MARIODAGGER: A Time and Space Efficient Autonomous Driver

Farzad Kamrani†, Andreas Elers‡, Mika Cohen†, Amir H. Payberah‡
†FOI - Swedish Defence Research Agency, Sweden
‡KTH Royal Institute of Technology, Sweden
†{farzad.kamrani,mika.cohen}@foi.se ‡{elers,payberah}@kth.se

***Abstract*—Imitation learning is a promising approach for training autonomous vehicles, where a set of state-action pairs from human demonstrated driving is used as training data in a supervised learning manner. Dataset Aggregation (DAGGER) is a common imitation learning algorithm, in which models are trained by iteratively collecting new data, aggregating it with old data, and retraining the model on the entire collected dataset. Data aggregation and retraining, however, lead to two main problems: (i) large memory consumption, and (ii) long training time. In this work, we present a fast and memory-efficient algorithm, called MARIODAGGER, that improves DAGGER by resolving the aforementioned problems. Unlike DAGGER that requires a collection of old and new data to train the models, MARIODAGGER uses only the new data and a few samples from the old data stored in a rehearsal buffer, which is updated iteratively using reservoir sampling. To prevent forgetting old knowledge, MARIODAGGER uses a recent regularization technique Elastic Weight Consolidation. We evaluate and compare MARIODAGGER with SAFEDAGGER, a recent variant of DAGGER that MARIODAGGER builds upon, in the context of autonomous vehicles, and show that MARIODAGGER achieves the same performance as SAFEDAGGER in half as many iterations, using significantly less memory space.**

***Index Terms*—Imitation Learning, DAgger, SafeDAgger, Catastrophic Forgetting, Elastic Weight Consolidation, Continual Learning**

I. INTRODUCTION

Autonomous Vehicles (AV) have received considerable attention lately within the Machine Learning (ML) community. According to the Autonomous Vehicle Outlook report [1], the global AV market is projected to reach $556.67 billion by 2026. However, training AVs is still challenging and there is a lot of research in progress to improve it [2].

Early attempts to train AVs are mainly based on supervised learning [3], [4]. In this approach, the training data is a set of *state-action* pairs, where each *state* represents the information available to the AV in a given moment (e.g., images recorded by a camera ahead of the vehicle), and the corresponding *action* is the expert driver's response (e.g., the steering wheel angles). The actions, then, are used as the labels for the states (camera images) and learned by using a supervised learning method.

In supervised learning, it is assumed that the data samples are *independent and identically distributed (i.i.d.)*, that is, both training and testing data are drawn independently from the same distribution. However, such an assumption is not valid in AVs scenarios, where under testing, each state is dependent on prior actions (i.e., predictions of the model). Any predictor inevitably introduces errors and these *compounding errors* change the distribution of future inputs, breaking the train-test i.i.d. assumption [5]. For instance, if the model predicts steering slightly left in a right turn, then, the next action should be a sharp right to prevent the vehicle from going off-road. However, such an action may be rare (or non-existing) in the training data, as the training data is generated by an expert. These compounding errors lead to an increasing deviation between the distribution of the training data (expert's behavior) and test data (model's behavior), lowering the performance of the model [5].

Imitation Learning (IL) is a promising solution to resolve the aforementioned problem of compounding errors in AVs [6], [7], [8], [9]. IL is built upon supervised learning methods and its goal is to learn the behavior of an expert from demonstrations (e.g., by training a model using a set of recorded state-action pairs generated by a human driver) [10]. However, unlike supervised learning, IL does not assume the input data samples are i.i.d.. In many IL algorithms, building a model is not a one-shot procedure. To resolve differences in the train and test distributions, the training data is augmented in an interactive manner to meet the test distribution and as new data is acquired, the model is retrained. However, retraining a model using new data usually overwrites already learned parameters, which is known as *catastrophic forgetting* [11], [12]. To mitigate this problem, the model is retrained on a collection of both newly acquired and the original data.

Dataset Aggregation (DAGGER) [13] is a well-known IL algorithm that uses data aggregation and model retraining to reduce the problem of compounding errors. DAGGER starts by training an initial model using a supervised learning method and thereafter lets the model control the AV while generating new labels (e.g., steering angle) under oversight/control of the expert. This creates new data that contain information about how to recover from the errors of the model. The model is trained on the aggregated data (union of the initial data together with the newly generated data). This process can be performed iteratively until the desired behavior is reached. Nonetheless, DAGGER suffers from two problems: (i) storing both old and new data requires a large memory space, and (ii) retraining a model using the aggregated data takes much time.

Continual Learning (CL) [14] is another approach to avoid catastrophic forgetting that enables models to handle shifts in data samples distribution and to learn new tasks incrementally, without forgetting earlier tasks. *Elastic Weight Consolidation* (EWC) [15] is a CL algorithm that uses a regularization technique to protect parameters that are important for a task.

978-1-7281-8470-8/20/$31.00 ©2020 IEEE
DOI 10.1109/ICMLA51294.2020.00230

To the best of our knowledge none of the existing solutions for training AVs consider the implicit connection between CL and learning on aggregated data. If we consider the problem of training a model on a different distribution as a new task, the problem can conveniently be viewed as an instance of CL, whereby its techniques can be used to relax the need for access to previously gathered data.

In this work, we present MARIODAGGER, a novel IL algorithm based on DAGGER that makes use of EWC to overcome catastrophic forgetting. MARIODAGGER does not need to preserve the entire training data to solve forgetting: the learning is only based on new data, and a few samples from the old data stored in a rehearsal buffer, which is updated iteratively using reservoir sampling [16]. MARIODAGGER will provide more flexibility and the possibility to extend a model to handle additional tasks or skewed input distributions different from the data that the model has been trained on. It is also more scalable and memory-efficient than the original DAGGER method, since it allows preceding large datasets to be discarded.

We evaluated and compared MARIODAGGER with SAFEDAGGER [17] that MARIODAGGER builds upon. SAFEDAGGER is a recent variant of DAGGER that collects data more efficiently, and only retrains models with data that is deemed difficult. Through experiments, we showed that MARIODAGGER outperforms SAFEDAGGER by reaching the same performance in half as many iterations, and uses significantly less memory space.

II. BACKGROUND

In this section, we briefly recall some basic concepts from IL, describe two prominent IL algorithms, DAGGER and SAFEDAGGER, and, finally, explain EWC as a CL solution for catastrophic forgetting.

A. Preliminaries

To train AVs, learning algorithms should make a sequence of predictions over time, based on the AVs environment. The environment is a set of *states* S, such that in each state, a number of *actions* $A(S)$ are feasible. A state $s \in S$ of an AV generally includes the vehicle's position, direction angle, velocity, acceleration, etc., and an *action* $a \in A(S)$ that demonstrates how the AV's driver acts in response to the state to maintain the vehicle in a safe state. In this work, we apply an end-to-end learning approach, in which states are represented by images taken by a camera ahead of the AV, and actions are measured only by the steering wheel angle. At each point in time $t = 1, 2, \cdots, T$, the learning algorithm receives a state-action pair (s_t, a_t) as a training *sample*.

A *policy* π is a function $\pi : S \rightarrow A(S)$ that maps each state s_t to an action a_t. Any selected action a_t at a state s_t by a policy π leads the AV to the next state s_{t+1}. The *loss* value l_t shows how inaccurate an AV is at time t based on an action a_t. The loss value will be zero if the AV drives well; otherwise, it will be larger values (e.g., if it deviates from its lane or crashes). If we consider a *trajectory* τ as a set of triples $\tau = \{(s_1, a_1, l_1), (s_2, a_2, l_2), \cdots, (s_T, a_T, l_T)\}$, then, our goal is to find the *best policy*, denoted by $\hat{\pi}$, such that it minimizes the expected loss over all states in τ:

$$\hat{\pi} = \arg\min_{\pi} \mathbb{E}_{\tau \sim \pi}[\sum_{t=1}^{T} l_t]. \quad (1)$$

We use neural networks to implement the policies, thus finding a policy means finding the network's parameters.

B. Supervised Learning

Assume there is an expert who knows how to drive, and the *expert policy* π^* is the policy used by the expert to choose actions in different states. One approach to satisfy Equation 1 is to model the problem as a supervised regression, inferring a policy model from the observed behavior of the expert. If we have n trajectories $\tau_1, \tau_2, \cdots, \tau_n$ generated by the expert policy π^*, then, we can define the training set D_{π^*} as follows: $D_{\pi^*} = \{(s, a) : \forall n, \forall (s, a, l) \in \tau_n\}$.

The set of states in D_{π^*} are called *reachable states* by π^* and is denoted by S_{π^*}. To build a regression model, we define the loss function $\mathcal{L}$ as the Mean Squared Error (MSE) of the selected actions by π and π^* in each state s_i (Equation 2). Hence, the MSE is the squared difference between the predicted steering wheel angle $\pi(s_i)$ and the ground truth given by expert $\pi^*(s_i)$. If $m = |D_{\pi^*}|$ is the total number of samples in the training set D_{π^*}, we have:

$$\mathcal{L}_{D_{\pi^*}}(\pi, \pi^*) = \frac{1}{m} \sum_{i=1}^{m} (\pi(s_i) - \pi^*(s_i))^2. \quad (2)$$

Considering this loss function, we can rewrite Equation 1 and define the best policy $\hat{\pi}$ as below:

$$\hat{\pi} = \arg\min_{\pi} \mathcal{L}_{D_{\pi^*}}(\pi, \pi^*). \quad (3)$$

The supervised learning approach assumes that the training and test data samples are i.i.d., but this assumption can be violated in AV settings, since the training data is generated by the expert policy π^* and the test data is generated by a learned policy π, so they are (in general) sampled from different distributions. It may happen that due to an action taken by π, the AV is led to a state s', which has not been visited by the expert (i.e., $s' \notin S_{\pi^*}$). The behavior of the learned model π, in the subsequent states that are not necessarily among the reachable states by S_{π^*}, is then generally unpredictable [18]. Therefore, applying standard supervised learning methods to train AVs often leads to poor performance.

C. Dataset Aggregation

Imitation Learning (IL) is one way to address the aforementioned problems. In IL, we try to learn a policy π by mimicking the expert policy π^*. DAGGER [13] is a well-established IL algorithm that iteratively collects new data, appends it to the previous data, and then retrains the model from scratch on all the data.

Algorithm 1 shows how DAGGER works. The initial training data D_{π^*} is generated by the expert, i.e., by running the

expert policy π^*, and based on it, the first policy $\hat{\pi}_0$ is trained using standard supervised learning (Equation 3). Then, in each iteration i, a new policy π_i is created based on the policy from the previous iteration $\hat{\pi}_{i-1}$, and expert policy π^* (Line 6). In Lines 7 and 8, a new dataset is generated using the new policy π_i, and appended to the old data. Finally, the policy of each iteration i is trained according to Equation 3 (Line 9). M in Line 5 is the number of iterations to train the policies, and β in Line 6 defines how much the expert is allowed to control the vehicle and correct the trajectory.

Algorithm 1 DAGGER

```
1:  procedure DAGGER(π*)
2:      Collect D_π* using π*
3:      D_0 = D_π*
4:      π̂_0 = arg min_π L_D0(π, π*)
5:      for i=1...M do
6:          π_i = β_i π* + (1 − β_i) π̂_{i−1}
7:          Collect D' using π_i
8:          D_i ← D' ∪ D_{i−1}
9:          π̂_i = arg min_π L_Di(π, π*)
10:     return π̂_M
```

DAGGER is an expensive algorithm because it queries the expert (calls the expert policy π^*) in all the collected states in each iteration i (Line 6 in Algorithm 1). SAFEDAGGER [17] is a variant of DAGGER that minimizes the number of queries to the expert policy. SAFEDAGGER uses a policy $\hat{\pi}$ that drives the vehicle as in DAGGER, but it uses a *safety classifier* c_{safe} to determine if a prediction by $\hat{\pi}_i$ in state s_i is good enough. If $c_{safe}(\hat{\pi}_i, s_i) = 1$, then the prediction is classified as good enough, and the policy $\hat{\pi}_i$ controls the vehicle, otherwise, the action returned by the expert policy π^* is used. In this way, it is assumed, SAFEDAGGER collects additional data only for difficult states that cause the model to fail at its task, e.g., a sharp turn that causes a trained model to drive off the road.

The pseudocode for SAFEDAGGER is shown in Algorithm 2. Lines 1-6 use the expert policy π^* to collect the initial training data D_{π^*} along with the safe data D_{safe}, and also to train an initial policy $\hat{\pi}_0$ as well as an initial safety model $c_{safe,0}$. The safety strategy at Line 8 is the process of giving control to the expert policy π^* when the control model $\hat{\pi}_i$ cannot drive safely, and Line 9 reduces the number of queries to the expert policy. Lines 10-12 aggregate data and update policies $\hat{\pi}_i$ and $c_{safe,i}$ by retraining them on the aggregated data. For more details about SAFEDAGGER please refer to [17].

Algorithm 2 SAFEDAGGER

```
1:  procedure SAFEDAGGER(π*)
2:      Collect D_π* using π*
3:      Collect D_safe using π*
4:      D_0 = D_π*
5:      π̂_0 = arg min_π L_D0(π, π*)
6:      c_safe,0 = arg min_c_safe L_{D0 ∪ Dsafe}(π̂_0, π*, c_safe)
7:      for i=1...M do
8:          Collect D' using safe strategy π̂_{i−1} and c_safe,i−1
9:          Subset selection: D' ← {(s, π*(s)) ∈ D' | c_safe,i−1(π̂_{i−1}, s) = 0}
10:         D_i ← D' ∪ D_{i−1}
11:         π̂_i = arg min_π L_Di(π, π*).
12:         c_safe,i = arg min_c_cafe L_{Di ∪ Dsafe}(π̂_i, π*, c_safe)
13:     return π̂_M and c_safe,M
```

D. Elastic Weight Consolidation

Neural networks that incrementally learn multiple tasks face the challenge of *catastrophic forgetting*: learning a new task can completely erase the previously learned knowledge [11], [19], or drastically reduces the performance of the model on previously learned tasks [11]. This is mainly due to the use of a single set of parameters θ in the network to learn mappings from input to output. As a consequence, a preceding task's mapping will suffer as the parameters are tuned for a succeeding task's mapping. This issue is generalized by the *stability-plasticity dilemma* [20], i.e., when learning new tasks, parameters should be stable enough to retain previous knowledge, but also plastic enough to learn new knowledge.

Continual Learning (CL) is a field within ML that aims at addressing the catastrophic forgetting problem using different techniques. *Elastic Weight Consolidation* (EWC) is a recent CL algorithm that has shown promising results in learning tasks incrementally without encountering catastrophic forgetting [15]. EWC is essentially a regularization technique that protects important parameters for each task by reducing their plasticity, ensuring that predominantly non-important parameters are modified during training. EWC assumes that for each task there are multiple parameter configurations for neural networks that deliver good performance.

Figure 1 illustrates schematically the intuition behind the EWC. The green and blue areas illustrate low error regions for tasks A and B, respectively, in a two-dimensional parameter space. Training a neural network on a task A, in practice means finding point θ^*_A (adjusting the network parameters successively to reach this point). If a network that is trained on task A is trained on a new task B, the optimization algorithm moves the parameters in the direction of the red arrow to point θ^*_B, which is optimized for task B at the cost of forgetting task A. The EWC algorithm tries to keep the parameter in a safe trajectory by penalizing movements in unsafe directions (by using *Fisher information matrix* [21]), and guides the optimization algorithm to move along the green arrow and toward point $\theta_{A,B}$, which is located in the intersection of areas with low error for both tasks A and B.

EWC uses the Fisher information matrix to measure parameters' importance for each learned task [21]. By saving the network's parameters after each training, it is possible to measure the deviation of the network's new parameters from the previous values. EWC calculates the Fisher information matrix by using a set of samples from previous tasks.

Equation 4 shows the EWC loss function of a model, which is trained on two tasks A and B, sequentially,

$$\mathcal{L}_{EWC}(\theta) = \mathcal{L}_B(\theta) + \sum_i \frac{\lambda}{2} F_i(\theta_i - \theta^*_{A,i})^2, \qquad (4)$$

where $\mathcal{L}_B(\theta)$ is the regular loss for task B (Equation 2). The hyperparameter λ signifies how important the old task A is compared to the new task B. F is the Fisher information matrix, θ_i is the network's current parameters, and $\theta^*_{A,i}$ is the set of parameters extracted previously by training the network

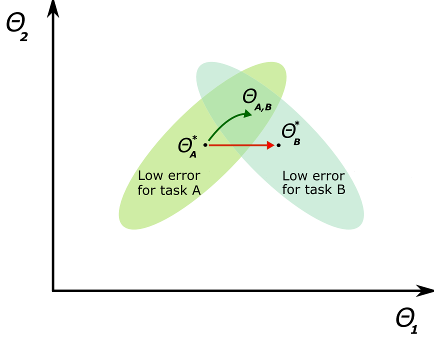


Fig. 1. The intuition behind EWC's imposed constraints. The green and blue areas illustrate low error regions for tasks A and B, respectively, in a parameter space. Training a network that is optimized on task A, with parameters at θ^*_A, on a new task B, moves the parameters in the direction of the red arrow to point θ^*_B and forgetting task A. EWC guides the optimization algorithm to move in a safe direction along the green arrow and toward point $\theta_{A,B}$.

on task A (i.e., parameters that minimize loss function $\mathcal{L}_A(\theta)$).

III. Our Algorithm (MarioDAgger)

As explained in Section II, DAgger is an IL algorithm that iteratively collects data, and trains the model using the entire dataset (newly collected, as well as old data). SafeDAgger offers a way of reducing the amount of collected data in each iteration. Although SafeDAgger collects less data than DAgger, it still needs to incrementally add data, and train the model using the entire dataset. To overcome this problem, we propose MarioDAgger, that similar to SafeDAgger, uses the idea of the safety model, but instead of training a model over all gathered data, it trains the model only on the latest collected data in each iteration, and a few samples of old data stored in the *rehearsal buffer*.

The rehearsal buffer is a small buffer that keeps a few random samples from all the previous iterations and is updated in every iteration. We use *reservoir sampling* [16] to update the rehearsal buffer. For a buffer with size k, the reservoir sampling simply stores the first k samples in the buffer. When the buffer is full, and the n-th sample arrives, the sample is stored in the buffer with the probability k/n. If a new sample is added to a full buffer, then, an item, uniformly selected at random, is evicted from the buffer.

Since MarioDAgger uses only a small part of the old data to train the model, it relies on EWC to protect the model against catastrophic forgetting. We define a *task* as driving on a track, and define the *task boundary* as driving along the entire track once. We use the task boundaries in EWC to compute the Fisher information matrix for tasks and to copy the model's parameters from previous ones. EWC uses this information to measure the deviation from the saved parameters for tasks thus far, and their importance to constrain the learning of new ones, and also to find a set of parameters similar to previous tasks' parameters.

It should be noted that although we keep a set of samples from previous tasks to calculate the Fisher information matrix, we do not use these samples while training the models. The changes between MarioDAgger compared to SafeDAgger are motivated by the fact that EWC enables a model to learn shifts in input distributions yet remain performant on previous input distributions. Thus, applying EWC to SafeDAgger is suitable to remove the need of saving previous data, lowering training time and memory requirements.

Algorithm 3 illustrates how MarioDAgger works. The initial dataset is collected to train the first policy $\hat{\pi}_0$, in Line 7 the Fisher matrix for the initial iteration is calculated, and in Line 8 the model parameters are stored. Moreover, the rehearsal buffer is initiated with a few samples of the first round using reservoir sampling in Line 10. Afterward, the policy is deployed over iterations, and in each iteration, it collects additional data where it fails (Lines 12-13). As seen in Line 14, the training data D_i of iteration i consists of new data collected in this iteration and the samples in the rehearsal buffer. It is a significant difference compared to both DAgger and SafeDAgger.

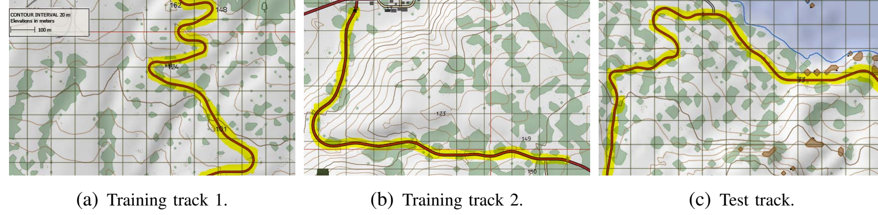
The model, then, is retrained in Line 15 to make it succeed in states where it has previously failed. At this stage, we use EWC to prevent the model from forgetting previous knowledge. Line 17 shows where the Fisher information matrix for the new task is calculated. Line 18 is where the parameters of the model are saved after training. These values are used in the EWC constraint during succeeding training iterations. Both Fisher information matrices and the saved parameters are stored in lists and the new results are appended to the corresponding lists. At the end of the iteration (Line 19), the rehearsal buffer is updated using reservoir sampling with a few samples collected in this iteration.

Algorithm 3 MarioDAgger

1: **procedure** MarioDAgger(π^*)
2: Collect D_{π^*} using π^*
3: Collect D_{safe} using π^*
4: $D_0 = D_{\pi^*}$
5: $\hat{\pi}_0 = \arg\min_\pi \mathcal{L}_{D_0}(\pi, \pi^*)$
6: $c_{safe,0} = \arg\min_{c_{safe}} \mathcal{L}_{D_0 \cup D_{safe}}(\hat{\pi}_0, \pi^*, c_{safe})$
7: F = calculate Fisher information matrix from D_0
8: θ^* = parameters of $\hat{\pi}_0$
9: $R \leftarrow \emptyset$
10: $R \leftarrow reservoirSampling(R, D_0)$
11: **for** i=1...M **do**
12: Collect D' using safe strategy $\hat{\pi}_{i-1}$ and $c_{safe,i-1}$
13: Subset selection: $D'' \leftarrow \{(s, \pi^*(s)) \in D' | c_{safe,i-1}(\hat{\pi}_{i-1}, s) = 0\}$
14: $D_i \leftarrow D'' \cup R$
15: $\hat{\pi}_i = \arg\min_\pi \mathcal{L}^{EWC}_{D_i}(\pi, \pi^*, F, \theta^*)$
16: $c_{safe,i} = \arg\min_{c_{cafe}} \mathcal{L}_{D_i \cup D_{safe}}(\hat{\pi}_i, \pi^*, c_{safe})$
17: F = calculate Fisher information matrix with data saved from D_{i-1}, and append the result to list of Fisher matrices
18: θ^* = append parameters of $\hat{\pi}_i$ to list of parameters
19: $R \leftarrow reservoirSampling(R, D'')$
20: **return** $\hat{\pi}_M$ and $c_{safe,M}$

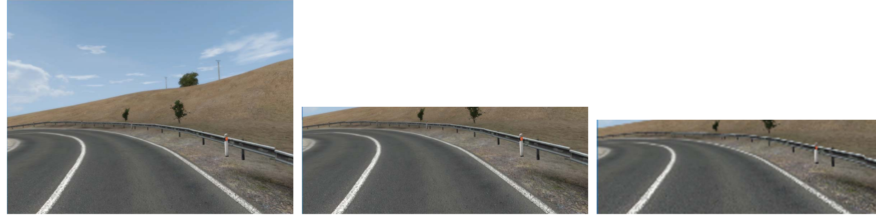
IV. Experiments

We deploy our models in the VBS3 simulator [22], a virtual training environment that facilitates collecting data, extracting metrics, and evaluating trained models. We define two metrics to evaluate the models:



(a) Training track 1. (b) Training track 2. (c) Test track.

Fig. 2. Bird's eye view of the training tracks (a) and (b), and the test track (c). Roads are the red lines, and the used roads are highlighted in yellow.



(a) Original image, 800x600 pixels. (b) After cropping, 800x300 pixels. (c) Downsampled to 200x66 pixels.

Fig. 3. (a) and (b) are the image before and after cropping, and (c) is the image after downsampling to a lower resolution.

1) The driven distance until the AV leaves its lane. This metric shows how far and how well an AV drives.
2) A true/false value indicating whether an AV manages to finish a track or not. Through this metric, we can also study if a model based on a policy π is monotonically improving or deteriorating, e.g., it finishes a training track in one iteration, but fails on the same track during the next iterations.

We studied the performance of four models in the experiments:

1) Naive model: the model is trained only on new data (i.e., the samples collected by the AV during the latest iteration).
2) SAFEDAGGER: the model is trained on both old and new data, which is cumulatively collected over iterations.
3) MARIODAGGER without a rehearsal buffer: similar to SAFEDAGGER, but instead of training the model over all recorded data in different iterations, it is trained only on the new data collected during the latest iteration. However, it uses EWC to protect against catastrophic forgetting.
4) MARIODAGGER: it uses the new data and a few random samples of the old data, stored in the rehearsal buffer, for training. It also uses EWC to prevent forgetting.

In the rest of this section, we first explain how we collect data to train the models, then briefly present the implementation, and finally, we show the results.

A. Data

The data is collected by driving a vehicle on roads in the VBS3 simulator with a fixed velocity of $20km/h$. The speed limit is imposed to enable the human expert to provide good labels. The weather is sunny, all roads are paved, and the view is unobstructed. The frames are recorded at a rate of $5Hz$ with 800×600 resolution and are saved alongside the corresponding angle of the steering wheel. We use two tracks for training, and one track for testing. The first training track is 1925 meters, and the second one is 1448 meters, and the test track is 1939 meters. Figure 2 depicts these tracks.

The initial dataset to train the models is collected by manually driving a vehicle (i.e., using the expert policy π^*) through the first training track (Figure 2(a)). This dataset consists of 901 samples in the form of $(\texttt{image}, \texttt{steering_angle})$, where the image represents the features, and the steering angle is the outcome variable. We use this data to train the initial policies in all the models (i.e., $\hat{\pi}_0$ in Algorithms 2 and 3). From the second iteration, we use the data collected by each model separately over iterations to train them. The number of collected samples for each iteration varies between 418 to 491 samples, which depends on how far the AV drives to finish the tracks. The test track is only used for testing the trained models, and we do not collect any data for training from this track. We evaluate the models until they either finish all tracks, or they exceed the maximum number of iterations (10 iterations in our experiments).

We encounter two situations during the data collection, either: (i) a model fails on one or both training tracks, or (ii) a model finishes both training tracks, but not the test track. In the first situation (i), we deploy the failed model on the track it fails, and a human expert provides correcting actions. We, then, record the data accordingly. In the second situation (ii), we do not use any correcting actions, and instead, we record data from environments in the training tracks that are very similar to failing environments in the test track. The recorded data in both cases represent difficult situations as the models fail. A state is *difficult* if a model encounters a state that it has barely or not at all been trained on such as a position where the vehicle is drifting into the other lane, something the human expert would never do. For each iteration, each model runs three times per track to evaluate the performance. Multiple

runs are necessary as the results vary slightly. Three runs per track are chosen as a trade-off between statistical validity and time cost. A model is allowed to drive until it either reaches the target, goes off road, or veers into the opposite lane.

B. Implementation

Before feeding the data into the models, we preprocess them in two steps to remove redundant information, and to reduce the training time. The collected images are first cropped to remove unnecessary sections and then downsampled to a lower resolution. As the road is the only important section in each image, cropping away everything slightly above the horizon creates images without redundant information. Downsampling the images to a lower resolution of 200×66 pixels further decreases the computational cost of training on the dataset. Figure 3 shows how different data preprocessing steps affect an example image from the dataset.

To process the collected images of the tracks, we implemented our policy as a convolutional neural network (CNN) based on an altered version of Nvidia's architecture for AVs [23], however, any other appropriate architecture can be used here. Figure 4 depicts the architecture of the implemented CNN. The neural network consists of four convolutional layers, three fully connected layers, and a single output giving the steering angle. To prevent overfitting, we use dropout between each of the fully connected layers. We also use l_2 regularization to improve the model generalization. The models are given images of 200×66 pixels as input, and predict the steering angle as its output. Table I shows more details about the neural network configuration and the models' hyperparameters. The models were developed using TensorFlow [24].

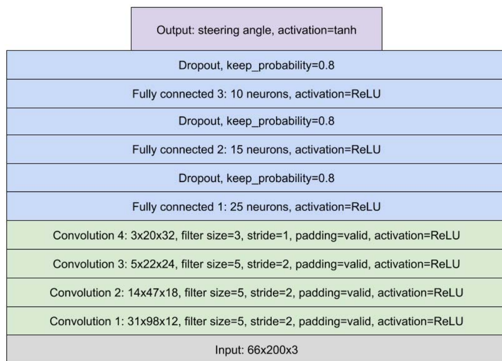


Fig. 4. The neural network architecture consisting of four convolutional layers, three fully connected layers, and one output.

C. Results

We start the result section by showing the performance of the naive approach, which is trained on new data without using EWC. Table II shows that the naive approach cannot maintain its performance after the initial training on the first track, and its performance drops due to catastrophically forgetting earlier knowledge. The model mainly fails; that is, the AV almost instantly drives off the road (with some exceptions for the third iteration). The naive model is only tested for four iterations as it is clear that it does not improve.

TABLE I
HYPERPARAMETERS

Optimizer = Adam
Learning rate = 0.0001
Batch size = 50
Dropout keep probability = 0.8
L2-regularization = 0.0001
λ in EWC (Equation 4) = 5000
Number of Fisher samples = 80
Rehearsal buffer size = 23

TABLE II
EMPIRICAL RESULTS OF NAIVE MODEL

	Training track 1		Training track 2		Test track	
	Distance driven (m)	Training track 1 finished	Distance driven (m)	Training track 2 finished	Distance driven (m)	Test track finished
Iteration 1						
Run 1	1927	yes	849.8	no	1462.4	no
Run 2	1926	yes	384.3	no	1458.7	no
Run 3	1929.1	yes	834	no	1455.8	no
Iteration 2						
Run 1	8.8	no	11	no	12	no
Run 2	8.7	no	11	no	11.9	no
Run 3	8.8	no	11	no	12	no
Iteration 3						
Run 1	16.8	no	308.2	no	181.3	no
Run 2	14.5	no	832.7	no	181.7	no
Run 3	15.6	no	823.8	no	181.6	no
Iteration 4						
Run 1	9.3	no	12.1	no	13.4	no
Run 2	9.3	no	12.1	no	13.4	no
Run 3	9.3	no	12.1	no	13.2	no

As Table III shows, SAFEDAGGER, which is trained on all data, improves its performance in each iteration except for the third iteration. In the third iteration, the model fails after 204 meters on the test track, which corresponds to a sharp right turn. In the next iteration (fourth), the model manages to finish all tracks. These results highlight that models trained with SAFEDAGGER do not improve monotonically as iteration two performs better than iteration three.

MARIODAGGER without the rehearsal buffer, which is trained only on new data with EWC, shows that it can resist catastrophic forgetting and improves its performance as the number of iterations increases (Table IV). However, MARIODAGGER without buffer requires more iterations compared to SAFEDAGGER to complete the test track, and it only manages to complete the test track in two out of three runs (due to the lack of space, we only show the iterations 1, 2, 9, and 10 in Table IV). As comparison, SAFEDAGGER requires four iterations before it completes all the test track in all runs. Notably, the performance of the MARIODAGGER without buffer seems to degrade severely between iterations one and two, although the model regains its performance in the following iterations. Nevertheless, it should be mentioned that the amount of data we use to train MARIODAGGER without buffer is much less than SAFEDAGGER, since the former uses only the new data, while the latter uses the whole old and new data. Therefore, although MARIODAGGER without buffer needs more iterations to learn, it uses less memory. Hence, it can be used in more memory-intensive cases, e.g., longer

TABLE III
EMPIRICAL RESULTS OF SAFEDAGGER

	Training track 1		Training track 2		Test track	
	Distance driven (m)	Training track 1 finished	Distance driven (m)	Training track 2 finished	Distance driven (m)	Test track finished
Iteration 1						
Run 1	1929.4	yes	1448.4	yes	502.7	no
Run 2	1929	yes	1448.1	yes	1888.1	no
Run 3	1930.2	yes	1448.1	yes	1472.9	no
Iteration 2						
Run 1	1927.1	yes	1449.6	yes	1552.6	no
Run 2	1926.6	yes	1447.1	yes	1941.9	yes
Run 3	1927.8	yes	1449.6	yes	621.3	no
Iteration 3						
Run 1	1927.3	yes	1448.1	yes	204.4	no
Run 2	1929.8	yes	1449.6	yes	204.5	no
Run 3	1928.4	yes	1449.2	yes	204.4	no
Iteration 4						
Run 1	1928.4	yes	1446.5	yes	1941.6	yes
Run 2	1927.1	yes	1447.5	yes	1940.9	yes
Run 3	1926.8	yes	1448	yes	1939.9	yes

TABLE IV
EMPIRICAL RESULTS OF MARIODAGGER WITHOUT BUFFER

	Training track 1		Training track 2		Test track	
	Distance driven (m)	Training track 1 finished	Distance driven (m)	Training track 2 finished	Distance driven (m)	Test track finished
Iteration 1						
Run 1	500.6	no	1448.3	yes	619.8	no
Run 2	501	no	1446.1	yes	620.3	no
Run 3	501.7	no	1446.9	yes	1259.6	no
Iteration 2						
Run 1	38	no	49.7	no	192.4	no
Run 2	37.9	no	49.5	no	106.8	no
Run 3	37.7	no	51.5	no	183.7	no
Iterations 3, 4, ..., 8						
Iteration 9						
Run 1	1924.4	yes	1444.8	yes	617.3	no
Run 2	1924	yes	1447.2	yes	607	no
Run 3	1925.7	yes	1445.1	yes	1372.9	no
Iteration 10						
Run 1	1923.9	yes	1446.8	yes	1938	yes
Run 2	1925.9	yes	1446.7	yes	1174.2	no
Run 3	1923.6	yes	1447.9	yes	1939	yes

TABLE V
EMPIRICAL RESULTS OF MARIODAGGER

	Training track 1		Training track 2		Test track	
	Distance driven (m)	Training track 1 finished	Distance driven (m)	Training track 2 finished	Distance driven (m)	Test track finished
Iteration 1						
Run 1	1929.2	yes	834.4	no	780.9	no
Run 2	1929	yes	834.2	no	1175.2	no
Run 3	1928.9	yes	832.9	no	1176.9	no
Iteration 2						
Run 1	1928.7	yes	1445.6	yes	1939.9	yes
Run 2	1926.3	yes	1445.5	yes	1939.1	yes
Run 3	1927.2	yes	1445.5	yes	1938.1	yes

tracks, or more frequent image capturing of the roads.

In the last experiment, we test MARIODAGGER, where we set the size of the rehearsal buffer to 23, which means that we inject 23 extra samples into the training data compared to MARIODAGGER without buffer. Table V shows that this approach has the best performance. This model manages to finish the test track in all three runs in iteration two. That is two iterations less than the SAFEDAGGER and eight iterations less than MARIODAGGER without buffer. It confirms that rehearsal is a highly useful technique with low cost, as 23 data samples correspond to approximately 5% of the training data used per iteration (the number of data samples in each iteration varies between 418 and 491). Interestingly, another study [25] also noted that rehearsing on 5% of old data together with EWC gives a significant increase in performance.

V. RELATED WORK

MARIODAGGER makes use of techniques from two relatively separate avenues of research within ML, namely Imitation Learning (IL) and Continual Learning (CL), and is naturally related to several works in these two areas.

The DAGGER algorithm [13] deals with the issue of compounding errors in sequential predictions by iteratively querying an expert for more data and retraining the model on past and new data combined. The superiority of DAGGER to other IL techniques such as SMILe [26] and SEARN [27] has been shown for different tasks, Super Tux Cart, Super Mario Bros [28] and handwriting recognition [13]. SAFEDAGGER [17] is an improvement upon DAGGER that aims to reduce the amount of correcting actions needed from the human expert, thus making the method less costly. SAFEDAGGER is evaluated via an autonomous driving scenario in TORCS [29], where it is shown that SAFEDAGGER reduces the number of actions needed by the human expert. Moreover, SAFEDAGGER trains a model faster and with less data compared to DAGGER, while achieving fewer crashes and less damage per driven lap.

To overcome catastrophic forgetting Kirkpatrick et al. present the EWC algorithm [15] as a regularization technique that protects tasks' important parameters by reducing their plasticity. EWC relies on there being multiple parameter configurations for neural networks that give good performance [30], [31], thus, it is possible to find a set of parameters for a new task, where the old task's important parameters are largely unchanged. EWC is evaluated with the permuted MNIST dataset [32], a common CL-benchmark in which pixels are permuted while labels are kept unchanged. The results show that a neural network can retain knowledge and perform well on multiple tasks when trained on tasks sequentially. However, the permuted MNIST test is criticized for giving unrealistically good results [33].

Rebuffi et al. present iCaRL [34] for learning tasks incrementally while recording a small set of examples for each class. iCaRL uses these sets to classify new data through nearest-mean-of-exemplars and to reduce catastrophic forgetting through rehearsal. The representation is updated by using a loss function combining classification and distillation loss. The results show that iCaRL performs better than the compared methods and that its accuracy is not biased towards recently learned classes as other methods are [35].

This work differs from the related work in the following ways. MARIODAGGER differs from DAGGER and SAFEDAGGER as it uses EWC to maintain previously learned knowledge while training only on the latest collected data instead of retraining on the union of old and new data. The work does not alter EWC, but evaluates EWC in a more realistic context of autonomous driving with shifting input distributions instead of the criticized permuted MNIST data test. MARIODAGGER takes the idea of a rehearsal buffer containing data from earlier tasks to investigate whether it can give a significant performance improvement with EWC.

VI. Conclusion

In this work, we present MarioDAgger, a novel IL algorithm that takes advantage of EWC, a CL algorithm, to overcome some of the challenges of the existing IL solutions, including (i) requiring large memory space to keep data (both new and historical data), and (ii) slow training, as they use all previously collected data to train the models. MarioDAgger only uses new data and a few random samples of the old data (around 5% of the new data) to train the model, and uses EWC to prevent forgetting the previous knowledge. In our experiments, MarioDAgger outperforms SafeDAgger, an enhanced implementation of DAgger by achieving the same results in half as many iterations, and using significantly less memory space.

Although the achieved results are promising, additional research is needed to verify them in more detail as there are multiple sources of uncertainty that may affect the outcome: the models are trained on different data and even on slightly different amounts of data and there may be varying degrees of human error during data collection. Moreover, the results raise further questions, such as the impact of the buffer size, and the buffer eviction policies, e.g., first-in, first-out, or random. These are left to future work.

Acknowledgments

This paper is based on the Master's thesis [36] of the second author. The work was supported by the FOI research project *Synthetic Actors*, which is funded by the R&D program of the Swedish Armed Forces.

References

[1] "Autonomous vehicle market outlook - 2026," AV-Market, accessed: 2020-04-30. [Online]. Available: https://www.alliedmarketresearch.com/autonomous-vehicle-market

[2] S. Grigorescu et al., "A survey of deep learning techniques for autonomous driving," *Journal of Field Robotics*, 2019.

[3] D. Pomerleau, "Alvinn: An autonomous land vehicle in a neural network," in *Advances in neural information processing systems*, 1989, pp. 305–313.

[4] Z. Chen at al., "End-to-end learning for lane keeping of self-driving cars," in *Intelligent Vehicles Symposium (IV)*. IEEE, 2017, pp. 1856–1860.

[5] A. Venkatraman et al., "Improving multi-step prediction of learned time series models," in *AAAI Conference on Artificial Intelligence*. AAAI Press, 2015, pp. 3024–3030.

[6] E. Rehder et al., "Driving like a human: Imitation learning for path planning using convolutional neural networks," in *International Conference on Robotics and Automation Workshops*, 2017.

[7] L. Sun et al., "A fast integrated planning and control framework for autonomous driving via imitation learning," in *Dynamic Systems and Control Conference*, vol. 51913. American Society of Mechanical Engineers, 2018, p. V003T37A012.

[8] S. Grigorescu et al., "Neurotrajectory: A neuroevolutionary approach to local state trajectory learning for autonomous vehicles," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 3441–3448, 2019.

[9] W. Schwarting et al., "Planning and decision-making for autonomous vehicles," *Annual Review of Control, Robotics, and Autonomous Systems*, 2018.

[10] I. Bratko et al., "Behavioural cloning: phenomena, results and problems," *IFAC Proceedings Volumes*, vol. 28, no. 21, pp. 143–149, 1995.

[11] M. McCloskey et al., "Catastrophic interference in connectionist networks: The sequential learning problem," in *Psychology of learning and motivation*. Elsevier, 1989, vol. 24, pp. 109–165.

[12] R. French, "Catastrophic forgetting in connectionist networks," *Trends in cognitive sciences*, vol. 3, no. 4, pp. 128–135, 1999.

[13] S. Ross et al., "A reduction of imitation learning and structured prediction to no-regret online learning," in *International Conference on Artificial Intelligence and Statistics*, 2011, pp. 627–635.

[14] N. Díaz-Rodríguez et al., "Don't forget, there is more than forgetting: new metrics for continual learning," *arXiv preprint arXiv:1810.13166*, 2018.

[15] J. Kirkpatrick et al., "Overcoming catastrophic forgetting in neural networks," *National Academy of Sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[16] K. Li, "Reservoir-sampling algorithms of time complexity o (n (1+ log (n/n)))," *ACM Transactions on Mathematical Software (TOMS)*, vol. 20, no. 4, pp. 481–493, 1994.

[17] J. Zhang et al., "Query-efficient imitation learning for end-to-end simulated driving," in *AAAI Conference on Artificial Intelligence*. AAAI Press, 2017, pp. 2891–2897.

[18] K. Asadi et al., "Lipschitz continuity in model-based reinforcement learning," *arXiv preprint arXiv:1804.07193*, 2018.

[19] R. Ratcliff, "Connectionist models of recognition memory: constraints imposed by learning and forgetting functions." *Psychological review*, vol. 97, no. 2, p. 285, 1990.

[20] M. Mermillod et al., "The stability-plasticity dilemma: Investigating the continuum from catastrophic forgetting to age-limited learning effects," *Frontiers in psychology*, vol. 4, p. 504, 2013.

[21] A. Ly et al., "A tutorial on fisher information," *Journal of Mathematical Psychology*, vol. 80, pp. 40–55, 2017.

[22] "VBS3," Bohemia Interactive Simulations, accessed: 2020-05-11. [Online]. Available: https://bisimulations.com/products/vbs3

[23] "End-to-end deep learning for self-driving cars," Nvidia, accessed: 2020-03-21. [Online]. Available: https://devblogs.nvidia.com/deep-learning-self-driving-cars/

[24] "An end-to-end open source machine learning platform," TensorFlow, accessed: 2020-05-11. [Online]. Available: https://www.tensorflow.org

[25] F. Kamrani et al., "Lagom intelligenta datorgenererade styrkor," FOI, 2018, FOI Memo 6587.

[26] S. Ross et al., "Efficient reductions for imitation learning," in *International Conference on Artificial Intelligence and Statistics*, 2010, pp. 661–668.

[27] H. Daumé et al., "Search-based structured prediction," *Machine learning*, vol. 75, no. 3, pp. 297–325, 2009.

[28] S. Ross et al., "No-regret reductions for imitation learning and structured prediction," in *In AISTATS*. Citeseer, 2011.

[29] "The open racing car simulator," TORCS, accessed: 2019-04-05. [Online]. Available: http://torcs.sourceforge.net/

[30] R. Nielsen, "Theory of the backpropagation neural network," *International Joint Conference on Neural Networks*, vol. volume I, p. pages 593–605, 1989.

[31] H. Sussmann, "Uniqueness of the weights for minimal feedforward nets with a given input-output map," *Neural networks*, vol. 5, no. 4, pp. 589–593, 1992.

[32] Y. LeCun et al., "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[33] R. Kemker et al., "Measuring catastrophic forgetting in neural networks," in *AAAI Conference on Artificial Intelligence*, 2018.

[34] S. Rebuffi et al., "icarl: Incremental classifier and representation learning," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 2001–2010.

[35] Z. Li et al., "Learning without forgetting," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 12, pp. 2935–2947, 2018.

[36] A. Elers, "Continual imitation learning: Enhancing safe data set aggregation with elastic weight consolidation," Master's thesis, KTH Royal Institute of Technology, 2019.