Designing Intelligent Drivers: Reinforcement Learning in a Racing Simulation

Albert Haladay
University of Nottingham

***Abstract*—This project investigates the performance of different experience-based agents in The Open RaceCar Simulation (TORCS), using OpenAI gym. Three agents were implemented in total, an agent with no memory, an agent that uses Q-learning, and an agent that uses deep Q-learning. The agent with no memory successfully raced, performing well, with a mean lap time of 92.2 ± 3.1 and a mean distance raced of 4303.6 ± 3748.7. The Q-learning model had the best results, with a mean lap time of 89.3 ± 3.0 and a mean distance raced of $10,346.4 \pm 7151.0$. In contrast, the deep Q-learning model was unsuccessful at performing the task, likely due to catastrophic forgetting.**

I. Introduction

All the Python code for this project can be found at https://github.com/UniformSoup/RacingAgents.

The purpose of this project is to evaluate the performance of different reinforcement learning techniques on an agent in a racing environment. There is considerable of motivation to investigate these types of autonomous driving agents, as self-driving vehicles have the potential to revolutionise transportation [1], and are becoming increasingly prevalent in today's society.

Investigating the performance of different models on driving agents will also help to understand which learning approach is the most suitable for evolving environments, like a racing simulation. This project's objective was to design three autonomous agents, each with different learning techniques, and then train them to use their past experiences to make autonomous decisions. Then after the agents have been trained, statistics about their ability to race will be collected to draw conclusions on their performance.

To achieve this goal, the OpenAI gym TORCS environment was used [2]. This environment provides a realistic racing simulation with plenty of tools for benchmarking the performance of an agent, which is why it has seen previous use with autonomous agents for deep reinforcement learning [3]. This aligns well with the objective of the project, as the agents can be subject to a detailed physical simulation, where precise performance data can be gathered from observation. After much contemplation, these three types of agents were chosen.

- **The Zero-Memory Agent:** This agent will do no learning and operate entirely from the information in the current state, as a baseline performance.
- **The Q-Learning Agent:** An agent that uses Q-learning [4] to discretise the state, and creates a table of state-action pairs to rank the expected reward from different actions taken in the past.
- **The Deep Q-Learning Agent:** This agent will use deep Q-learning [5], an evolution of Q-learning where the table of expected rewards will be approximated with a neural network, to handle continuous states.

These agents were chosen as they demonstrate increasing levels of complexity. This was done to ensure that the autonomous driver was not constrained by its underlying code, without making the code overcomplicated.

II. Methodology

A. The Environment

The experimental setup for the environment was running on Ubuntu 22.04 and needed to be built from source using the instructions in the Gym-TORCS repository. Integrating the environment with Python also requires the installation of the *OpenAI gym, numpy* and *tensorflow* packages.

Fig. 1. Racing in TORCS.

1) State Space: There is a lot of information that we can extract from the environment to construct our state, the available variables include:

- The speed[1]
- Opponent positions
- The engine RPM
- The track sensors
- The wheelspin velocity
- A crash flag

The track sensors are a list of distances from the driver's perspective to the edges of the track. There are 19 sensors in total, facing forward and spanning from $[-90°, 90°]$, the

[1]The speed is relative to the origin of the environment, not the agent.

sensors are also normalised between zero and one with a maximum distance of 200 meters. These sensors will be the main focus of our agents as they provide substantial information about the environment, without cheating by telling the agent exactly what to do. If too much information were to be given in the state, such as the angle of the car relative to the track, the problem would become too simple and could be solved without any form of memory at all.

The Gym-TORCS environment was also modified to include the current lap time and the distance the agent has raced into the state, these are used as metrics to evaluate performance.

2) Action Space: The action space is quite straightforward, consisting of two elements, the direction and the throttle. The direction is a variable in the range $[-1, 1]$ representing the angle that the agent should be turning at, whereas the throttle is in the range $[0, 1]$ and is used to regulate the speed of the agent. There are optional actions that could have been added as well, such as the gear and breaking, but the gear can be automated and as such is not relevant. breaking could have been added, but this would have required more modifications to the environment.

Algorithm 1: Pseudocode for the zero-memory agent.

```
In  : An array of 19 sensors $sensors$, two floating
      point variables composing the previous action,
      $dir$ and $gas$, and a constant $threshold$ value.
Out: The new direction and throttle of the car.
function ACTION($sensors, dir, gas, threshold$):
    if $min(sensors) < threshold$ then
        $leftmin \leftarrow min(sensors[0:9])$
        $rightmin \leftarrow min(sensors[10:end])$
        if $leftmin > rightmin$ then
            $dir \leftarrow exp\,(-rightmin)$
        else
            $dir \leftarrow -\,exp\,(-leftmin)$
        end
        $gas \leftarrow 0.8 * gas$
    else
        $direction \leftarrow 0.1 * direction$
        $gas \leftarrow min(1.25 * gas, 1)$
    end
return $dir, gas$
```

B. Belief, Desire and Intention

If the agents were to be characterised in terms of human cognition, the belief-desire-intention model would be used. With this model, it could be said that the agents *believe* that they have a list of sensor values describing their distance to the edges of their environment. The agents would *desire* to navigate the track in the safest possible way, without crashing. Finally, the agents would *intend* to steer away from the edges of the track when they get too close. The following agents are implemented with this model in mind.

C. The Zero-Memory Agent

This agent was developed to provide a standard measure of performance, it does not have any memory, modifying only the previous action, and as such has limited capacity for intelligent behaviour. It was programmed to reduce the throttle and steer away when close to the edges, and accelerate when it is far enough away. The pseudocode of the agent is shown in Algorithm 1, this code checks to see if any of the sensors is below a certain threshold (for our agent this is 0.03875, corresponding to 7.75 meters), if they are it modifies the direction of the agent to steer away and reduces the throttle. If the angles are above the threshold the direction is slowly reduced back to zero and the throttle is increased.

D. The Q-Learning Agent

This agent was intended to improve upon the previous agent, by using memory to learn from experience. Q-learning is an iterative learning process that uses a discretised version of the state and action space, to create a table of state-action pairs. This table is then used to estimate the expected reward for taking an action in a certain state. The expected reward is estimated with a Q-value, which is calculated iteratively using Equation 1. This equation has two hyperparameters, the learning rate α and the discount factor γ. In short, this equation updates the expected reward of the current state-action pair using the highest expected reward in the next state and the reward for taking the current action. The learning rate is the intensity at which the Q-values are adjusted, and the discount factor represents how much future rewards are considered when choosing an action.

$$Q(s,a) \leftarrow (1-\alpha)\,Q(s,a) + \alpha\,(r + \gamma\, max\,[Q(s',a')]) \quad (1)$$

The Q-value update equation allows us to calculate the expected reward for taking an action, but this is not enough, the agent will have to explore the environment and make randomised actions to evaluate the expected reward for each state-action pair in the Q-table. This brings us to the last two hyperparameters for Q-learning – the exploration rate ϵ and decay rate τ. The exploration rate is used in the training of the Q-learning agent, and it represents the probability that actions should be taken randomly instead of the action with the highest expected reward. The decay rate is the rate at which the exploration rate is decreased, as the agent needs to have a trade-off between exploration (choosing a random action) and exploitation (choosing the action with the highest expected reward).

What does this look like for our agent? The pseudocode for this is shown in Algorithm 2, which defines two functions, a function to learn (which just updates the Q-table), and a function that chooses the action to take in the current state. The throttle of the car is determined similarly to the zero-memory agent, the zero-memory agent reduces the throttle when there is an obstacle, whereas the Q-learning agent sets the throttle as a multiple of the distance from the bonnet to the track edge, this allows for a more continuous acceleration and deceleration.

Algorithm 2: Pseudocode for the Q-learning agent.

```
In  : The current discretised state $s$, the current action
      index in the list of actions $a$, the reward $r$, the
      next state $s'$, the learning rate $\alpha$, discount factor
      $\gamma$ and the Q-table $table$.
Out: The updated Q-table.
function LEARN ($s, a, r, s', \alpha, \gamma, table$) :
    $qvalue \leftarrow table[s][a]$
    $newqvalue \leftarrow max(table[s'])$
    $table[s][a] \leftarrow (1-\alpha)\, qvalue + \alpha\, (r + \gamma\, newqvalue)$
return $table$
In  : The current state $s$, list of actions $actionList$,
      qtable $table$, and the exploration rate $\epsilon$.
Out: The action the agent will take.
function ACTION ($s, actionList, table, \epsilon$) :
    $direction \leftarrow$ NULL
    $throttle \leftarrow$
      $max\,(1, 5 * \underline{bonnetDistanceToTrack}(s))$
    if $random() < \epsilon$ then
        $direction \leftarrow random(actionList)$
    else
        $direction \leftarrow actionList\,[max(table[s])]$
    end
return $direction, throttle$
```

1) Training: The Q-table was trained with the following hyperparameters, a learning rate of 0.15, a discount factor of 0.5, and an exploration rate of 0.2, the decay rate was set to 0.995 and was applied to the exploration rate every time the agent crashed. Until this point the state and its method of discretisation have not been mentioned, this is because the values used in binning the state can also be considered hyperparameters. The state of the agent is a tuple of three values, the binned sensor distance from the bonnet to the track, the smallest binned sensor distance on the left side, and the smallest sensor binned distance on the right side. Nine intervals compose the bins in total, which are evenly distributed between zero and 0.1 (or zero and 20 meters). There are also seven actions in total, those actions being to set the angle to one of the following values 0.0, ± 0.1, ± 0.2, ± 0.4. This may seem oversimplified, but the agent has to store $9^3 * 7$ (5103) parameters already, so increasing the number of bins will make the model increasingly harder to train. The reward for the Q-learning agent was modified from the original Gym-TORCS reward, the original reward was calculated using the angle of the car relative to the track, and the speed of the car. This meant that the code rewarded the agent's progress, which was bad for the Q-learning agent, as it learned to accelerate directly into the wall. This was fixed by modifying the Python source code to punish the agent when the angle between it and the track grew large, this resulted in a model that successfully converged, and was able to navigate the track.

E. The Deep Q-Learning Agent

The deep Q-learning agent was intended to be the successor of the Q-learning agent, unfortunately, it was not able to successfully navigate the track. The deep Q-learning agent was made to follow the same structure as the Q-learning agent, but instead of a table to estimate the expected rewards, a neural network would be used to approximate the Q-values. The agent would perform a run of the environment, and record all the states and actions it took, then a model would be trained to predict those results, the pseudocode for this is shown in Algorithm 3. This algorithm uses experience replay to train the neural network, storing the values of the previous run and training the network once the episode is finished by calculating the expected reward from the stored experience. It also uses a target network [6], which is a method developed by Google's DeepMind group to stabilise the training of the agent by calculating the future expected reward with a delayed version of the network used to predict the current expected reward. Without this, the network would use its predictions to evaluate itself, leading to highly unstable training.

1) Training: The hyperparameters of the deep Q-learning network are as follows, the model was trained using the ADAM optimiser, with a learning rate of 0.01, a discount factor of 0.99, an exploration rate of 0.3, and a decay rate of 0.995. Additionally, the update rate for the target network was set to 0.05. The neural network for this agent consists of the 19 sensors as inputs, then four layers of 32 neurons with a ReLU activation function, and seven outputs, representing the seven actions that the agent can take, which are the same as described in the Q-learning agent. The structure of the agent's neural network can be seen in Figure 2. Unfortunately, this model was unable to converge given an extended period, and due to the network being a black box model, the cause is very difficult to identify.

F. Data Collection

During the training of the Q-learning agent, the mean reward for each training episode was gathered and saved to a file. Once the agents had finished training, the only thing left was to gather performance data. The two metrics of importance for an agent with a desire to race are the lap time and the distance before a crash.

To gather this data the Python source code was modified to pass along the distance raced and last lap time from TORCS as an observation of the environment. This data was then captured on each episode and repeated with each agent until 50 laps had been completed. This was a difficult task as it takes a significant time for 50 laps to elapse, and any mistake or failure requires that the process be restarted.

The data was gathered using the procedure in Algorithm 4, this procedure performs multiple runs of the agent on the environment, storing the lap time when it changes and the distance raced when the agent crashes. From these lists, we can gather three metrics, the distribution of lap times, the distances raced, and the number of times the agent crashed. Since the zero-memory agent moves deterministically, it is

Algorithm 3: Pseudocode for the deep Q-learning agent.

```
In  : The current discretised state $s$, the current action index in the list of actions $a$, the reward $r$, the next state $s'$, a boolean to indicate if the agent crashed $done$, and a list to store these values for the training of the model $memory$.
Out: The updated list of training information $memory$
function LEARN ($s, a, r, s', done, memory$):
    $memory.append((s, a, r, s', done))$
return $memory$
In  : The current state $s$, list of actions $actionList$, the deep Q-learning network $model$, and the exploration rate $\epsilon$.
Out: The action the agent will take.
function ACTION ($s, actionList, model, \epsilon$):
    $direction \leftarrow$ NULL
    $throttle \leftarrow$
      $max\,(1, 5 * \underline{bonnetDistanceToTrack}(s))$
    if $random() < \epsilon$ then
        $direction \leftarrow random(actionList)$
    else
        $direction \leftarrow$
          $actionList\,[argmax(model.predict(s))]$
    end
return $direction, throttle$
In  : The deep Q-learning network $model$, the target network $target$, the update rate for the target model $\tau$, the discount factor $\gamma$, and the list of past experiences $memory$.
Out: The updated networks.
function TRAIN ($model, target, \tau, \gamma, memory$):
    $s, a, r, s', d \leftarrow memory$
    $qvalues \leftarrow model.predict(s))$
    $futureQvalues \leftarrow target.predict(s'))$
    for $i$ in $length(memory)$ :
        $qvalues[i, a[i]] \leftarrow$
          $r[i] + \gamma * max(futureQvalues) * (!d)$
    end
    $model.train(s, qvalues)$
    for $i$ in $length(model.weights)$ :
        $target.weights[i] \leftarrow \tau * (model.weights[i] -$
          $target.weights[i]) + target.weights$
    end
    $memory \leftarrow [\,]$
return $model, target$
```

initialised to a random starting direction and throttle when the environment is reset; Similarly, the Q-learning agent does not do any learning, but the exploration rate is set to 0.02 to make it move randomly 2% of the time. This randomness is added as an experimental control to gather more realistic testing data.

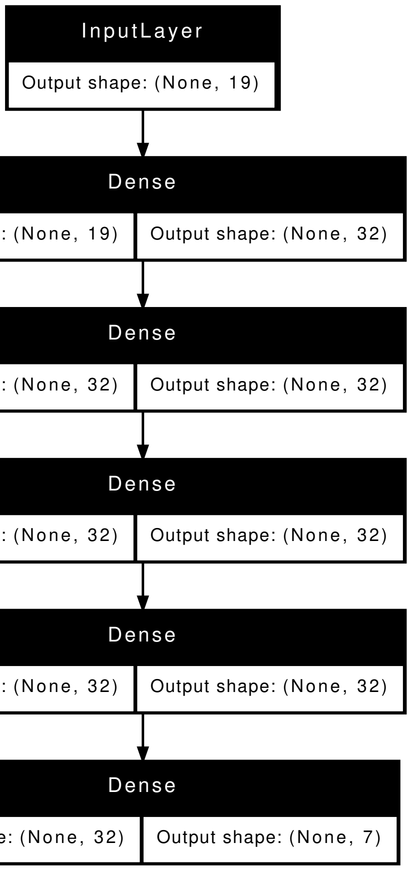

Fig. 2. The structure of the neural network for deep Q-learning.

III. Evaluation

Visualised in Figure 3 is the mean reward for each episode of the Q-learning agent's training process. This figure shows the agent beginning with a very bad score, which slowly increases until it eventually plateaus, the curve is also very noisy, which is almost certainly due to the random nature of the agent's actions during training.

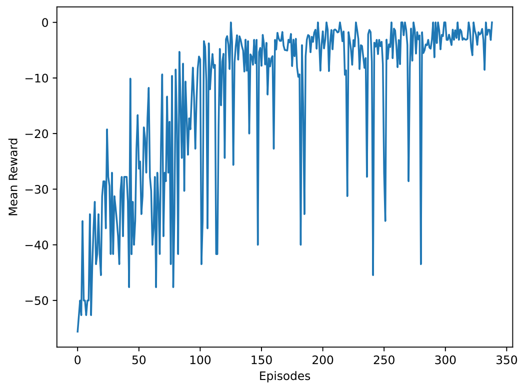

Fig. 3. The mean reward for the Q-learning agent for each episode of its training.

In Figure 4, a distribution of lap times was collected for both

Algorithm 4: Pseudocode for the data collection.

```
In  : The agent from which the data will be gathered
      $agent$, and the number of laps to run for
      $lapCount$.
Out: A list of lap times $laptimes$, and a list of
      distances that the agent travelled before crashing
      $distances$.
function RESULTS($agent$):
    $done \leftarrow False$
    $env \leftarrow \underline{TorcsEnvironment}()$
    $distances \leftarrow [\,]$
    $laptimes \leftarrow [\,]$
    $state \leftarrow env.reset()$
    while $length(laptimes) < lapCount$ do
        $action \leftarrow agent.action(state)$
        $state, done \leftarrow env.step(action)$
        if $state['lastLaptime']! = 0$ then
            if $length(laptimes) ==$
             $0 or state['lastLaptime']! =$
             $laptimes[end]$ then
                $laptimes.append(state['lastLaptime'])$
            end
        end
        if $done$ then
            $distances.append(state['distanceRaced'])$
            $state \leftarrow env.reset()$
        end
    end
return $laptimes, distances$
```

agents using Algorithm 4. This figure clearly shows that the Q-learning agent has a much lower mean lap time. The average lap time for the Q-learning agent is 89.3 ± 3.0, whereas the zero-memory agent has an average lap time of 92.2 ± 3.1.

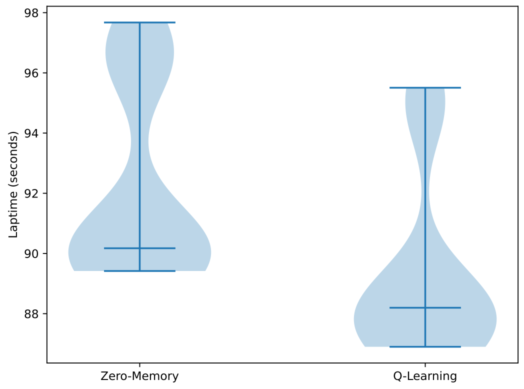

Fig. 4. The distribution of lap times for the Q-learning agent and the zero-memory agent.

In Figure 5, a list of distances before the agents crashed was collected for both agents using Algorithm 4. The average distance raced for the Q-learning agent is $10,346.4 \pm 7515.0$, whereas the zero-memory agent has an average distance raced of 4303.6 ± 3748.7. From this data, we can see that the average distance raced of the Q-learning agent is higher, but with a significantly larger spread compared to the zero-memory agent. Although the zero-memory agent performs worse on average, it travels a more consistent distance, likely due to the constant random actions taken by the Q-learning agent.

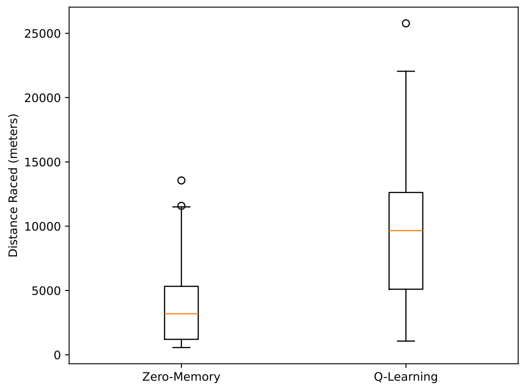

Fig. 5. The distribution of distances raced for the Q-learning agent and zero-memory agent.

The final metric gathered was the number of crashes each agent made whilst attempting to race 50 laps, this data is shown in Figure 6. This figure shows the zero-memory agent crashing a total of 26 times, giving it a lap-to-crash ratio of $25 : 13$. The same metric can be calculated for the Q-learning agent, with 11 crashes made throughout the 50 laps, giving it a better lap-to-crash ratio of $50 : 11$.

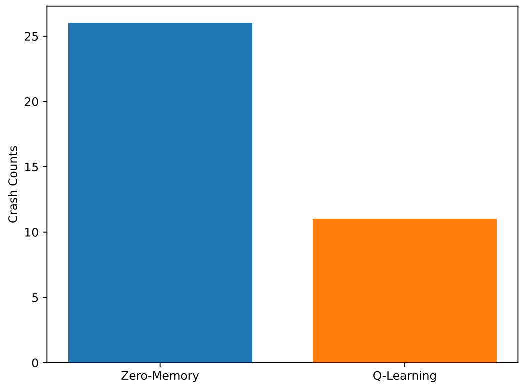

Fig. 6. The number of crashes during the 50 laps for the Q-learning agent and zero-memory agent

Now that all the results have been gathered, it is time to return to the original question in this project, what is the performance of these techniques?

Whilst the zero-memory agent successfully drives around the track, it exhibits very little intelligence, as if it was driving whilst blindfolded. Its lack of knowledge of the environment creates scenarios where the agent drives towards a bend with an angle of attack that leaves it unable to avoid a crash.

The most successful learning technique by far is Q-learning, it travels further and faster than the zero-memory agent. This is a direct result of the agent's intelligent behaviour. The zero-memory agent has no prior experience in the environment and travels with no direction, avoiding walls when they appear. The Q-learning agent addresses this and closely follows the edges of the track with significantly less wasted movement, this knowledge of the environment allows it to achieve its desires with great success.

The deep Q-learning agent performed the worst, as it was unable to learn anything about the environment. This is a common problem in deep Q-learning called catastrophic forgetting [7], where the agent immediately forgets old information after learning new information and is unable to learn multiple tasks at once. To combat this, a target network was used to slow down and stabilise the training, so that the neural network can converge. If the deep Q-learning agent were subject to more fine-tuning of the hyperparameters, there is still a possibility that it could navigate the track, but the difficulty and time required in training such an agent is extremely high.

IV. Conclusion

In summary, the zero-memory agent was successful in navigating the track with a mean lap time and mean distance raced of 92.2 ± 3.1 and 4303.6 ± 3748.7. However, this agent exhibited a severe lack of intelligent behaviour, crashing a total of 26 times while attempting 50 laps of the track.

The Q-learning agent used its knowledge of the environment to perform the task effectively, improving upon the zero-memory agent with better lap time and racing distances of 89.3 ± 3.0 and $10,346.4 \pm 7151.0$ respectively. It also crashed a total of 11 times during its attempt at 50 laps, which is significantly less than the zero-memory agent. Given more time, the agent could further be improved with adjustments to the reward system, by penalising the agent for being too close to an edge.

The deep Q-learning agent used a learning technique that was ultimately ineffective, suggesting that it is not a suitable method for an autonomous racer. This could have been improved through modifications to the reward system, or a change in hyperparameters. Another possibility would be to remove the 'deep Q-learning' entirely and train a neural network with a single continuous output for the direction of the agent.

To conclude, the best-performing agent was the Q-learning agent, with the zero-memory agent in second place due to a lack of intelligence. The deep Q-learning agent failed to perform the task likely due to the hard problem of catastrophic forgetting.

References

[1] L. M. Martinez and J. M. Viegas, "Assessing the impacts of deploying a shared self-driving urban mobility system: An agent-based model applied to the city of lisbon, portugal," *International Journal of Transportation Science and Technology*, vol. 6, no. 1, pp. 13–27, 2017.

[2] "Torcs environment for openai gym," https://github.com/dosssman/GymTorcs, 2013.

[3] K. Güçkıran and B. Bolat, "Autonomous car racing in simulation environment using deep reinforcement learning," in *2019 innovations in intelligent systems and applications conference (ASYU)*. IEEE, 2019, pp. 1–6.

[4] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[5] F. Tan, P. Yan, and X. Guan, "Deep reinforcement learning: from q-learning to deep q-learning," in *Neural Information Processing: 24th International Conference, ICONIP 2017, Guangzhou, China, November 14–18, 2017, Proceedings, Part IV 24*. Springer, 2017, pp. 475–483.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[7] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.