Implementation of Genetic Pseudo Rehearsal

Bhasker Sri Harsha Suri and Kalidas Yeturu(✉)

Indian Institute of Technology Tirupati, Tirupati 517506, Andhra Pradesh, India
{cs18s506,ykalidas}@iittp.ac.in

Abstract. Deep neural networks suffer from catastrophic forgetting problem when they are deployed in a continual learning scenario. In our main work, we proposed *Genetic Pseudo rehearsal*, where we generated synthetic data of the previous task using Genetic Algorithms and *pseudo rehearsed* the neural network on it. We demonstrated the computational and memory efficiency offered by our proposed method. In this work, we discuss the implementation details of our proposed algorithm and the experimental setup in detail.

1 Introduction

1.1 Catastrophic Forgetting

Deep neural networks suffer from *Catastrophic forgetting* problem when deployed in a continual learning scenario [1]. Robins [3] proposed the concept of *pseudo rehearsal*, where a generator synthetically generates the data for the previous task on which the neural network is *rehearsed*. Generative replay [4] is a good example of *pseudo rehearsal* where the training data for the previous task is synthetically generated using a Generative adversarial network(GAN).

1.2 Genetic Pseudo Rehearsal

In our main work [5], we proposed to generate synthetic data using Genetic Algorithms instead of GANs. We demonstrated that we could achieve higher efficiencies in computational and memory resource consumption by forgoing the data's photo-realism. In this work, we try to explain the implementation details of our main work.

1.3 Organization of the Paper

The paper has been divided into five main sections. In Sect. 2, we give insights into the implementation of operations like Mutation, cross-over functions, which are an integral part of the Genetic algorithm. We also explain the *Enrichment function* used to enrich the data generated after the genetic algorithm phase.

B. Kerautret et al. (Eds.): RRPR 2021, LNCS 12636, pp. 167–172, 2021.
https://doi.org/10.1007/978-3-030-76423-4_13

To increase the work's availability to a broader audience, we implemented our work in 3 different formats. In the first format, the entire code has been made available in ready-to-run Google Colab notebooks. A detailed description of this implementation is provided in Sect. 3 of the paper. In the second format, the proposed algorithm has been implemented as a Python *function call*(.py file). A function called *Generate_Genetic_data* has been implemented where the user can pass their respective models and parameters as arguments, and the function returns the generated synthetic data as output. The file also gives access to other functions that are part of our main work, like *mutation, cross-over, and Enrichment phase* and *Agreement score*. A detailed description of this format is given in Sect. 4 of the paper.

In order to make our work available to users who build neural networks using frameworks other than Tensorflow, we implemented the work as an *service* as well. The proposed algorithm can be deployed on a local server, and users can generate synthetic data for neural networks that were implemented in any Deep learning framework. The description of this work is present in Sect. 5 of the paper.

2 Overview of the Code

The proposed algorithm was implemented in Python 3 with NumPy [2] library handling all the vector operations that arose at various points in the algorithm. The experiments in the main work used neural networks implemented using Keras with Tensorflow in the backend. The code is made available at the following Github link: https://github.com/BhaskerSriHarsha/Genetic-Pseudo-Rehearsal.

2.1 Representation of Images

The experimentation took place in an image classification setting; hence, it is essential to understand how images were represented in the Genetic Algorithm. The algorithm's crux lies in the process of evolving *genes* of organisms in a population through iterative selection using a fitness function. Each image was considered an organism, and the vector format representation of the pixel values of the generated images was considered the *genes* in the algorithm. So, instead of taking the images in standard picture formats like .jpg or .png, they were considered as *numpy arrays*.

2.2 Fitness Function

Each synthetically generated image was given to the neural network for classification. The softmax confidence of the neural network at the final layer for the target class was considered the *fitness score* of each organism. The fittest 25% of the organisms in a given generation were propagated to the next generation for evolution.

2.3 Mutation Function

The mutation function is responsible for perturbating the synthetic images' pixels with a probability *"p"*. To implement this, *np.random.choice()* function was used. The function generated a vector of size same as that of the given image, in which each element has a probability p of being a value between 0 and 1. The mutation magnitude of each pixel was decided using *np.random.normal()* function, which generated a value between 0 and 1 by sampling a normal distribution.

2.4 Crossover Function

In addition to the mutation function, the cross-over operation was also used to create the next generation of organisms. The cross-over function accepts two NumPy arrays as inputs and the index through which the arrays will be crossed over. The NumPy arrays (images) are clipped at the given *index* and the second half of the arrays are exchanged. The arrays that were passed as arguments are crossed over and ready to be used, and the function does not return anything as it uses the "*Call by Object reference*" property of the Python language.

2.5 Enrichment Phase

Sklearn package was used to implement the *Gaussian mixture models* that was critical in the enrichment phase of the algorithm. The *GaussianMixture* model can be imported from *sklearn.mixture* package and it accepts the *number-of-components* and *data* as arguments. The function fits N number of centers to the *data* where $N =$ *number-of-components*. All the functions mentioned above were used in all the three formats in which our work was implemented. Though the environment changes for the three implementations, the core logic and code for the functions mentioned above remained constant.

3 Colab Notebooks

All the experiments that were reported in the main paper were run on Google Colab notebooks. The experiments used a Tesla P100 GPU and an Intel Xeon Dual Core 2.5 GHz processor. The Neural network was implemented in Keras with Tensorflow in the backend. Numpy was used to represent the images in a vector form; however, *Matploblib* library was used to display the images and generate the final graphs for the experiments. The experiments discussed in the main paper are made available in a ready-to-run form in the **Jupyter Notebooks** folder of our Github repository. The notebooks can be executed directly without any modifications on the Google Colaboratory platform. *Genetic_Rehearsal.py* file, which is also present in the folder, needs to be present in the working directory as it contains the supporting functions to run the algorithm. However, the main code to create the synthetic data using the Genetic algorithm and the

Enrichment phase is written in the notebook. It has to be noted that the purpose of these Google Colab notebooks is to aid in the reproducibility of experiments that were described in the main work. The notebooks generate the synthetic data for networks that are already declared in the notebooks. In case the reader wants to generate synthetic data for their neural network, it is required that the user swaps the default neural network in the notebook with their network.

4 Python Library Files

For users who wish to run the code on their local systems instead of Google Colab environment, the proposed algorithm has been implemented as a python library and can be accessed as a *function call*. To run it, visit the official GitHub repository and download the *Genetic_Rehearsal.py* file in the folder ***.py files*** to your working directory. A requirements file (*requirements.txt*) has been provided to aid the users in creating the virtual environment required to run the code. The main Genetic algorithm that generates the synthetic data is implemented in the function *Genetic_data_Generator()*. It accepts the *model*, *shape* of the image sample in the dataset, *target classes* for which the synthetic data is to be generated, *size of population* in a given generation, *number of cultures*, *number of generations* for which the evolution continues, pixel *mutation probability* and finally the pixel *mutation type* as input arguments. The function returns the generated synthetic data and the respective labels for the individual samples in the form of a list with the first element as data and the second element as labels.

The *Enrichment()* function performs the enrichment operation that was described in the main work using *Gaussian mixture models* from *Sklearn library*. The function takes the target *data*, *labels* of the data, target *model*, *number of centers* for the Gaussian mixture model, *number of classes* and *number of samples* to be generated as parameters. For Step 1 of the Enrichment phase, the number of centers is set equal to the number of classes, and for Step 2 of the Enrichment phase, the number of centers is set to 1. The explanation for this can be found in our main work. Please note that to avoid memory overflows, set the default datatype of the NumPy arrays as *float32* instead of the default *float64*. Since we are dealing with images in this particular application, a Numpy array with *float32* as datatype is sufficient and memory-efficient. In addition to the main functions, the file also has additional functions like *duplicate_remover()*, *duplicate_counter()*, *agreement_score()* etc. which were used in the *Ablation studies* section of the main paper.

An in-situ documentation of each function can be obtained using the *help()* function. For example, the command *help(duplicate_remover)* will print the documentation for the *duplicate_remover* function when executed.

5 Data Generation as a Service

To extend our algorithm's availability to users who have already developed their neural network models using frameworks other than Tensorflow, we are offering

the proposed algorithm as a *service* which can be deployed on a local server. Neural networks developed using any deep learning framework can access our proposed algorithm as a service using HTTP methods.

The Genetic algorithm that generates synthetic data runs on a local server and will be referred to as *GA-service*. The Neural network is deployed on the local machine and will be called as *model-service* from here on. A continuous interaction between the *GA-service* and *model-service* generates the desired synthetic samples. The entire process of generating synthetic data has been split between the *GA service* and *model-service*. Whenever the *GA-service* requires the neural network predictions, the *model-service* requests the current generation of images and returns the softmax confidence for each image to the *GA-service* as a string. The predictions received by the *GA-service* are used by the fitness function to select the fittest individuals for the next generation. The whole process continues until the generated synthetic data reaches a threshold level of fitness.

The algorithm at the *model-service* is described in Algorithm 1. The "/" symbol describes the path from where the server-side script was deployed.

Algorithm 1: Algorithm at the *model-service*

```
status_flag = 0;
target_labels = "1,2,3";
POST('/', data = target_labels, timeout = 1);
status_flag = GET('/training');
while status_flag == 1 do
    images_flag = GET("/flag");
    if images_flag == 1 then
        images = GET('/images');
        predictions = model(images);
        POST('/predictions',data = predictions);
        POST('/reset_flag',data = 0);
        POST('/ready',data = 1);
    end
    status_flag=GET('/training');
end
synthetic_data=GET('/synthetic_data');
```

The synchronization between the two services is achieved by monitoring three variables: *status_flag, images_flag* and *ready_flag* on the server by the client. *status_flag* is responsible for letting the *model-service* know that the generation of synthetic images is still active and on-going. When the *GA-service* flips the status_flag to 0, it means that the required synthetic data is ready, and the evolution procedure can be stopped. *images_flag* variable says that the *GA-service* has prepared the current generation of images, and the *model-service* can acquire them using the GET method. *ready_flag* is used by *GA-service* to know whether predictions for the previously sent images are ready to be collected by the *GA-service* from the *model-service*.

The *model-service* starts the synthetic data generation process by sending POST command to the *GA-service*. The POST command carries the *target labels* for which the synthetic data is to be generated as data. The *model-service* then monitors the *status_flag* on the *GA-service* using the GET command. As soon as the *status_flag* is set to 1, the procedure to generate the synthetic images begins. The *GA-service* begins by generating random images as the first generation. It then sets the *images_flag* variable to 1, indicating the *model-service* to collect the images. The *model-service* collects those images using a GET command and returns the softmax predictions of each image to the *GA-service* in the form of a string. This process continues till a generation reaches a certain threshold level of fitness. The *GA-service* ends the procedure by setting the *status_flag* back to 0. Finally, the *model-service* collects the generated synthetic data using a GET method.

To install the setup, download the *GA-service.py* file available in the *API* folder of the Github repository. The requirements file (*requirements.txt*) is also provided in the folder, which can be used to set up the virtual environment required to run the *GA-service* and *model* side codes. A template (*model-service.py*) for the *model-service* side is provided, which can be used by any deep learning framework in Python. If the user uses any language other than Python, the reference algorithm (Algorithm 1) provided on the Github page can be used to write the *model-service*'s code. Currently, the *target labels* can be sent as parameters to the *GA-service* from the *model-service*. The package will be updated to send more parameters concerning the genetic algorithm soon.

6 Conclusion

In this work, we discussed the implementation details of Genetic Pseudo Rehearsal. The proposed algorithm was implemented in three different formats to increase the work's availability to the research community. All the formats were discussed in the current work to aid the reproducibility of the research.

References

1. French, R.M.: Catastrophic forgetting in connectionist networks. Trends Cogn. Sci. **3**(4), 128–135 (1999)
2. Harris, C.R., et al.: Array programming with NumPy. Nature **585**(7825), 357–362 (2020)
3. Robins, A.: Catastrophic forgetting, rehearsal and pseudorehearsal. Connect. Sci. **7**(2), 123–146 (1995)
4. Shin, H., Lee, J.K., Kim, J., Kim, J.: Continual learning with deep generative replay. In: Advances in Neural Information Processing Systems, pp. 2990–2999 (2017)
5. Suri, B.S.H., Yeturu, K.: Pseudo rehearsal using non photo-realistic images. In: International Conference on Pattern Recognition (ICPR) (2020)