INTERNATIONAL JOURNAL OF ADVANCE RESEARCH, IDEAS AND INNOVATIONS IN TECHNOLOGY

ISSN: 2454-132X
Impact factor: 4.295
(Volume 4, Issue 4)
Available online at: www.ijariit.com

Improved random-forest image classification using generative adversarial network

Ashutosh Mahesh Pednekar
ashutoshpednekar15@gmail.com
Vellore Institute of Technology, Vellore, Tamil Nadu

ABSTRACT

Mankind has always hoped to do better. Ensemble learning [12] is one of the techniques that help drastically improve results of machine learning algorithms. But these classifiers need a lot of data in order to perform really well. But even in today's data-driven world, it is not that easy to get hold of that magnitude of data. This is where the concept of synthetic data [2] comes into play. Synthetic data is not real but boasts many characteristics of real data, and the latter can sometimes become indistinguishable from the former. This data can be put to good use, such as training our ensemble classifiers so that we can get even better results. That is what this paper aims to illustrate. I'm using random-forest ensemble classifier for classifying MNIST handwritten digits dataset, and the Generative Adversarial Network (GAN) for generating the said synthetic data.

Keywords: *An ensemble, Generative adversarial networks, Mnist, Random-forest classifier*

1. INTRODUCTION

This paper tries to illustrate the usage of synthetically generated data as training data for an ensemble classifier. The practice of using several classifiers together, to improve the accuracy of the results is quite popular. But the results would be even better if it was trained on more data. Of course, it is possible to get real data from the environment, or through surveys. But this data is still limited. Wouldn't it be wonderful, if we could just synthesize training data on the fly? This would literally make the training data unlimited. But we can't just put in random values that will ruin our classifier, as it would then be trained over inaccurate data thus giving inaccurate results. That is clearly not acceptable. A better approach would be if we could somehow figure out a way to create data that is similar, or better yet, indistinguishable from out real data. This was only theoretical in earlier days, but it is definitely possible with today's technology.

Nowadays, with the advent of neural networks, and the fact that we now have the computational resources to handle these heavy calculations, we can now think of more innovative and intelligent ways to generate synthetic data. The model I'm using in the context of this paper is the "Generative Adversarial Network", short for GAN. The primary application of GAN is to generate synthetic facial images. Ledig, Christian, et al 3] have used this network to generate Photo-Realistic Single Image Super-Resolution. The working principle of GAN and random forest are illustrated in the following sections. The main idea behind this paper is to propose and illustrate the use of generative models to synthesize more training data to feed our random forest classifier so that it trains better and gives more accurate results.

2. ARCHITECTURE OVERVIEW

This section explains the architecture and the functioning of the proposed system. The following sections each illustrate the two components of the said system. Following those, is the high-level abstract architecture of the entire system combined together. So let's take a quick look at the working principle of the random forest classifier.

2.1 Random-Forest Classifier

The working of the random forest classifier is explained in detail, in a further section. The following figure shows the basic working of the said classifier.

The detailed explanation of how the Random-Forest classifier works is given in a further section of this paper. Now let's look at the cream, the generative adversarial networks, and understand what goes on inside it, and how good is it.

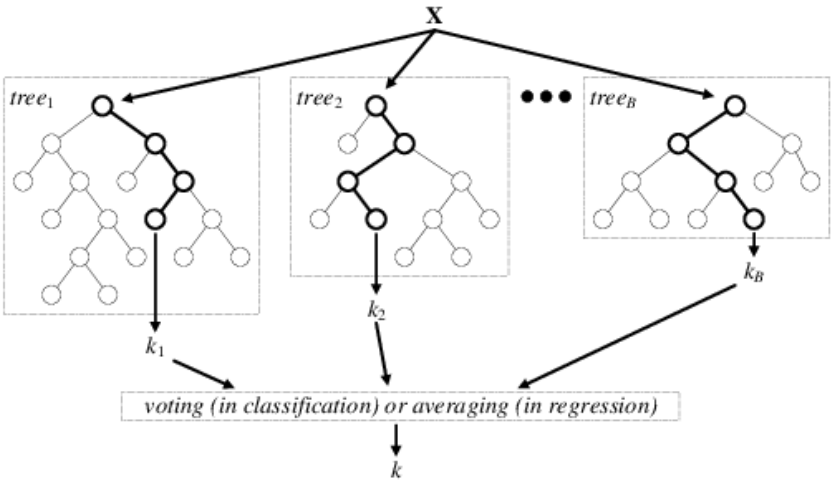


Fig. 1: Random forest classifier

2.2 GAN component

Generative Adversarial Networks are a new type of Adversarial Networks, proposed by "Ian Goodfellow" in 2014. It is usually implemented using deep neural networks. GAN's are very powerful, and efficient in generating "Realistic" outputs which can seldom be distinguished from a "Real" data sample.

The model is illustrated in the following figure:

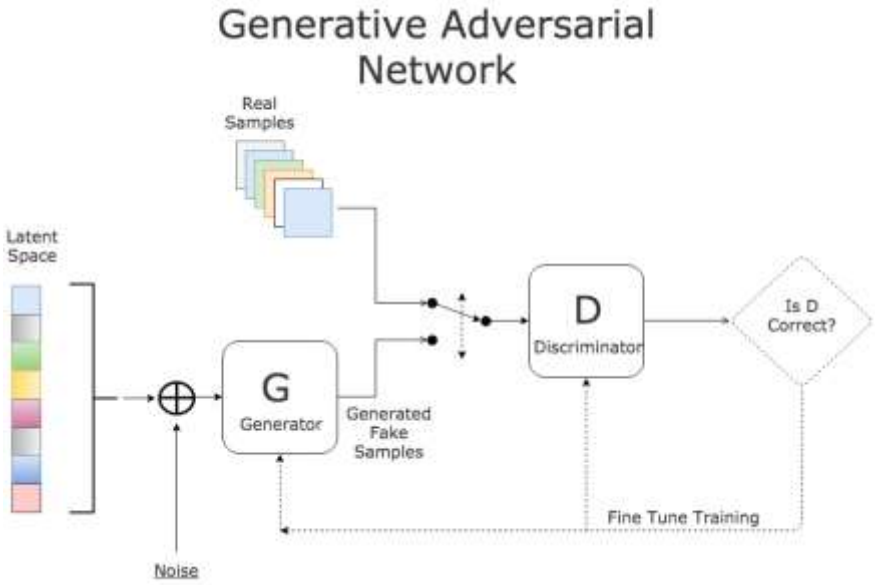


Fig. 2: Generative adversarial network

3. RANDOM-FOREST ALGORITHM OVERVIEW

One of the first machine learning algorithm in most our learning journey is the Decision Tree Classifier. It is one of the most intuitive, straight-forward classification algorithm, introduced by Quinlan, J. Ross [5]. Unlike probabilistic classifiers like Naive-Bayes, the Decision tree classifier uses the concept of entropy. The entropy of a feature or a column is given below.

$$E(S) = \sum_{i=1}^{c} - p_i \log_2 p_i$$

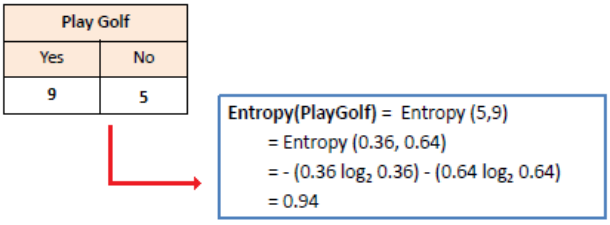
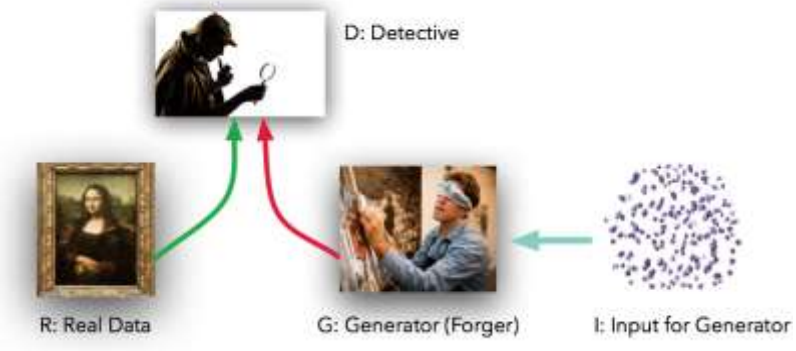


Fig. 3: Entropy calculation

First, we calculate the entropy of the class and store it somewhere handy. Then we go ahead and calculate the entropies of all other features. Now, we calculate the 'Gain', i.e, the absolute difference between the class entropy, and the entropy of each feature. The feature with the maximum entropy is considered our root node. Then the dataset is partitioned according to the different values of the root node. And that is followed by recursive execution of a similar procedure for these partitions. What we finally get is a huge Decision tree. It is very useful to help make quick decisions, and to classify data conveniently. But sometimes this may lead to "overfitting", i.e, giving accurate results only for the training data, but bizarre results for the test data. This can be avoided by pruning the tree or setting a maximum depth beforehand. But this was still not good enough. Thus came the introduction of Random Forest[6] classifier, which takes an average of the results, in case of regression, and voting in case of classification, the results of many decision trees in an ensemble, thus giving much better results. A forest of trees is impenetrable as far as simple interpretations of its mechanism go. In some applications, analysis of medical experiments, for example, it is critical to understand the interaction of variables that is providing the predictive accuracy. A start on this problem is made by using internal out-of-bag estimates, and verification by reruns using only selected variables. This is the classification algorithm being used in the context of this paper.

4. GENERATIVE ADVERSARIAL NETWORK OVERVIEW



Goodfellow, Ian, et al. proposed a novel approach for generating synthetic data, that is very similar to the real data. It consists of a Generative model G, which captures the distribution of the data, and a Discriminative model D that computes the probability estimate saying that a sample belongs to the training data rather than the generator G.

This can be thought of as follows. This explanation may sound a little informal, please bear with me. Consider a student in an acting school and a newly appointed teacher. The student has to play the role of, say a politician. So the student starts his acting. It is the job of the teacher to point out the flaws in his acting, and say whether the character is a real politician or just an actor. The teacher and the student, both initially have very little intuition of how a politician acts. The teacher refers to a Biography of a popular politician (i.i, Training Data) for reference. Say, after a few trials, the teacher figured out some salient feature that a politician is supposed to possess, say confidence. The teacher will let the student know that this is what is expected in a real politician, and the student will try his best to act in such a way that the teacher believes that he is a real politician. Now, the teacher finds out that a politician must dress in a particular way. So the student will try to imitate that. They repeat these rounds for hundreds, or maybe thousands of time until the student now seems indistinguishable from a real politician. Note that he is *still just an actor* though, *not* a real politician.

Let us now rephrase this in more formal, computational terms. Our actor/student is played by a generator model G, while the role of the teacher is played by the Discriminative model D. The better the model D is, at differentiating between real and fake data, the better will the Generative model G will be at generating more realistic data, as it will be able to take better advantage of the Discriminator's new knowledge. The number of times the training takes place may be of the order of hundreds of thousands, or even millions. The generator uses gradient descent, and the discriminator learns using the gradient ascent to converge.

They are created from two primary modules. a generator, and a discriminator. The Generative part is trying to create some sample and deceive the discriminator to declare it as a "Real" data sample. Mathematically they can be represented like this:

$$\min \max V(D, G) = E_{x \sim p\ data\ (x)} [\log D(x)] + E_{z \sim p\ z\ (z)} [\log(1 - D(G(z)))]$$

In the proposed adversarial nets framework, the generative model is pitted against an adversary: a discriminative model that learns to determine whether a sample is from the model distribution or the data distribution. The generative model can be thought of as analogous to a team of counterfeiters, trying to produce fake currency and use it without detection, while the discriminative model is analogous to the police, trying to detect the counterfeit currency. Competition in this game drives both teams to improve their methods until the counterfeits are indistinguishable from the genuine articles.

5. IMPLEMENTATION

The task is to create a Generative adversarial network that will generate MNIST digits that seem to be indistinguishable from the original MNIST dataset. These new data are augmented to the original training dataset. Now, we train the Random-Forest classifier and classify the MNIST dataset.

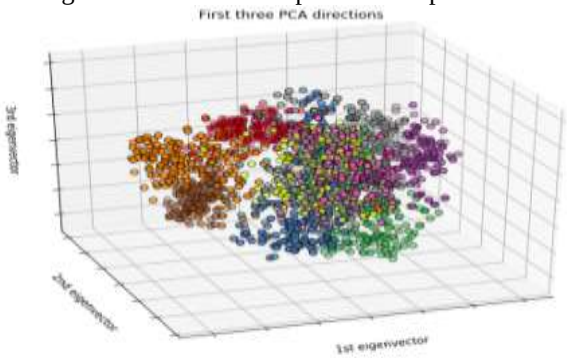
- Real MNIST training images
- A generator network that takes in a random noise vector and produces a synthetic image
- A discriminator network (a Convolutional Neural Network) that learns to discriminate between real and synthetic images. It can be thought of as a binary classifier (1 for real image, 0 for fake)
- An optimization procedure that jointly updates both the neural nets by means of SGD. This is the crucial part because we need to train the generator network to deceive the discriminator network, which in turn means that we have unique gradient flows and labels.
- Tensorflow - Our choice of Deep Learning framework
- Python, along with its basic libraries like numpy, pandas, matplotlib, etc.

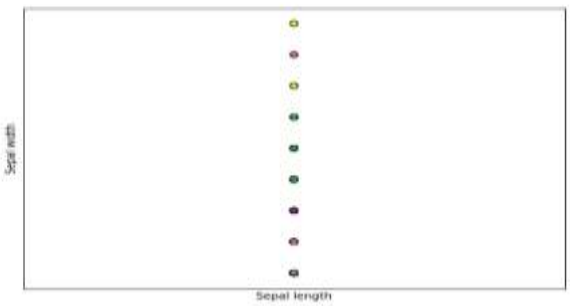
Loading the MNIST dataset from sci-kit-learn:

- For visualization
  Here, we load only the first two features from the datasets, as it is not feasible to visualize data in higher dimensions. So here is the code cell illustrating the same.

```
In [1]: %matplotlib inline

In [2]: print(__doc__)


        # Code source: Gaël Varoquaux
        # Modified for documentation by Jaques Grobler
        # License: BSD 3 clause

        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D
        from sklearn import datasets
        from sklearn.decomposition import PCA

        # import some data to play with
        digits = datasets.load_digits()
        X = digits.data[:, :2]  # we only take the first two features.
        y = digits.target

        x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
        y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

        plt.figure(2, figsize=(8, 6))
        plt.clf()

        # Plot the training points
        plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Set1,
                    edgecolor='k')
        plt.xlabel('Sepal length')
        plt.ylabel('Sepal width')

        plt.xlim(x_min, x_max)
        plt.ylim(y_min, y_max)
        plt.xticks(())
        plt.yticks(())

        # To getter a better understanding of interaction of the dimensions
        # plot the first three PCA dimensions
        fig = plt.figure(1, figsize=(8, 6))
        ax = Axes3D(fig, elev=-150, azim=110)
        X_reduced = PCA(n_components=3).fit_transform(digits.data)
        ax.scatter(X_reduced[:, 0], X_reduced[:, 1], X_reduced[:, 2], c=y,
                   cmap=plt.cm.Set1, edgecolor='k', s=40)
        ax.set_title("First three PCA directions")
        ax.set_xlabel("1st eigenvector")
        ax.w_xaxis.set_ticklabels([])
        ax.set_ylabel("2nd eigenvector")
        ax.w_yaxis.set_ticklabels([])
        ax.set_zlabel("3rd eigenvector")
        ax.w_zaxis.set_ticklabels([])

        plt.show()
```

In the above-illustrated code, I've also carried out 'Principal Component Analysis' (PCA), more specifically Eigenvector decomposition so that we can have a better insight into how the data-points are separated.

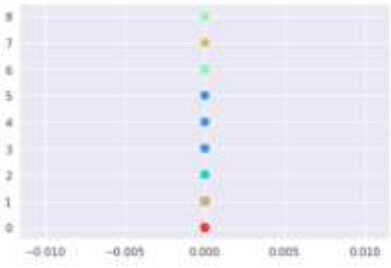




- For actual implementation

Unlike before, where we were using only two features from the dataset for visualization purposes, here we are loading the entire dataset.

```
In [13]: from sklearn import datasets

digits = datasets.load_digits()
X = digits.data
y = digits.target
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='rainbow');
```



Function for creating an individual decision tree:

```
In [103]: from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier()
```

But running this function for multiple iterations over different test data, we get completely different results. That's a clear sign of over-fitting. This needs to be avoided. Pre-pruning and Post-pruning are effective ways, but we can obtain way better results if we go for ensemble methods - Enter Random forest!

Function for creating this ensemble of classifiers:

```
In [116]: def fit_randomized_tree(X, y, random_state=0):

    clf = DecisionTreeClassifier(max_depth=15)

    rng = np.random.RandomState(random_state)
    i = np.arange(len(y))
    rng.shuffle(i)
```

Here, what we are doing is basically creating decision trees for random subsets of data from out train sample. Once we fit these decision trees on our data, all that is to be done is to essentially average out their results. This ensures a much better fit, overall.

Fit the Random Forest Classifier

Fit the model on the train set

```
In [199]: clf = RandomForestClassifier(max_depth=5, n_estimators=10, max_features=1)
clf.fit(X_train, y_train)

Out[199]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=5, max_features=1, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

Run inference on the test data

```
In [200]: score = clf.score(X_test, y_test)
score

Out[200]: 0.95
```

In order to ensure proper results, and to avoid averaging error, I've used the predefined function from sci-kit-learn instead of using my own function to fit the data. So, as we can clearly see, the score obtained is **0.95**

Now is the time for the cream on the cookie! According to Yann LeCun, "adversarial training is the coolest thing since sliced bread". I'm inclined to believe so because I don't think sliced bread ever created this much buzz and excitement within the deep learning community. Hence, I tried deploying Generative adversarial networks on the MNIST data. I'm using the TensorFlow library from Google. Also please note; for saving time and for faster results, I've used Google Colab instead of my local installation of sensor flow and jupyter notebook…

Importing the necessary libraries: -

```
[1] import tensorflow as tf
    import random
    import numpy as np
    import matplotlib.pyplot as plt

    %matplotlib inline
```

Loading Data:

```
[2] from tensorflow.examples.tutorials.mnist import input_data
    mnist = input_data.read_data_sets("MNIST_data/")
```

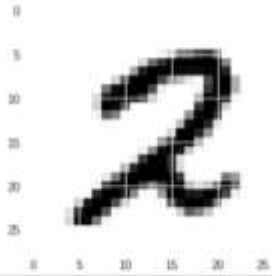

Plotting sample data, i.e. MNIST image:

```
[3] x_train = mnist.train.images[:55000,:]
    x_train.shape
```

```
(55000, 784)
```

Let's look at what a random image might look like.

```
[4] randomNum = random.randint(0,55000)
    image = x_train[randomNum].reshape([28,28])
    plt.imshow(image, cmap=plt.get_cmap('gray_r'))
    plt.show()
```



Functions for creating CNN's in tensorflow:

```
[5] def conv2d(x, W):
      return tf.nn.conv2d(input=x, filter=W, strides=[1, 1, 1, 1], padding='SAME')

    def avg_pool_2x2(x):
      return tf.nn.avg_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
```

Function for defining the Discriminator network:

```
[6] def discriminator(x_image, reuse=False):
        with tf.variable_scope('discriminator') as scope:
            if (reuse):
                tf.get_variable_scope().reuse_variables()
            #First Conv and Pool Layers
            W_conv1 = tf.get_variable('d_wconv1', [5, 5, 1, 8], initializer=tf.truncated_normal_initializer(stddev=0.02))
            b_conv1 = tf.get_variable('d_bconv1', [8], initializer=tf.constant_initializer(0))
            h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
            h_pool1 = avg_pool_2x2(h_conv1)

            #Second Conv and Pool Layers
            W_conv2 = tf.get_variable('d_wconv2', [5, 5, 8, 16], initializer=tf.truncated_normal_initializer(stddev=0.02))
            b_conv2 = tf.get_variable('d_bconv2', [16], initializer=tf.constant_initializer(0))
            h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
            h_pool2 = avg_pool_2x2(h_conv2)

            #First Fully Connected Layer
            W_fc1 = tf.get_variable('d_wfc1', [7 * 7 * 16, 32], initializer=tf.truncated_normal_initializer(stddev=0.02))
            b_fc1 = tf.get_variable('d_bfc1', [32], initializer=tf.constant_initializer(0))
            h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*16])
            h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

            #Second Fully Connected Layer
            W_fc2 = tf.get_variable('d_wfc2', [32, 1], initializer=tf.truncated_normal_initializer(stddev=0.02))
            b_fc2 = tf.get_variable('d_bfc2', [1], initializer=tf.constant_initializer(0))

            #Final Layer
            y_conv=(tf.matmul(h_fc1, W_fc2) + b_fc2)
        return y_conv
```

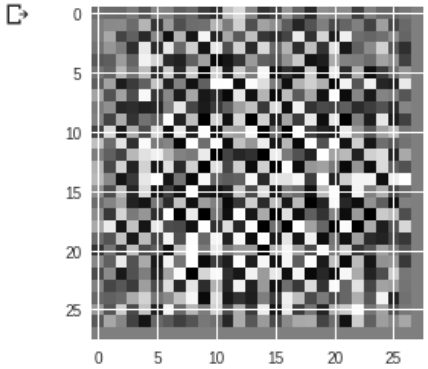
Function for creating the Generator Network:

```
[7] def generator(z, batch_size, z_dim, reuse=False):
        with tf.variable_scope('generator') as scope:
            if (reuse):
                tf.get_variable_scope().reuse_variables()
            g_dim = 64 #Number of filters of first layer of generator
            c_dim = 1 #Color dimension of output (MNIST is grayscale, so c_dim = 1 for us)
            s = 28 #Output size of the image
            s2, s4, s8, s16 = int(s/2), int(s/4), int(s/8), int(s/16) #We want to slowly upscale the image, so these values will help
                                                                      #make that change gradual.

            h0 = tf.reshape(z, [batch_size, s16+1, s16+1, 25])
            h0 = tf.nn.relu(h0)
            #Dimensions of h0 = batch_size x 2 x 2 x 25

            #First DeConv Layer
            output1_shape = [batch_size, s8, s8, g_dim*4]
            W_conv1 = tf.get_variable('g_wconv1', [5, 5, output1_shape[-1], int(h0.get_shape()[-1])],
                                      initializer=tf.truncated_normal_initializer(stddev=0.1))
            b_conv1 = tf.get_variable('g_bconv1', [output1_shape[-1]], initializer=tf.constant_initializer(.1))
            H_conv1 = tf.nn.conv2d_transpose(h0, W_conv1, output_shape=output1_shape, strides=[1, 2, 2, 1], padding='SAME')
            H_conv1 = tf.contrib.layers.batch_norm(inputs = H_conv1, center=True, scale=True, is_training=True, scope="g_bn1")
            H_conv1 = tf.nn.relu(H_conv1)
            #Dimensions of H_conv1 = batch_size x 3 x 3 x 256

            #Second DeConv Layer
            output2_shape = [batch_size, s4 - 1, s4 - 1, g_dim*2]
            W_conv2 = tf.get_variable('g_wconv2', [5, 5, output2_shape[-1], int(H_conv1.get_shape()[-1])],
                                      initializer=tf.truncated_normal_initializer(stddev=0.1))
            b_conv2 = tf.get_variable('g_bconv2', [output2_shape[-1]], initializer=tf.constant_initializer(.1))
            H_conv2 = tf.nn.conv2d_transpose(H_conv1, W_conv2, output_shape=output2_shape, strides=[1, 2, 2, 1], padding='SAME')
            H_conv2 = tf.contrib.layers.batch_norm(inputs = H_conv2, center=True, scale=True, is_training=True, scope="g_bn2")
            H_conv2 = tf.nn.relu(H_conv2)
            #Dimensions of H_conv2 = batch_size x 6 x 6 x 128

            #Third DeConv Layer
            output3_shape = [batch_size, s2 - 2, s2 - 2, g_dim*1]
            W_conv3 = tf.get_variable('g_wconv3', [5, 5, output3_shape[-1], int(H_conv2.get_shape()[-1])],
                                      initializer=tf.truncated_normal_initializer(stddev=0.1))
            b_conv3 = tf.get_variable('g_bconv3', [output3_shape[-1]], initializer=tf.constant_initializer(.1))
            H_conv3 = tf.nn.conv2d_transpose(H_conv2, W_conv3, output_shape=output3_shape, strides=[1, 2, 2, 1], padding='SAME')
            H_conv3 = tf.contrib.layers.batch_norm(inputs = H_conv3, center=True, scale=True, is_training=True, scope="g_bn3")
            H_conv3 = tf.nn.relu(H_conv3)
            #Dimensions of H_conv3 = batch_size x 12 x 12 x 64

            #Fourth DeConv Layer
            output4_shape = [batch_size, s, s, c_dim]
            W_conv4 = tf.get_variable('g_wconv4', [5, 5, output4_shape[-1], int(H_conv3.get_shape()[-1])],
                                      initializer=tf.truncated_normal_initializer(stddev=0.1))
            b_conv4 = tf.get_variable('g_bconv4', [output4_shape[-1]], initializer=tf.constant_initializer(.1))
            H_conv4 = tf.nn.conv2d_transpose(H_conv3, W_conv4, output_shape=output4_shape, strides=[1, 2, 2, 1], padding='VALID')
            H_conv4 = tf.nn.tanh(H_conv4)
            #Dimensions of H_conv4 = batch_size x 28 x 28 x 1

        return H_conv4
```


Generating a sample image:

```
[9]  sample_image = generator(z_test_placeholder, 1, z_dimensions)
     test_z = np.random.normal(-1, 1, [1,z_dimensions])
```

```
[11] my_i = temp.squeeze()
     plt.imshow(my_i, cmap='gray_r')
     plt.show()
```



Preparation for training:

```
[12] batch_size = 16
     tf.reset_default_graph() #Since we changed our batch size (from 1 to 16), we need to reset our Ten

     sess = tf.Session()
     x_placeholder = tf.placeholder("float", shape = [None,28,28,1]) #Placeholder for input images to t
     z_placeholder = tf.placeholder(tf.float32, [None, z_dimensions]) #Placeholder for input noise vect
```

```
[13] Dx = discriminator(x_placeholder) #Dx will hold discriminator outputs (unnormalized) for the real
     Gz = generator(z_placeholder, batch_size, z_dimensions) #Gz holds the generated images
     Dg = discriminator(Gz, reuse=True) #Dg will hold discriminator outputs (unnormalized) for generate

g_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=Dg, labels=tf.ones_like(Dg)
```

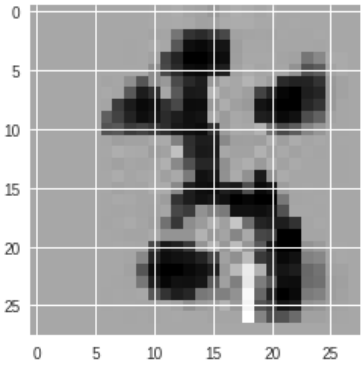
Training the GAN:

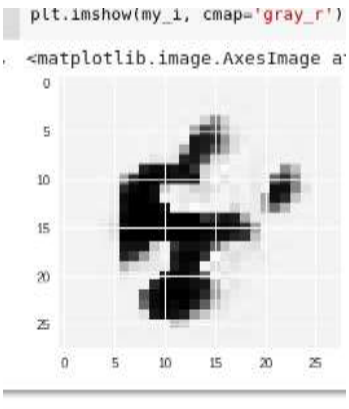
```
iterations = 3000
for i in range(iterations):
    z_batch = np.random.normal(-1, 1, size=[batch_size, z_dimensions])
    real_image_batch = mnist.train.next_batch(batch_size)
    real_image_batch = np.reshape(real_image_batch[0],[batch_size,28,28,1])
    _,dLoss = sess.run([trainerD, d_loss],feed_dict={z_placeholder:z_batch,x_placeholder:real_imag
    _,gLoss = sess.run([trainerG,g_loss],feed_dict={z_placeholder:z_batch}) #Update the generator
```

Looking at a sample image:

```
[19] sample_image = generator(z_placeholder, 1, z_dimensions, reuse=True)
     z_batch = np.random.normal(-1, 1, size=[1, z_dimensions])
     temp = (sess.run(sample_image, feed_dict={z_placeholder: z_batch}))
     my_i = temp.squeeze()
     plt.imshow(my_i, cmap='gray_r')
```

```
<matplotlib.image.AxesImage at 0x7f87686f0790>
```





As seen in the above figure, the generated image is now starting to obtain the features of the digit **8** in an MNIST dataset. The one on the right resembles **4**. Of course, these images are not yet ready to be augmented to the dataset. But running this entire training process with the right hyper parameter tuning would definitely produce more convincing results.

I tried augmenting the dataset to my existing dataset anyway, and the results were significantly increased.

```
[27] from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import StandardScaler
     from sklearn.datasets import make_moons
     from sklearn import datasets
     from sklearn.ensemble import RandomForestClassifier

     X,y=make_moons(noise=0.85, random_state=1)
     X = StandardScaler().fit_transform(X)
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.4, random_state=42)

     clf = RandomForestClassifier(max_depth=5, n_estimators=10, max_features=1)
     clf.fit(X_train, y_train)

     score = clf.score(X_test, y_test)
     score

     0.975
```

The score after augmenting GAN generated data is **0.975**

6. OBSERVATION

Clearly, it can be observed that augmenting GAN generated synthetic data to the training dataset significantly increases the accuracy score. In my case, it was increased by 2.6%; Also it is worth noting that the synthetic generated in my experiments were not that convincing to the human eye. Still, I got a 2.6% rise in accuracy. This can be further improved by using proper hyperparameter tuning, and pre-processed data to obtain even better results. These observations are certainly very reassuring.

7. CONCLUSION

The unavailability of large datasets for classification purposes can be compensated by augmenting synthetic data. This can be achieved through generative models like 'generative adversarial networks'. The tradeoff here is between the extra computational overhead of the generative model versus the effort and resources needed for data collection. The results obtained in this paper definitely illustrate the great potential of adversarial networks and augmented datasets.

8. REFERENCES

[1] Liaw, Andy, and Matthew Wiener. "Classification and regression by randomForest." *R News* 2.3 (2002): 18-22.
[2] SUE, LEURGANS. "Linear models, random censoring and synthetic data." *Biometrika* 74.2 (1987): 301-309.
[3] Ledig, Christian, et al. "Photo-realistic single image super-resolution using a generative adversarial network." arXiv preprint (2016).
[4] Friedl, Mark A., and Carla E. Brodley. "Decision tree classification of land cover from remotely sensed data." *Remote sensing of environment* 61.3 (1997): 399-409.
[5] Quinlan, J. Ross. "Induction of decision trees." *Machine learning* 1.1 (1986): 81-106.
[6] Breiman, Leo. "Random forests." *Machine learning* 45.1 (2001): 5-32.
[7] Goodfellow, Ian, et al. "Generative adversarial nets." *Advances in neural information processing systems*. 2014.
[8] https://medium.com/@awjuliani/generative-adversarial-networks-explained-with-a-classic-spongebob-squarepants-episode-54deab2fce39 [ONLINE].
[9] Huang, Xun, et al. "Stacked generative adversarial networks." *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Vol. 2. 2017.
[10] Arjovsky, Martin, and Léon Bottou. "Towards principled methods for training generative adversarial networks." *arXiv preprint arXiv:1701.04862* (2017).
[11] Strobl, Carolin, et al. "Bias in random forest variable importance measures: Illustrations, sources, and a solution." *BMC Bioinformatics* 8.1 (2007): 25.
[12] Dietterich, Thomas G. "Ensemble methods in machine learning." *International workshop on multiple classifier systems*. Springer, Berlin, Heidelberg, 2000.
[13] Wu, Zhaohua, and Norden E. Huang. "Ensemble empirical mode decomposition: a noise-assisted data analysis method." *Advances in adaptive data analysis* 1.01 (2009): 1-41.
[14] Bolón-Canedo, Verónica, Noelia Sánchez-Maroño, and Amparo Alonso-Betanzos. "A review of feature selection methods on synthetic data." *Knowledge and information systems* 34.3 (2013): 483-519.
[15] Jaderberg, Max, et al. "Synthetic data and artificial neural networks for natural scene text recognition." *arXiv preprint arXiv:1406.2227* (2014).
[16] Data Augmentation Generative Adversarial Networks Anthreas Antoniou, Amos Storkey, Harrison Edwards
[17] Scikit-learn and Tensorflow documentation for a code reference.

