Implementation of Generative Adversarial Networks in HPCC Systems using GNN Bundle

Ambu Karthik
Computer Science and Engineering
R. V. College of Engineering
Bengaluru, India
Email: ambukarthik.cs18@rvce.edu.in

Prof. Jyothi S.
Computer Science and Engineering
R. V. College of Engineering
Bengaluru, India
Email: jyothis@rvce.edu.in

Dr. Shobha G.
Computer Science and Engineering
R. V. College of Engineering
Bengaluru, India
Email: shobhag@rvce.edu.in

Roger Dev
Sr. Architect, Machine Learning
Lexis Nexis Risk Solutions
Alpharetta, Georgia, United States
Email: Roger.Dev@lexisnexisrisk.com

***Abstract*—HPCC Systems, an open source cluster computing platform for big data analytics consists of Generalized Neural Network bundle with a wide variety of features which can be used for various neural network applications. To enhance the functionality of the bundle, this paper proposes the design and development of Generative Adversarial Networks (GANs) on HPCC Systems platform using ECL, a declarative language on which HPCC Systems works. GANs have been developed on the HPCC Platform by defining the Generator and Discriminator models separately, and training them by batches in the same epoch. In order to make sure that they train as adversaries, a certain weights transfer methodology was implemented. MNIST dataset which has been used to test the proposed approach has provided satisfactory results. The results obtained were unique images very similar to the MNIST dataset, as it were expected.**

***Index Terms*—HPCC systems, Generative Adversarial Networks, Deep Learning, Artificial Neural Networks, Enterprise Control Language**

I. Introduction

The Open source HPCC (High performance Computing Cluster) system is a cluster computing platform used for Big Data applications [1]. For managing the data flow in HPCC while preserving the cluster-computing capabilities, a special language called Enterprise Control Language, abbreviated as ECL was developed using C++. This language has many functions, data types, language embedding and various bundles as support for the user's convenience. This makes it a powerful language for big data processing and computation. HPCC Systems platform also has a tool called ECL Watch on the cluster-computing server which enables you to manage multiple nodes, input files so they are accessible to all nodes (called spraying), output files by merging processed data from all nodes (called despraying) and to see the various outputs and logs of the process that is run [2].

Due to the cluster computing scalability of the platform, it provides for processing many petabytes of data, making it immensely powerful for its applications in Machine learning. For this, a bundle for performing Machine learning operations on the ECL language was developed and is available as an open source bundle hosted on github [3]. This was further used to develop an ECL bundle capable of using Neural Networks which is called Generalised Neural Networks [4]. More about this will be covered in Section II.

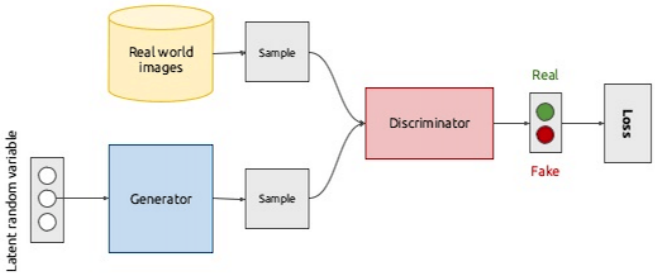


Figure 1: Architecture of GAN

Technology over the previous decade developed in statistical learning using various machine learning metrics and deep learning neural networks, through which the feature recognition capabilities for images, audio files or even text improved to amazing levels. In the last decade, there came a marvellous idea to utilise the feature recognition capabilities of these models to be able to generate data. Utilising a minimax function with a pair of neural networks behaving as adversaries gave birth to a superior neural network framework capable of generating completely new data with similar patterns, which due to their behaviour was given the name Generative Adversarial Networks, abbreviated as GANs.

GANs are a pair of models which behave as adversaries to each other in order to compete and learn from experience, and generate artificial data which is similar to the data given to the model while training.

The Generator model forms data patterns from a latent random variable, which is also simply called noise that is generated

in each epoch. The Discriminator model uses data fed into it in order to train, followed by using it to back-propagate to the Generator that uses the feedback from the Discriminator network to get better at producing artificial data of the kind input into the discriminator. The framework explained above is shown in Fig.1. This framework of neural networks was first implemented and presented by Ian J. Goodfellow et. Al in 2014 [5].

As HPCC Systems was a distributed computing system, there were challenges in implementing Neural Networks in the same. Neural Networks have been implemented in the GNN Bundle described above which uses Tensorflow in the back-end for the Neural Network computations that are needed to be performed. GANs in HPCC Systems were implemented using the GNN bundle by the team working on the same. The paper focuses on how it was implemented along with the results.

II. Generalised Neural Networks Bundle (GNN)

The Generalised Neural Network Bundle, is a bundle that provides a generalised ECL interface to Keras over Tensorflow in Python 3.6. It has various functions that help in the development and usage of Neural Networks very similar to how they would be used in Keras. The documentation for the same has provided on the HPCC Systems website [6].

GNN Bundle utilises the HPCC Systems multi-node processing capability by treating each node on the cluster as a thread of processing which fills the gap between ECL and Python. This makes sure that processing happens in parallel as it should happen in a cluster-computing system.

Although, the Neural Network model is trained epoch-wise but the processing in each epoch occurs in parallel over the nodes which increases the processing speed when there is a large amount of data that is given as input.

The bundle is stable for purely Sequential Keras models but is unstable for the Keras Functional models. Due to this, even GANs implemented have a limitation that it purely runs Sequential GAN models of various kinds.

Below are various important modules along with features that have been used widely in the following GAN implementation:

A. The GNNI Module

GNNI is the main Neural Network module of the GNN bundle. The functions in GNNI are very important when it comes to managing the Neural Network as it doesn't dwell on many details to be given by the user. Given below are features of the following module:

- **Create Session**: Create a Tensorflow session which share global variables for a set of Neural Network models.
- **Define models**: Defines a Sequential Keras model given the Session ID, Definition of each layer as a list of strings and compile definition as a string.
- **Fit data**: Fits the given data to the model.
- **Predict data**: Predicts the output using the model according to input data.
- **Weights transfer**: Retrieving weights from model, storing weights to a JSON file, Input weights from JSON file, Setting weights to a model can all be done with various functions. Helps for saving state of models or transfer between separate but similar models.
- **Evaluation of model**: Evaluation of model to get various metrics or even just getting the previous loss for a fit can be obtained

B. The Tensor Module

For the input of data, Python 3.6 has a numpy package capable of keeping track of the data as an n-dimensional array which is provided as a tensor for handling the input data. For the same framework in ECL, there was a special ECL record structure called t_Tensor implemented which stores more information than an n-dimensional array while also able to convert to input data for Tensorflow. Given below are the main usable features of Tensor module:

- **Building Tensors**: Tensors may be built from a dataset structured in certain way to build the t_Tensor dataset fully by using a defined function in this module.
- **Aligning Tensors**: Tensors can be in multiple different lists as per how they are defined. A few functions help in aligning tensors so there isn't a dimension mismatch
- **Get data from Tensor**: The dataset can be extracted from a t_Tensor using a certain function
- **Miscellaneous features**: Functions to reshape the tensor, add multiple tensors element-wise, get the number of records in a tensor are a few features which could be described as miscellaneous useful features.

These features although adequate for basic numerical record, there was a need of much more processing in order to convert a collection images to the required t_Tensor type. For this, a module was added on to the GNN bundle which is covered in the Section II-C.

C. The Image Module

Processing images to give as input to the Neural Network models developed in GNN is a difficult job due to ECL being a structured language. Defining structure for an image was required along with converting a compressed image format like JPG into a matrix of numbers. After the matrix was obtained, we would need to transform the dataset into a specific structure in order to make the Tensor. Due to these hurdles to process an image, there was a need to make a module which would perform these operations for ease of the user. Given below are the main features that were provided:

- **MNIST dataset**: Functions were implemented to input the images and labels of both train and test datasets of MNIST from the compressed unsigned bytes files obtained online [7].
- **Input Image**: The Image that was input in any format was taken as a stream of bytes, then converted into a n-dimensional array in Python using OpenCV 2 functions. This matrix was flattened and then returned along with it's dimensions as a dataset in ECL. To ensure that all images are of the same format, the images are either cropped

to certain dimensions in the center or resized to certain dimensions.

- **Image to Tensor**: The Image was in the form of a list of unsigned integers. This set of numbers was converted to a t_Tensor as is required for Neural Networks in GNN to take input.
- **Tensor to Image**: The t_Tensor which was given from the Neural Network as output was converted into a set of unsigned integers to convert back into an Image for output in various ways.
- **Image output**: The set of Numbers were converted into a numpy matrix and encoded back into an image in JPG or PNG or BMP format for seeing the output. Other than this, a function was implemented to get a collection of images in a grid to see many predictions of generator instead of just one.

III. Implementation of GAN for HPCC Systems

A. Input phase

The input of data given to the GAN needs to be pre-processed so as to suit the needs of GNN bundle. All the data that is required to be given for training needs to be transformed into a dataset defined as $TensData$ which has the record structure: [indexes, value]. $TensData$ simulates the representation of an n-dimensional array in numpy module with the indexes and value together to represent each position in an array.
All the data required to be used need to be transformed into the structure of $TensData$ using various transformation operations in ECL. For a dataset that is readily available as CSV or XLS, it is possible to easily transform these. For inputting images into the neural network, the Image module described in Section II-C may be used. It provides straightforward functions to perform the operation described above for input.

B. Definition of models

Generative Adversarial Networks by the definition are required to have 2 neural network models, namely the Generator and the Discriminator acting as adversaries to each other.
Here, the models are required to be given by the user. The user decides on the Generator and Discriminator model definitions and adds definition of each layer into a set of strings. Each model also has a compile definition which is typically a string containing the compile function. Both the models usually must have just one compile definition together. Examples for these are in Section V.
Given these definitions, Generator and Discriminator are defined using the GNNI function $DefineModel(..)$. Along with this, a combined model which is the Generator stacked over the Discriminator needs to be defined. Definition of this combined model is done by appending Generator layer definitions with Discriminator layer definitions, with each layer in discriminator set to non-trainable. This is defined using the same compile definition as the two models.

C. Training phase

Each epoch of GAN training happens in the following stages:

1) Generate data from random latent variable using Generator.
2) Fit real data with the label 1 to Discriminator.
3) Fit generated data with the label 0 to Discriminator.
4) Fit random latent variable with the label 1 to Combined model. Since Discriminator is non-trainable, Generator is trained with random latent variable while fitting as per Discriminator weights.

Here, the modified weights from Discriminator needs to be transferred to the combined model so that the Fitting of the data to Discriminator is effective in the Epoch. Along with this, the trained weights of the Combined model need to be given to the Generator to predict from noise. If this isn't done, the training of the previous epoch is rendered useless.
In Keras API of Python 3.6, it is possible to make the Combined model by adding the Generator and Discriminator as objects which link their layer weights. In ECL, we need to manually transfer weights to simulate this.
For this, a weights transfer mechanism was implemented using $GetWeights(..)$ and $SetWeights(..)$ functions in GNNI module of GNN bundle. Figure 2 explains the weights transfer methodology for an n^{th} epoch:

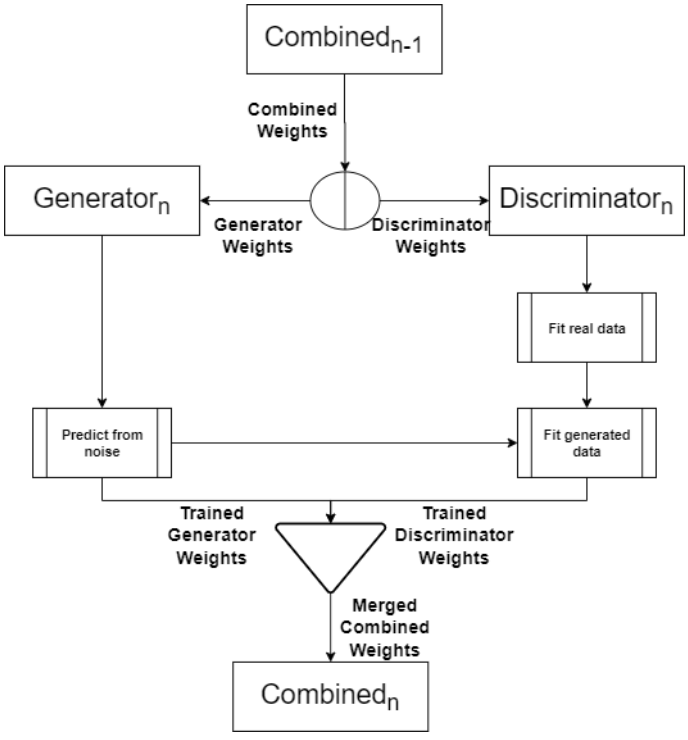


Figure 2: Weights transfer methodology

D. Output phase

After the epoch-wise training of the GAN model is performed, both the Generator and Discriminator models are returned from the Module. These models can both be given their respective inputs for checking the output or for an application.

*1) **Generator model**:* Generator model takes an input of a random latent variable, and if required, multiple sets of the random variable to predict data. These predictions are obtained as a t_Tensor, whose data can be extracted using the $GetData(..)$ function in Tensor module. The extracted data can be stored or output to understand the Generator's performance. For an output of images, the Image module in GNN bundle contains functions to easily output the images as is required by the user.

*2) **Discriminator model**:* Discriminator model takes an input of data to classify them as fake or real data. This capability of the Discriminator helps in applications where fraudulent data are to be detected when the number of fraudulent cases are smaller. Machine learning evaluation metrics like AUC, F1 score etc. can be performed on the output dataset with expected to see how the Discriminator has performed.

IV. Choice of dataset for testing

When working on any model which is used to detect features on an image, MNIST dataset is commonly used. As many may refer to it, the MNIST dataset is the "Hello World" of Deep learning as it is a rather straight forward dataset with minimal noise and clear distinction. It is also said commonly,

If it does not work on MNIST, it will not work at all.

Even though that may not be completely true, it is still a great dataset to check the working of most Neural Network implementations. MNIST dataset was initially used by Yann LeCun in his paper about deep learning titled "Gradient based learning applied to document recognition" [8]. This dataset is a very simplistic dataset which has handwritten numbers in white with a black background of an uniform size of 28x28 pixels. This enables it to be easily computed and enables neural networks to greatly understand features of the images.
Classification performed on these numbers with a convolutional neural network enabled the models to get accuracy of upto 99.8% [9]. Due to it being such a good dataset for Neural Networks to use, it works great for us to test GANs on MNIST. Other than these reasons, Ian J. Goodfellow et. Al on their paper titled "Generative adversarial networks", tested their GAN model on MNIST as well [5].

V. Tested GAN models

There were two prominent models in GAN which were Sequential in nature for pure generation of data. The output of the generated dataset were checked after subsequent epochs to verify the training that was performed, which are shown in Section VII. These models were obtained from a github repository called "Keras-GAN" by Erik Linder Noren [10].
The Generator and Discriminator models of these are described as follows:

A. Simple Dense layer based Generative Adversarial Networks

This model is described as a GAN architecture with purely dense based networks with no special transformations except for Batch Normalisation, Dropout and Activation layers. This model tries to see pattern in the image as a whole set by predicting each pixel as per the Dense layer weights obtained after each training epoch.
The Generator and Discriminator models of the Simple GAN can be seen in Figure 3.

As can be seen in Figures 3a and 3b, the network consists of purely Dense based models for data manipulation which enables the GAN to be trained pixel-wise. Due to this, the training is much slower than it's other counterparts and the generated images aren't smooth, due to the pixels generated being different. The output of the following model can be seen in Section VII-A. The results in the next model, Deep Convolutional Generative Adversarial Networks (Section V-B), abbreviated as DCGAN, can be seen to provide smoother images compared to Simple GAN.

B. Deep Convolutional Generative Adversarial Networks

This model is described as a GAN architecture with convolutional transformations in the Neural Network. This model consists of multiple convolutional layers which helps the models in recognizing various features in the image.
The Generator and Discriminator models of the Deep Convolutional GAN can be seen in Figure 4.

As can be seen in Figures 4a and 4b, the network consists of convolutional layers along with Dropout layers which increases the feature recognition capability of the network. Due to the improved feature recognition capability, the generated images are much smoother and more unique in nature. The outputs of the following model can be seen in Section VII-B. It can be observed that DCGAN gives smoother and more unique images as compared to Simple GAN.

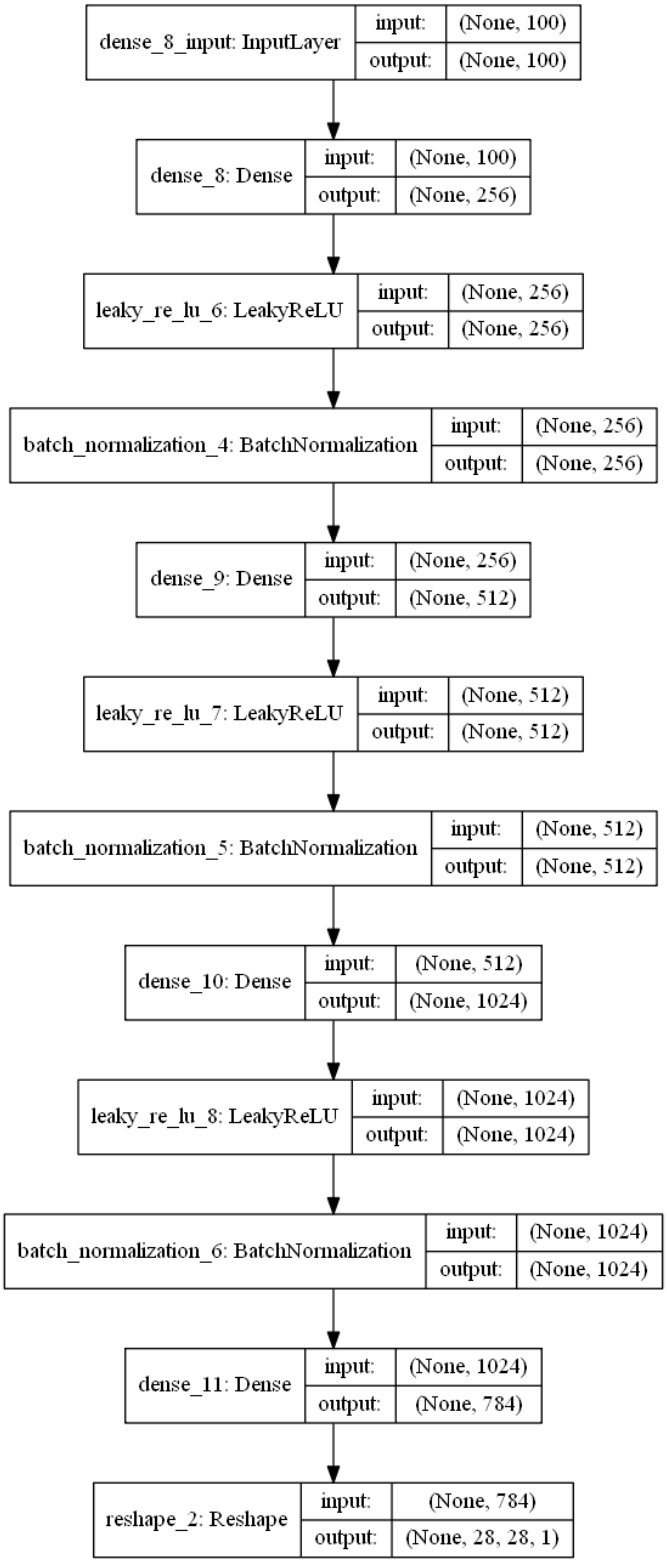
VI. Few notable features

In the GAN bundle that was implemented for ECL [11], there are a few features that were implemented which are worth having a discussion about:

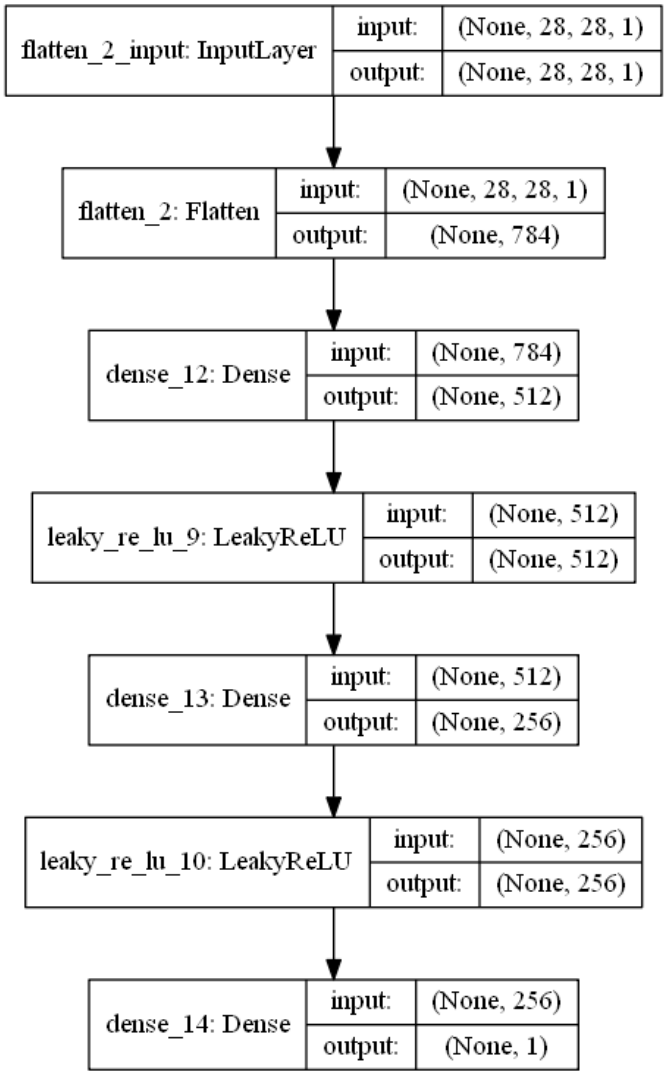
A. Saving of trained models

A record structure made to store details of the Generator and Discriminator after being trained in order to fit more data over the trained network or predict to get more snapshots.
The record structure has the following details:
Layer specifications : [Layer definitions, Compile Definition],
Model Weights,
Command to despray images,
Rows in output grid image,
Columns in output grid image,
Batch Size to produce noise,
Latent dimensions for Random latent variable,
Number of epochs for storing in string

Doing this helps to not lose the model trained and reuse it for various purposes so that the time spent in training isn't deemed worthless. It also helps to experiment over it as and when required after the training is done.
In $Test/GANtest.ecl$ and $Test/DCGANTest.ecl$, the models are automatically saved at the end so that the user can

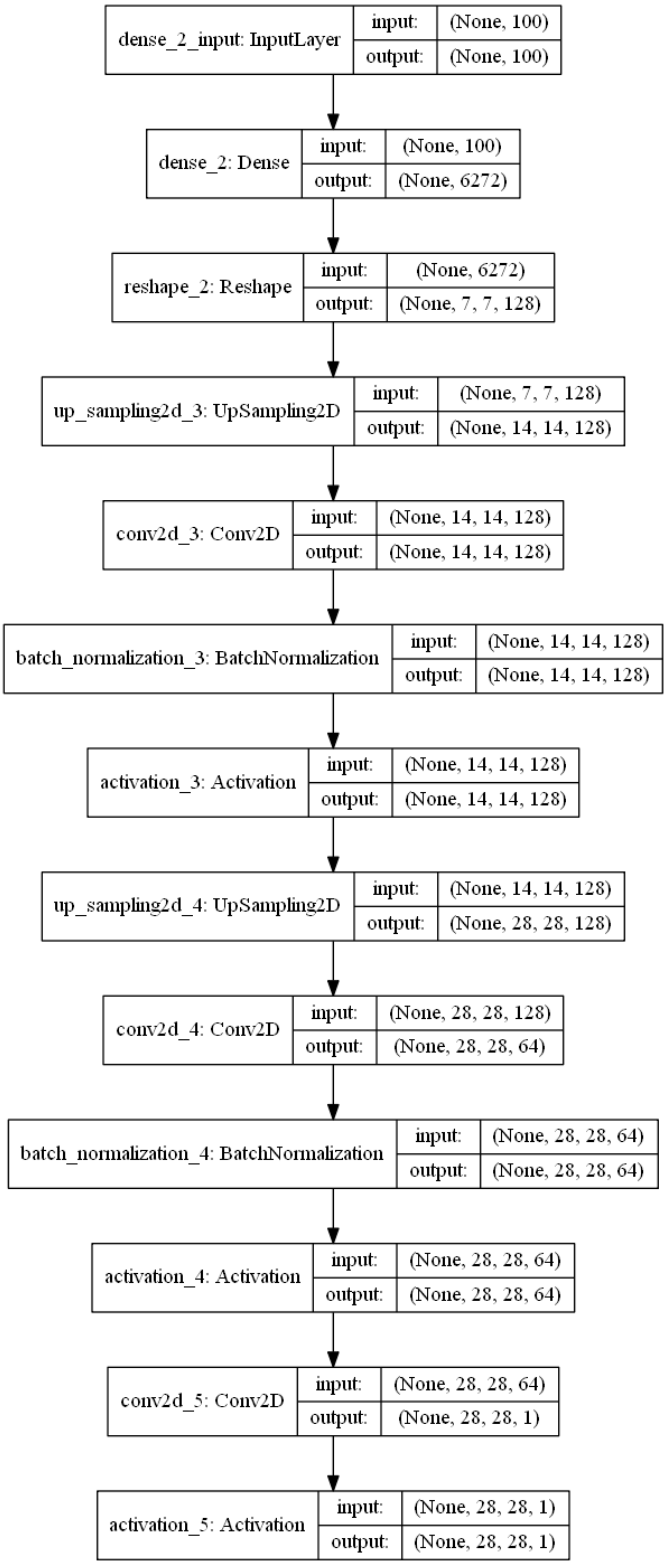


(a) Generator model

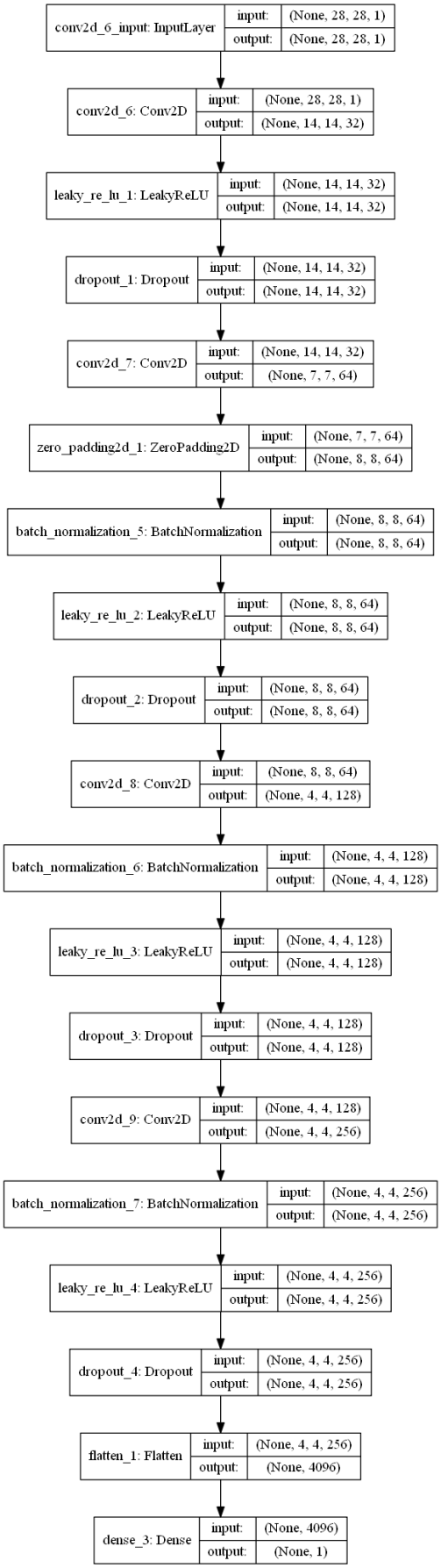


(b) Discriminator model

Figure 3: Simple Dense layer based Generative Adversarial Networks



(a) Generator model



(b) Discriminator model

Figure 4: Deep Convolutional Generative Adversarial Networks

experiment with the saved model as per their convenience. $Test/SaveTest.ecl$ contains the code snippet of how to save a model, this can be taken as inspiration for using anywhere else.

B. Prediction of models

The saved model after training as described in Subsection VI-A helps for the user to Predict for the model saved as required.
This helps in getting new snapshots of the data by the Generator or to test the Discriminator's distinguishing capabilities on various datasets as required due to a saved model.
An example of how this works is provided in the $Test/PredictTest.ecl$ file in the GAN bundle. This helps to understand how to use the feature.

VII. Results

Using the above implementation, two architectures of GANs that were described in Section V have been tested on HPCC Systems using the GAN implementation and executed. Snapshots of the Generator predictions were taken after 2000, 3000 and 4000 epochs respectively, which shows us how the model gets better as it's trained. The outputs of the architectures are as follows:

A. Simple Generative Adversarial Networks

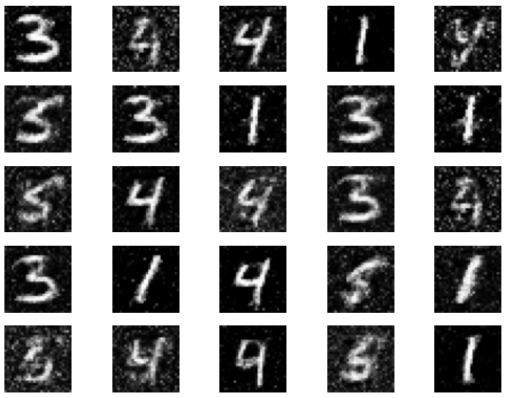
It can be noticed that Simple GAN produces results very similar to the MNIST dataset, but there are noisy pixels around them due it being a purely dense network relying on the density of each tensor. This is rectified by Deep Convolutional GAN which produces smoother images, with very unique images as compared to Simple GAN.

B. Deep Convolutional Generative Adversarial Networks

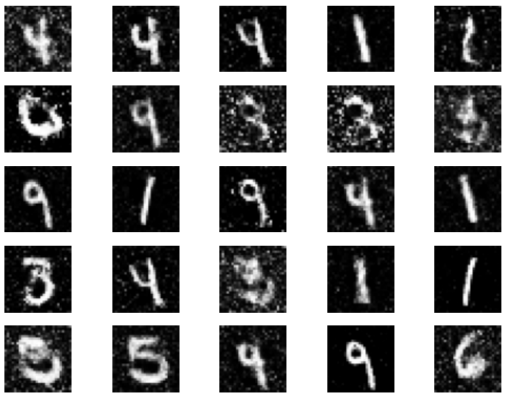
The images produced by Deep Convolutional GAN are much more smoother and unique in nature compared to simple GAN. The convolutional layers implemented in the model detect various features of the MNIST dataset in order to put them together to generate similar digits. The digits generated by both the GAN models get better with more number of training epochs for the GAN.

VIII. Conclusion

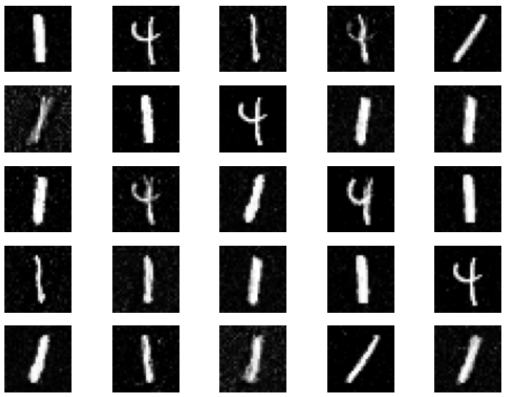
Generative Adversarial Networks were successfully implemented on the HPCC Systems Platform to be able to execute any Keras Sequential GAN model. The implementation has been stable for the models tested on the platform.
This implementation was made into an ECL bundle to be installed on any system containing the required dependencies. The GAN bundle so created depends on ML_Core and GNN bundles. So, to use the following implementation, these dependencies need to be installed on the platform.



(a) Epoch 2000

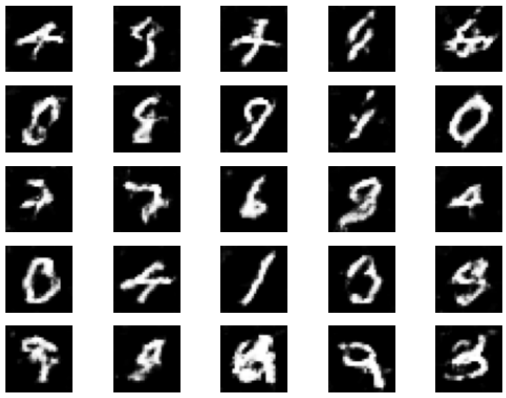


(b) Epoch 3000

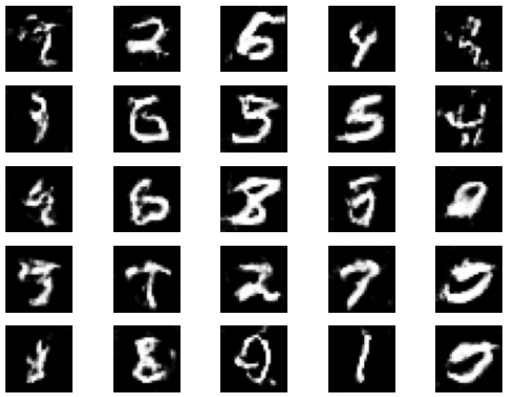


(c) Epoch 4000

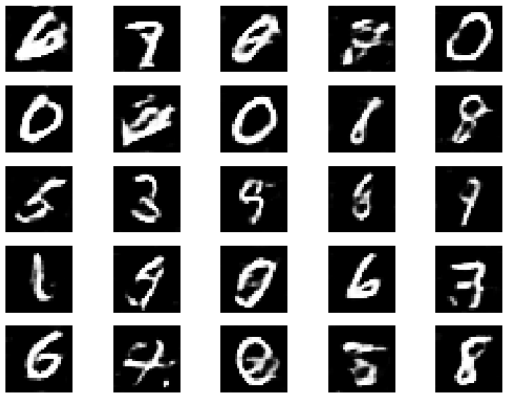
Figure 5: Generated images from Simple GAN



(a) Epoch 2000



(b) Epoch 3000



(c) Epoch 4000

Figure 6: Generated images from Deep Convolutional GAN

IX. Future enhancements

The implementation for GAN in HPCC Systems has much room for improvement with many more features possible with the proposed architecture above.

As mentioned above, this implementation of GAN works for purely Sequential models which eliminates the possibility of multiple inputs or outputs as is required for a few types of GAN models. So, an architecture for GAN could be implemented using a GNN Neural Network model similar to the Keras Functional Model API.

There is quite a confusion for the user as to how the combined model looks like in case the user runs into an error while using the GAN module. Visualising neural network models so as to understand the model better would be very useful.

The debugging of the GAN module could be strange as the error which is logged isn't usually the cause of the error. Improving these debugging prospects of GAN to efficiently track the issue so the user can fix errors would reduce the time taken by the user in solving an error.

References

[1] HPCC Systems, "About HPCC Systems", 2020, [Online] Available: https://hpccsystems.com/about. [Accessed July 1, 2020]

[2] HPCC System, "ECL Language Reference", 2020, [Online] Available: https://d2wulyp08c6njk.cloudfront.net/releases/CE-Candidate-7.8.24/docs/EN_US/ECLLanguageReference_EN_US-7.8.24-1.pdf . [Accessed July 1, 2020]

[3] HPCC Systems, "ML Core Bundle source code", 2020, [Online] Available: https://www.github.com/hpcc-systems/ML_Core. [Accessed July 1, 2020]

[4] HPCC Systems, "Generalised Neural Networks Bundle source code", 2020, [Online] Available: https://www.github.com/hpcc-systems/GNN. [Accessed July 1, 2020]

[5] Goodfellow, I.J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A.C., & Bengio, Y. (2014). Generative Adversarial Networks. ArXiv, abs/1406.2661.

[6] HPCC Systems, "Generalised Neural Networks Bundle documentation", 2020, [Online] Available: https://d2wulyp08c6njk.cloudfront.net/pdf/ml/GNN.pdf. [Accessed July 11, 2020]

[7] Yann LeCun, "The MNIST database of Handwritten digits", 2020, [Online] Available: http://yann.lecun.com/exdb/mnist/. [Accessed July 12, 2020]

[8] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner: Gradient-Based Learning Applied to Document Recognition, Proceedings of the IEEE, 86(11):2278-2324, November 1998.

[9] Baldominos, A., Saez, Y., & Isasi, P. (2019). A Survey of Handwritten Character Recognition with MNIST and EMNIST. Applied Sciences, 9(15), 3169. doi:10.3390/app9153169.

[10] Erik Linder Noren, Github, "Keras-GAN", 2020. [Online] Available: https://github.com/eriklindernoren/Keras-GAN. [Accessed July 12, 2020]

[11] Ambu Karthik, Github, "GAN", 2020, [Online] Available: https://github.com/dragonfist453/GAN. [Accessed July 12, 2020]