. In this field, we explore the study and construction of algorithms that can learn from and make predictions on this data. Machine learning tasks are classified into three broad categories:

- **Supervised Learning:** Our data has input variables, x, and output variables, y. Here we use an algorithm to learn the mapping from the input function to the output in the form $y = f(x)$. We want to approximate the mapping function so well such that when we query a new input x sample we can predict the output y for that sample. When we train the algorithm our data essentially has a teacher to supervise the learning process.
- **Unsupervised Learning:** Our data only has an input x and no corresponding output label y. Here, we want to model the structure or distribution in the data to learn more from it. Unsupervied algorithms have to discover and present the structure in the data without a teacher.
- **Semi-Supervised Learning:** Here, we have some input x data that has associated output y labels and some x data that does not. For instance, we might have a collection of photos where some photos have a caption and some do not.
- **Reinforcement Learning:** Here our program interacts with a dynamic environment in which we must perform a certain goal. The program is given feedbak in terms of rewards

and punishments. For instance, we might try and navigate a robot through a maze. When training it we punish it when it makes an incorrect turn and reward it when it makes a correct turn.

We can further categorize machine learning tasks considering the output of the system:

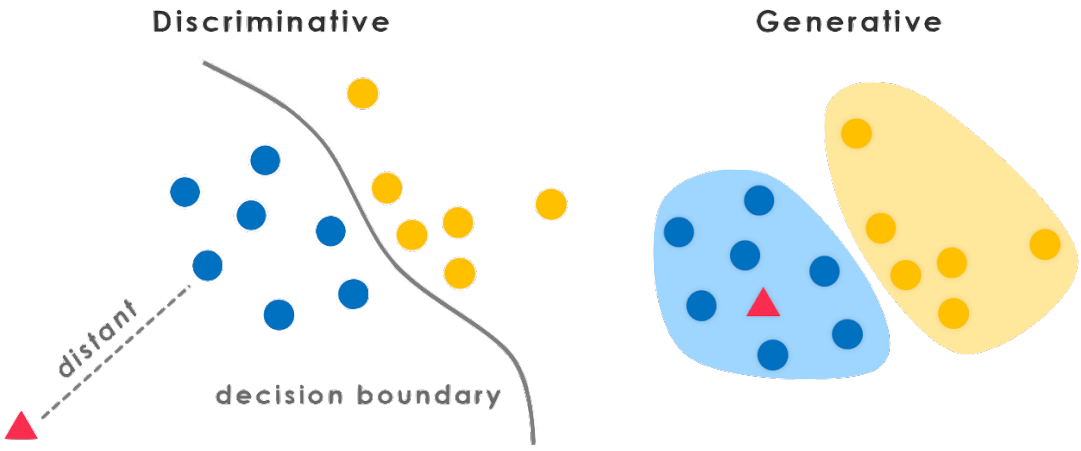
- **Classification:** The output variable is from a class. For instance the output variable could be round, square, or triangular.
- **Regression:** The output variable is a real number.
- **Clustering:** We seek to discover groupings in the data. For instance we want to cluster customers based on purchasing data that we have.
- **Density Estimation:** Constructing an estimate based on observed data of an unobservable probability density function.
- **Dimensionality Reduction:** We want to reduce the dimensionality of our dataset by reducing the number of random variables under consideration. However, we want to preserve it's true predictive nature with the reduced feature representation.

Generative vs Discriminative Models:

Generative models model how the data was generated in order to categorize a signal. This type of model cares how data was generated to categorize that signal. It tries to answer, "Which category is most likely to generate the signal based on generation assumptions?" Here, the distribution of individual classes is modeled. The generative model concerns the specification of the joint probability $p(x, y)$. The data X and label Y are taken into a model and we learn $p(x|y)$ and $p(y)$. We can then learn $p(y|x)$ indirectly as we know that $p(y|x) \propto p(x|y)p(y)$. Some examples of generative models are Gaussian Mixture Models, Hidden Markov Models, Naive Bayes, Latenet Dirichlet Allocation, and the Restricted Boltzmann Machine.

Discriminative models do not care about how the data was generated. They simply categorize a given signal. Here, the boundary between classes is learned. A discriminative model concerns the specification of the conditional probability $p(y|x)$. This conditional probability is learned directly. Some exampls of discriminative models are Logistic Regression, Support Vector Machines, Boosting, Conditional Random Fields, Linear Regression, Neural Networks, and Random Forests.

We can better observe these two types of models with the figure below:



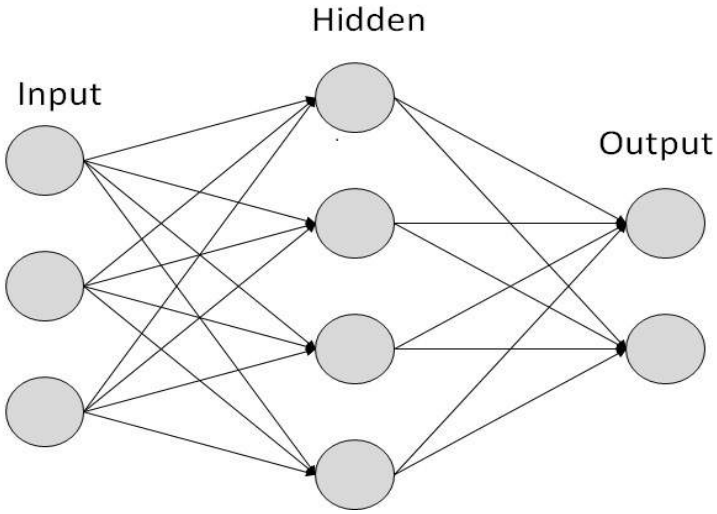
There is a simple test to tell whether a model is generative or discriminative. If the model can generate a new set of training data, including all features and labels, given the condition that all unknown parameters in the model are known then the model is generative. Otherwise, it is discriminative. If we look at a Naive Bayes model we see that it can generate feature data and label data. Support Vector Machiens, on the otherhand, cannot generate data in any feature space.

Deep Learning:

Deep learning is a subfield of machine learning that is concerned with algorithms, like Artificial Neural Networks (ANNs), which is inspired by the structure and function of the brain. These ANNs are generally presented as systems of interconnected neurons which exchange messages between each other. The neural network model mimics a neuron which contains the dendrites, nucleus, axon, and terminal axon. The neurons transmit information via synapse between the dendrites of one terminal and the axon of another. Computer scientists inherited this idea from the brain starting in the 1940's. However, during that time, computing power was limited and expensive and thus neural networks did not showcase their power. Recently, in 2011, neural networks resurfaced as computing power became stronger and cheaper.

Neural Networks

Neural Networks are the focus of deep learning. An artificial neural network looks as follows:

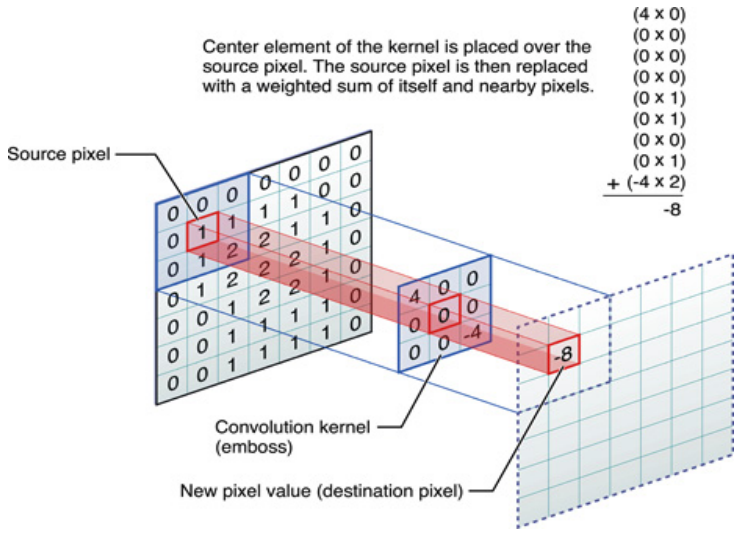


Above we have an input layer, the hidden layer, and the output layer. The circles are our nodes, or neurons. The lines connecting them are the weights and information being passed along. If we have a single hidden layer, we have an artifical neural network. If we have more than one hidden layer then we have a deep neural network. Our input data is randomly weighted and then passed into the hidden layer. Here the weights are summed and an activation function like the sigmoid activation is applied on the sum. We feed forward through the neural network from the input to the hidden layers to the output. We then go backwards and begin adjusting the weights to minimize our loss function. This is called forward/backward propogation.

Convolutional Neural Networks

A convolutional neural network is a neural network comprised of one or more convolutional layers and followed by one or more pooling layers and fully connected layers.

Convolutional layers essentially take a weighted sliding window over the input to create a new output. We see an example as follows:

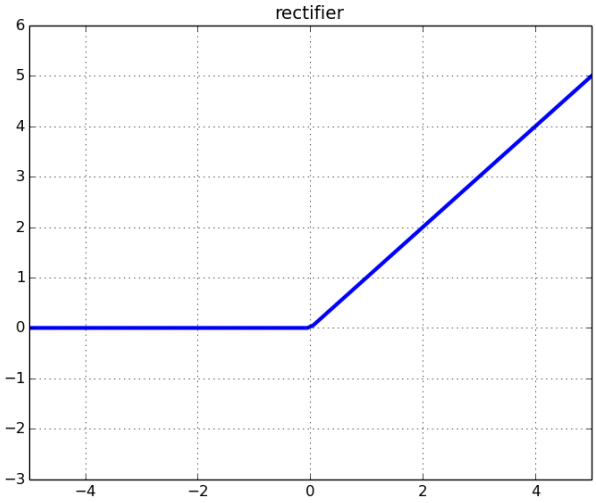


Here we take a 3 x 3 sliding window over our 7 x 7 matrix and perform a weighted sum to generate a new value. We slide this window over all locations in the matrix.

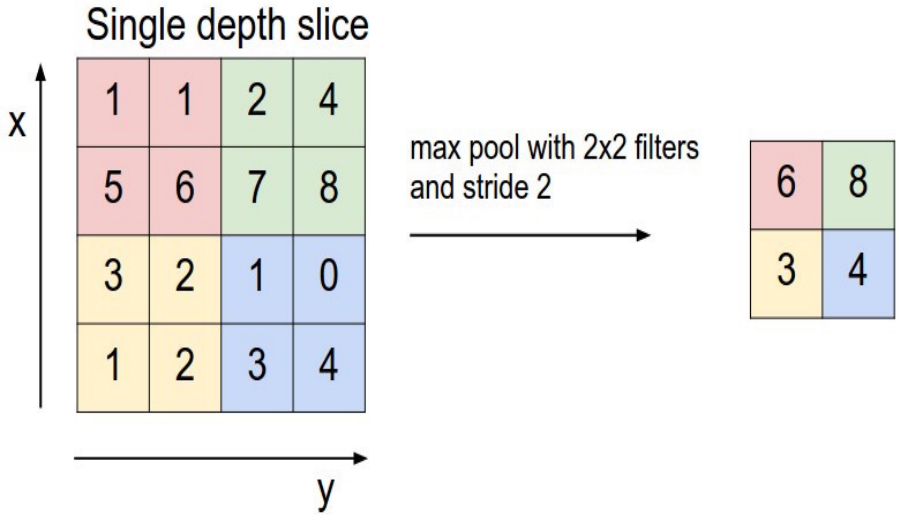
In a rectified linear unit (RELU) layer any input value less than zero is set to zero and any value greater than or equal to zero retains its value. A RELU layer takes the function

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

This activation function will be applied in an elementwise fashion. It is graphically depicted as follows:



A pooling layer's function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. We see an example of max pooling as follows:



Here we divide a 4 x 4 matrix up into 2 x 2 cells and take the maximum value from each cell.

In a fully connected layer, after several convolutional and max pooling layers, all neurons in the previous layer (whether it be fully connected, pooling, or convolutional) are connected to to every single neuron it has. Fully connected layers are not spatially located anymore (you can visualize them as one-dimensional), so there can be no convolutional layers after a fully connected layer. The output from the convolutional layer represents high-level features in the data. This output could be flattened and connected to the output layer, but adding a fully-connected layer is usually a cheap way of learning non-linear combinations of the features. So the convolutional layers provide a meaningful low-dimensional invariant feature space and the fully connected layer is learning a possibly non-linear function in that space.

An example of a convolutional neural network architecture would be

- Input
- Convolutional Layer
- Max Pooling Layer
- Convolutional Layer
- Max Pooling Layer
- Fully Connected Layer

Batch Normalization

In 2015 Sergey Ioffe and Christian Szegedy released a paper called Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Training a deep neural network is complicated by the fact that the distribution of each layer's inputs changes during training as the parameters of the previous layers change. This slows down training by requiring lower learning rates and careful parameter initialization. This phenomenon is referred to as internal covariate shift.

Batch normalization is a technique to provide any layer in a neural network with inputs that are zero mean/unit variance. The mean and variance are measured over the whole mini-batch, independently for each activation. A learned offset β and multiplicative factor γ are then applied. This allows us to use much higher learning rates and be less careful about initialization. It also prevents the vanishing gradient, makes it easier to get out of local minima, and acts as a regularizer, in some cases eliminating the need for a dropout layer. We can observe the formal algorithm for the batch normalization transform applied to activation x over a mini-batch from the paper below:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Zero-Sum Game

In game theory, a zero-sum game is a mathematical representation of a situation in which each participant's gain or loss of utility is exactly balanced by the losses or gains of the utility of the other participants. Here, if the total gains of the participants are added up and the total losses are subtracted, the sum will always be zero. So, zero-sum games are basically situations where if one person wins the other participant or participants lose. This is sometimes called a win lose or lose win situation.

For instance, say Mike and Fred bet on the outcome of a tennis match. There is no draw in tennis so one of the betters must win and one of them must lose. Here, we see that tennis itself is a zero-sum game. The bet between the two participants is also a zero-sum situation. If Mike wins then Fred loses. If Fred wins then Mike loses.

Nash Equilibrium

Nash Equilibrium is a concept of game theory where the optimal outcome of a game is one where no player has an incentive to deviate from his chosen strategy after considering an opponent's choice. In Nash Equilibrium, each player's strategy is optimal when considering the decisions of other players. Every player wins because everyone gets the outcome they desire.

We now have a solid grasp of some key concepts that will appear in Generative Adversarial Networks. Let us now proceed to talk about GANs.

What are Generative Adversarial Networks (GANs)?

Why Are GANs Exciting:

Take a look at the image below:

Within fractions of a second, the human brain can realize that this image is a sofa. If a human is asked to draw a sofa, they easily can. Ask a computer what this image is and it will have no idea. To a computer, this image of a sofa is a matrix of numbers that has color values for each picture. Ask a computer to draw you a sofa and you will most likely not be presented with an image of a sofa. The computer does not understand what is actually in the images. However, what if we showed the computer tens of thousands of images of sofas. Then, the computer might be able to generate a sofa when asked to do so. With generative adversarial networks we can feed thousands and thousands of (sofa) images into the model and train the computer to understand the concepts without explicitly teaching them the semantics of these concepts. This is exciting researchers as it is a huge leap from what current systems are doing.

The GAN Framework:

Generative Adversarial Networks are a branch of unsupervised machine learning where two neural networks compete against each other in a zero-sum game framework. We essentially have a game between two players: the generator and the discriminator. The generator creates samples that come from the same distribution as the training data. The generator is generating "fake" or "synthetic" data. The discriminator examines the synthetic generated samples and the

real samples and tries to determine whether they are real or fake. The generator neural network is trained to produce fake data that better fools the discriminator neural network. The discriminator neural network is trained to better distinguish real data from fake data. Here the two networks control each others loss functions. An equilibrium occurs if the first network learns to perfectly model the true distribution. At this point, the discriminator can do no better than chance at predicting whether the data is fake or real. It is important to note that GANs are generative models. They take the training set, consisting of samples drawn from the distribution p_{data} and learn to represent an estimate of that distribution, p_{model}. GANs then generate samples from p_{model}.

Simplified GAN Framework:

Let's look at a simple example to better understand GANs. The basic idea is that we have two neural networks and we make them fight against each other so that they both become stronger. The first neural network is called the discriminator and second neural network is called the generator. Let's say that the discriminator is a police officer and the generator is a criminal. The criminal is trying to counterfit money and the police officer is trying to decide whether the money is real or fake.

You might be looking at the above image and be like, "hmmm this is strange, why are the discriminator and generator both childen?" Well, at first the discriminator is a brand new police officer. This police officer at first does not know what counterfit money looks like. The generator is a brand new counterfitter and has not yet mastered any skills to generate fake money. At first

the generator might generate some fake money like this:

We look at the above and are like "Wow that is a terrible fake!" However, the discriminator knows nothing about fake money and is a beginner just like the counterfitter, so he thinks it's real and lets it pass. We now tell the discriminator, "No that is fake money." We show him an image of real money and he now looks for details in this image so that he can tell real money from fake money in the future. He might now realize that real money has numbers in all the corners. The generator keeps generating fake money that looks like the money above, but it is all getting rejected as fake. Now, we tell the generator that the discriminator knows there needs to be numbers at all the corners to trick the discriminator. The counterfitter now starts to create bills like:

Now, the discriminator is fooled again and starts accepting these bills. The discriminator will again look for new ways to tell if a bill is fake. Now, it might find for instance that a bill needs a face on it. The generator will keep generating images, just like above, and the discriminator now will recognizing these as fake. The generator's loss will increase and it will find a way to make better fake images. The entire process will repeat until both the generator and discriminator are experts.

Expert Discriminator

Expert Generator

Once an expert, the generator will be generating images like:

The discriminator and generator eventually become experts and the discriminator is looking for the tiniest details while the generator is generating increadible counterfits. Now the images that are being generated should impress the human eye.

Mathematical GAN Framework:

Now that we have simplified the understanding of GANs, let us talk about the mathematics behind GANs.

Definition:

Let us represent the discriminator as a function D and the generator as the function G. D takes in x as an input and uses $\theta^{(D)}$ as parameters. G takes in z as input and uses $\theta^{(G)}$ as parameters. Here we let x, z both be latent variables.

Cost Functions:

The discriminator and the generator both have cost functions. The discriminator wants to minimize $J^{(D)}\left(\theta^{(D)}, \theta^{(G)}\right)$ and must do so while only controlling $\theta^{(D)}$. The generator wants to minimize $J^{(G)}\left(\theta^{(D)}, \theta^{(G)}\right)$ and must do so while only controlling $\theta^{(G)}$. Each player's cost function depends on the other's parameters, but each player cannot control the other's parameters. The solution is represented as a Nash Equilibrium, $\left(\theta^{(D)}, \theta^{(G)}\right)$ which is a local minimum of $J^{(D)}$ with respect to $\theta^{(D)}$ and a local minimum of $J^{(G)}$ with respect to $\theta^{(G)}$.

Let us be more clear about the cost functions. The cost function used for the discriminator is:

$$J^{(D)}\left(\theta^{(D)}, \theta^{(G)}\right) = -\frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}}\log D(x) - \frac{1}{2}\mathbb{E}_z \log(1 - D(G(z)))$$

This is similar to the standard cross entropy cost that is minimized when training a standard binary classifier with a sigmoid output.

We know that we have a zero-sum game so the sum of all player's costs must equate to zero. Thus, we cans tate the cost function of the generator as:

$$J^{(G)} = -J^{(D)}$$

We see this is the case if we are using the case of minimax. If, for instance, we are in a heuristic non-saturating game then the equation above does not perform well. Instead, the cost function becomes:

$$J^{(G)} = -\frac{1}{2}\mathbb{E}_z \log D(G(z))$$

In this game, the generator maximizes the log probability of the discriminator being mistaken. In the case of the minimax game, the generator minimizes the log-probability of the discriminator being correct.

GAN Training:

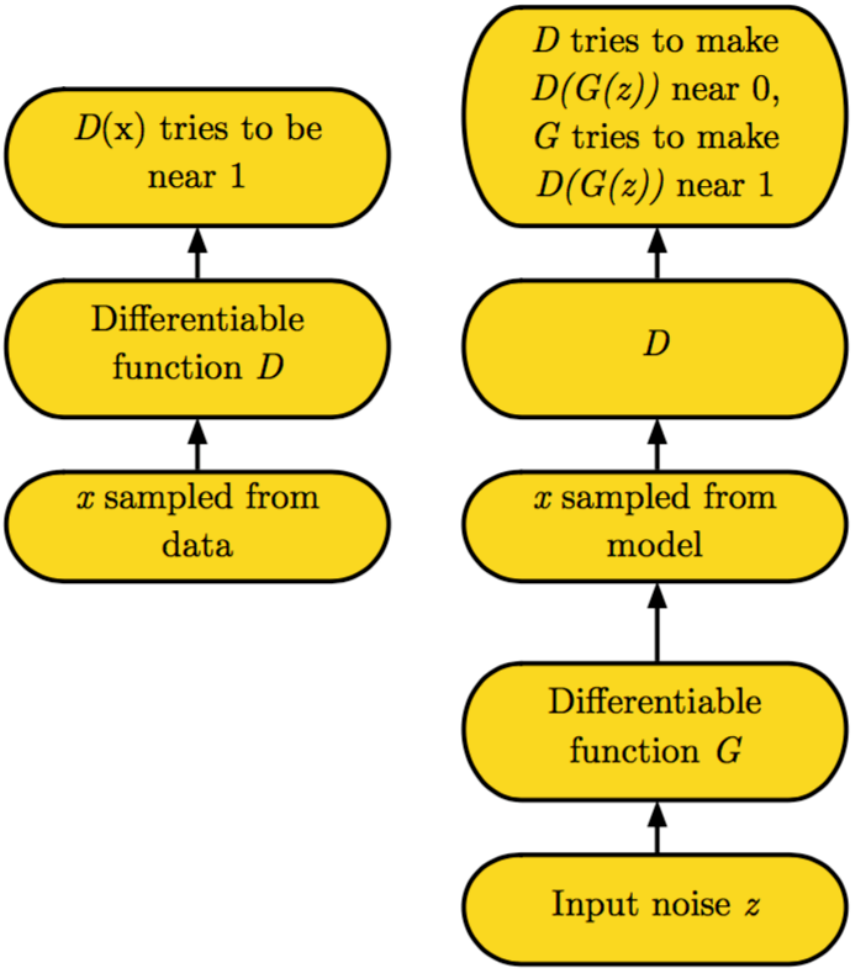
When the generative adversarial network is trained, there is a simultaneous process of stochastic gradient descent (other papers have explored other methods). With each step a minibatch of x values from the dataset and a minibatch of z values from the model's proior over latent variables are sampled. Then, two gradient steps are made at the same time. $\theta^{(D)}$ is updated to reduce $J^{(D)}$ and $\theta^{(G)}$ is updated to reduce $J^{(G)}$.

GAN Pipeline:

As mentioned previously there are two scenarios in our game.

1. x training examples are randomly sampled from the training set and used as input for the discriminator, which is a differentiable function D. The discriminator outputs a probability that x is real or fake, which is denoted as $D(x)$. The goal here is for $D(x) = 1$ or as close to 1 as possible.
2. Input noise z is passed into the generator randomly (noise usually comes from a random uniform distribution) from the model's prior over latenet variables. The generator G creates a fake sample denoted as $G(z)$. This $G(z)$ is then passed to the discriminator. The generator wants $D(G(z))$ to be near 1 while the discriminator tries to make $D(G(z))$ near 0.

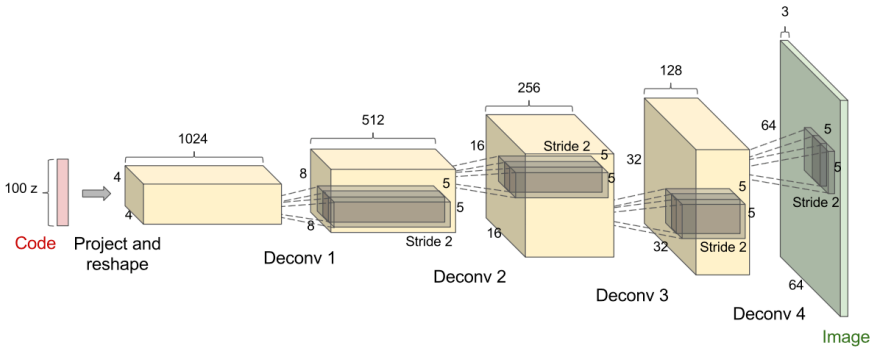
Ian Goodfellow, in his NIPS 2016 Generative Adversarial Networks Tutorial offers a nice diagram of these two game scenarios:



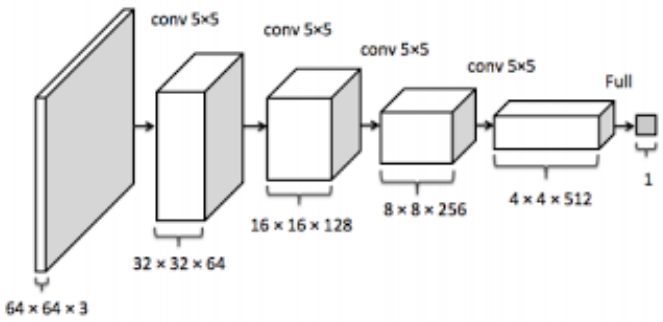
The Nash equilibrium of this game corresponds to $G(z)$ being drawn from the same distribution as the training data and $D(x) = 0.5 \; \forall \; x$.

Deep Convolutional Generative Adversarial Networks (DCGANs):

In 2016 Alec Radford, Luke Metz, and Soumith Chintala in 2016 submitted Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks at the International Conference on Learning Representations (ICLR). For computer vision tasks, deep convolutional neural networks have had great success. In this paper, Radford proposed the following architecture:



In this specific diagram, 100 random numbers from a uniform distribution (latent variables) are passed in and a 64 x 64 x 3 image is produced. The yellow networks in the above image are deconvolutional layers, the reverse of convolutional layers, and the green layer is a fully connected layer. The above image serves as the paper's implementation of the generator. For the discriminator, a 64 x 64 x 3 image is taken as input and passed through 3 conovlutional layers and then a fully connected layer. A probability is outputted of the image being real or fake. This discriminator architecture, not shown in the original paper, would look as follows:



Radford provides the following architecture guidelines in the conference paper:

- 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.

Radford proposes that this architectural topology of DCGANs makes them stable to train in most settings compared to the traditional GAN architecture.

Improving Performance of GANs:

Training a generative adversarial network can be computationally taxing and there can be a high degree of unstability. However, there are some tricks to combat these problems:

1. Normalize Inputs between -1 and 1
2. Use tanh as last layer in the generator output
3. Try sampling from a gaussian distribution
4. Use batch normalization
5. Use a LeakyReLU layer instead of a sparse gradient like ReLU or MaxPool
6. Use a DCGAN model
7. Use the Adam optimizer

A full list of tips and tricks can be found here.

Applications & Current Research:

1. Fashion - Generative Adversarial Networks can be used to produce photorealistic articles of clothing. This can help inspire fashion designers to create and design new clothing trends. GANs can be trained to learn to represent the attributes of style without ever being explicitly told what the representations should look like. Recently, Donggeun Yoo, Namik Kim, Sunggyun Park, Anthony S. Paek, and In So Kweon published their paper Pixel-Level Domain Transfer. In this paper they use a variation of generative adversarial networks to input a target image consiting of a dressed person and output a synthetic piece of clothing from the input image. Their goal looks as follows:

A source image.

Possible target images.

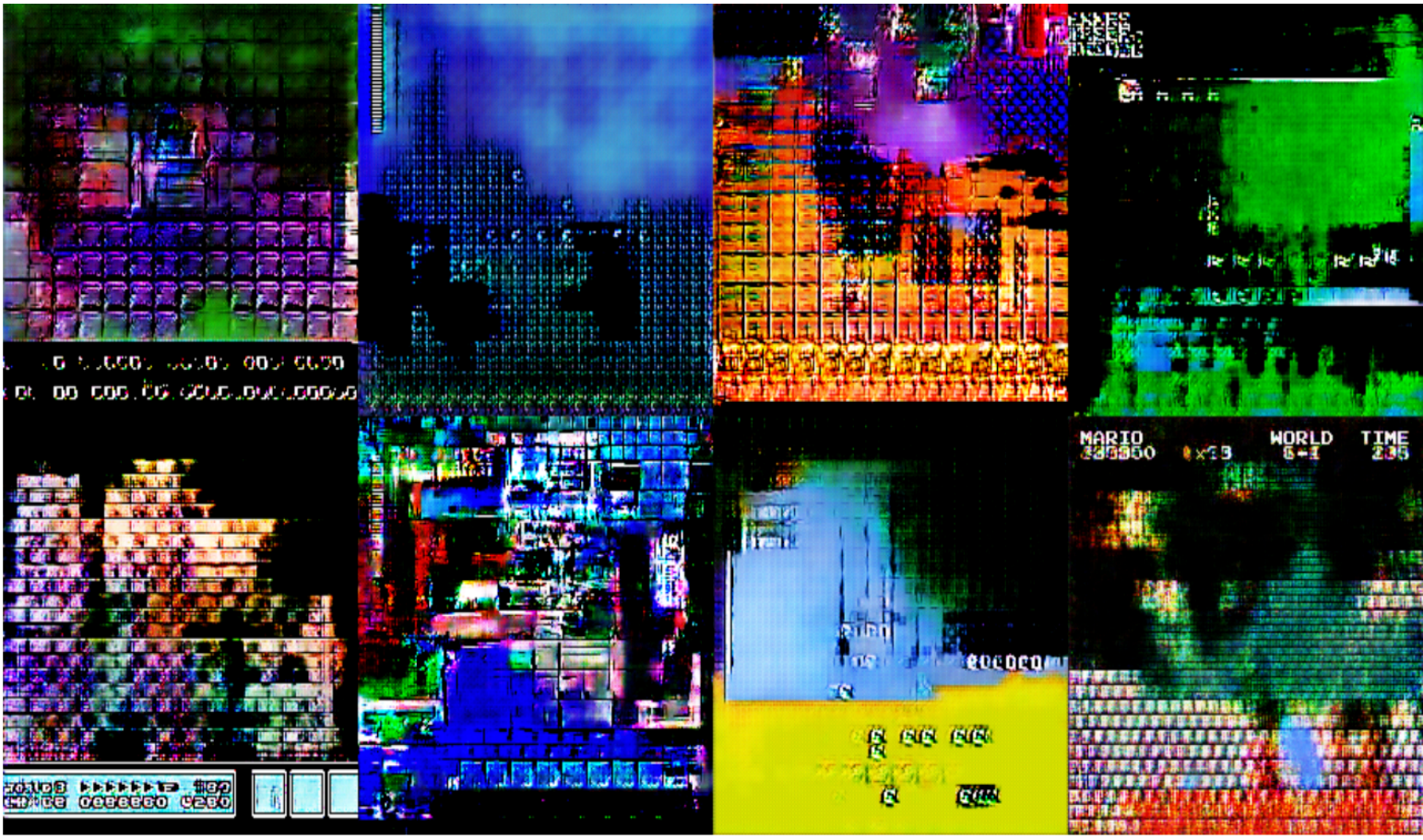
Here, an image of a person wearing clothing is inputted and a desired article of clothing resembling the input clothing is desired to be the output. After running the experiment, Yoo and his team arrived at results like the following:

Here, they found that they were able to generate clothing based on a desired input. GANs could be potentially groundbreaking in the fashion industry and help designers to great extents.

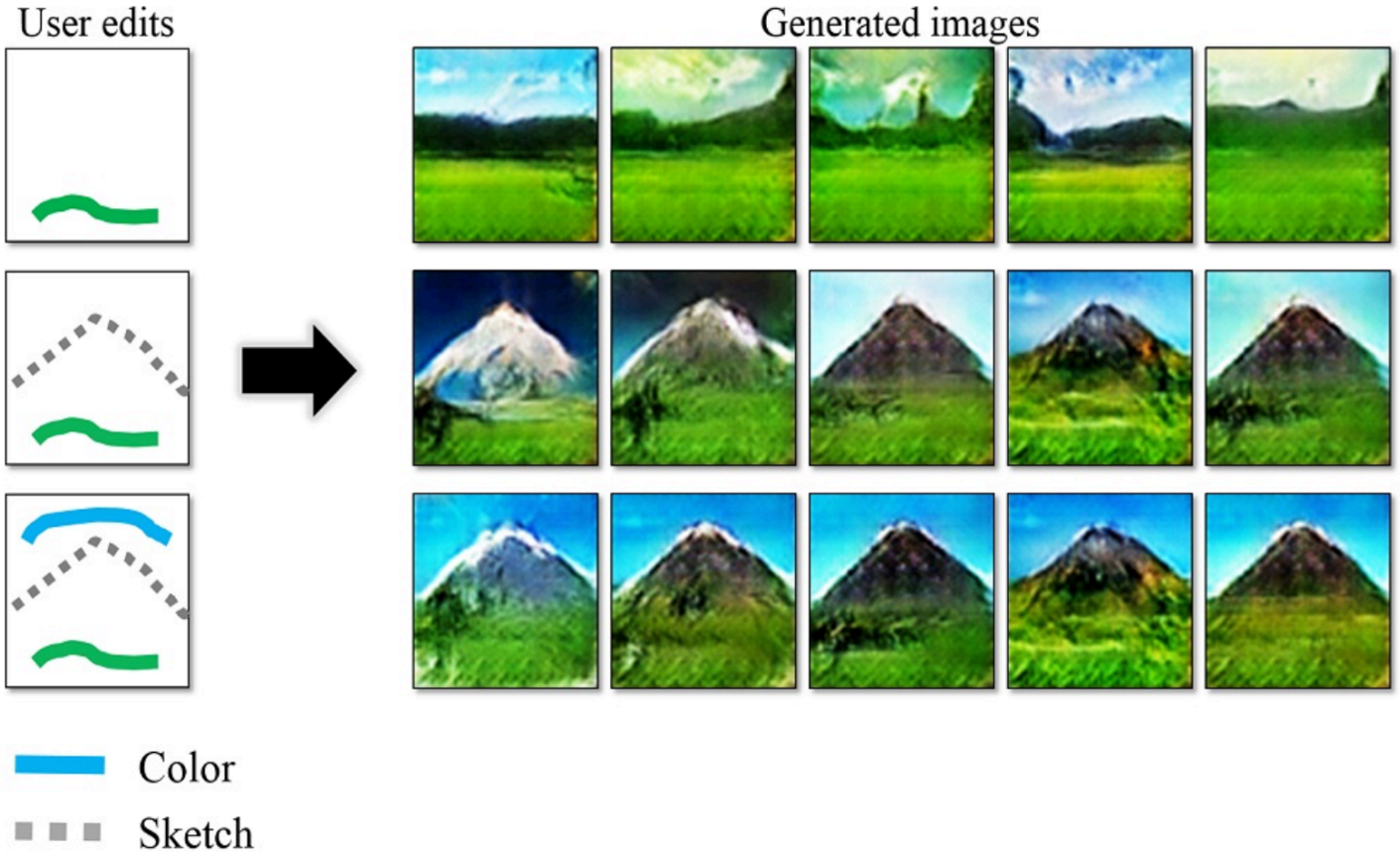
2. Art - GANs can be used to generate art and help assist users in generating art. Adam Geitgey, in his 7 part Machine Learning is Fun series, used generative adversarial networks

to generate 8-bit pixel art from video games. He used input images that look like:

and was able to train his network to produce images like:



GANs can also be used to assist users in generating art. A research prototype called iGAN, developed by UC Berkley and Adobe CTL, do just this. The user draws some shapes and colors with a few strokes and the GAN produces photo-realistic samples that best satisfy the user edits in real time. For example:

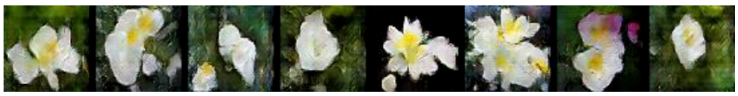
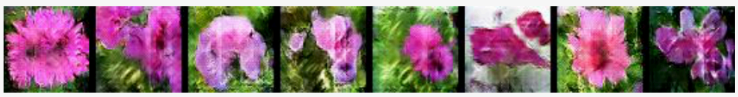
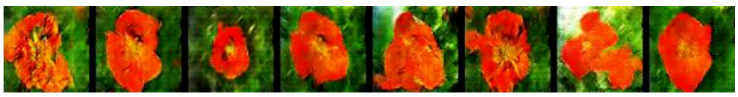


Here we see that the user attempts to draw mountains and the GAN produces them. We can see this process in real time as follows:

This can greatly assist artists and aspiring artists to produce high-quality sketeches in short periods of time.

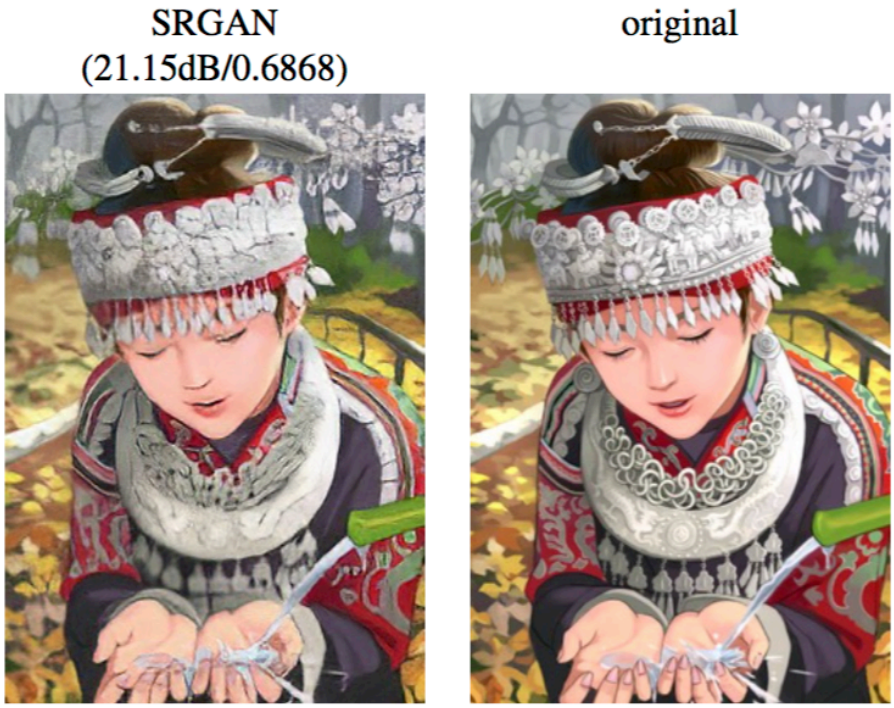
3. Text-to-Image Synthesis - Scott Reed, Zeynep Akata, Xinchen Yan, and Lakanugen Logeswaran, in their paper Generative Adversarial Text to Image Synthesis, described an automatic synthesis of realistic images from text. They used GANs to enter in keywords and create images based off of these keywords. For example:

This work bridged the advances in text and image modeling, by translating visual concepts from characters to pixels. Thier model was able to plausibly generate images of birds and flowers from detailed text descriptions. After this paper was released, individual users like Paarth Neekhara built open sourced models to perform this text to image synthesis. After training for 200 epochs on a GPU Neekhara arrived at fairly accurate results like the following:

Caption	Generated Images
the flower shown has yellow anther red pistil and bright red petals	
this flower has petals that are yellow, white and purple and has dark lines	
the petals on this flower are white with a yellow center	
this flower has a lot of small round pink petals.	
this flower is orange in color, and has petals that are ruffled and rounded.	
the flower has yellow petals and the center of it is brown	

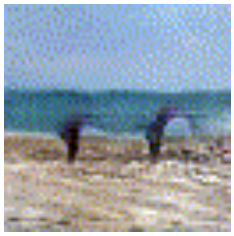
Text to Image synthesis has worked great for birds and flowers and now researchers are training on other types of images to add variety to the current systems.

4. Super Resolution - Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, and Wenzhe Shi, all from Twitter, published Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network in 2016. The goal of this paper was to estimate a high-resolution image from a low-resolution image, the task of super-resolution. The team build SRGAN, a generative adversarial network for image super-resolution. They were able to infer photo-realistic natural images for 4 x upscaling factors. For example:



Ian Goodfellow, creator of the GAN, made note of this success in his tutorial lecture on generative adversarial networks.

5. Video - Carl Vondrick, Hamed Pirsiavash, and Antonio Torralba were able to Generate Videos with Scene Dynamics in their 2016 NIPS paper. Here, they used large amounts of unlabeled video to learn a model of scene dynamics for video recognition and video generation using generative adversarial networks. They were able to generate videos such as:



by feeding in videos of beach scenes. The above video is not real, it is a hallucinated video from the GAN.

6. Encryption - Martín Abadi and David G. Andersen, from Google Brain, released Learning to Protect Communications with Adversarial Neural Cryptography. Here, three neural networks were set up: Alice, Bob, and Eve. The goal was to limit what Eve could learn by eavesdropping on Alice and Bob. In GAN fashion, the researchers wanted Alice and Bob to

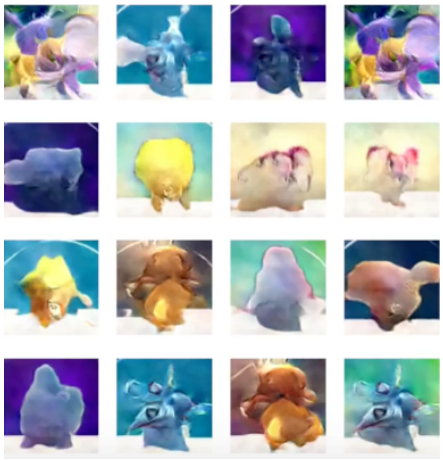
defeat the best possible version of Eve. All in all, the networks were able to encrypt and protect communications.

7. Law Enforcement - Generative Adversarial Networks are great for creating synthetic facial images. They have great importance in law enforcement. Traditionally, if a crime is committed, a witness is asked to describe the suspected criminal and a sketch is drawn by hand. GANs can quickly produce sketches based on facial features given by the witness similar to how iGAN works.

8. Healthcare - There are synthetic patient record databases like the MIHN FHIR server used for research in academia and in industry. This data is meant to mimic real data by giving patients certain diagnoses and conditions and presrcibing them medications. Instead of being done by hand, GANs could input authentic patient records and generate synthetic patient data used in these databases.

9. Speech Enhancement - In SEGAN: Speech Enhancement Generative Adversarial Network, Santiago Pascual, Antonio Bonafonte, and Joan Serra used generative adversarial networks to create SEGAN for speach enhancement. The model works as an encoder-decoder fully-convolutional structure, which makes it fast to operate for denoising wave-form chunks. The results show that, not only the method is viable, but it can also represent an effective alternative to current approaches.

10. Generating Pokemon - Yota Ishida used a pokemon go dataset as input to a generative adversarial network. He was able to reconstruct synthetic pokemon as follows:



Ian Goodfellow also used this example in his NIPS tutorial as it shows how we can use GANs not only for serious research efforts but also as a means of generating images of our interest.

Research Problems:

Generative Adversarial Networks are the hot topic right now and there is still a lot to be discovered.

Non-Convergence:

Currently there are research efforts in trying to solve the non-convergence issue. There is no current theoretical arguement that a generative adversarial network should converge nor a theoretical arguement that one should not converge. In terms of practice they do not always converge. On small problems, they sometimes converge and sometimes do not converge. On large problems, like the ImageNet at resolution 128 x 128, Goodfellow notes that he has never seen convergence. Currently there is no set of conditions under which a GAN will converge or not converge, but that is currently being researched.

Mode Collapse:

Another area within non-convergence, is mode collapse. Here, several input values are mapped to the same output point. Complete mode collapse is rare, but partial mode collapse is much more common. With this issue, GANs are limited to problems where one input can be mapped to many distinct outputs, for instance text to image synthesis. Here an input description of a small yellow bird could be mapped to many correct distinct outputs.

Equilibrium Algorithms:

To train a GAN we need to find the equilibrium of a game. We don't always find this equilibrium with gradient descent. Currently, there is no good algorithm that finds the equilibrium, hence adding training unstability.

Overall Training:

All in all, training a generative adversarial network is difficult. The function the networks try to minimize has no closed form and thus the optimization problem is unstable. Research is being hevily devoted to efficient training of a generative adversarial network.

Further Resources

A lot of work has been done on generative adversarial networks. Here are some alternate links worth checking out:

Presentations:

- Ian Goodfellow AIWTB 2016 Lecture
- Ian Goodfellow NIPS 2016 Workshop
- Ian Goodfellow 2016 NIPS Tutorial
- Soumith Chintala Facebook London Machine Learning Meetup Lecture
- Two Minute Papers Image Editing with GANs
- Siraj Raval Fresh Machine Learning with GANs

Research Papers:

- 2014 Ian Goodfellow Generative Adversarial Nets
- Music Generation using GANs
- Improved Techniques for Training GANs
- Learning to Protect Communications with Adversarial Neural Cryptography
- Generative Adversarial Text to Image Synthesis
- Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network
- Generating Videos with Scene Dynamics
- Pixel-Level Domain Transfer
- Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks
- Conditional generative adversarial nets for convolutional face generation
- Connecting Generative Adversarial Networks and Actor-Critic Methods

Building our own GAN:

Now that we understand what a generative adversarial network is and how it works, let's proceed by creating our own GAN. We will specifically implement a deep convolutional generative adversarial network (DCGAN) and generate facial images like we described in the motivating example. For this post click the button below.

Let's Build A GAN

References

Below are all references I used in constructing this tutorial. Each reference item is a link. You can find the specific link referenced by going to the section with the corresponding desctiption:

- Images
  1. Motivation - Generating Facial Images Example
  2. History - Goodfellow/Schmidhuber Twitter Complaint
  3. History - Goodfello/Schmidhuber Reddit Discussions
  4. History - Goodfellow Quora Response
  5. Prerequisites - Generative vs. Discriminative Example
  6. Prerequisites - Artifical Neural Network Example
  7. Prerequisites - Convolutional Layer Example
  8. Prerequisites - Rectified Linear Unit Layer Example
  9. Prerequisites - Max Pooling Layer Example
  10. Prerequisites - Batch Normalization Algorithm
  11. What are Genearative Adversarial Networks - Motivational Couch Image
  12. What are Genearative Adversarial Networks - Beginner Police/Criminal GAN description
  13. What are Genearative Adversarial Networks - Expert Police/Criminal GAN description
  14. What are Generative Adversarial Netowrks - Dollar Bill
  15. What are Generative Adversarial Networks - GAN pipeline
  16. Deep Convolutional Generative Adversarial Networks - Generator DCGAN architecture
  17. Deep Convolutional Generative Adversarial Networks - Discriminator DCGAN architecture
  18. Applications & Current Research - Fashion Images
  19. Applications & Current Research - Art Images
  20. Applications & Current Research - iGAN
  21. Applications & Current Research - Text-to-Image Synthesis Image 1
  22. Applications & Current Research - Text-to-Image Synthesis Image 2
  23. Applications & Current Research - Super Resolution Image
  24. Applications & Current Research - Video with Scene Dynamics GIF
  25. Applications & Current Research - Generating Pokemon Image

- Content
  1. Motivaiton - Yann LeCun Quote
  2. History - Schmidhuber Predictibility Minimization
  3. History - Goodfellow's Invention of GANs
  4. History - Schmidhuber's NIPS Submission Feedback & Goodfellow's Response
  5. History - Goodfellow GANs paper revision

Generative Adversarial Networks

Part 2: Building a Generative Adversarial Network

By: Adam Lieberman

Introduction

Radford's paper Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks lays the framework and architecture for building a deep convolutional generative adversarial network (DCGAN). Convolutional layers have been great for image based deep learning regarding tasks like image classification in the field of computer vision. Additionally, DCGAN models can improve stability during training so that we hopefully do not encounter issues like mode collapse. Our goal is to build a DCGAN that can generate synthetic facial images. We will follow the steps Radford used in his paper to build out our model.

Environment Setup

Below we have a description of each library we will use. Please click the links under installation and documentation to install and learn more about each library.

- **Programming Language:**
    - Python 3 - Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. We will be using version 3.x. This can be obtained from the official Python website or through the Anaconda distribution, which contains python 3 and many useful scientific computing libraries.
        - Python Installation: Python Installation or Anaconda Installation
        - Pip Installation: Pip Installation

- **Libraries:**
    - TensorFlow - TensorFlow is an open-source software for machine intelligence. It is currently a very popular choice for developing deep learning modles.
        - Installation: TensorFlow Installation
        - Documentation: TensorFlow Documentation

      Once TensorFlow is installed you will have access to TF-Slim, a lightweight library for defining, training, and evaluating complex models in TensorFlow. This makes writing TensorFlow code easier and quicker while still allowing us to see the fine details of our model and its layers.

    - Numpy - Numpy is a package for scientific computing that contains many useful operations for a multi-dimensional data structure called an ndarray (np array).
        - Installation: Numpy Installation
        - Documentation: Numpy Documentation

    - Matplotlib - Matplotlib, from the creators of numpy, is a plotting library that allows for custom charts like scatter plots, bar charts, line graphs, etc.
        - Installation: Matplotlib Installation
        - Documentation: Matplotlib Documentation

    - Scipy - Scipy, from the creators of numpy, is an alternate library for mathematics, science, and engineering.
        - Installation: Scipy Installation
        - Documentation: Scipy Documentation

    - Sklearn - Sklearn has efficient tools for data mining, analysis, and machine learning.
        - Installation: Sklearn Installation
        - Documentation: Sklearn Documentation

    - PIL - PIL, short for Python Imaging Library, is a powerful library to handle and manipulate images.
        - Installation: PIL Installation
        - Documentation: PIL Documentation

    - Random - Random allows us to create pseudo-random number generators for various

distributions.

- Installation: Installed with Python
- Documentation: Random Documentation

Data

We will be using data from the Labeled Faces in the Wild (LFW) database. This database consists of more than 13,000 grayscale images of faces collected from the web. Additionally, each face has been labeled with the name of the person pictured. The data can be downloaded from here. Additionally, we can access the datasets module in sklearn and pull the data using the fetch_lfw_people function. This will return our data in a dataset object, which is a dictionary like structure. Let us now proceed to write a class and a few fuctions to read in and format our data.

We start by creating a class called Data_Set. In this class we take in our vector of images. In our init method we will calculate the number of samples we have in the images vector, normalize the images, and set the number of epochs completed and index of the epochs to 0. We will need the number epochs completed and the index of the epochs when we train our generative adversarial network. We will also create some getters for the number of images, number of samples, and the number of epochs completed. Finally, we will create a method called batch_next. This takes in a batch size and will perform a sequential pull of this batch size for our training data. Our class is as follows:

```
class Data_Set(object):

    #Parameter Initialization
    def __init__(self, images):
        self.num_examples = images.shape[0]
        self.images = np.multiply(images.astype(np.float32), 1.0 / 255.0)
        self.epochs_completed = 0
        self.index_in_epoch = 0

    #Getters/Setters for our class
    def images(self):
        return self.images

    def num_examples(self):
        return self.num_examples

    def epochs_completed(self):
        return self.epochs_completed

    #Return the next batch, used in model training
```

```
    def batch_next(self, batch_size):
        start = self.index_in_epoch
        self.index_in_epoch += batch_size
        if self.index_in_epoch > self.num_examples:
            self.epochs_completed += 1
            perm = np.arange(self.num_examples)
            np.random.shuffle(perm)
            self.images = self.images[perm]
            start = 0
            self.index_in_epoch = batch_size
        end = self.index_in_epoch
        return self.images[start:end], None
```

Next, we create a function called pull_data(). This function creates an empty Data_Sets object which we create inside the function and then uses the fetch_lfw_people method from the sklearn datasets module. Within fetch_lfw_people module we use the following parameters:

- Slice: This provides a custom 2D slice in terms of (height, width) which allows us to extract a specific part of each of the JPEG images. We want to extract the faces in each image which corresponds to slice_ = (slice(70, 195, None), slice(70, 195, None)).

- Resize: The Ratio used to resize the each facial picture. This value is specified as a float and defaults to 0.5. The original images are 250 x 250 pixels, but the default slice and resize arguments reduce them to 62 x 74. We, however, want our our images to be 28 x 28. To do so we can set the resize = 0.224.

Once we call the fetch_lft_people module we obtain a dataset object. We can then use the .data function to extract a vector of the facial images. We can then create the Data_Set object, which we created in the class above, by passing in our vector of the facial images. We create our pull_data function as follows:

```
def pull_data():
    class Data_Sets(object):
        pass
    data_sets = Data_Sets()

    #Get LFW people data and resize it to 28 x 28, slice for faces
    lfw_people = fetch_lfw_people(slice_=(slice(70, 195, None), slice(70, 195, N

    #Create Data_Set object
    d = lfw_people.data
    data_sets.train = Data_Set(d)
    return data_sets
```

We can now obtain our data by doing the following:

```
data = pull_data()
```

We can check the size of our data as follows:

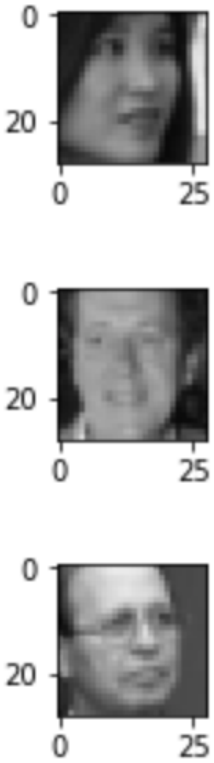
```
print(data.train.images.shape)
>>>
(13233, 784)
```

Above, we see that we have 13,233 samples where each sample is a vector of length 784. We see that this is a flattened 28 x 28 vector as 28 x 28 = 784.

Now that we have our data, let us see what a sample of it looks like. To do so we write a function called display_images which takes in our data and some indices we want to display from the data. Here we use data.train.images to obtain an ndarray of normalized image values. We loop over our indices and look at the image inside data.train.images of that particular index and unnormalize it. We also reshape it into a 28 x 28 image and then use the python library PIL's imshow function to show the image. We do this for every image in the list of indices we specify. Our function is as follows:

```
def display_images(data,indices):

    #Obtain the images inside data
    im_data = data.train.images

    for i in range(len(indices)):

        #Set image sizing
        %matplotlib inline
        plt.figure(figsize=(1,1))

        #Unnormalize the image and reshape it
        im = Image.fromarray(np.multiply(im_data[indices[i]],255.0).reshape(28,2

        #Show the image
        p = plt.imshow(im)
        plt.show(p)

display_images(data,[4,45,599])
```

Once we call display_images(data,[4,45,599]) above, we see the 4^{th}, 45^{th}, and 599^{th} images in our dataset. The images are as follows:



DCGAN architecture:

Let us now briefly talk about Deep Convolutional Generative Adversarial Networks so that we have some background before we implement one. In 2016 Alec Radford, Luke Metz, and Soumith Chintala published Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks. In computer vision tasks, convolutional networks and supervised learning has had great success and popularity. However, unsupervised learning with convolutional neural networks has recieved less attention. Deep Convolutional Generative Adversarial Networks (DCGANs) demonstrate that they are a strong candidate for unsupervised learning. The generator/discriminator pair learns a hierarchy of representations from object parts to scenes in both the generator and discriminator. Since we are working with facial image data, implementing a DCGAN is a great choice. The architecture from the paper, which we will follow, is described as:

- 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

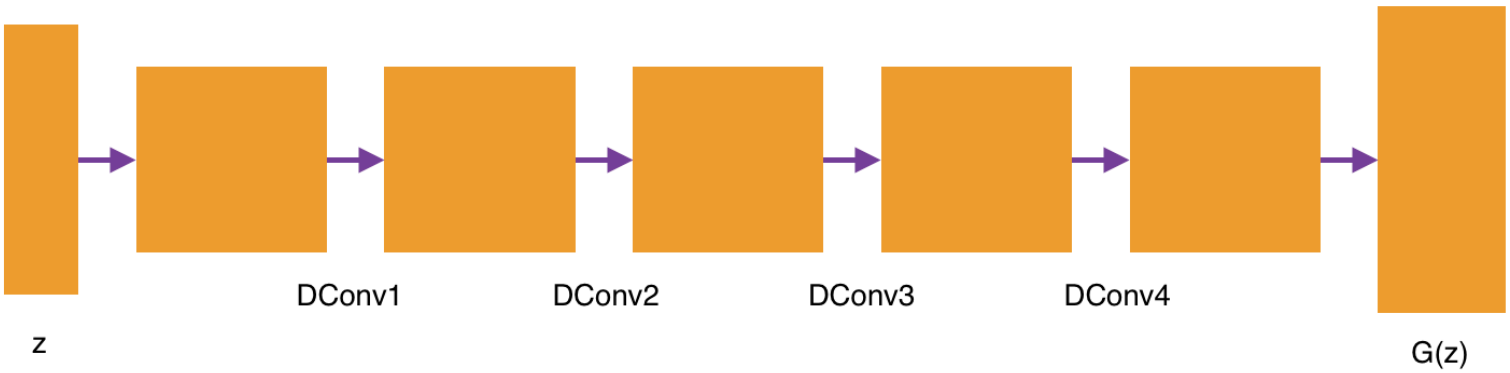
Leaky Rectified Linear Unit

Before we move on to creating our generator and discriminator let us first construct a leaky rectified linear unit layer. When we use GANs we want to avoid sparse gradients to prevent suffering in terms of GAN stability. This means we should stear clear from ReLU and MaxPool layers. Instead we can use this leaky rectified linear unit layer. Let us create a function called lrelu that takes in x, a leak value, and the name "lrelu". Inside we use tf.variable_scope("lrelu") and set f1 to 0.5 * (1 + leak) and f2 to 0.5 * (1 - leak). We then return f1 * x + f2 * abs(x). We will use the leaky rectified linear unit layer in our discriminator as noted in the DCGAN architecture above. We build our function as follows:

```
def lrelu(x, leak=0.2, name="lrelu"):
    with tf.variable_scope(name):
        f1 = 0.5 * (1 + leak)
        f2 = 0.5 * (1 - leak)
        return f1 * x + f2 * abs(x)
```

Generator

Let us now construct our generator. The architecture looks as follows:



Above we see that we have our z input (uniformly distributed random numbers). We will create four deconvolutional layers. Each layer will have batch normalization. The first three layers will use the rectified linear unit (RELU) activation function, while the fourth deconvolutional layer will use the tanh activation function as specified in the paper. We will then output an image G(z) with

dimensions 32 x 32. Our architecure for the generator will look as follows:

- Fully Connected Layer
- Reshape
- Deconvolutional Layer 1 - batch normalization, relu activation
- Deconvolutional Layer 2 - batch normalization, relu activation
- Deconvolutional Layer 3 - batch normalization, relu activation
- Deconvolutional Layer 4 - batch normalization, tanh activation

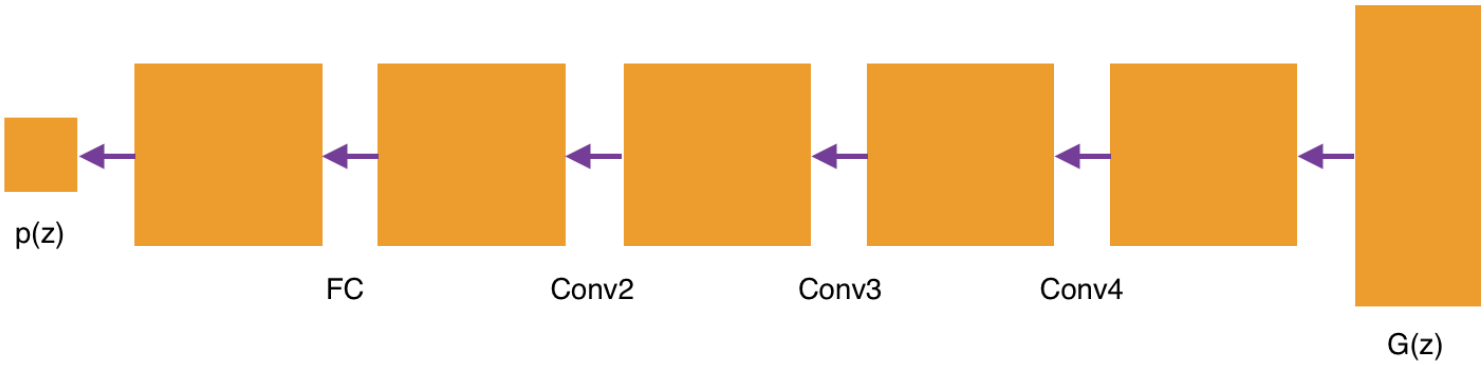
All in all, our generator will take in a vector consisting of random numbers from a uniform distribution, pass through four fractionally-strided convolutions (called deconvolutions), and then output an image with shape 32 x 32. We construct our generator model, called generator, with an input called z_shp. This is the shape of our z in the above architecture descrption. Our function is as follows:

```
def generator(z_shp):

    #Commonly Used Variables
    PADDING = "SAME"
    STRIDE = [2,2]

    #Our first dense (fully connected) layer
    g1_dense = slim.fully_connected(z_shp,4*4*256,normalizer_fn=slim.batch_norm,
        activation_fn=tf.nn.relu,scope='g1_dense',weights_initializer=initialize

    #Reshape
    g1_dense_reshape = tf.reshape(g1_dense,[-1,4,4,256])

    #Dconv Layer 1, batch normalization, relu activation
    g2_dconv = slim.convolution2d_transpose(\
        g1_dense_reshape,num_outputs=64,kernel_size=[5,5],stride=STRIDE,\
        padding=PADDING,normalizer_fn=slim.batch_norm,\
        activation_fn=tf.nn.relu,scope='g2_dconv', weights_initializer=initializ

    #Dconv Layer 2, batch normalization, relu activation
    g3_dconv = slim.convolution2d_transpose(\
        g2_dconv,num_outputs=32,kernel_size=[5,5],stride=STRIDE,\
        padding=PADDING,normalizer_fn=slim.batch_norm,\
        activation_fn=tf.nn.relu,scope='g3_dconv', weights_initializer=initializ

    #Dconv Layer 3, batch normalization, relu activation
    g4_dconv = slim.convolution2d_transpose(\
        g3_dconv,num_outputs=16,kernel_size=[5,5],stride=STRIDE,\
        padding=PADDING,normalizer_fn=slim.batch_norm,\
        activation_fn=tf.nn.relu,scope='g4_dconv', weights_initializer=initializ
```

```
    #Dconv Layer 4, batch normalization, tanh activation
    g5_dconv = slim.convolution2d_transpose(\
        g4_dconv,num_outputs=1,kernel_size=[32,32],padding=PADDING,\
        biases_initializer=None,activation_fn=tf.nn.tanh,\
        scope='g5_dconv', weights_initializer=initializer)

    return g5_dconv
```

Discriminator

Now that we have our generator to generate synthetic images, let us now create our discriminator. Our discriminator achitecture is as follows:



Here we will input a 32 x 32 image and pass it through three convolutional layers that have a leaky rectified linear unit activation. We will then pass through a fully connected layer with a sigmoid activation function and will return a single valued probability representing whether the generated image is a "real" image or a "fake" image. We create the discriminator as follows:

```
def discriminator(bottom, reuse=False):
    PADDING = "SAME"
    STRIDE = [2,2]

    #Conv Layer 1, No batch normalization, leaky relu activation
    d1_conv = slim.convolution2d(bottom,16,[4,4],stride=STRIDE,padding=PADDING,\
        biases_initializer=None,activation_fn=lrelu,\
        reuse=reuse,scope='d1_conv',weights_initializer=initializer)

    #Conv Layer 2, batch normalization, leaky relu activation
    d2_conv = slim.convolution2d(d1_conv,32,[4,4],stride=STRIDE,padding=PADDING,
        normalizer_fn=slim.batch_norm,activation_fn=lrelu,\
        reuse=reuse,scope='d2_conv', weights_initializer=initializer)
```

```
    #Conv Layer 3, batch normalization, leaky relu activation
    d3_conv = slim.convolution2d(d2_conv,64,[4,4],stride=STRIDE,padding=PADDING,
        normalizer_fn=slim.batch_norm,activation_fn=lrelu,\
        reuse=reuse,scope='d3_conv',weights_initializer=initializer)

    #Dense Layer (Fully connected), sigmoid activation
    d4_dense = slim.fully_connected(slim.flatten(d3_conv),1,activation_fn=tf.nn.
        reuse=reuse,scope='d4_output', weights_initializer=initializer)

    return d4_dense
```

DCGAN Construction

Now that we have our generator and discriminator we can now create our deep convolutional generative adversarial network. We will create a variable called z_size which will be the size of the z vector used for our generator. We will then initialize all weights for our network. We do so here because we want to initialize the same weights. If we pass this into each weights_initializer parameter in our tf.slim code above we might be assigned different weights. We then create an input for the generator and discriminator and create the images for the random vectors and probabilites for the real images. We can then create the optimization objective and then apply gradient descent. We create our DCGAN model as follows:

```
tf.reset_default_graph()

z_size = 100

#Initialize Network weights
initializer = tf.truncated_normal_initializer(stddev=0.02)

#Input for Generator
z_in = tf.placeholder(shape=[None,z_size],dtype=tf.float32)

#Input for Discriminator
real_in = tf.placeholder(shape=[None,32,32,1],dtype=tf.float32)

#Creating Images for ranom vectors of size z_in
Gz = generator(z_in)

#Probabilities for real images
Dx = discriminator(real_in)

#Probabilities for generator images
Dg = discriminator(Gz,reuse=True)
```

```
#Optimize the discriminator and the generator
d_log1 = tf.log(Dx)
d_log2 = tf.log(1.-Dg)
g_log = tf.log(Dg)
d_loss = -tf.reduce_mean(d_log1 + d_log2)
g_loss = -tf.reduce_mean(g_log)

tvars = tf.trainable_variables()

#Use the Adam Optimizers for discriminator and generator
LR = 0.0002
BTA = 0.5
trainerD = tf.train.AdamOptimizer(learning_rate=LR,beta1=BTA)
trainerG = tf.train.AdamOptimizer(learning_rate=LR,beta1=BTA)

#Gradients for discriminator and generator
gradients_discriminator = trainerD.compute_gradients(d_loss,tvars[9:])
gradients_generator = trainerG.compute_gradients(g_loss,tvars[0:9])

#Apply the gradients
update_D = trainerD.apply_gradients(gradients_discriminator)
update_G = trainerG.apply_gradients(gradients_generator)
```

Saving Generated Images

Our GAN will create synthetic images during training. Let us now write a function called save_generated_images to save our generated images. Here we will take in a vector of images, a size, and a path to save them. We will create a vector of zeros that is of size image height x width multiplied by the size we pass in (for instance size = [6,6]). We will then extract the images we want to save and return a block of images (6 x 6) so we can see how our model improves over time. We do so as follows:

```
def save_generated_images(images,size,image_path):
    images = (images+1.)/2.
    height = images.shape[1]
    width = images.shape[2]
    img = np.zeros((height * size[0], width * size[1]))
    for idx, image in enumerate(images):
        a = idx % size[1]
        b = idx // size[1]
        img[b*height:b*height+height, a*width:a*width+width] = image
    sve = scipy.misc.imsave(image_path,img)
    return sve
```

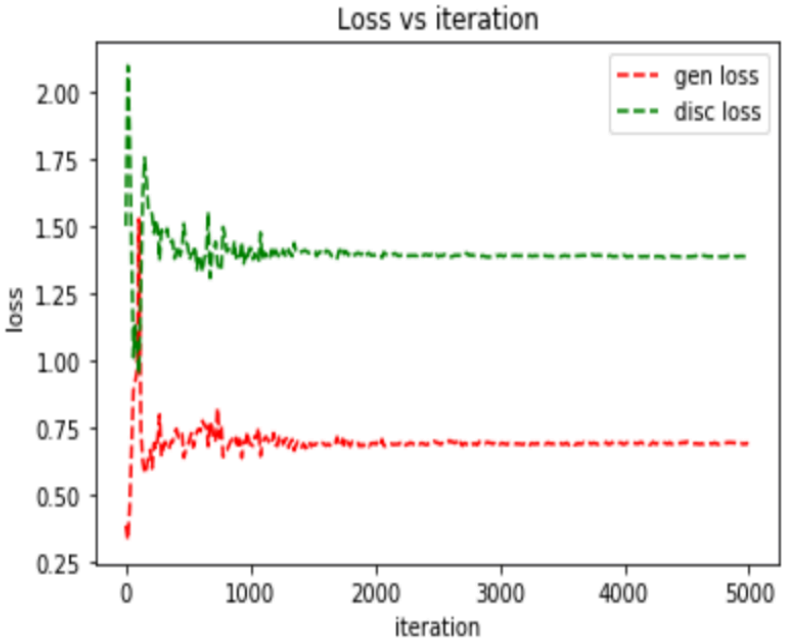
DCGAN Training

Let us now train our network on our machine (I am using a retina macbook pro with 8gb of ram). Since we are not using a GPU right now (we will do so in the next section), we will use 5000 iterations and a batch size of 128. Here, we will choose a sample batch from our facial image data using the class function batch_next and shape it into a 32 x 32 image. We will then generate a random batch, z, and update the generator and discriminator. Every 15 iterations we will calculate our loss for both the generator and discriminator and plot them on an updating plot. We will also save some sample generated images (36 sample generated images in a 6 x 6 grid) from the generator at this stage. We will save these images into a directory called facial_figs. Every 1000 iterations we will save our model attributes into a directory called facial_models. We train our model as follows:

```
#size setups
batch_size = 128
iterations = 5000

#tf setup
init = tf.global_variables_initializer()
saver = tf.train.Saver()

#plot setup
plt_g = np.array([])
plt_d = np.array([])
plt_x = np.array([])

with tf.Session() as sess:
    sess.run(init)
    for i in range(iterations):
        print("Progress: ",i,"/",iterations)

        #Choose sample batch from data
        xs,xt = data.train.batch_next(batch_size)

        #Make sure our data is between (-1,1)
        xs = np.lib.pad(((np.reshape(xs,[batch_size,28,28,1]) - 0.5) * 2.0),((0,

        #Updating Discriminator Once and Generator Twice
        random_z_batch = np.random.uniform(-1.0,1.0,size=[batch_size,z_size]).as
        _,dLoss = sess.run([update_D,d_loss],feed_dict={z_in:random_z_batch,real
        _,gLoss = sess.run([update_G,g_loss],feed_dict={z_in:random_z_batch})
        _,gLoss = sess.run([update_G,g_loss],feed_dict={z_in:random_z_batch})
```

```
        #Training Stats
        if i % 15 == 0:
            #Plot our generator loss and discriminator loss
            print("Gen Loss: " + str(gLoss) + " Disc Loss: " + str(dLoss))
            plt_g = np.append(plt_g,float(gLoss))
            plt_d = np.append(plt_d,float(dLoss))
            plt_x = np.append(plt_x,i)
            plt.gca().cla()
            plt.plot(plt_x,plt_g,'r--',label='gen loss')
            plt.plot(plt_x,plt_d,'g--',label='disc loss')
            plt.xlabel('iteration')
            plt.ylabel('loss')
            plt.title('Loss vs iteration')
            plt.legend()
            display.clear_output(wait=True)
            display.display(plt.gcf())

            #Get sample images from the generator
            z2 = np.random.uniform(-1.0,1.0,size=[batch_size,z_size]).astype(np.
            newZ = sess.run(Gz,feed_dict={z_in:z2})

            #Save our generated images
            if not os.path.exists('./facial_figs'):
                os.makedirs('./facial_figs')
            save_generated_images(np.reshape(newZ[0:36],[36,32,32]),[6,6],'./fac

        #Save our model every 1000 iterations
        if i % 1000 == 0 and i != 0:
            if not os.path.exists('./facial_models'):
                os.makedirs('./facial_models')
            saver.save(sess,'./facial_models/model-'+str(i)+'.cptk')
            print("Saved Model")
```

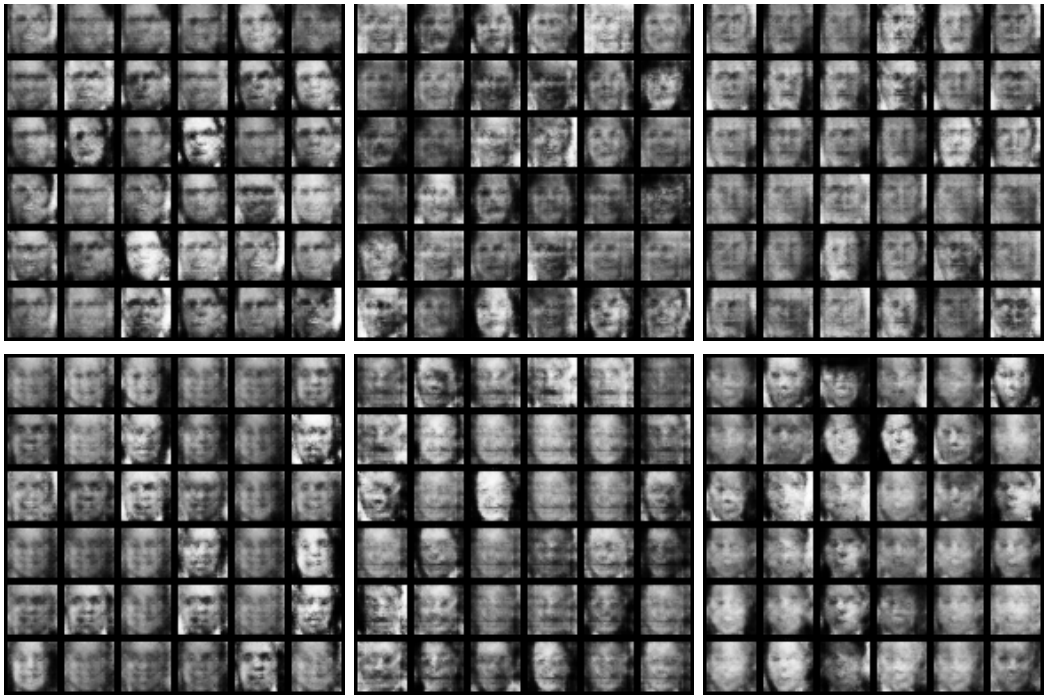
The code above should take approximately 8 hours using the CPU on a retina Macbook Pro with 8gb of ram. When the model finished our plot of the loss function against the number of iterations looked as follows:



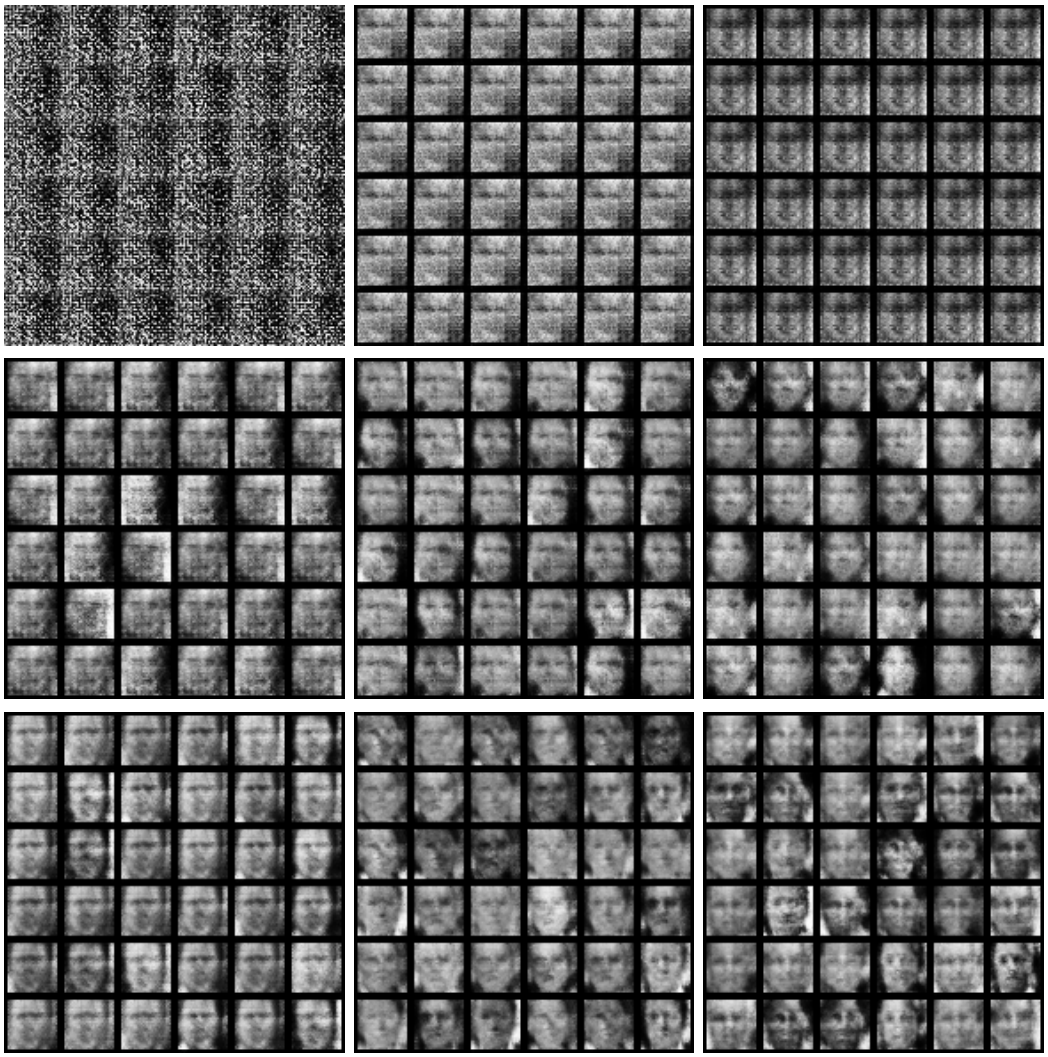
We see that our generator's loss was initially lower than the discriminator's loss meaning the generator was generating images that was fooling the discriminator. However, the discriminator soon learned to catch these fake images and it's loss dropped lower than the discriminators. Soon, the generator started generating better quality images that fooled the discriminator. We see that the generator and discriminator loss functions started to stabalize at about 1000 iterations with the generator's loss being about 0.65 and the discriminator's loss being about 1.40.

DCGAN Generator Image Samples

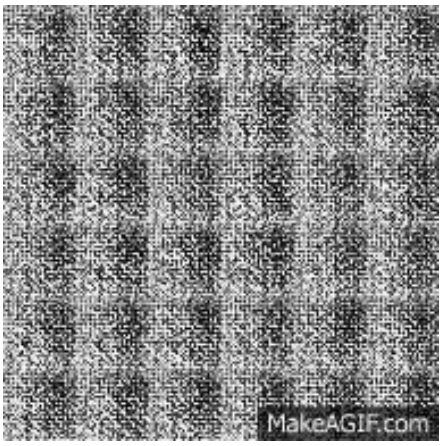
Below are some sample images that were generated during training:



All of the above images were from iterations 4000 - 5000. We can clearly see that facial images have started to form and we see clearly defined facial features like eyes, nose, mouth. The images are still a little blurry, but we should expect this after a low number of iterations. Let's take a look at some of the earlier images generated earlier in the training process:



We see that our generator started off generating low quality pixelated images, but started to generate images that looked more and more like faces, hence fooling the discriminator. We can find a timelapse gif of the training process below:



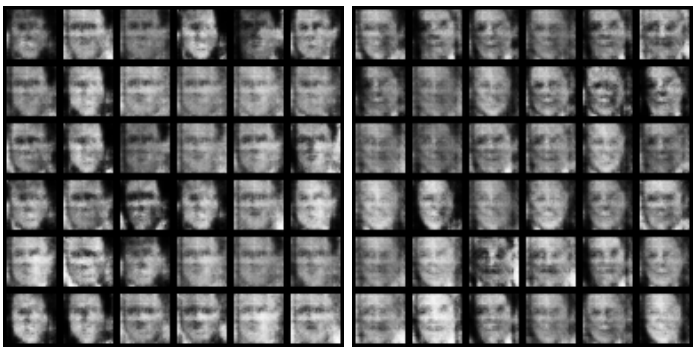
Query Our DCGAN

Now that we have a trained model, where we can clearly see facial images, let us write a function to query new facial images. We first load in our saved model from the facial_models directory. Then, we pass in a vector of random uniformly distributed numbers of the batch size specified previously to the generator and generate a set of 6 x 6 images, which will be saved in a directory called synthetic_faces. We wrap the random number process and image generation in a for loop so we can repeat this k times. Our code is as follows:

```
batch_size_sample = 36
num_img = 10
init = tf.initialize_all_variables()
saver = tf.train.Saver()
with tf.Session() as sess:
    sess.run(init)

    #Load previos Model and weights
    ckpt = tf.train.get_checkpoint_state('./facial_models')
    saver.restore(sess,ckpt.model_checkpoint_path)

    #Generate 10 synthetic facial iamges
    for k in range(num_img):
        #Generate random uniform to generate synthetic images
        rand_z = np.random.uniform(-1.0,1.0,size=[batch_size_sample,z_size]).ast
        newZ = sess.run(Gz,feed_dict={z_in:rand_z})

        #Save synthetic image
        if not os.path.exists('./synthetic_faces'):
```

```
            os.makedirs('./synthetic_faces')
        save_generated_images(np.reshape(newZ[0:batch_size_sample],[36,32,32]),[
```

Let us now take a look at some of the images we generated:



Again, we can clearly see facial images produced by our generator.

DCGAN On A GPU

We see that we had good success in generating facial images by training 5,000 iterations over 8 hours on our machine's CPU. Using a GPU will allow us to train many more iterations in much less time. Amazon Web Services (AWS) offers a low cost hourly charge to run code on their powerful machines. We will make use of the p2.8xlarge machine for training our model. This machine has 8 GPUs, 32 vCPU, 488 GiB of memory, 19968 parallel processing cores, 96 GiB of GPU memory, and 10 Gigabit network performance. This is a very powerful machine and at the cost of $\$7.20$ per hour we can train for 50,000 iterations in about 85 minutes. This might sound expensive, but we are going to use AWS spot pricing. Here, the current spot price (always changing) is $\$0.905$ per hour. So, for less than $\$2$ we can train our DCGAN model on a high powered machine packed with multiple GPUs for 50,000 iterations. To do so, we need to enable our TensorFlow code to use a GPU. Now, when we create our DCGAN model by combining the generator and discriminator, we need to set DEVICE = '/gpu:0' and wrap our code inside with tf.device(DEVICE). We make the change as follows:

```
DEVICE='/gpu:0'
with tf.device(DEVICE):
    tf.reset_default_graph()

    z_size = 100

    #Initialize Network weights
    initializer = tf.truncated_normal_initializer(stddev=0.02)
```

```
    #Input for Generator
    z_in = tf.placeholder(shape=[None,z_size],dtype=tf.float32)

    #Input for Discriminator
    real_in = tf.placeholder(shape=[None,32,32,1],dtype=tf.float32)

    #Creating Images for ranom vectors of size z_in
    Gz = generator(z_in)

    #Probabilities for real images
    Dx = discriminator(real_in)

    #Probabilities for generator images
    Dg = discriminator(Gz,reuse=True)

    #Optimize the discriminator and the generator
    d_log1 = tf.log(Dx)
    d_log2 = tf.log(1.-Dg)
    g_log = tf.log(Dg)
    d_loss = -tf.reduce_mean(d_log1 + d_log2)
    g_loss = -tf.reduce_mean(g_log)

    tvars = tf.trainable_variables()

    #Use the Adam Optimizers for discriminator and generator
    LR = 0.0002
    BTA = 0.5
    trainerD = tf.train.AdamOptimizer(learning_rate=LR,beta1=BTA)
    trainerG = tf.train.AdamOptimizer(learning_rate=LR,beta1=BTA)

    #Gradients for discriminator and generator
    gradients_discriminator = trainerD.compute_gradients(d_loss,tvars[9:])
    gradients_generator = trainerG.compute_gradients(g_loss,tvars[0:9])

    #Apply the gradients
    update_D = trainerD.apply_gradients(gradients_discriminator)
    update_G = trainerG.apply_gradients(gradients_generator)
```

We can now apply our previous code, shown below, to start training the model on the AWS instance. We will save all generated images and model parameters to the directories gpu_facial_figs and gpu_facial_models.

```
#size setups
batch_size = 128
iterations = 50000

#tf setup
init = tf.global_variables_initializer()
```

```
saver = tf.train.Saver()

#plot setup
plt_g = np.array([])
plt_d = np.array([])
plt_x = np.array([])

with tf.Session() as sess:
    sess.run(init)
    for i in range(iterations):
        print("Progress: ",i,"/",iterations)

        #Choose sample batch from data
        xs,xt = data.train.batch_next(batch_size)

        #Make sure our data is between (-1,1)
        xs = np.lib.pad(((np.reshape(xs,[batch_size,28,28,1]) - 0.5) * 2.0),((0,

        #Updating Discriminator Once and Generator Twice
        random_z_batch = np.random.uniform(-1.0,1.0,size=[batch_size,z_size]).as
        _,dLoss = sess.run([update_D,d_loss],feed_dict={z_in:random_z_batch,real
        _,gLoss = sess.run([update_G,g_loss],feed_dict={z_in:random_z_batch})
        _,gLoss = sess.run([update_G,g_loss],feed_dict={z_in:random_z_batch})


        #Training Stats
        if i % 15 == 0:
            #Plot our generator loss and discriminator loss
            print("Gen Loss: " + str(gLoss) + " Disc Loss: " + str(dLoss))
            plt_g = np.append(plt_g,float(gLoss))
            plt_d = np.append(plt_d,float(dLoss))
            plt_x = np.append(plt_x,i)
            plt.gca().cla()
            plt.plot(plt_x,plt_g,'r--',label='gen loss')
            plt.plot(plt_x,plt_d,'g--',label='disc loss')
            plt.xlabel('iteration')
            plt.ylabel('loss')
            plt.title('Loss vs iteration')
            plt.legend()
            display.clear_output(wait=True)
            display.display(plt.gcf())

            #Get sample images from the generator
            z2 = np.random.uniform(-1.0,1.0,size=[batch_size,z_size]).astype(np.
            newZ = sess.run(Gz,feed_dict={z_in:z2})

            #Save our generated images
            if not os.path.exists('./gpu_facial_figs'):
                os.makedirs('./gpu_facial_figs')
            save_generated_images(np.reshape(newZ[0:36],[36,32,32]),[6,6],'./gpu
```

```
        #Save our model every 1000 iterations
        if i % 1000 == 0 and i != 0:
            if not os.path.exists('./gpu_facial_models'):
                os.makedirs('./gpu_facial_models')
            saver.save(sess,'./gpu_facial_models/model-'+str(i)+'.cptk')
            print("Saved Model")
```

Now, we have over 50,000 generated images during training. Let us take a look at a couple of the images generated during training:

Let us also look at a sample video of the training process. We only show a sample as there are too many images and the video file becomes too large to upload:

After training on the GPU for 50,000 iterations, our images look much more realistic.

Code

We have successfully built a DCGAN model and generated synthetic facial images on both a CPU and GPU. All code for this tutorial is available for download in an iPython format here.

References:

- Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks
- Introduction to GANs with Tensorflow
- GANs with Tensorflow
- DCGAN with Tensorflow
- LFW Sklearn Dataset Documentation
- LFW dataset