README

Authors: Adnan Al Medawer, Leo Hjulstrom, Lukas Toral, Xiaoxu Wu

Data source

We have used images from the quickdraw dataset https://github.com/googlecreativelab/quickdraw-dataset. It consists of over 50 million simple drawings made by players in the game Quick, Draw! https://quickdraw.withgoogle.com/. The simplified 28x28 grayscale numpy bitmap images have been used for this project.
The generator code is written in pure Python. The discriminator models have been written using Keras (https://keras.io/about/) and Tensorflow (https://www.tensorflow.org/learn). Some inspiration and minimal code snippets have been used from the book "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition, A. Géron, 2019, O'Reilly, ISBN: 9781492032649" and its corresponding github page (https://github.com/ageron/handson-ml2/blob/master/10_neural_nets_with_keras.ipynb)

Code description

The source code of this project can be found in a GitHub repository https://github.com/Wuxiaoxue888/ethics.

Main idea

We were intrigued by the idea of generating images. As our inspiration we used the Generative Adversarial Network (GAN) that utilises two models - Generator and Discriminator. The discriminator is a model trained on "real" (original images) and is used to evaluate the images generated by the generator model. The essence of this structure is that the generator model is forced to continuously improve to learn how to fool the discriminator model into thinking that the new images are real.

Our implementation

Discriminator

From the description of the discriminator in the section Main idea, it is obvious that the model plays an important role in the whole architecture. Let's quickly imagine our system without it. We ask the generator to create new images. It is important that we provide feedback to the generator model to evaluate how good the new generated images are. If we were to do this manually, the process would be very slow as we would have to evaluate every single picture. Thankfully we can utilise the discriminator model to do the evaluation automatically without any human interaction.

Implementation

Our discriminator is implemented as a neural network model trained on the original images. To be more precise it is a feed-forward neural network with 784 input nodes, one hidden layer with 100 neurones and one output node. The hidden layer neurones use ReLU as their activation function and the one output neuron use Sigmoid for its activation function. As hinted by the presence of the Sigmoid activation function on the output neuron, it is a regression NN that outputs the confidence (probability), that a given image is from the original set. To optimise the network we use Stochastic gradient descent that tries to minimise the loss given by the Binary Cross-entropy loss function. The motivation to use this loss function is because our problem is basically a binary classification - either the given image is real or fake.

We experimented with different ways of generating fake images for the discriminator model to train on. All the way from images with random noise to slightly altered real images. Although our discriminator model achieved 99%+ validation accuracy, it still struggled to select for what we would call "real looking images". We believe this has to do with the fake images it was given during training. It is important that they are varied enough so that the discriminator model can choose between real, and many different types of fake images. Even though we implemented retraining of the discriminator during the generation process, a main challenge in this project was to get the discrimantor to select for "real looking images".

Multiple discriminators

During our experiments with the discriminator we faced a problem that the images generated are too similar. To deal with this, we decided to create a pool of discriminator models whose hyperparameters are the same but their training data differ. As described in the section Data source, we used 25000 real and fake images. Those images are split into smaller sets for each model: if we train 10 discriminator models, each model will receive 25000/10 = 2500 both real and fake images.

When an image is evaluated by the discriminator, we randomly select a discriminator model that will be asked to evaluate it. The reason we did this was that after few thousand generations, the images start to become extremely similar, by using multiple models, we get a different ranking of the images each generation which increases the variation.

Retraining discriminators

To force the generator to continuously improve, the discriminator has to get better as well. That is why we implemented the retraining of our discriminators. What this means is that when we generate enough ("enough" being defined in code and is configurable) of new images, we give them to our discriminators so that they can recognise the generated images as fake.

Generator

The generator is a model responsible for creating art. There are many ways to implement such model, we have decided to a use genetic algorithm.

Implementation

Our generator follows the classic paradigm - create initial population, evaluate the fitness of the population, select individuals that proceed to the next generation, create offsprings and mutate selected individuals. The cycle repeats itself until a final population is reached which represents our generated images.

Initial population

The initial population consists of two types of images: randomly generated or doped. Let's start with the simpler one - the doped images. Our population can be boosted by injecting a certain number of real images by a process called doping. These images are randomly selected from the real set and represent the perfect solutions that will be mated with the weaker one to create a better image in the end. On the other hand, the randomly generated images are not so aesthetically pleasing but a bit more interesting. We start with a blank 28x28 pixel image. Each pixel has a probability of 0.1% to become black. There are certain things that can change this probability. The closer this pixel is to the middle the higher probability it has that it will become black. We combine this process with second probability that applies only if the selected pixel has a neighbouring black pixel. If it has one it has a 15% probability to turn black which forces the initial images to have some shapes inside them.

We tried different ways of generating the randomly generated images for the initial population, and different combinations of randomly generated and doped images. We achieved the best results by only having doped images in the initial population, combined with continuous doping throughout the evolutionary process. With this approach, the generated images are less original and unique; instead they are more of slight variations and combinations of real images.

Evaluating fitness of the population

The fitness of the population is evaluated using the discriminator models. The whole population is given to a model which returns a probability estimate for each image that defines how sure the model is whether this image is real or not. Higher probability means better fitness of the image.

Selection functions

We won't go into too much detail in this section as we use three standard selection functions for genetic algorithms: Select top N images from the population ordered by fitness, Tournament selection and Roulette wheel selection.

Crossover functions

Our crossover function works by splitting the chromosomes of the parent images either vertically or horizontally. For a pair of two parents it is not certain which is the best orientation to use or where exactly to split them. To address this, we have created an algorithm that searches every possible split in both orientations, and chooses the split that minimizes all the pixel differences at the combined intersection.

Mutations

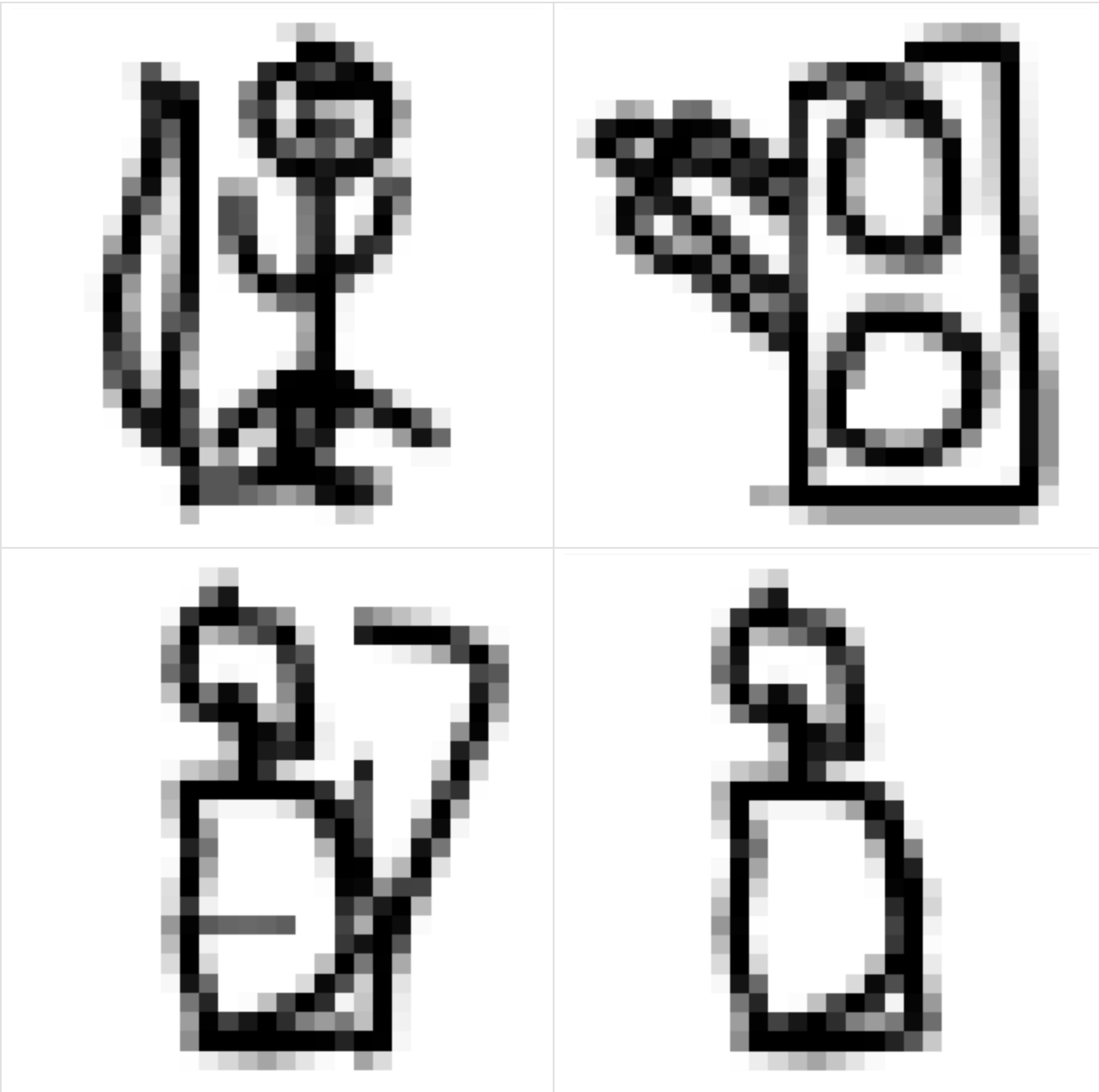
As per good practise in genetic algorithms, mutations help introduce something new in the chromosome regardless of if it is good or bad. When a new image is created there is a 20% chance that it will be mutated.
In our code there are three types of mutations:

- type 1: A random pixel is selected. Its value is updated by a random amount. An adjacent pixel is then randomly chosen and its value is updated by a similar amount as the first pixel. This process iteratively repeats 6-10 times.
- type 2: A random pixel is selected. If the pixel is on the edge, it and neighbouring pixels are turned white. Else, if the pixel is almost white (value lower than 0.4) it and neighbouring almost white pixels will be turned white. Otherwise, a black straight line is drawn in the image.
- type 3: A random portion of the image is turned white.

While selecting the mutation function, a random combination of the three functions is drawn and applied to the image's chromosome.
Visualizations of how the different mutation functions, and the crossover function behave can be found in the experimentation.ipynb notebook.

Results

Here are 3 examples of our favorite generated images.



The fourth image is the closest looking image in the real dataset for the third generated image. The first two examples are not that similar to any image in the real dataset. These results are from the 100th generation after running the program.

Since we chose to use a lot of doping in order to get better results, the generated images are more similar to the real images than if less doping would be used. An automatic way to exclude generated images that are too much of a copy of a real image is recommended for future work. We have evaluated our results of this project by judging the final generated images ourselves, by how much we like them. For future work, the generated images should be evaluated using a more systematic approach with external evaluators, and be compared to some form of baseline model. The main challenge we see with the current implementation is the slow evolutionary process, caused by the slow evaluation of the populations by the discriminator. If the discriminator predictions could be sped up, it would allow for way more generations to be created, which is a key part of an evolutionary algorithm.