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 our 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.
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:  

![Generative Adversarial Network](image)

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.

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

![Entropy calculation](image)
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:

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

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]:
# 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.datasets import load_digits
from sklearn.decomposition import PCA

data = load_digits()
X = data['data'][:, :2]  # we only take the first two features.
y = data['target']

# Plots the training points
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Set1, edgecolor='w', s=40)
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 get a better understanding of interaction of the dimensions
# plot the first three PCA dimensions
fig = plt.figure(figsize=(8, 3))
ax = Axes3D(fig, elev=10, azim=20)
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='w', s=40)
ax.set_title('First three PCA directions')
ax.set_xlabel('1st eigenvector')
ax.set_ylabel('2nd eigenvector')
ax.set_zlabel('3rd eigenvector')
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.
Unlike before, where we were using only two features from the dataset for visualization purposes, here we are loading the entire dataset.

Function for creating an individual decision tree:

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:

Here, what we are doing is basically creating decision trees for random subsets of data from our 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.

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:

Loading Data:
Plotting sample data, i.e. MNIST image:

```python
x_train = mnist.train.images[:50000, :]
K_train.shape
```

Let's take a look at what a random image might look like:

```python
randomNum = random.randint(0, 59999)
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:

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

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

Function for defining the Discriminator network:

```python
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_w_conv1', [5, 5, 1, 32], initializer=tf.truncated_normal_initializer(stddev=0.03))
        b_conv1 = tf.get_variable('d_b_conv1', [32], initializer=tf.constant_initializer(0))
        h_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1) + b_conv1)
        h_pool1 = tf.nn.avg_pool(h_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
        #Second Conv and Pool Layers
        W_conv2 = tf.get_variable('d_w_conv2', [5, 5, 32, 64], initializer=tf.truncated_normal_initializer(stddev=0.03))
        b_conv2 = tf.get_variable('d_b_conv2', [64], initializer=tf.constant_initializer(0))
        h_conv2 = tf.nn.relu(tf.nn.conv2d(h_pool1, W_conv2) + b_conv2)
        h_pool2 = tf.nn.avg_pool(h_conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
        #First Fully Connected Layer
        fc1 = tf.get_variable('d_w_fc1', [7 * 7 * 64, 1024], initializer=tf.truncated_normal_initializer(stddev=0.03))
        b_fc1 = tf.get_variable('d_b_fc1', [1024], initializer=tf.constant_initializer(0))
        h_pool2_flat = tf.reshape(h_pool2, [-1, 7 * 7 * 64])
        h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
        #Second Fully Connected Layer
        fc2 = tf.get_variable('d_w_fc2', [1024, 1], initializer=tf.truncated_normal_initializer(stddev=0.03))
        b_fc2 = tf.get_variable('d_b_fc2', [1], initializer=tf.constant_initializer(0))
        y_conv = tf.nn.sigmoid(tf.matmul(h_fc1, W_fc2) + b_fc2)
    return y_conv
```

Function for creating the Generator Network:

```python
def gen_wgan(batch_size, z_dim, reuse=False):
    if reuse:
        tf.get_variable_scope().reuse_variables()  
        # First Layer
        g_conv1 = tf.get_variable('g_w_conv1', [5, 5, 32, z_dim], [32, 32], [1, 1], [0, 0], [32, 32], [0, 0], tf.nn.relu)
        # Second Layer
        g_conv2 = tf.get_variable('g_w_conv2', [5, 5, 64, 32], [64, 64], [1, 1], [0, 0], [64, 64], [0, 0], tf.nn.relu)
        # Third Layer
        g_conv3 = tf.get_variable('g_w_conv3', [5, 5, 128, 64], [128, 128], [1, 1], [0, 0], [128, 128], [0, 0], tf.nn.relu)
        # Fourth Layer
        g_conv4 = tf.get_variable('g_w_conv4', [5, 5, 256, 128], [256, 256], [1, 1], [0, 0], [256, 256], [0, 0], tf.nn.relu)
        # Output Layer
        g_out = tf.get_variable('g_w_out', [5, 5, 28*28, 256], [28*28, 28*28], [1, 1], [0, 0], [28*28, 28*28], [0, 0], tf.nn.sigmoid)
    return g_out
```
Generating a sample image:

```
[9] sample_image = generator(z test placeholder, 1, z dimensions)
test_z = np.random.normal([-1, 1, [1, z dimensions]])
```

```
plt.imshow(my_i, cmap='gray_r')
plt.show()
```

Preparation for training:

```
[12] batch_size = 16
```

```
# Since we changed our batch size (from 1 to 16), we need to reset our Ten
sess = tf.Session()
```

```
[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
```

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)
  _, g_loss = sess.run([trainerD, d loss], feed dict=[z placeholder: z batch, x placeholder: real imag]
  _ , g_loss = 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')
```

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.

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

[16] Data Augmentation Generative Adversarial Networks Anthreas Antoniou, Amos Storkey, Harrison Edwards