An Empirical Study of Continual Learning for Autonomous Driving Under Simple Circumstances

Reza Moravej
University of Toronto
`ADD Email address here`

Ruiwen Li
University of Toronto
`ruiwen.li@mail.utoronto.ca`

Xingxing Yan
University of Toronto
`estrella.yan@mail.utoronto.ca`

Abstract

A desirable property of neural networks is to learn new tasks in a sequential matter continually. A particular application domain where such property is handy is Autonomous driving. However, neural networks suffer from a phenomenon known as catastrophic forgetting, where the model performance on previously learned tasks degrades when learning new tasks. There have been many approaches to overcome this issue, including EWC[1], Generative replay [2]. We study the performance of these two approaches when training the model to learn to drive in different. We find that these continual learning algorithms helps the model with generalizability and less forgetting for autonomous driving tasks. Our code is available on Github [1].

1 Introduction

Deep neural networks have shown to be extremely powerful for image classification tasks. However, most of the time it requires to train on the data that is i.i.d distributed. If we obtain new data, we need to completely retrain the model with the whole dataset along with the new data, which is not an ideal solution due to privacy issues and limited computation resources. One possible solution is train the existing model consciously on the incoming stream of data without access to the data from previous tasks.

The main idea behind continual learning is to train a neural network for a number of sequential tasks while maintaining the *knowledge* learned from previous tasks, where the data from preceding tasks are no longer available when training the new ones. The main obstacle for continual learning is a phenomena known as *catastrophic forgetting*. The idea is that when a model learns a new task, it loses the information obtained from previous tasks.

Autonomous driving is a primary scenario where continual learning is applicable. Ideally, the vehicle should be able to drive in environments it has not seen during training. For instance, if the model has seen clean roads and sunny sky, it should still be able to drive in snow with different lighting conditions. To achieve this goal, we will have to continuously train the model to not only learn to drive in different environments, but also to not forget what it has learned from the past environments.Training a model to drive can be very expensive (both in terms of the sheer size of

[1]https://github.com/liruiwen/CSC412-project

the dataset and computational time for training the model), thus training from scratch on different environment is not a feasible solution. To the best of our knowledge, there has not been any study that compares the performance of continual learning algorithms on an autonomous driving task.

There have been many approaches to overcome the problem of catastrophic forgetting, including regularization-based, rehearsal-based (real data replay), pseudo-rehearsal-based (generative replay), parameter-isolation-based. We will mainly focus on regularization-based and pseudo-rehearsal-based methods since they fit the context of this course. We evaluate the performance of two popular continual learning algorithms, Elastic weight consolidation (EWC) [1], and Generative replay[2] on an end-to-end autonomous driving task.

2 Model and Environment Description

There has been a lot of recent work on autonomous driving which take advantage of state of the art RL and computer vision techniques. For the purpose of this project, we consider training a simple End to End model similar to the one presented by Bojarski *et al.* [3]. The model consists of a CNN along with a number of linear layers. Given an input image of the vehicle surroundings, the model outputs a steering direction. The network is trained to minimize the mean squared error between the steering command output by the network and the command given by the expert driver.
We will use CarRacing-v0 environment in openAI gym, since training on more *realistic* environments can be costly. Since openAI gym is meant for training RL agents, we have to collect a dataset of (image, steering direction) pairs to train the model using supervised learning. The *expert* steering direction will be generated for each image using a PID controller. [2]

3 Figure or Diagram

To generate different tasks, we will apply different filters, along with some noise, to each image in the dataset. The aim is to have a single model perform well on all the datasets (i.e. tasks). As shown in Table 1, We generate two datasets; one applying different scale of Gaussian noise to the original image and the other applying different color filters, each strategy generates a new dataset and each dataset consists of 3 tasks.
For the dataset with Gaussian noise, we use the original image for Task 1, apply Gaussian noise $\mathcal{N}(0.4, 0.2)$ to the original image for Task 2, and apply Gaussian noise $\mathcal{N}(-0.4, 0.2)$ to the original image for Task 3. For the dataset with color filer, we use the original image for Task 1, apply a cool color filer to the original image for Task 2, and apply a warm color filter to the original image for Task 3. Each of the dataset consists of 4445 training images and 9414 test images. Our class labels are integers from 0 to 19, which is assigned based on the degree of the steering direction. If the car is steering left, it has a class label close to 0. In contrast, if the car is steering right, it has a class label close to 19. Due to the natural of the autonomous driving problem, the class label is highly unbalanced in our datasets, 44.1% of samples is of Class 9, while 0.4% of samples is of Class 0 or Class 19, i.e. for most images in our datasets, the car is going straight. Figure 3 in appendix shows the class distribution of our datasets.

4 Formal Description

In our project, we only consider domain-incremental learning setting where a model learns from a stream of data composed by several tasks, and there can be overlapping in class labels for different tasks. Formally, let $\mathcal{D} = D_1, ..., D_N$ over $X \times Y$ denotes a stream of data, where X is the input images, Y is the input labels, and N is the number of tasks. We define a model $f : X \rightarrow \mathbb{R}^c$ that takes an input image and outputs the probability of each class. At each time step t, the model receives a batch of data B_t^n from task D_n. A continual learning algorithm A can be defined as:

$$A_t : \langle f_{t-1}, B_t^n \rangle \rightarrow f_t$$

[2]For this purpose borrow starter code from CSC2626 A1 at
https://github.com/florianshkurti/csc2626w21/blob/master/assignments/A1/A1.pdf
We provide a summery of the code structure and what we have implemented in Appendix B.

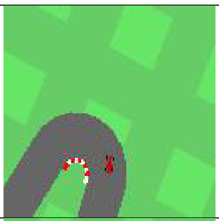
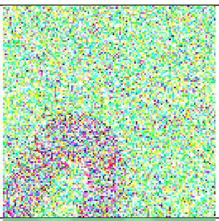
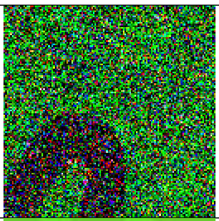
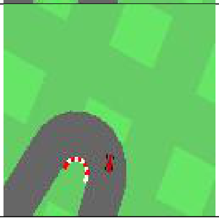
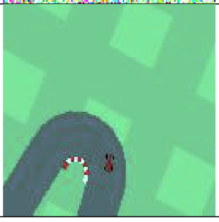
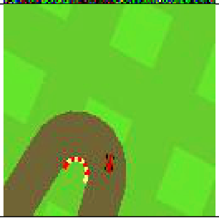
Filter or Noise Type	Task 1	Task 2	Task 3
Gaussian			
Color Filter			

Table 1: The two dataset used in our experiment, one generated by applying Gaussian noise and the other generated by applying color filters.

Algorithm 1 shows our training protocol.

Algorithm 1: CL Training

```
Input : CL Algorithm $A$, number of epoches $E$
Initialize: Model $f$;
for $n \in \{1, \dots, N\}$ do
    for $e \in \{1, \dots, E\}$ do
        for $B_n \sim D_n$ do
            $f \leftarrow A(f, B_n)$
```

We use Average Task End Accuracy as our metrics for performance measure. It is defined as follows:

$$\text{Average Task End Accuracy} = \frac{1}{ij} \sum_{i} \sum_{j} a_{ij}$$

where a_{ij} is the accuracy evaluated on the test set of Task j after trained on the train set of Task i.

5 Related work

Continual Learning There have been many approaches to overcome the problem of catastrophic forgetting, including regularization-based, rehearsal-based (real data replay), pseudo-rehearsal-based (generative replay), parameter-isolation-based. Rehearsal-based methods store real data examples from previous experience in a memory buffer and replay them in some way to reduce forgetting. Pseudo-rehearsal-based methods do not explicitly store data. Instead, they learn a distribution over which the data is assumed to be generated. In this project, we focus on a regularization-based (EWC) and pseudo-rehearsal-based (Generative Replay) methods.

EWC EWC [1] is a regularization-based method that utilizes quadratic regularization over the model parameters that are important to the previous task to prevent them from changing too much when training the current task. The importance of each parameter is approximated by the diagonal of Fisher Matrix. Chaudhry *et al.* [4] propose an online optimized version of EWC named EWC++ which is used in our project.

Generative Replay Shin *et al.* [2] propose a pseudo-rehearsal-based method that replays synthetic data sampled from a learned generative model.A generative model (a GAN in this case) is trained to generate data similar to the data from the past tasks. The network is jointly optimized using an ensemble of generated past data and real current data. We describe the details of generative replay in Appendix A.

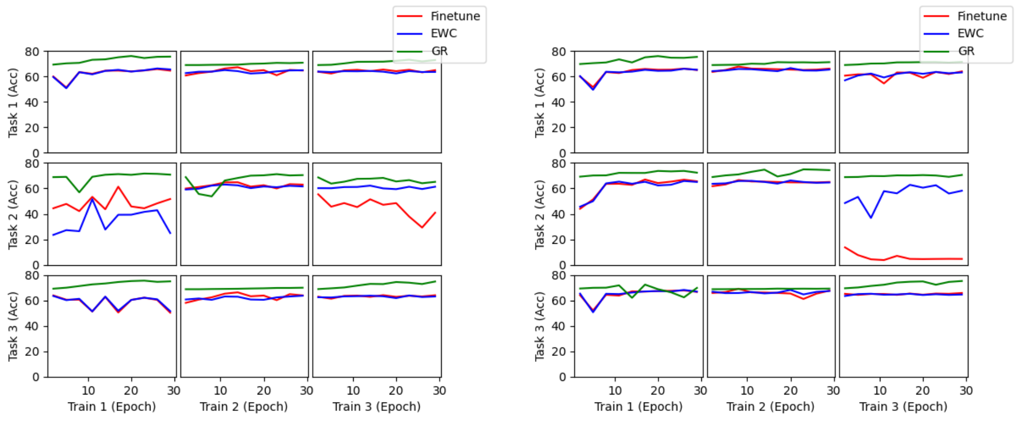
6 Results and Demonstrations

In our experiment, we compare the performance of EWC++ and Generative Replay (GR) along with the lower bound method: finetune and the upper bound method: i.i.d training on the datasets we generated from Section 3. The fine tuning method sequentially trains the model on different tasks without any strategies to prevent forgetting. The i.i.d training method merges and shuffles all data from different tasks into a single *super task* and trains the model on this single *super task*. Note that the i.i.d training method may suffer from issues such as generalizability, it does not deal with catastrophic forgetting as it has access to all data at once.
We use a small Convolutional Neural Network with 4 Conv layers and 3 FC layers activated using ReLU as our learning model (the model is similar to the one presented by Bojarski *et al.* [3]). We use Adam optimizer and a learning rate of 0.001 for training. We train each task for 30 epochs, and the final performance is evaluated based on 3 independent runs. We use a batch size of 256, and minimize the Cross Entropy Loss during training. In order to reduce the bias towards dominated class labels, we rescale the weight given to each class by multiplying the inverse of class distribution density of each class when calculating the Cross Entropy Loss. For EWC++ and GR, we evaluate the performance of several groups of hyperparameters on the train set, and choose the best group for evaluation on the test set.

For the sake of easier fine-tuning, the input image size is resized to 32 * 32 for Generative Replay, which is different from other methods (128 * 128). The base model (solver) is also modified for generative replay; it uses a feed forward network a single hidden layer, different the CNN that other methods use.

Figure 1 shows the training curves of finetune, EWC++, and GR. The x-axis represents training epochs, in which case we sequentially train on Task 1, Task 2, and Task 3. The y-axis the test accuracy for each task. Figure 4 shows a comparison of the performance between finetune, EWC++, and GR at the end of each Task (for example, the plot at column 3, row 1 indicates accuracy of the models trained on tasks 1,2 and 3 and tested on task 1).

Table 2 shows the Average Task End Accuracy of finetune, i.i.d, EWC++, and GR.



(a) Gaussian (b) Color Filter

Figure 1: Training curves of finetune, EWC, and GR.

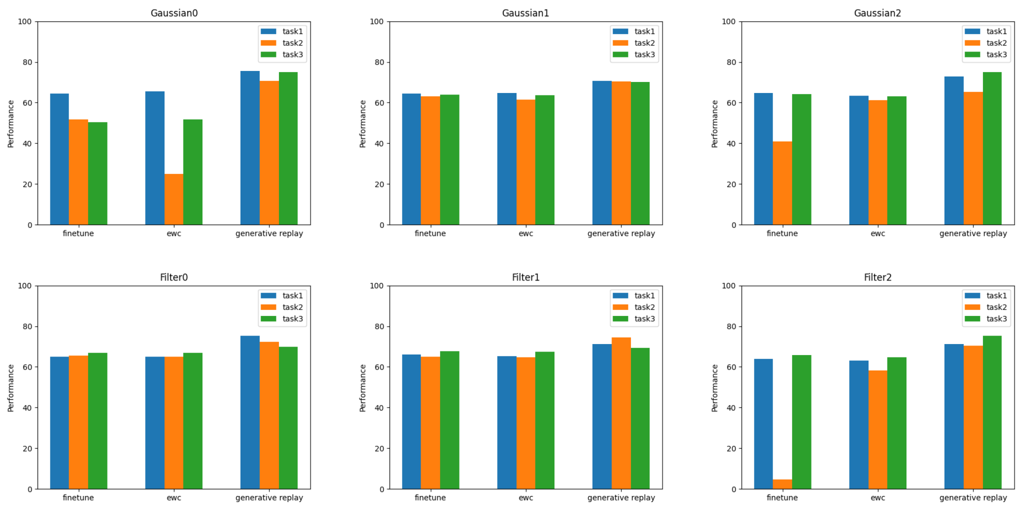


Figure 2: Task End Accuracy of finetune, EWC, and GR.

Method	Gaussian	Color Filter
finetune	58.6 ± 11.2	58.9 ± 1.9
i.i.d	64.3 ± 0.3	66.2 ± 2.0
EWC++	57.8 ± 9.0	64.5 ± 3.2
GR	71.7 ± 3.4	72.1 ± 2.3

Table 2: Average Task End Accuracy of finetune, EWC, and GR

7 Discussion and Limitations

Our result shows that EWC++ can help prevent forgetting. In Figure 1 (a) and (b), we see that the performance of finetune drops for the subplots in the second row and the third column. This indicates that the model forgets what it learned from Task 2 after trained on Task 3 in both datasets. EWC++ is an effective method to prevent forgetting as the model still keeps high accuracy on Task 2 after trained on Task 3. Also, we see that for both datasets, training on Task 1 can result in high accuracy on Task 3 and training on Task 3 doesn't have much impact on the accuracy of Task 1. This may indicate that data in Task 3 still preserves many important features after we augment it from the Task 1 data (i.e. the original image) by applying Gaussian noise $\mathcal{N}(-0.4, 0.2)$ or a warm color filter.

In general, the catastrophic forgetting phenomenon is less obvious than our expectation. An important reason may be that the class labels in our datasets is highly unbalanced (see Fig 3). This is a common problem in autonomous driving, as for the majority of the time the vehicle just has to drive straight. This can cause the model to overfit and predict the label in majority (driving straight) and get good performance. We suspect this to be the primary reason for all continual learning algorithms to perform similarly on all tasks (see fig2). We hoped to address this issue by weighted Cross Entropy Loss for training, but such approach did not seem to solve the issue. For future work, ww suggest studying reducing prediction biases via algorithms such as DAGGER [5].

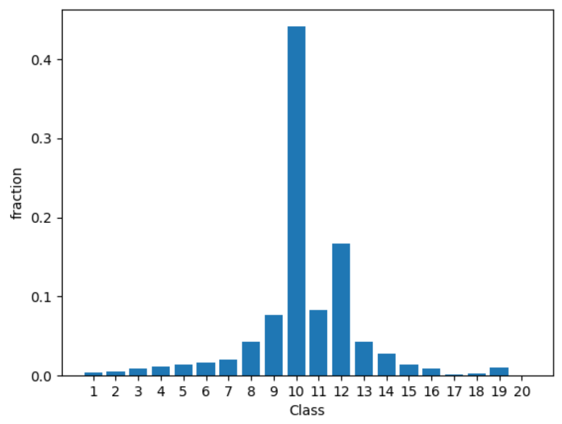


Figure 3: Class distribution of our datasets. Class 10 corresponds to straight steering, 1 is a sharp left steer and 20 a sharp steer to the right

8 Conclusion

We studied the performance of continual learning approaches on an end-to-end autonomous driving task. While all approaches overfitted on the data and predicted to drive straight for the majority of steps, EWC and GR performed better than the naive fine-tuning method. The more sophisticated methods actually learned to overfit the data better and learned to drive straight in all tasks. While this is not ideal for an autonomous driving task, it indicates that EWC and GR were able to learn the same thing across all tasks and preserve previously learned knowledge, hence achieving higher accuracies.

References

[1] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

[2] Hanul Shin, Jung Kwon Lee, Jaehong Kim, and Jiwon Kim. Continual learning with deep generative replay. *arXiv preprint arXiv:1705.08690*, 2017.

[3] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.

[4] Arslan Chaudhry, Puneet K Dokania, Thalaiyasingam Ajanthan, and Philip HS Torr. Riemannian walk for incremental learning: Understanding forgetting and intransigence. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 532–547, 2018.

[5] Stephane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning, 2011.

Appendix

9 Appendix A: Continual Learning with Deep Generative Replay

A scholar H is defined as tuple (G, S), where a generator G is a generative model that produces real-like samples and S is a task solving model parameterized by θ. Notice that in our particular case, the solver is the neural net (CNN + feedforward) that accepts an image an outputs a steering direction.
Recall that given $\mathcal{D} = D_1, ..., D_N$ corresponding to detests for N tasks, the goal is to train a single model on all datasets *once* such that the unbiased sum of losses among all tasks $(\mathbb{E}_{(x,y)}[L(S(x, \theta), y)])$ is minimized. Here D is the entire data distribution and L is a loss function.

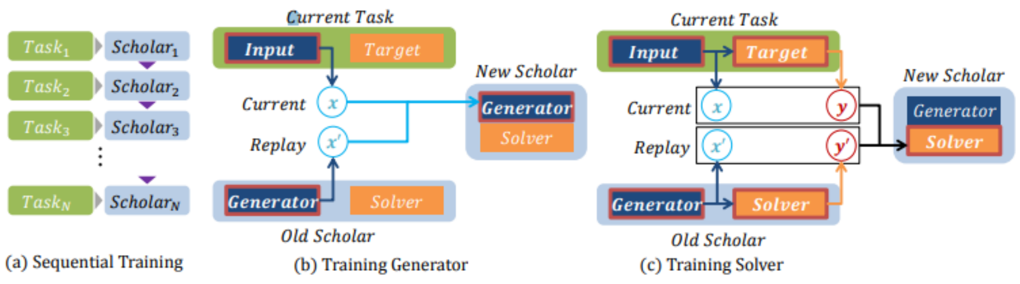


Figure 4: fig: the Generative Replay Training Process, figure from [2]

For each task, the generate replay algorithm trains a generator which learns to generate the learns to reconstruct cumulative input space from all previous tasks. For each new task, a new solver is also trained to couple the inputs and targets drawn from the same mix of real and replayed data. The figure above depicts the training process for the solver and the generator.
More formally, the loss function of the $i - th$ solver is the following:

$$L_{train}(\theta_i) = r\mathbb{E}_{(x,y)\sim D}[L(S(x;\theta_i), y)] + (1-r)\mathbb{E}_{x'\sim G_{i-1}}[L(S(x';\theta_i), S(x', \theta_{i-1}))]$$

Here r is the ratio of mixing real data with generated data. The test loss is defined slightly differently as the model is evaluated on original tasks:

$$L_{test}(\theta_i) = r\mathbb{E}_{(x,y)\sim D}[L(S(x;\theta_i), y)] + (1-r)\mathbb{E}_{(x,y)\sim D_{past}}[L(S(x';\theta_i), y)]$$

10 Appendix B: short descriptions of code files

agents: this is the directory that contains the continual learning models: iid, ewc, finetune and generative replay

histograms: this is the directory that contains histograms of accuracies for our models performed on pre-existing image test sets.

dataset.zip: this contains the original images used for training and testing.

steering_plots: the directory that contains the steering direction vs time plots for the tests on the go

Weights: the directory that contains the tuned parameters obtained from training

racer.py: this is the file that runs the models on the environment. We load the learned weights by giving it the path to the saved weights .

dataset_loader.py: it applies filters/gaussian noises to data images to generate different data sets.

driving_policy.py: this is the CNN+Feed Forward model. It gets used in **full_state_car_racing_env.py**.

car_racing.py: a top-down racing environment. It is used in **full_state_car_racing_env.py**.

full_state_car_racing_env.py: the environment for cars to drive autonomously.

main.py: we use this to train the models.

pid.py: this is the PID controller used in **full_state_car_racing_env.py** to get optimal expert steering direction.

results.txt: this is where we store the accuracies of model predictions on pre-existing image test sets.

task0/1/2_filter/gaussian.png: these are the sample images in datasets.

utils.py: helpers used to implement models.

draw_plots: we use this to plot graphs.