Continual Learning Report

Loris Cino

This report presents the results and steps taken to resolve the assignment concerning the course on Continual Learning for the Data Science Ph.D. program for the year 2023/2024. The objective of this project is to implement a Continual Learning framework based on a Deep Generative Replay buffer. The MNIST dataset and a DCGAN were used to conduct the experiments. During the experiments, numerous technical issues were encountered that affected the performance of the proposed methodology. In particular, the instability problems of the generative models are highlighted by the fact that the proposed methodology relies on learning from the data generated in previous steps.

During the text I will try to answer to the following questions.

1) Measure the memory and (estimate) computational requirements for generative and vanilla replay with raw samples. Is it comparable in memory requirements?

2) How would that solution scale with n tasks? Is it linear, quadratic, or something else?

3) What are the downsides of your approach?

4) What are your ideas for making this solution more memory- and computationally-efficient?

Preliminar Experiments

For the framework structure, having a solver and a generator with excellent performance is essential for the overall functioning of the pipeline. Since $scholar_{i+1}$ is trained on the output produced by $scholar_i$, according to the theory of error propagation, the performance of subsequent scholars degrades very quickly if the previous ones make numerous errors. Therefore, part of the time was dedicated to ensuring that the initial models are capable of performing the assigned tasks without issues.

Test solver

As a solver, a CNN from the EfficientNet family was used, specifically the EfficientNet-B0 network. This network is more than sufficient for classifying images from the MNIST dataset. An experiment demonstrated that its vanilla implementation achieves a validation accuracy of approximately 98% after just a few epochs.

Test generator

Finding an adequate generator was much more difficult. The initial experiments were conducted using a GAN based on a fully connected neural network. Although the generative model in question

managed to generate images of decent quality and of almost all numbers, it created artifacts in the images that caused problems in subsequent tasks. For this reason, it was decided to use a GAN based on convolutional layers (DCGAN). After a lengthy process of hyperparameter tuning, particularly the learning rate, good results were achieved. Setting the learning rate of the discriminator an order of magnitude lower than that of the generator was necessary.

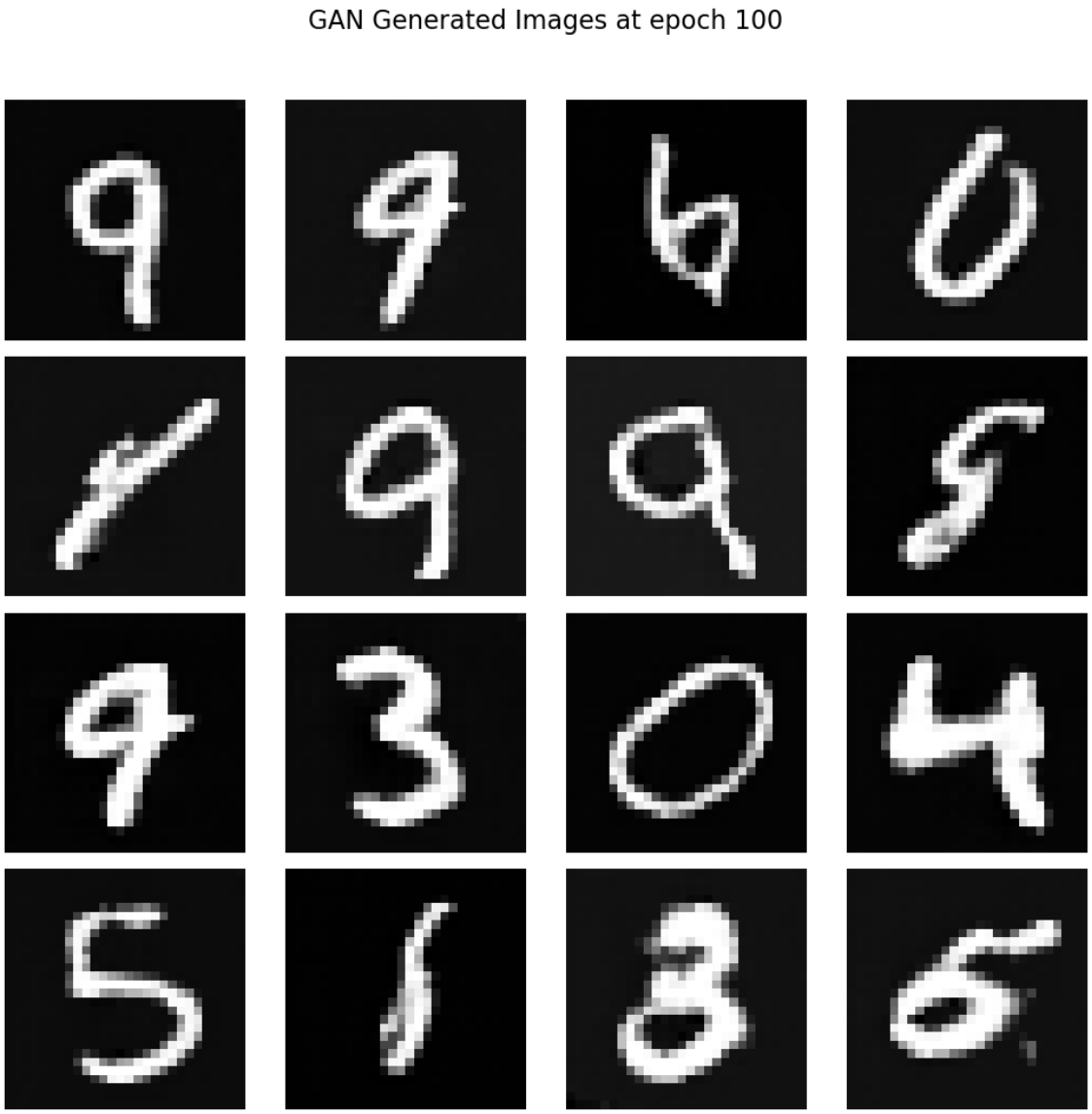

Figura 1 Images generated by the DCGAN after 100 epochs.

Although the generated images are decent, once the same DCGAN is applied to the continual learning task, the images have considerably lower quality.

Metodology

In the paper [1], various ways of implementing the proposed framework are suggested. The first choice to make is whether to train a generator for each task or to continue training the same generator. In this implementation, an approach more similar to the latter was chosen. This allows the memory

requirement not to increase with the number of tasks, but, as we will see later, it could lead to problems in generating new samples. The memory and computational requirements scale linearly with the increasing number of tasks. The resources needed are comparable with the replay buffer. In fact, the approach actually implemented was to train the same generator architecture from scratch because, starting from the weights of the previous training, the generator continued to generate only images of the previous task. A possible solution to this problem could have been modifying the learning rate. Additionally, it was decided to retrain the solver from scratch because empirically it was seen to slightly improve performance. For training the generators, a further hyperparameter tuning process was necessary since the parameters previously used did not yield good results; specifically, the learning rate was lowered further. During the experiments, efforts were made to have a dataset as balanced as possible by generating more images and selecting them to have a balanced dataset, but this requires generating more data and discard the data points belonging to the most common class. Have a balanced dataset is more difficult with the increasing number of tasks. This was only possible for the first tasks because of the deterioration in performance.

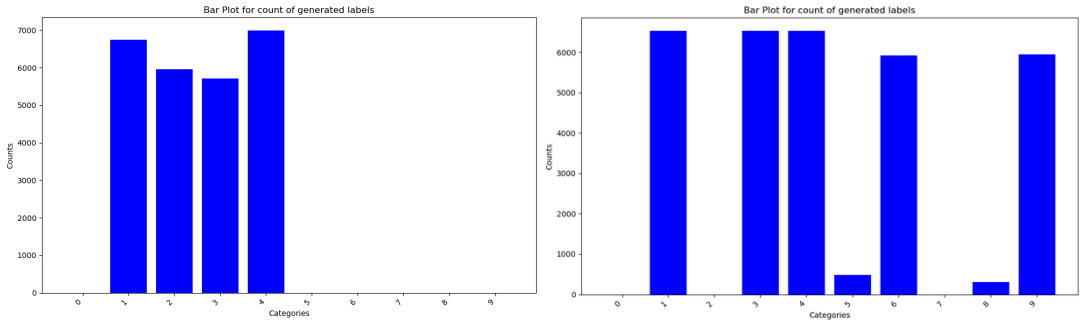

Figura 2 Balancing of the generated data sample. On the second task (left) and on the last task (right)

As can be seen from Figure 2, in the initial tasks, the number of generated samples is uniform across all classes, but this is no longer true in the later tasks.

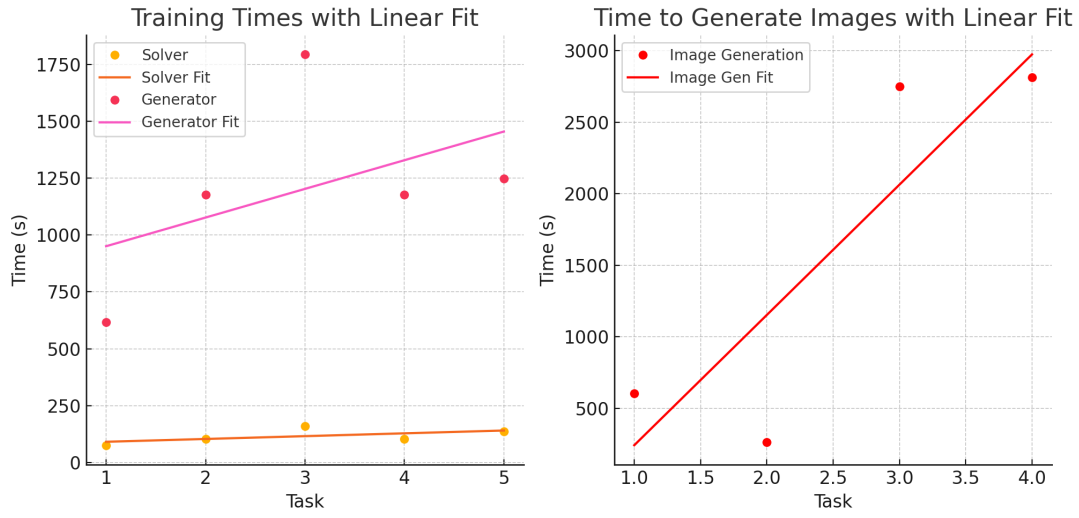

Figure 3 Time in seconds for different part of the training loop as number of tasks increase.

Figure 3 shows that the training time for both the solver and generator increases linearly with the number of tasks. However, the time needed to generate images for the previous tasks is not as clear. It is possible that setting an upper bound on the number of generated images affected the results. I expect that the time would increase dramatically with the number of tasks. I attribute this to the effort required to balance the dataset.

Results

Despite this process, the performance of the solution degrades rapidly as the tasks progress. This can be seen both in the quality of the generated images and in metrics such as ACC metric, A metric, FWT (Forward Transfer), and BWT (Backward Transfer). For the experiment with the learning rate set to 1e-4 the FWT is equal to -0.127 and the BWT is equal to -0.121.

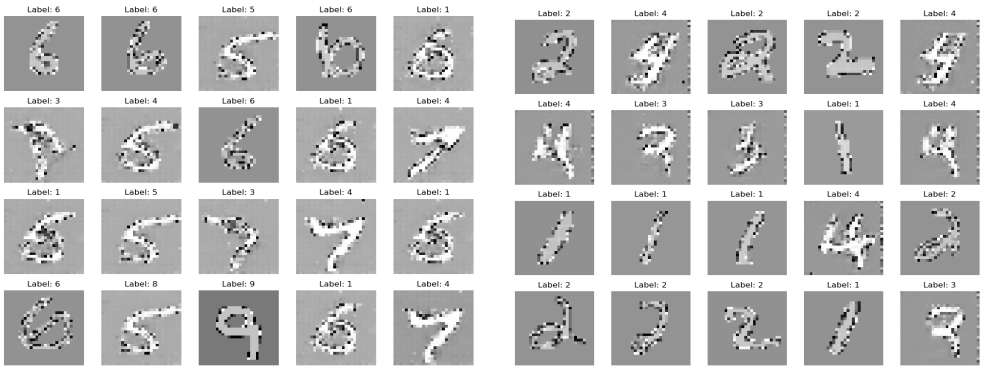

Figura 4 Images generated on the second task (left) and images generated on the last (right)

Another way to highlight the problems of the architecture is to look at the images generated in the various tasks. As can be seen from Figure 4, in the initial tasks, the generated images are of good quality both in terms of image and label. In the subsequent steps, we see that some classes are not generated and there are some image classification issues.

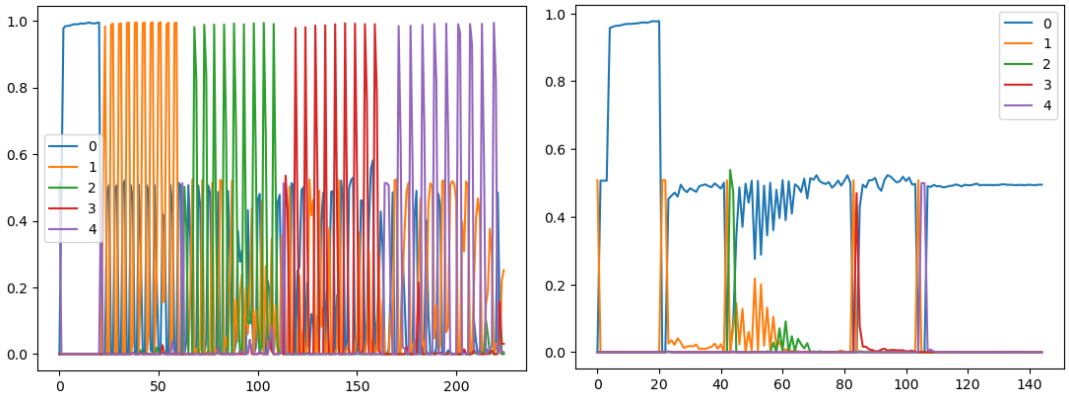

Figura 5Accuracy over time with of two different esperiment. The difference is the learning rate 1e-3 (left) and 1e-4 (right)

The Figure 5 shows the performance on the different task, in both cases they are not good. Probably when the learning rate is high the network is not able to learn for the noise in the data. When the learning rate is low the network learns from the data, but it learns noise.

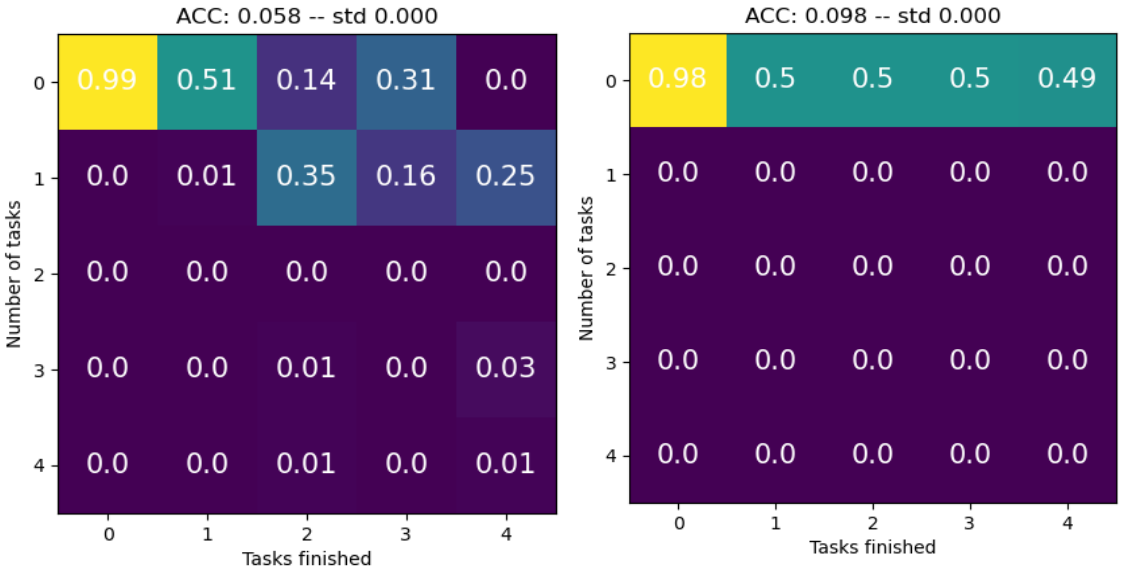

Figura 6 Accuracy over task finished. The left is where the learning rate is 1e-3 and the right 1e-4.

Strangely, the network is not able to learn even the current task despite the solver having performance close to 99% on the training set. It is not easy to explain the reason for this; perhaps it learns to classify the images by differentiating them from the generated ones. Further experiments are needed to understand the reason.

Improvements

One of the most impactful and easy-to-implement improvements would be to use a generator that allows you to generate images belonging to a specific class, such as Conditional Generative Adversarial Nets, this could make the framework more computationally efficient because allows to generate just the number of images needed. Additionally, modifying the learning rate or other hyperparameters, such as batch size, could probably improve the solution's performance, particularly by providing different hyperparameters for the various tasks. The framework proposed needs more memory for the incrementing number of tasks only for the increasing number of data point. It is not easy to improve it, maybe just a batch generation could be useful.

Bibliografia

[1] H. Shin, J. K. Lee, K. Jaehong e K. Jiwon, Continual Learning with Deep Generative Repla, ArXiv, 2017.