Extended Generative Adversarial Networks

John Hancock
College of Engineering and Computer Science
Florida Atlantic University
Boca Raton, United States of America
jhancoc4@fau.edu

***Abstract*—In this work we explore extending generative adversarial networks (GAN's). Authors of previous research document GAN's with one generator and one discriminator. In this work we explore GAN's with more than one generator or discriminator. We term the type of multiple generative adversarial networks in this work chained generative adversarial networks. We investigate the performance of these extended GAN's, with a special focus on the accuracy of discriminators in these chained generative adversarial networks.**

***Index Terms*—generative adversarial networks, gan, neural networks, deep learning**

I. Introduction

In this work our goal is to investigate the behavior of generative adversarial networks when one generative adversarial network uses the output of a previous generative adversarial network for input. Specifically we design a way to take the output of a base generative adversarial network, that is the output of both the generator and the discriminator of the base generative adversarial network, and use that as the input for a subsequent generative adversarial network. We then extend this iterative use of the output of previous generative adversarial networks 5 times, and investigate the changes in the performance of the subsequent generative adversarial networks, as well as the quality of the artifacts that the generative adversarial networks produce.

Generative adversarial networks are pairs of neural networks that generate artifacts in an iterative fashion. Goodfellow *et al*. Introduce Generative adversarial networks in their paper, "Generative adversarial nets" [1]. This paper describes a stable system comprised of a generator and a discriminator where the generator iteratively learns to create artifacts that the discriminator iteratively learns to discriminate whether the current input it sees is an artifact from the generator, or an instance of some dataset.

In [1] Goodfellow *et al*. Give a strong formal justification for the stability of their design, which inspired subsequent researchers to find good implementations of their design. One successful implementation is from Radford, Metz, and Chintala expose in [4]. This design is referenced in other works such as [11], the so-called least squares deep convolutional generative adversarial network. There are many other bodies of research that exploit the architecture in [4], and we see how the initial idea of Goodfellow *et al.* is extensible in and of itself, but also that subsequent refinements of it also take on lives of their own.

Alternatives to generative adversarial networks for generating artifacts that would appear to be instances of a dataset include Deep Boltzman Machines [14], and variational autoencoders [15].

The inspiration for this work is the notion that a generator, when armed with the previous result of another generator training to create the same kinds of artifacts might be able to create a better quality artifact. We find that this is not the case. However, our results show that some information may be preserved because when we pass the output of a previous generator to a subsequent generator, along with the value of the loss function associated with the value from that generator, then we find some noticeable impact on the accuracy of the discriminator in the subsequent generative adversarial network.

At the time of this writing, we do not find a similarly structured collection of generative adversarial networks that we we propose in the architecture we review in this work.

Additionally we would like to point out to the reader early on that the source code for the architecture and results we cover in this work is available with details in reference [12].

II. Related Work

The seminal work on generative adversarial networks is the paper entitled, "Generative Adversarial Nets" [1]. In this work, Goodfellow *et al.* give the formal basis for generative adversarial networks. They prove that under the proper conditions, a system of neural networks comprised of a generator and a discriminator will converge to a stable state. The discriminator has two sources of input: the output of the generator, and the instances of some dataset. The discriminator outputs a probability that the input it received is from the dataset. The generator and the discriminator are locked in an adversarial relationship in the sense that the generator's optimizer is driven using a loss function that incurs a high loss for when the discriminator gives a low probability that its input is from the dataset. At the same time the discriminator's own optimizer uses a loss function that incurs a high loss when the discriminator gives a probability that its input is from the generator when it is actually from the dataset, and vice-versa. The experiments we perform in this work show that it could be possible to chain generative adversarial networks in such a way that the ultimate discriminator is always capable of determining where its input is coming from, *i.e.* the generator, or the dataset. We are not capable of providing a formal

derivation for our empirical results at this time, so we do not state for certain that this is the case.

The formal derivation of the stability of generative adversarial networks that Goodfellow *et al.* find in [1], we believe, gave researchers confidence in pursuing implementations inspired by the idea that one could employ neural networks to create artifacts that sufficiently resemble instances of some dataset. A milestone in the evolution of generative adversarial network implementations is the work of Radford, Metz, and Chintala, in their work, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks" [4]. This work provides an architecture for generative adversarial networks using convolutional neural networks. Many researchers employ or cite this architecture in the work. Examples include:Fussell and Moews in [6], who incorporate Radfor, Metz,and Chintala's design into generative adversarial networks for generating images of galaxies. Goodfelow *et al.* mention that Radford, Metz, and Chintala's work in [4] is notable for its performance due to their architecture, and hyperparameter choices in their deep learning text, [8]. In our experiments we use a Keras implemenation of Radford, Metz, and Chintala's deep convolutional generative adversarial network as a basic building block. We found the keras implementation of Radford, Metz, and Chinatala's generative adversarial network design in [7].

A subsequent refinement of [4] is the least squares generative adversarial network [11]. In this work Mao *et al.* leverage Radford, Metz, and Chintala's architecture with a least squares loss function in the discriminator to produce higher quality artifacts from their generative adversarial networks. We feel that the existence of derivative works from the architecture that Radford, Metz, and Chintala give indicates that their architecture is a good starting point for exploratory work in generative adversarial networks, and another reason we chose to use their architecture as a starting point for our design.

We find much interest in using generative adversarial networks as transformative agents that manipulate their inputs into outputs that retain some aspects of their inputs; however, the output is unique in some way that has aesthetic appeal and appears in some sense natural. A popular application of generative adversarial networks in this domain is the pix2pix software that Isola, Zhu, Zhou, and Efros cover in [3].

In [3] the authors show examples of applications of GAN's to produce realistic images of things that do not exist, based on actual images, such that the produced images do not appear to be synthetic to human observers. Even though this may be the case, we posit that the result of the experiment we perform below shows that one should be capable of training a series of discriminators that would be able to distinguish the generated output from instances of some dataset that the generated output is meant to resemble. Often times, this point will be moot because the generated output is not intended to substitute for actual instances of some dataset. Nevertheless there is perhaps an application of our method for the purposes of forgery detection.

Researchers in the field of generative adversarial networks have turned their attention to leveraging the output of generators as inputs to successive stages of artifact generation.

We found recent interest in the medical application of generative adversarial networks to creating simulated images of scar tissue in human hearts in [2]. In this work Lau *et al.* use a first GAN to generate the shape of the scar tissue in an image, and then a second GAN to generate a refined image that uses the first image as input.

Further research along the lines of Lau *et al.*'s work lead us to the discovery of the StackGAN architecture of Zhang *et al.* [5]. The StackGAN architecture is similar to the architecture Lau et. al. propose. It consists of a stage-I generative adversarial network and a stage-II generative adversarial network. The stage-I generative adversarial network generates a basic image, and the stage-II generative adversarial network generates a refinement of the image coming from stage-I. Our work differs from Zhang *et al.*'s work in that we are interested in the behavior of the discriminator, and not the quality of the artifacts that the generator produces.

However, our work indicates that if we augment the loss function of the discriminator involved in generating the first image with the value of the loss function for generating the first image, and use that as input for a subsequent chain of generators then eventually the discriminators will be capable of distinguishing actual versus generated images with a high degree of accuracy. At this time we do not have a formal justification for this belief, but only the empirical result of our experiment.

Research for related work involving generative adversarial networks that use inputs other than pure noise brings us to the domain of conditioned generative adversarial networks [9]. Since Goodfellow *et al.* define generative adversarial networks as using vectors of noise as inputs we felt it important to confirm that researchers explore using different kinds of inputs for generators. Mizra and Osindero in [9] deliver a proof-of-concept work that adds labels to both the generator and discriminator inputs. The resulting generative adversarial network, after training, has a generator that is then capable of producing artifacts with labels as conditions. In [9] Mizra and Osindero give a table of generated MNIST [10] digits. The table shows that when their generative adversarial network is conditioned with a particular label as part of its input, it will produce an artifact that resembles the class of instances of the dataset that the label is a member of. We speculate that a formal justification for our result that implies successive discriminators are more capable of classifying inputs as generated, versus coming from an actual dataset, may stem from the notion that we are giving a component of the previous discriminator's loss function value as input to subsequent generators. We feel this is similar to the notion of giving data labels as inputs to the generator and discriminator.

After performing this research for related work, we did not find work that exactly matches our procedure for linking generative adversarial networks. We find that the output of our chained generative adversarial networks is unremarkable, however we feel the reader may wish to take note of the

increased accuracy of the series of discriminators that we find empirically.

III. Chained generative adversarial networks

This work is centered on a design for generative adversarial networks we call **chained generative adversarial networks**. The figure below illustrates the architecture of a chained generative adversarial network at a high level.

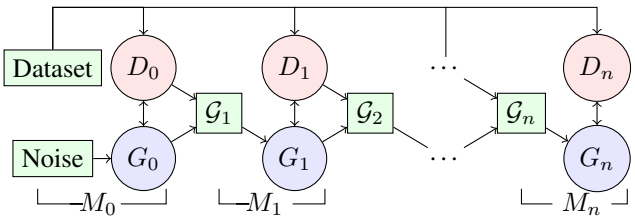

Fig. 1. Chained GANs architecture

The reader should follow figure 1, above from left to right. The chained generative adversarial networks are composed of models, that we denote as M_0, M_1, ..., M_n. Each model is then composed of a generator, labeled G_0, G_1, ..., G_n, and a discriminator, labeled D_0, D_1, ..., D_n. The first generator, G_0 is the same generator that E. Linder -Norén writes in [7], and it takes a noise vector as input. Likewise, the first Discriminator, D_0 is untouched from the implementation in [7], and it takes input from both some dataset, and the generator G_0. Please note the bidirectional arrows linking the generators and discriminators in the diagrams above. This is to indicate that the generator's output becomes input for the discriminator, as well as to indicate that we compute a loss function value based on the output of the discriminator that we feed back to the optimizer of the generator neural network.

To the right of the first generator and discriminator pair, we see the first generator, G_0, and discriminator, D_0 are linked via arrows pointing right to a box labeled with a $\mathcal{G}_1$. We are now transitioned to the second model, M_1, of our system. However, before we delve into M_1, we should point out that we pass the array of artifacts that G_0 creates, as well as the value of the loss function that we calculate in training D_0, to $\mathcal{G}_1$. One should envision these as traveling along the arrows pointing in to $\mathcal{G}_1$. Thus, $\mathcal{G}_1$ serves as the input for the generator G_1 of the second model M_1 of our chained generative adversarial network architecture. The second, and subsequent models of the chained generative adversarial network, M_i, do not use noise vectors as their inputs, rather they use the appropriate $\mathcal{G}_i$ as input. We compute $\mathcal{G}_i$ as follows: the generators G_i all generate an array of arrays of output values. We flatten the arrays and augment them with the value of the loss function that we compute when training the respective D_i. Hence each flattened array is augmented with the same loss function value for a training step of one of the M_i.

To give a concrete example of how the process of computing $\mathcal{G}_i$ works, we ask the reader to consider the output of the generator E. Linder-Norén gives in [7]. This generator, which is our generator G_0, generates a batch of 32 28×28 arrays meant to be interpreted as members of the MNIST hand written digits [10] dataset. Note that these 28×28 arrays are not actually data from the MNIST dataset, but they are fakes meant to appear to be instances of the dataset. G_0 creates the batch of 32 28×28 fake images by starting with a batch of 32 noise vectors, and then applying several convolutional transformations to them. Once G_0 produces the batch of 32 28×28 fake MNIST images, we can use this batch of fake images as input for D_0. We do this passing the batch of 32 28×28 images as an input to the train_on_batch function that Keras provides for the D_0 discriminator neural network. Train_on_batch returns the value of the loss function that it computes in order to optimize the neural network it is invoked on - in this case the neural network of the first discriminator, D_0. We save the value of this loss function for training D_0 with the batch of 32 28×28 fake MNIST images, plus a reference to the batch of 32 28×28 fake MNIST images for use in training the next model, which in this case will be M_1. Now we need to find the input values for the generator G_1, that is a component of M_1. In order to do this, we use the reference we saved for the batch of 32 28×28 fake MNIST images, and we flatten each of the 32 images into vectors of length 784. ($784 = 28 \times 28$). We then add one element to each of these 32 arrays, which is the value of the loss function for training D_0 with the output of G_0 as the input for D_0. At this point we have computed the value of $\mathcal{G}_1$ that we can use as input for the generator G_1 of the second model M_1 in our chained generative adversarial network architecture.

The functioning of subsequent models, M_2, M_3, ..., M_n in the architecture we describe in the digram 1 is similar to what we have describe for the functioning of the second model, M_1. We feel this is important for the reader to grasp, and bear in mind as we report the result of the accuracy we record for the discriminators D_i in this architecture. Although models M_1 and later function with the same mechanics, there appears to be an increase in the accuracy of later discriminators that we show in the experiments results.

Now that we have given an overview of the high level functioning of the overall chained generative adversarial networks system, we give some details on the interactions of the components of the individual models. The next diagram we discuss is the diagram for the functioning of the first generative adversarial network in the chained generative adversarial network system, the model M_0.

Before we delve into the details on models, we would like the reader to bear in mind that the loss function we use in the implementation of this design is the binary cross-entropy loss function, which is suitable because the discriminator is a classifier that decides if its input is in the class of generated instances from G_0, or real instances from the input dataset.

Please see figure 2.

As noted in the discussion of our high-level architecture, the first model of the chained generative adversarial network feeds noise vectors as inputs to the generator of the generative adversarial network in M_0. We represent this with the box labeled, "Noise vector," in the lower left hand side of figure

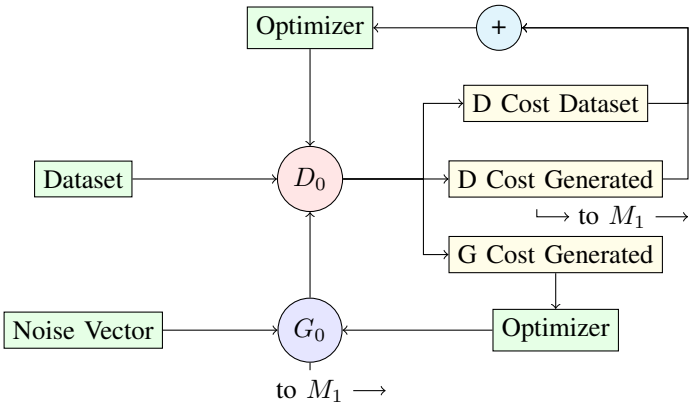

Fig. 2. Chained GANs base Model, M_0

2 above, and the reader should take this as the starting point when perusing this diagram. We pass the noise vector as the input value for the predict function Keras provides for neural networks, which in this case is the generator, G_0. This allows us to get an output value from the generator G_0, without changing any parameters of the generator G_0 neural network. We are now following the arrow emanating from the top of G_0 going to D_0 in the diagram above. Also, note that there is a second arrow pointing into the neural network D_0 that indicates D_0 is getting input from a dataset. The discriminator of a generative adversarial network generally receives inputs from two sources: the first source is the generator, G_0 in this case, and the second is some dataset, which in this work is the MNIST dataset. For neural networks, Keras defines a train_on_batch function requires at least two inputs, which are the input values for training, along with the expected output values for each particular input value. For the discriminator D_0 of our first generative adversarial network M_0 pictured above, invoke the train_on_batch function of D_0 twice. Once where we give the output of G_0 as input to D_0, and the expected value of 0 for every generated output from G_0 because we wish to train D_0 to give an output of 0, which represents a probability of 0 that the output of the generator G_0 could have come from the dataset, which is the MNIST dataset in our work. Calling the discriminator D_0's train_on_batch function with the output of the generator G_0 and 0 as input values has the side effect of optimizing D_0 for one training step, and the resulting output value of this train_on_batch function will be the value of the loss function the Keras library computes when optimizing the discriminator D_0. We make a similar call to the discriminator D_0's train_on_batch function but we use a batch of instances of the MNIST dataset, and expected values of all 1's for the input value of train_on_batch because now we wish to train the discriminator to output values of 1 whenever it has an input from the genuine dataset. In this work an input of the genuine dataset would be an image from the MNIST dataset. Finally, we calculate a third cost used in training the generator, but the manner this is done is somewhat circuitous and roundabout. One should notice in the lower right-hand side of figure 2 the box labeled, "G Cost Generated," We compute this cost from the discriminator D_0 again, but we take advantage of the Keras library by defining a Model that is comprised of D_0, and G_0 where we specify that the D_0 component is not trainable. Then we invoke the train_on_batch function of this model, which for our purposes is presently labeled M_0, which starts the process from generating a batch of 32 28×28 fake images out of noise vectors using generator G_0, and so on, leading up to D_0 finally outputting a value of a loss function. Note that in the code that accompanies this work [12], as well as in the original code by E. Linder-Norén that models such as M_0 depicted above are named as variables that start with the word, "combined," in the variable name.

This concludes our discussion of the basic generative adversarial network that serves as the building block for the system that is the chained generative adversarial network we discuss in this work. Now we move on to the slightly different architecture of the successive models that constitute the remainder of the chained generative adversarial network. Please see the digram in figure 3.

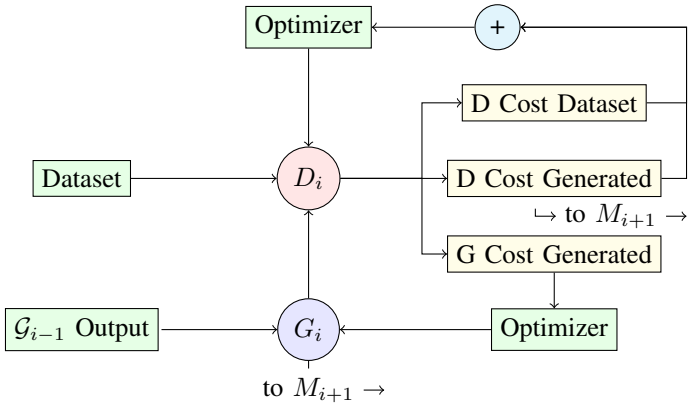

Fig. 3. Chained GANs successor model M_i

As one can surmise from the similarities in figures 3, and 2 there is not much functional difference between the generative adversarial networks that constitute the models M_0, and the subsequent models M_1, M_2, $\ldots M_n$. The key difference is the input to the generator, depicted as G_i above. The input to the generator is now the value of the $\mathcal{G}$ function that we describe in the commentary on the high level functioning of the chained generative adversarial network that accompanies figure 1. Otherwise the functioning of the generative adversarial networks that comprise models M_1, M_2, $\ldots M_n$ is the same.

Now we move on to discuss the internals of the discriminators in our chained generative adversarial networks.

The discriminators D_0, D_1, $\ldots D_n$ in a chained generative adversarial network are convolutional neural networks. For our work, because we rely so heavily on the Keras library, the Keras Conv2D function is critical to the functioning of the discriminator. The input layer of the discriminator takes batch of 32 28×28 images and performs a 32 convolutions (one for each filter) that result in a batch of 32 14×14 arrays. The output size is cut in half from 28×28 to

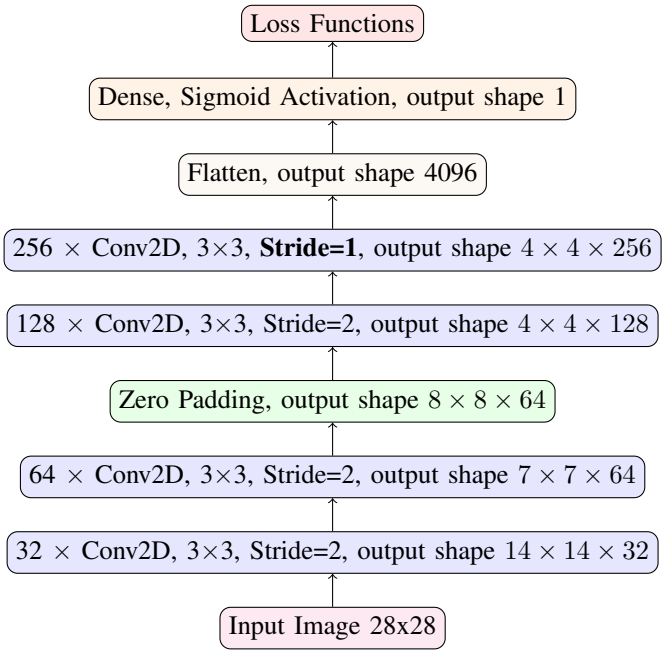

Fig. 4. Discriminator architecture

14×14 by dint of using a stride value of two. The next layer of the discriminator is a similar convolution that cuts the outupt to a batch of 7×7 arrays. Note that we must use Keras' ZeroPadding layer functionality in order to continue - this results in a batch of 8×8 arrays that are easier to then generate smaller feature maps from as we proceed with convolutions. Finally we flatten the feature maps to a 4,096 element dense layer with a sigmoid activation function. It is the output of this sigmoid activation function that we interpret as the probability that the discriminiators D_0, D_1, ..., D_n give that its input is a genuine instance of a dataset. It is most important for the reader to note that we omit some layers in the digram for the discriminators of the chained generative adversarial networks in order to keep the diagram to a manageable size. Following every convolutional layer in the neural networks that constitute the discriminators, we have a leaky rectified linear unit (LeakyReLU) layer that serves as the activation function for that layer. In our work, we use the hyperparameter α value of 0.2 for the LeakyRelU layers. Also accompanying every convolutional layer except the first convolutional layer is a batch normalization layer that we apply before the LeakyReLU layer. Batch normalization refers to the process of rescaling the output of the convolutional layer so that all values have zero mean and unit variance. Radford, Metz, and Chintala write in [4] that this practice has the effect of stabilizing learning. Finally after each LeakyReLU layer we have a dropout layer that is also not pictured in the diagram above, but is nevertheless a used multiple times in the discriminators. While the authors of [4] do not mention using any dropout layers, we note that Goodfellow *et al.* recommend using dropout in [8] where they write, "Dropout seems to be important in the discriminator network." We take this as the reason that E. Linder-Norén uses dropout layers in [7], and we use his Keras implemenation of deep convolutional generative adversarial networks as the basis for our chained generative adversarial networks.

This concludes our discussion of the discriminator portion of the models of the chained generative adversarial network system, and we move on to give the details of the generators. Please see the diagram in figure 5 below, and the commentary that accompanies it.

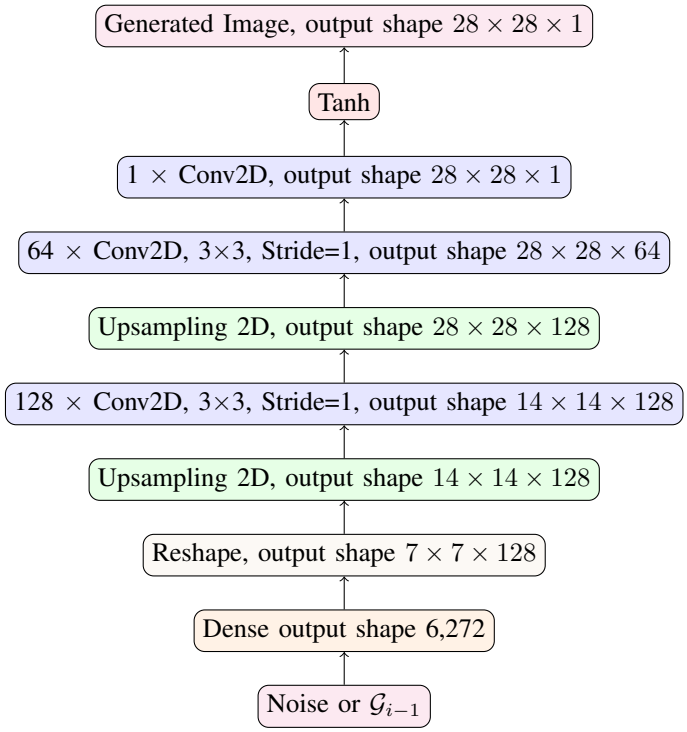

Fig. 5. Generator architecture

The generators of the chained generative adversarial network system share a common architecture, the main difference between any of them is that which we highlighted above. This is the difference that the first generator uses vectors of random noise for input, whereas subsequent generators use the augmented output of the previous generators via the $\mathcal{G}$ function we described above.

Each generator begins with a batch of vectors that we feed to a dense layer. The first generator will use a a batch of random valued vectors where the values are drawn from the normal distribution on $(0, 1)$. Otheriwse the input to the first layer of the generator will be a vector with elements that are the values of the $\mathcal{G}$ function. Regardless of how we obtain this input, in all the generators we feed this to a dense network that we reshape to a batch of $7 \times 7 \times 128$ feature maps and feed to subsequent upsampling and convolutional layers. Convolutional layers are necessary so that we have trainable parameters, and we require upsampling to expand feature maps into arrays large enough that we can interpret as fictitious instances of some dataset. In order to keep the diagram to a manageable size we omit relu activations in every layer except the final layer, which we

expliclitly depict as a tanh layer in the diagram above. Radford, Metz and Chintala explain that we use a tanh activation in the output layer in order that we may more easily interpret the output of the generator as an image in [4].

Finally, to further elucidate the architecture of the chained generative adversarial network system, we present a section of pseudocode that summarizes the training steps of the components of the system.

We give pseudocode for the key training steps of the CGAN's below. Please keep in mind the following definitions and notations when reading the pseudocode:

- a model M_i is a system of two neural networks D_i and G_i. We codify this relationship with the notation $M_i = (G_i, D_i)$. Model and GAN are synonyms.
- This work covers systems of generators and discriminators (models) that are ordered in the sense that successor models use the output of predecessor models for their inputs. Hence D_0 is the first discriminator, and G_0 is the first generator. D_i is a successor discriminator, and G_i is a successor generator. D_n, and G_n, are the last discriminator, and generator, respectively.
- The entire collection of models (GAN's) is $\mathbf{M}$.
- $\mathbf{Z}$ is a list of vectors $\hat{\mathbf{z}}_i$, where the components of each $\hat{\mathbf{z}}_i$ is a random number sampled from the normal distribution on $(0, 1)$.
- $G_0(\mathbf{Z})$ is the output of the zeroth generator G_0 for the input values of $\mathbf{Z}$.
- $D_0(G_0)$ is the output of the zeroth discriminator for the input value of $G_0 = G_0(\mathbf{Z})$.
- $\mathbf{X}$ is a list of instances of a dataset $\mathbf{S}$. In GAN's the discriminator is a classifier that attempts to label its inputs as either genuine instances of $\mathbf{S}$ or synthetic imposters that some generator has created. In the code that accompanies this work, 1 is the desired output value of the discriminator if it is given an element of $\mathbf{X}$ as an input value, and 0 is the desired output value of the discriminator if it is given the output value of some generator as an input value.
- $L(D_i(\mathbf{X}))$ is the loss function the model M_i (GAN) uses in conjunction with the optimizer for the output value of a discriminator D_i for input values $\mathbf{X}$.
- $L(D_i(G_i))$ is the loss function evaluated on the output of discriminator D_i when D_i uses the output of G_i as input.
- $L(M_i)$ is the value of the loss function of with the value of the combined model $M_i = (G_i, D_i)$. In our code, we train a combined model to update the generator, however, in that combined model the discriminator is configured to be **not** trainable. Therefore the optimizer only updates the generator component of the GAN during the portion of training where the generator is trained.
- $\mathcal{G}_i = \mathcal{G}(G_{i-1}, L(D_{i-1}(G_{i-1})))$ is the input for generator G_i that we calculate as the function $\mathcal{G}$ that flattens the output of G_{i-1} to 1-dimensional array, and then concatenates the value of the loss function $L(D_{i-1}(G_{i-1}))$ to it. One should keep this in mind in contrast with the vectors of random numbers $\mathbf{Z}$ that are the input values for the first generator, G_0.

Please see the pseudocode labeled 1 for the pseudocode for the training step of the chained GAN's. The pseudocode utilizes all the notation in the previous list.

Data: Inputs: $\mathbf{Z}$, $\mathbf{X}$
Result: Trained system of GAN's $\mathbf{M}$
initialization;
use standard Keras layer components to instantiate n identical generator neural networks G_0, G_1, ..., G_n;
use standard Keras layer components to instantiate n identical discriminator neural networks D_0, D_1, ..., D_n;
create combined models M_0, M_1, ..., M_n such that $M_i = (G_i, D_i)$;
for *number of epochs* **do**
  note: Keras *train batch* function has the side effect of updating the neural network's trainable parameters;
  $L(D_0(G_0)), D_0(G_0) \leftarrow$ train batch $(D_0(G_0(\mathbf{Z})))$;
  $L(D_0(\mathbf{X})) \leftarrow$ *train batch* $(D_0(\mathbf{X}))$;
  $L(M_0) \leftarrow$ *train batch* (M_0);
  for *i = 1 to number of successor GAN's* **do**
    $\mathcal{G}_i \leftarrow \mathcal{G}(G_{i-1}, L(D_{i-1}(G_{i-1})))$;
    $L(D_i(G_i)), D_i(G_i) \leftarrow$ train batch $(D_i(G_i(\mathcal{G}_{i-1})))$;
    $L(D_i(\mathbf{X})) \leftarrow$ *train batch* $(D_i(\mathbf{X}))$;
    $L(M_i) \leftarrow$ *train batch* (M_i);
  end
end

Algorithm 1: Pseudocode for chained GAN's training steps

We hope the reader has a clear idea of the functioning of the chained generative adversarial network system at its various levels. We now move on to discuss the experiments we perform with this system.

IV. Experiments

The main purpose of the experiments we perform is to investigate the behavior of the generators and discriminators in the chained generative adversarial network system as the output of the generators flows through the system. Specifically we wish to know how the accuracy and the loss changes when we look at values of these metrics for the generator G_0, discriminator D_0, and subsequent generators and discriminators G_1, D_1, and so on.

The tools we use to conduct the experiments we report on below are as follows. We started with the python code in [7] from Github. We then forked this code and modified it to the architecture we present in the previous section. In addition, we incorporated a Tensorflow tensorboard callback function to facilitate recording the data we report below. In order to get a feel for the repeatability of the results, we conducted 3 trials. We compiled the latest tensorflow source code using commit a49887b0d24c9c1a97c5915147473bb09f854ebc from November 25th. We compiled this version of tensorflow

specifically for a g3s.xlarge Amazon Web Services EC2 instance to take advantage of the Nvidia Tesla M-60 GPU that Amazon includes with g3s instances. In addition we the AWS g3s instance we used runs Ubuntu version 16, and we use the instructions in [13] in order to compile tensorflow from source. In order to modify the code in [7] to achieve our implementation, we used the Pycharm integrated development environment.

The baseline method is the behavior of the first generative adversarial network in the chained generative adversarial network system. The output of the generators is most instructive in describing the functionality of the base system. For these experiments, the output is images that resemble MNIST images. First, consider the output of the untrained generative adversarial network. The image below is from the Keras implementation of Radford, Metz, and Chintala's deep convolutional generative adversarial networks, from [7].

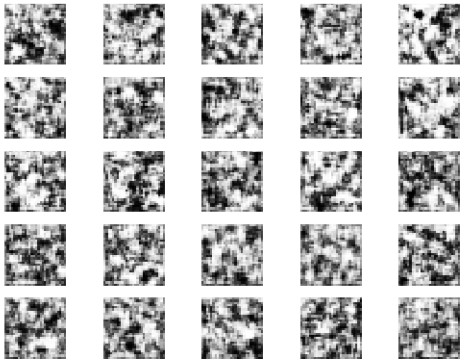

Fig. 6. First output of base model generated images

Note how in the image above, the images resemble static on television screens because they are images with random greyscale values as pixels. The next image shows the behavior of the base model after convergence. Note that the images now appear to be more organized, and they resemble handwritten digits:

We give the images of the output of the baseline system, which is the Keras implementation of the deep convolutional generative adversarial network from [7], to emphasize the functionality of the code we copy as a starting point for our work. It clearly defines a deep convolutional generative adversarial network with a tangible result of a progression of generated images that start off as noise, and progress to images that closely resemble images of the MNIST dataset, which is what the generative adversarial network in [7] is designed to generate.

The performance measures we look at for our experiments are: accuracy of the generators and discriminators, and the value of the loss functions for the generators and the discriminators. We leave it as a topic for future work to include more metrics.

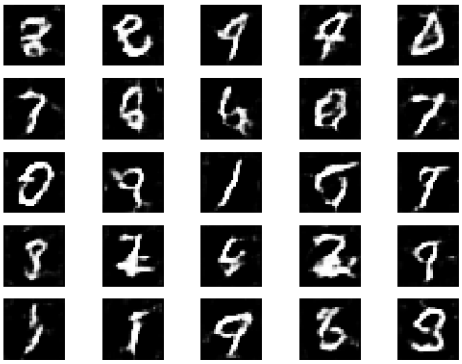

Fig. 7. Epoch 1,750 output of base model generated images

In order to conduct our experiments we extend the baseline of 4,000 epochs to 8,000 epochs to attempt to detect if the chaining of the output of subsequent generators has some instability that that the baseline system does not encounter.

For our experiments we fix the number of generative adversarial networks in the system of chained generative adversarial networks at 6. We have one baseline generative adversarial network, chained to 5 successive generative adversarial networks M_1, through M_5. Thus the first generative adversarial network embodied in M_0 has generator G_0, and discriminator D_0, and so on.

First of all we show the performance measures for the baseline system. The loss function for generators of the baseline system is as follows:

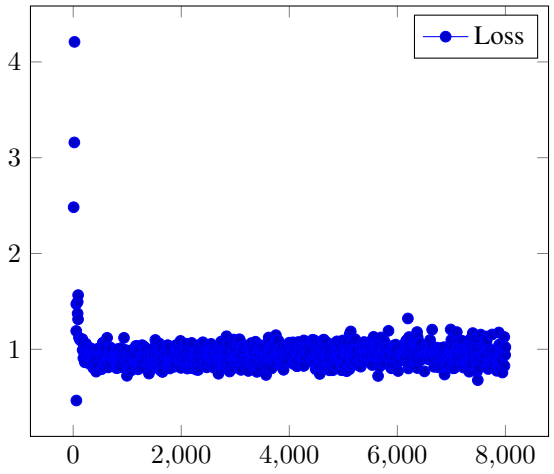

Fig. 8. Baseline generator loss, x-axis is epoch number, y-axis is value of binary cross entropy loss function

We see the expected behavior that the generator trains to quickly minimize the value of its loss function, and then subsequently the value of the loss function hovers randomly in a narrow range.

The loss functions of the chain of generative adversarial networks that we give the architecture for in the previous section are in the charts below:

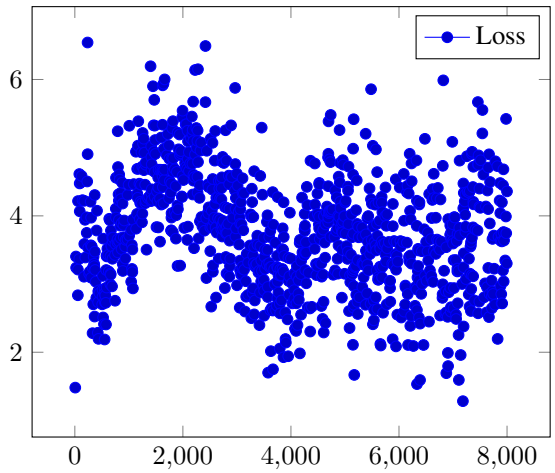

Fig. 9. generator G_1 loss, x-axis is epoch number, y-axis is value of binary cross entropy loss function

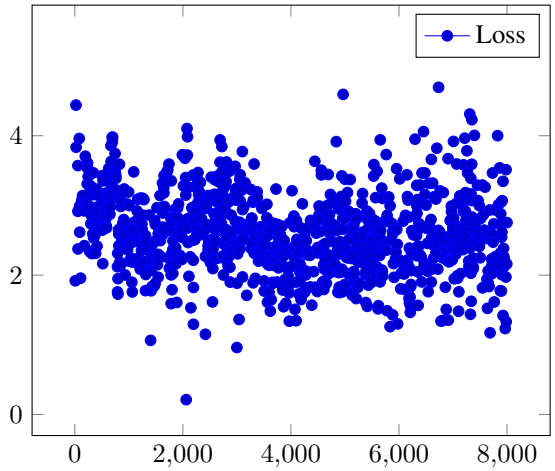

Fig. 10. generator G_2 loss, x-axis is epoch number, y-axis is value of binary cross entropy loss function

We find nothing interesting in the successive generator's accuracies in figures 9 through 13. They resemble scatter plots of random values ranging from 2 to 6. However we see there is some compelling reason why this may be the case, the generators appear to flail around randomly because the discriminators are able to give accurate probabilities that their input data is genuine data from a dataset, and the generators are unable to learn to fool the discriminators as they should be in the convergence state that Goodfellow *et al.* describe in [1].

First we report the performance of the baseline discriminator loss function and accuracy:

Again as in the case with the generator, in figure 14 we see that the discriminator's loss function starts high and then quickly converges to a stable value.

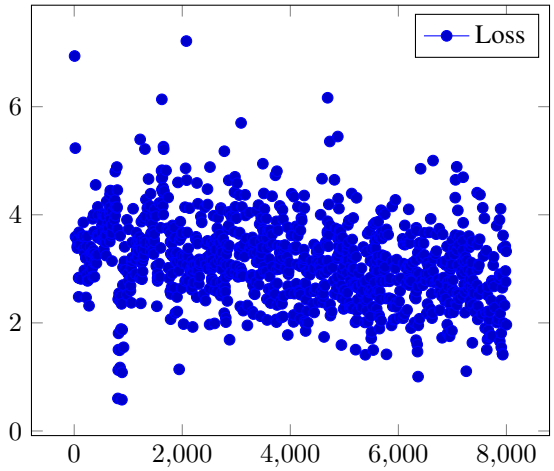

Fig. 11. generator G_3 loss, x-axis is epoch number, y-axis is value of binary cross entropy loss function

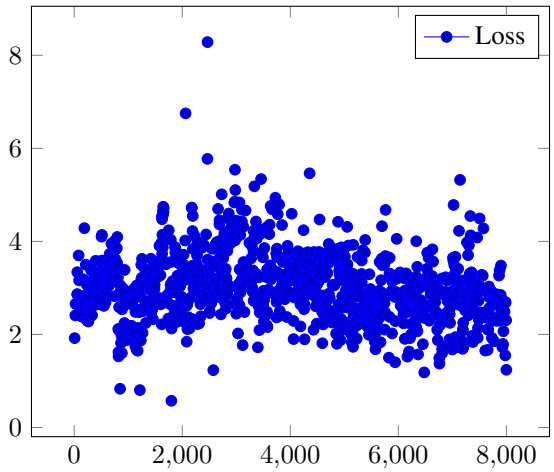

Fig. 12. generator G_4 loss, x-axis is epoch number, y-axis is value of binary cross entropy loss function

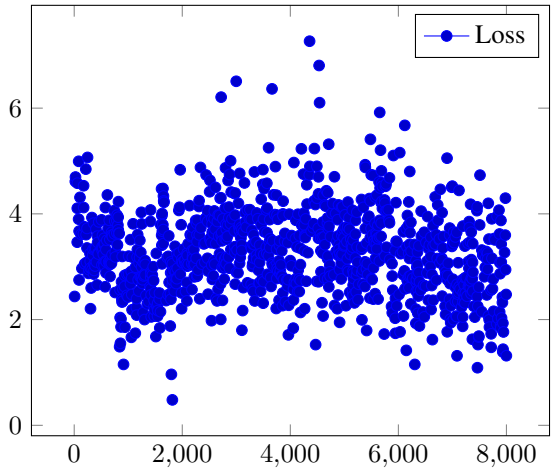

Fig. 13. generator G_5 loss, x-axis is epoch number, y-axis is value of binary cross entropy loss function

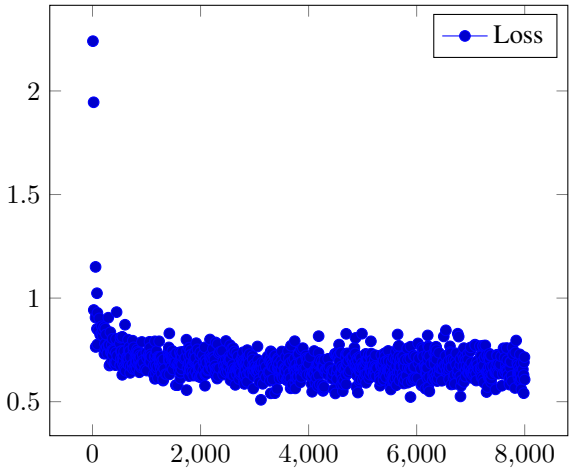

Fig. 14. Baseline discriminator D_0 loss, x-axis is epoch number, y-axis is value of binary cross entropy loss function

Next we look at the accuracy of the baseline discriminator network.

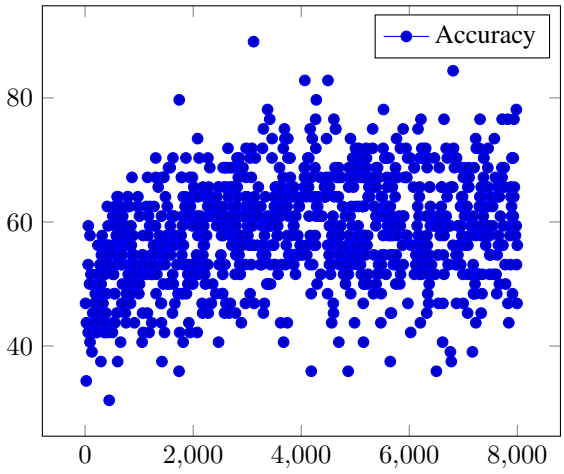

Fig. 15. Baseline discriminator D_0 accuracy, x-axis is epoch number, y-axis is percentage of correctly classified input values

In figure 15 we see the result Goodfellow *et al.* predict in [1]; the discriminator accuracy resembles a scatterplot of random values around the 60% mark. This is because the generator learns to fool the discriminator and the discriminator can do no better than give about a 0.5 probability that the input it is given is coming from a real dataset versus from a generator, in this case it is the generator G_0 we discuss above.

Now we move on to look at the plots for the subsequent discriminators D_1 through D_5:

We note nothing important in figure 16. However, we see that figure 17 reflects a higher accuracy for the discriminator D_1. We move on to show charts for the accuracies for the remaining discriminators since we do not find anything interesting in the loss values for the remaining discriminators.

We notice the conintued trend of higher accuracies for subsequent discriminators. It also appears that there may be a

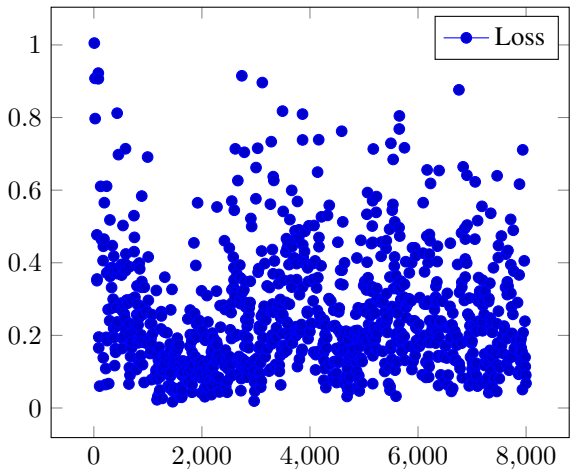

Fig. 16. Baseline discriminator D_1 loss, x-axis is epoch number, y-axis is value of binary cross entropy loss function

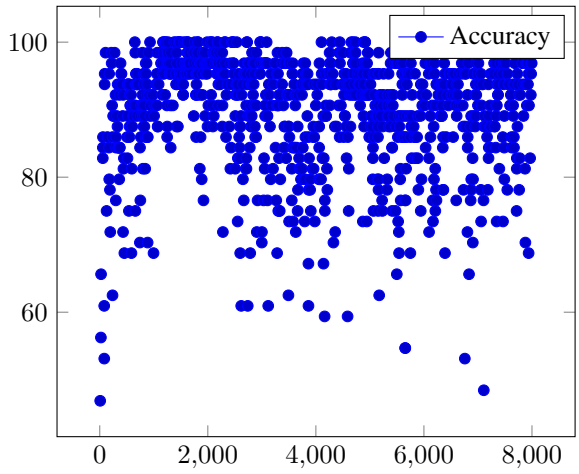

Fig. 17. Discriminator D_1 accuracy, x-axis is epoch number, y-axis is percentage of correctly classified input values

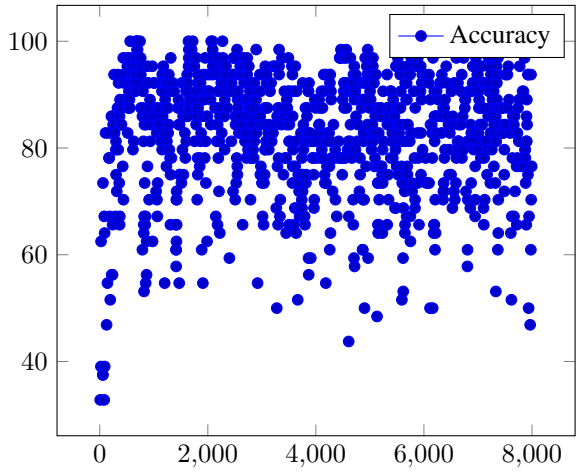

Fig. 18. discriminator D_2 accuracy, x-axis is epoch number, y-axis is percentage of correctly classified input values

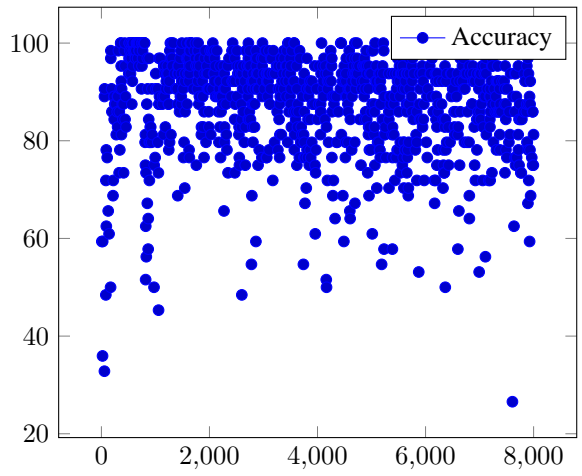

Fig. 19. discriminator D_3 accuracy, x-axis is epoch number, y-axis is percentage of correctly classified input values

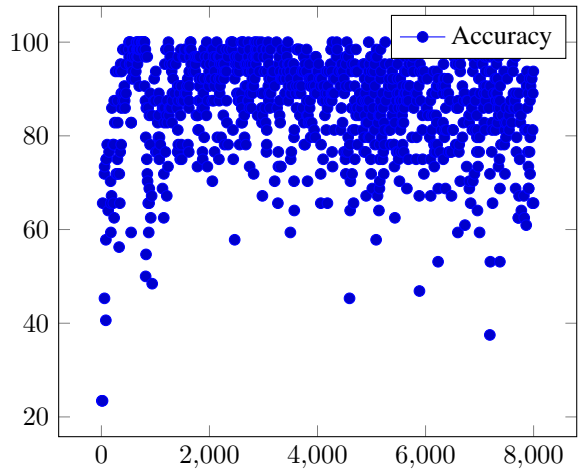

Fig. 20. discriminator D_4 accuracy, x-axis is epoch number, y-axis is percentage of correctly classified input values

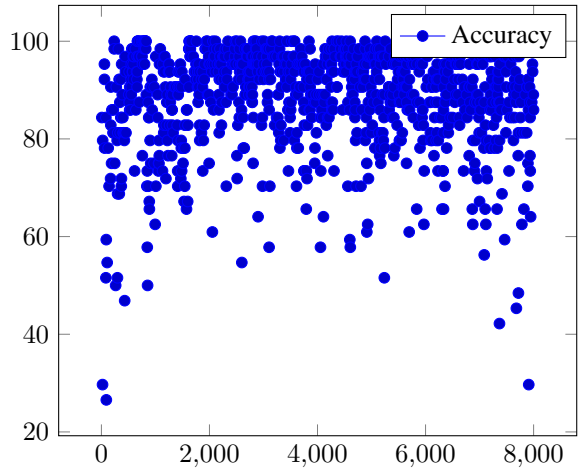

Fig. 21. discriminator D_5 accuracy, x-axis is epoch number, y-axis is percentage of correctly classified input values

greater percentage of accuracy values captured that are closer to the top of the range.

Therefore we decided to repeat this experiment two more times, and compute summary statitics of the accuracy values for the discriminators. We export the data that tensorboard captures while we run these experiments as csv files, and report mean accuracy values along with the standard deviations of the accuracy values, to give the reader an indication of the spread of these accuracy values for each of the three trials we perform.

Trial	Discriminator	Accuracy	Standard Deviation
1	0	58.35	8.44
1	1	89.91	8.63
1	2	82.61	11.35
1	3	87.38	10.37
1	4	86.53	10.71
1	5	88.42	10.41
2	0	59.76	8.61
2	1	91.62	8.65
2	2	87.62	10.29
2	3	88.0	11.38
2	4	91.16	9.64
2	5	92.13	9.58
3	0	59.08	8.71
3	1	90.90	8.83
3	2	90.19	10.39
3	3	90.03	9.95
3	4	90.54	10.04
3	5	92.76	9.34

TABLE I
MEAN ACCURACY FOR DISCRIMINATORS, NON-BASELINE DISCRIMINATOR ACCURACY IS CONSISTENTLY HIGHER

We notice that the mean accuracy for the non-baseline discriminators in table I is higher than 0.5. This would indicate that the chained discriminators are not confused about whether the output they are seeing is from a generator, or a dataset. The generators G_0 through G_5 produce output in the same format, and the discriminators D_0 through D_5 get input from either the respective generator or the dataset. However, there seems to be some observable effect happening that when the generators generate output not from noise, but from some values based on the output of a previous generator, that the discriminators are better able to classifiy their input as either coming from a generator or coming from an actual dataset.

These emprical results support the statement in the previous pararagraph, but we do not have a theory as to why this is the case at this time.

Finally, we pick a randomly chosen batch of images generated from the M_5 generative adversarial network to illustrate that, to the human viewer, the output of the generative adversarial newtork is different or of noticably lower quality than the images we see in 22.

V. CONCLUSIONS

In this work we present an architecture for chaining generative adversarial networks together.

The key to our implementation is to use the output of a basis generative adversarial network that works along the

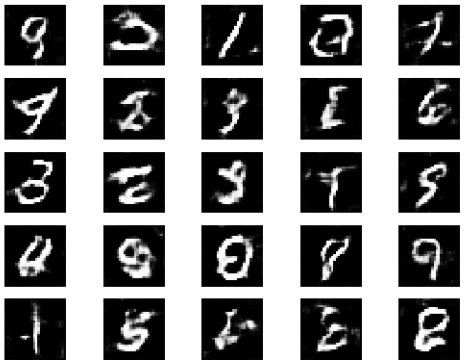

Fig. 22. Epoch 1,750 output of model M_5 generated images

lines of the deep convolutional generative adversarial network that Radford, Metz, and Chintala give in [4] as the input for similarly functioning generative adversarial networks in an iterative fashion.

We devise a function that incorporates one of the values of the loss functions that are typically computed to train the component generator and discriminator of a generative adversarial network with the output of the generator of a generative adversarial network, and we use the output of this function as a way to tie successive members of a group of generative adversarial networks together.

We report data from experiments that supports the hypothesis that when a generator's input is not from a random source, but is the output of a generator trained in a generative adversarial network to fit the same dataset, then the discriminators in the successive generative adversarial networks are able to identify instances of the dataset, versus generated instances with a higher degree of accuracy.

Interestingly, the quality of output generated from the successive generative adversarial networks that do not use random noise as inputs for the generators does not appear to be of remarkably lower quality. This implies that the architecture we give is a stable method for linking multiple generative adversarial networks together to perform some function.

We leave for future work the tasks of determining the repeatability of this work over larger combinations of hyperparameters that we have fixed in the code in [12], as well as assessing the repeatability of these results with different baseline generative adversarial network components, as opposed to the deep convolutional generative adversarial framework that we copy from Radford, Metz and Chintala from [4]. Another avenue of future research is to gather additional metrics from the component generative adversarial networks and see if other metrics follow some noticeable trend as discriminator accuracy appears to follow in our results. Finally, we offer no formal justification for the increased accuracy of the discriminators that we observe, therefore it is an important future work to find such a derivation should it exist.

VI. References

References

[1] I. Goodfellow *et al.*, "Generative Adversarial Nets," Neural Information Systems Processing Conference, 2014. [Online]. Available: https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf. [Accessed Dec. 2, 2018].

[2] Lau *et al.*, "ScarGAN: Chained Generative Adversarial Networks to Simulate Pathological Tissue on Cardiovascular MR Scans," August 2018. Available:arXiv:1808.04500 [cs.CV], https://arxiv.org/pdf/1808.04500v1.pdf. [Accessed November 12, 2018].

[3] P. Isola, J.Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-Image Translation with Conditional Adversarial Networks," Available: arXiv:1611.07004 [cs.CV], http://www.umi.com/proquest/. [Accessed November 23, 2018].

[4] S. Chintala, L. Metz, and A. Radford, Unsupervised representation learning with deep convolutional generative adversarial networks. 2016. [Online]. Available: arXiv:1511.06434v2 [cs.LG] https://arxiv.org/pdf/1511.06434.pdf.

[5] Zhang *et al.*, "StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks," August 2017. Available: arXiv:1612.03242 [cs.CV] , https://arxiv.org/pdf/1612.03242.pdf. [Accessed November 12, 2018].

[6] L. Fussell and B. Moews, "Forging new worlds: high-resolution synthetic galaxies with chained generative adversarial networks," December 2018. Available: arXiv:1811.03081 [astro-ph.IM], https://arxiv.org/pdf/1811.03081.pdf. [Accessed December 7, 2018].

[7] E. Linder-Norén, Keras implementations of Generative Adversarial Networks, November, 2018. [Online]. Available: https://github.com/eriklindernoren/Keras-GAN. [Accessed December 1, 2018].

[8] Y. Bengio, A. Courville, I. Goodfellow. (2016). "Chapter 20 Deep Generative Models," 2016. [Online] Available: https://www.deeplearningbook.org/contents/generative_models.html. [Accessed: November 20, 2018].

[9] M. Mirza and S. Osindero, "Conditional Generative Adversarial Nets," November 2014. Available: arXiv:1411.1784 [cs.LG], https://arxiv.org/pdf/1411.1784.pdf. [Accessed November 17, 2018].

[10] Y. LeCunn, C. Cortes and C. J.C. Burges, "The MNIST Database of Handwritten Digits," yann.lecun.com. [Online]. Available: http://yann.lecun.com/exdb/mnist/ [Accessed Nov. 29, 2018].

[11] X. Mao, *et al.*, "Least Squares Generative Adversarial Networks,", April 2017. Available: arXiv:1611.04076 [cs.CV], https://arxiv.org/pdf/1611.04076.pdf. [Accessed November 18, 2018].

[12] J. Hancock., "chain-gan.py" December 2018. [Online]. Available: https://github.com/jhancock1975/Keras-GAN/blob/term/chain-gan/chain-gan.py. [Accessed December 10, 2018].

[13] Tensorflow.org, "Build from source," [Online]. Available: https://www.tensorflow.org/install/source. [Accessed: Nov. 12, 2018].

[14] R. Salakhutdinovand and G. Hinton, "Deep Boltzmann Machines," Proceedings of the 12th International Confe-rence on Artificial Intelligence and Statistics (AISTATS). 2009. Available: http://proceedings.mlr.press/v5/salakhutdinov09a/salakhutdinov09a.pdf. [Accessed November 30, 2018].

[15] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes", May 2014. Available arXiv:1312.6114 [stat.ML] https://arxiv.org/pdf/1312.6114.pdf [Accessed December 1, 2018]