Generative continual learning

Marco Montagna

June 15, 2024

1 Introduction

For this assignment, I implemented a model for classifying the MNIST dataset. The problem is posed in the class incremental setting, where the model needs to classify the images into one of ten classes while only having access to samples from two classes at a time. Using a naive approach of training a classifier on each task sequentially leads to the model learning to solve only the current task and catastrophically forgetting the previous tasks almost immediately after starting the new training. To alleviate this problem one of the possible approaches is to use replay. The samples from the new task are mixed with those from previous tasks. This way the model can learn to classify the new samples without immediately forgetting how to solve the old tasks. There are different approaches to achieve this and the goal of this assignment is to implement a model that uses a generative model to produce samples from the previously tackled tasks.

2 Implementation details

The dataset used is MNIST, which contains around 60k examples for training. They are evenly divided into the ten classes that compose the dataset. The model used to classify the images is a simple multiplayer perception since it is more than enough to solve the problem.

For the model that creates the images, the choice was made to use a GAN. GANs are composed of two main modules: the generator and the discriminator. The discriminator aims at distinguishing between the real images and the ones produced by the generator, while the ladder aims at fooling the discriminator. These two modules can have any architecture and two were tested: the first used MLPs for both models while the second used convolutional models. After testing both, the GAN using the convolutional modules was found to be more reliable and produced better results. For this reason, the final experiments use this model.

The main training loop starts with training the *solver* on the first task, then training the *generator* on it. Before the second task, the generator creates a predetermined number n_{gen} of samples, which are then mixed with the current dataset. These new samples are generated from a random distribution and are labeled using the solver. This combined dataset is then used to train the solver. After that, the generator is reset and trained on the combined dataset. This is repeated for all the tasks.

Training the solver is much quicker than the generator, and for this reason, the number of epochs for training the two differs. It was also observed that when the generator has to learn more classes it is beneficial to train it for longer. This is why the number of epochs that it is trained for can be specified for each task independently.

The generated samples are obtained by feeding random inputs to the generator. This means that there are no guarantees that the generator will produce the same number of samples for each class. I tried to force the number of samples to be the same by labeling the samples and then selecting only the ones that I need, but this approach proved to be quite difficult to train since it could fail both if the generator collapsed and never produced outputs from a certain class, which is a common problem with GANs, or if the solver did not predict samples from one class, which is also common for later tasks. For this reason, this approach was abandoned after some testing and I accepted the risk of forgetting some numbers if the generator stopped producing them.

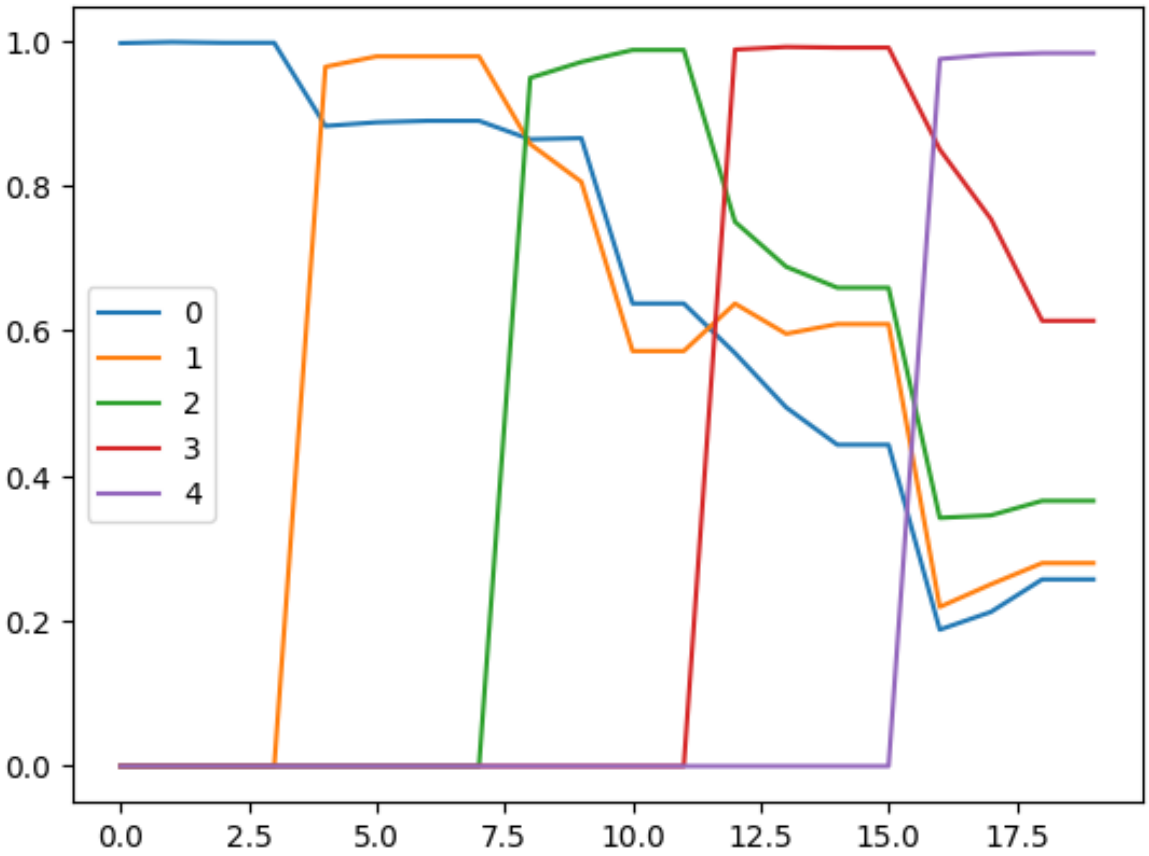

Figure 1: Training validation accuracy for $n_{gen} = 12k$.

3 Training

Figure 1, 2, and 3 show the validation accuracy for three runs, where n_{gen} was set to respectively $12k$, $1.2k$, and $120k$. These numbers were chosen because each task has roughly that many training samples, meaning that if the model produces roughly the same number of samples for each class, the data distribution should be balanced in the first case. In the second case instead, there are much fewer examples from the generated samples so we expect the model to remember less the previous tasks. In the last case, the model has much more generated examples, leading to potential problems in learning the new tasks.

4 Results

For $n_{gen} = 12k$ the approach is able to slow down the rate at which the model forgets the previous tasks, which is evident by looking at the results in figure 4. As expected the further in the past a task is the worse the performance. This is probably due to the generative model being trained on its own outputs, which degrades the quality of the generated images over time.

For $n_{gen} = 1.2k$ the results it seems that the model, when training on task $n+1$ performs very well on task n as well. When training on task $n+2$ though the performance on task n drops significantly. This is probably due to the generative model not having enough samples from task n to learn from and thus forgetting that task. It instead has high-quality data for task $n+1$ and thus is able to generate good images for that task.

For $n_{gen} = 120k$ the model is able to remember only the first task. This could be caused by the generator, on the second task, being trained on a large number of images from the first task, and much fewer from the second task. This could lead the generator to focus on samples belonging only to the

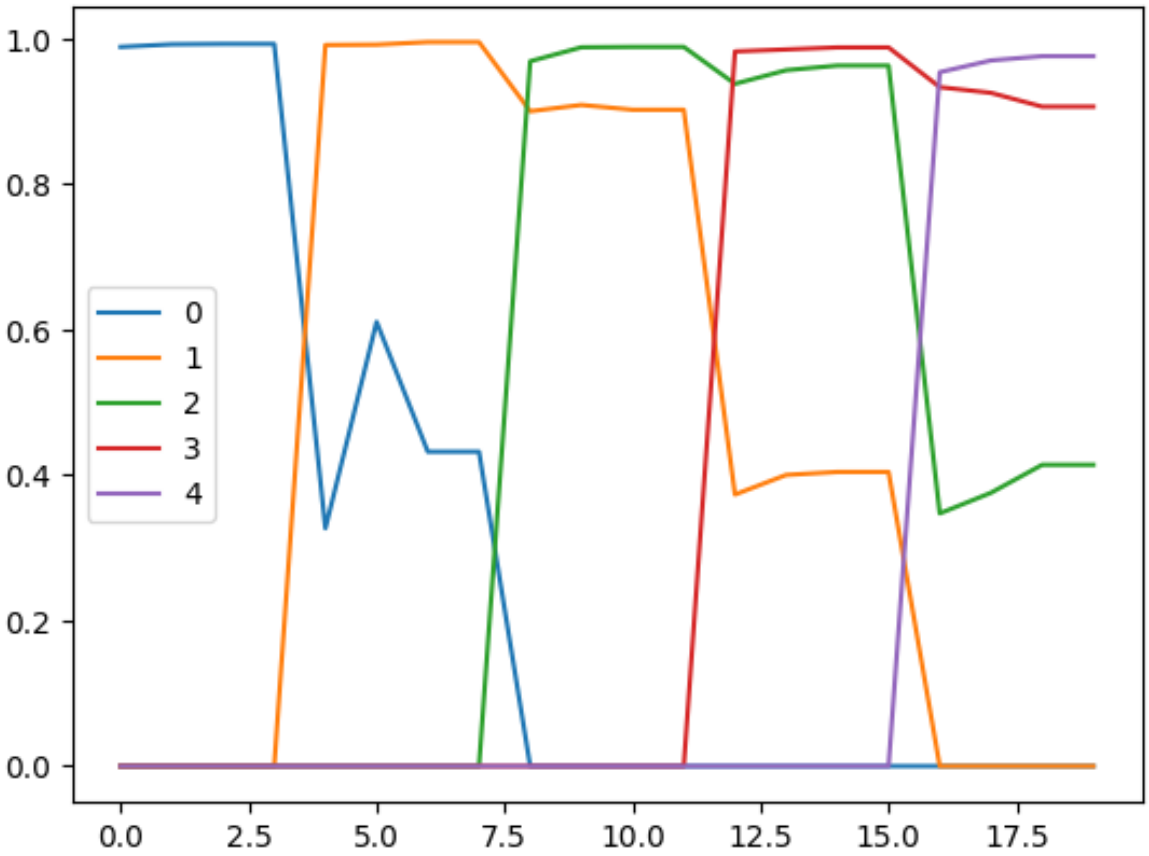

Figure 2: Training validation accuracy for $n_{gen} = 1.2k$.

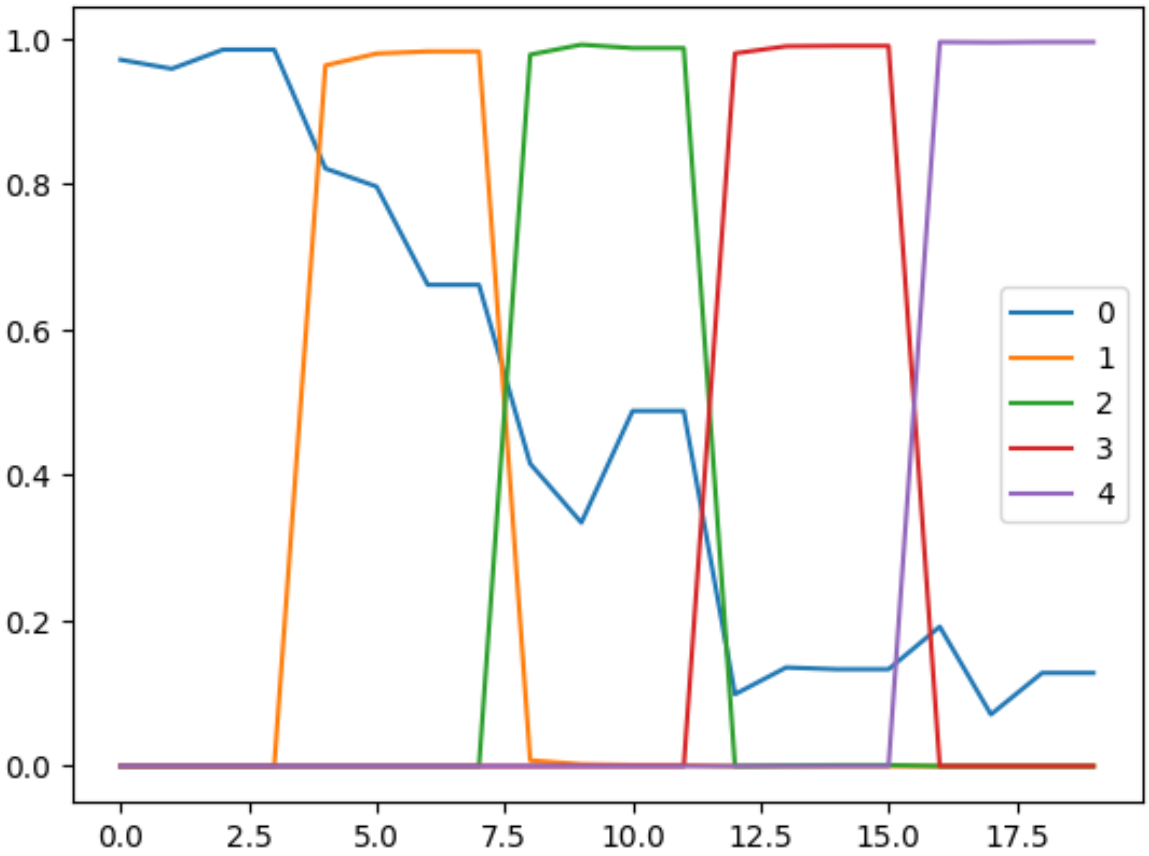

Figure 3: Training validation accuracy for $n_{gen} = 120k$.

first task. This bias is then propagated to the later tasks as well.

Regarding the memory and computational requirements I measured the size of my model to be around 5.7MB, while the size of the MNIST dataset is around 180MB for the training set. This means that the generative approach is around 30 times more efficient memory-wise. It is also important to note that the memory requirements for the vanilla replay approach increase linearly with the number of tasks, while they do not change for the generative approach, since we only need the weights of the model to generate any number of samples for however many tasks. Regarding computational complexity the generative model is more expensive than the vanilla approach. This is especially evident during training since this model needs to be retrained from scratch for each new task. Moreover, if we want to preserve the number of samples for each class, the length of the training dataset grows linearly with the number of tasks, meaning that the computational cost of using a generative model grows quadratically with the number of tasks. Another drawback I see in this approach is that training the generative model on its own outputs degrades the quality of the results the more tasks we add, which means that the model will have lower and lower accuracy for previous tasks, with no real possibility of backward transfer. Table 4 shows that as expected backward transfer is always negative.

	BWT	FWT
$n_{gen} = 12k$	-0.609	-0.100
$n_{gen} = 1.2k$	-0.661	-0.116
$n_{gen} = 120k$	-0.954	-0.131

One way to make the model more computationally efficient could be to fix the number of generated samples independently of which tasks we are on. This would make the computational cost linear in the number of tasks. As seen in the cases where $n_{gen} = 1.2k$ and $n_{gen} = 120k$ unbalanced classes are reflected in the generated images distribution, which creates a feedback loop that precludes the generator from learning the real distribution of images. This means that to keep the number of generated images constant we would have to take fewer and fewer training samples for the new tasks. This is not ideal, since it would make learning new tasks more challenging. Still, it could be interesting to explore the trade-off between the computational cost and the model performance.

One way to improve the memory cost of the vanilla replay could be to mix it with the generative approach. The model could save a small number of of samples from each task and use them to train the generator. This would reduce the tendency of the generator to produce worse samples the further back in time a task is. Still, having fewer samples would reduce the capabilities of the generator and in turn of the solver so the trade-off between memory requirements and performance should be studied.

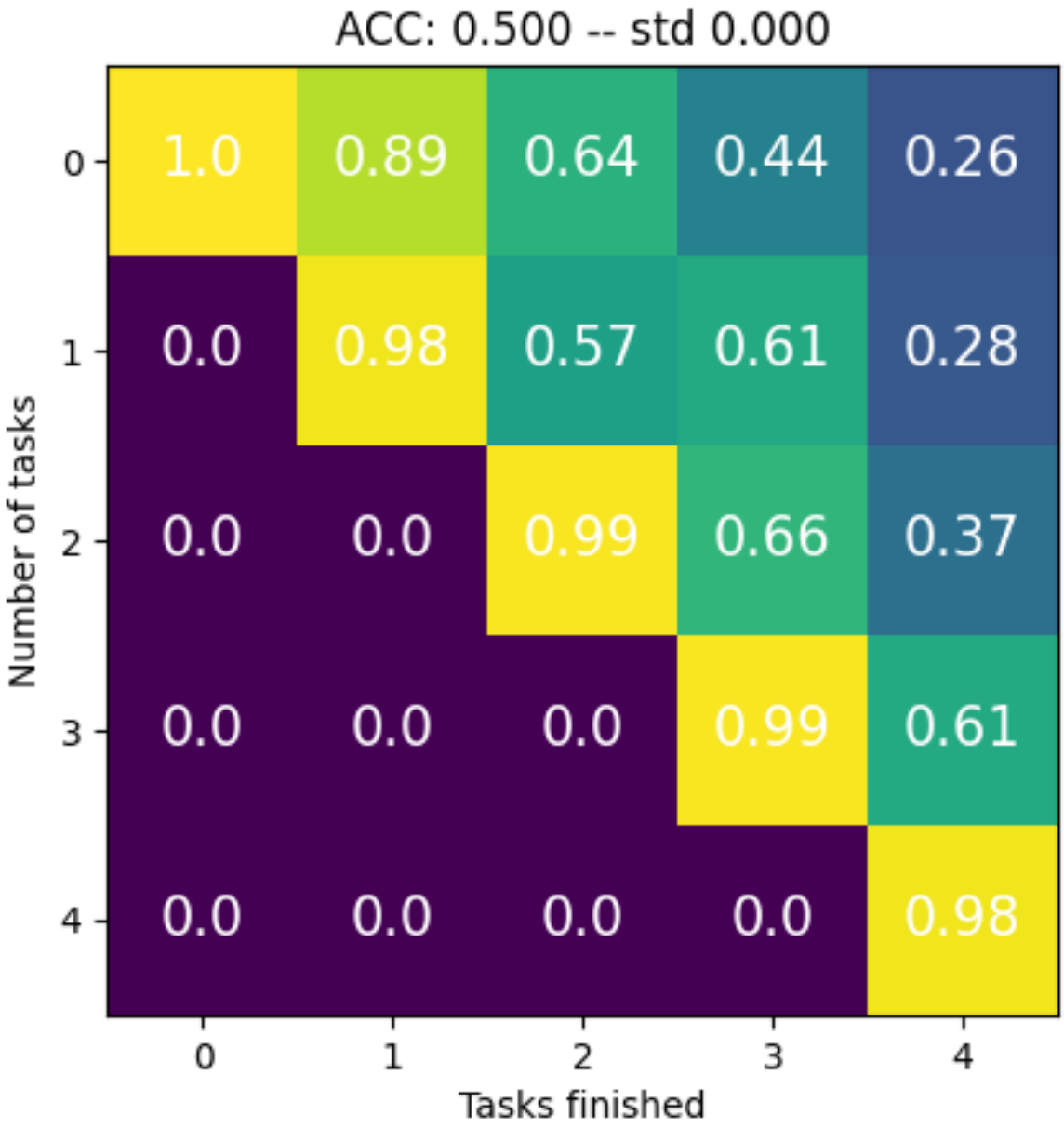

Figure 4: Task based validation accuracy for $n_{gen} = 12k$.

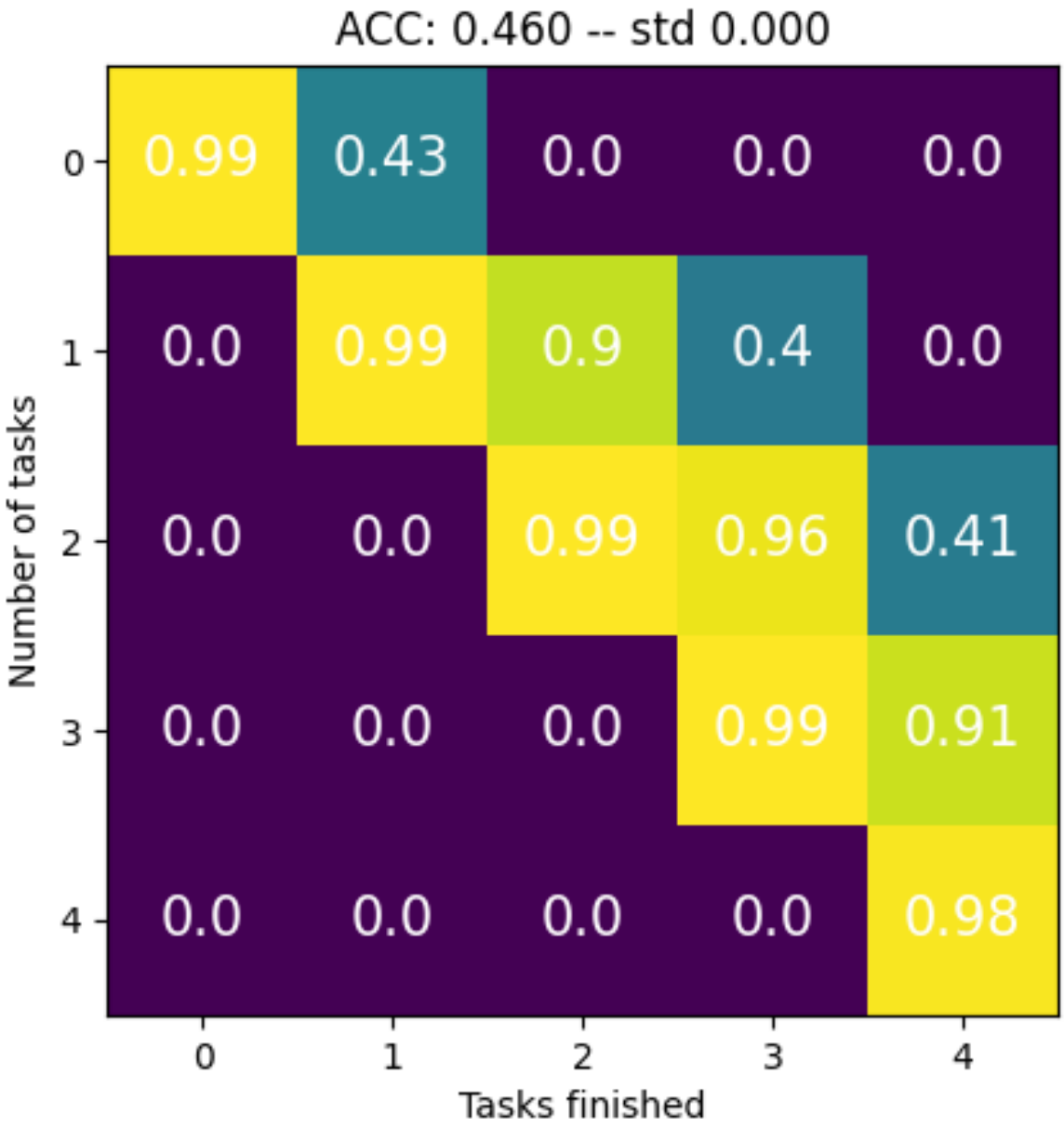

Figure 5: Task based validation accuracy for $n_{gen} = 1.2k$.

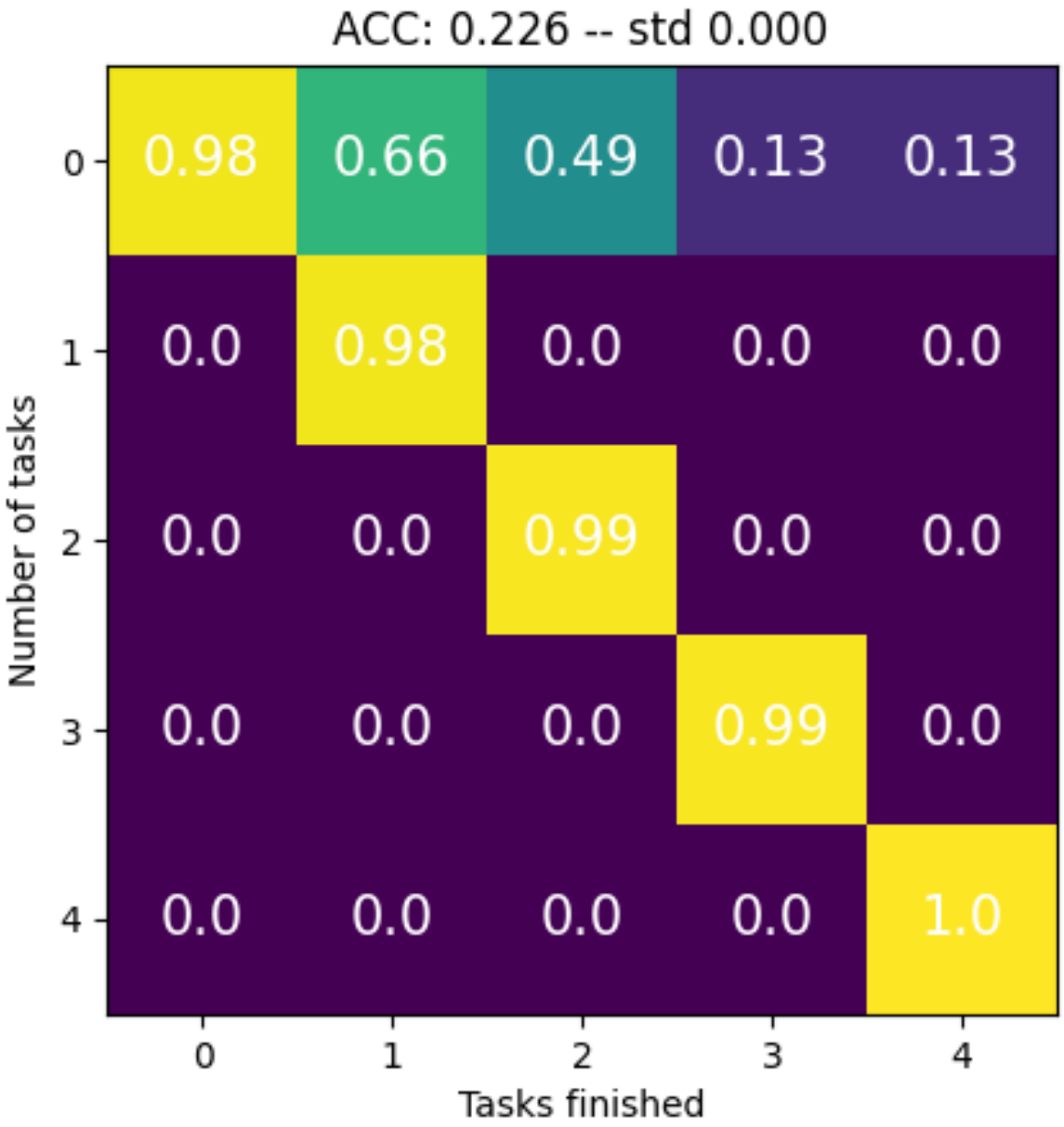

Figure 6: Task based validation accuracy for $n_{gen} = 120k$.