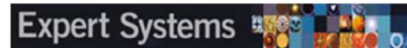
View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Repository@Nottingham

Received: 11 May 2020 | Revised: 15 September 2020 | Accepted: 6 November 2020

DOI: 10.1111/exsy.12665



WILEY

Solving the Rubik's cube with stepwise deep learning

Colin G. Johnson

School of Computer Science, University of Nottingham, Nottingham, UK

Correspondence
Colin G. Johnson, School of Computer Science, University of Nottingham, Jubilee Campus, Wollaton Road, Nottingham NG8 1BB, UK.
Email: colin.johnson@nottingham.ac.uk

Abstract

This paper explores a novel technique for learning the fitness function for search algorithms such as evolutionary strategies and hillclimbing. The aim of the new technique is to learn a fitness function (called a *Learned Guidance Function*) from a set of sample solutions to the problem. These functions are learned using a supervised learning approach based on deep neural network learning, that is, neural networks with a number of hidden layers. This is applied to a test problem: unscrambling the Rubik's Cube using evolutionary and hillclimbing algorithms. Comparisons are made with a previous LGF approach based on random forests, with a baseline approach based on traditional error-based fitness, and with other approaches in the literature. This demonstrates how a fitness function can be learned from existing solutions, rather than being provided by the user, increasing the autonomy of AI search processes.

KEYWORDS
artificial intelligence, evolutionary computation, fitness functions, human-like AI, loss functions

1 | INTRODUCTION

The aim of this paper is to present a new kind of fitness function that can be applied to evolutionary and hillclimbing algorithms. Instead of the fitness being defined directly from an error function, a pre-training process is used to learn a fitness function from a set of solved examples of the problem class. Call these functions *Learned Guidance Functions*. These smooth out the fitness landscape by taking a set of solved examples for a problem, and learning a new fitness function based on the distance taken to move between state in the solved examples and the solved state. This function can then be applied to previously unseen examples.

This paper is an extended version of a conference paper (Johnson, 2019). This paper additionally contains a new set of experiments comparing hillclimbing and evolutionary approaches, a new set of experiments that investigate the structure of the learned models, and a more thorough contextualisation of the work with other literature.

Fitness functions—also known as error functions, loss functions, and objective functions—are a key component of most approaches to machine learning. These are problem-specific functions that measure how far a point in the search space is from being a solution to the problem at hand, or a ranking function that allows two points to be compared. This ability to specify the important aspects of the problem through a single function—in many cases a single scalar function—is at the core of the metaheuristic approach to problem solving. By containing the domain-specific aspects of the problem primarily to this function, improvements to search and optimisation algorithms redound to improvements in many different problems.

There are difficulties with such an approach. Most obviously, such functions typically have many local minima. This is often seen as a core part of the problem to be solved. A large amount of the evolutionary computation literature is dedicated to operators and other techniques that allow the search to escape local minima and ensure a balanced exploration of the search space. That is, avoiding these structural problems in the fitness function is seen as a problem to be tackled as part of the search process.

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.
© 2021 The Authors. *Expert Systems* published by John Wiley & Sons Ltd.

Another approach focuses on how the fitness function itself can be transformed to avoid these problems. This has a long history in the evolutionary computation, typified by work on *fitness scaling* (Grefenstette, 1986; Hopgood & Mierzejewska, 2009; Kreinovich et al., 1993; Ware et al., 2003). The principle aim of fitness scaling is to prevent premature convergence of the search algorithm, by composing a scaling function with the fitness function that does not change the ranking of points in the search space but ensures a more even distribution of the fitness values allocated to those points. A more recent version of this transformation approach is exemplified by geometric semantic genetic programming (GSGP) (Moraglio et al., 2012), which attempts to reconfigure the problem so that a much simpler search process such as hillclimbing can be used.

The aim of these approaches is to create a transformed problem that can be solved more effectively. However, these transformations sometimes involve some kind of tradeoff. For example, in basic GSGP, this tradeoff is against the size of the solution, though more recent implementations have used a caching strategy to make implementation more efficient (Vanneschi et al., 2013). This idea of reconfiguring the fitness function prior to the main evolutionary algorithms being run is one source of inspiration for the work in this paper; this has been explored elsewhere in evolutionary computation in work showing how a good choice of genotype–phenotype mapping can be used to create a smoother landscape (Asselmeyer et al., 1996).

A domain-specific smoothing the search landscape is in the form of *pattern databases* (Culberson & Schaeffer, 1998). These consist of patterns in the search space such that any point that matches the pattern has the same cost of solution—typically, these represent symmetries of the underlying problem. If a solution of a particular cost is known for one problem that matches the pattern, then any other solution matching the pattern will have at most that cost to solve because all of the moves to the solution can be similarly transformed. As with the approaches discussed earlier, the idea here is to transform the search space. The key difference is that pattern databases are based on domain knowledge. Some work has used learning methods to generalize from pattern databases—for example, by using neural networks to learn how pattern databases can be combined (Mehdi Samadi & Felner, 2008).

Another important source of inspiration is the view that traditional fitness functions take a very narrow view of the problem; while a traditional fitness function is a good guide as to which elements of the population to choose for the next generation, it is a very simple representation of the complexity of a problem. Instead, it is argued, rather than a fitness function that returns a single number or a ranking, we should be using more complex *fitness drivers* that give us more information about the population member, allowing a more directed application of operators (Krawiec, 2016; Krawiec et al., 2016). However, such fitness drivers can require more domain-specific knowledge than a traditional fitness function. One of the aims of this paper is to give a generic method by which more information about problems can be incorporated into the evolutionary search, in this case by pre-training.

A more fundamental problem for evolutionary algorithms is that for some problems, defining the fitness function is difficult, because each problem has a different goal state. Call these *non-oracular problems*. As an example, consider the protein-folding problem in bioinformatics (Dobson, 2003). Biological proteins consist of a sequence of amino acids, which then fold into a three-dimensional shape, which is (with a few exceptions such as prions) entirely dependent on the sequence. To define this as a traditional evolutionary search is problematic, because we do not have access to a measure of how far a particular configuration is from the solution—indeed, if we did know what configuration we were searching for, we would have solved the problem! Therefore, evolutionary computing approaches to these types of problems have focused on learning parameters in, or functional forms of, a domain-specific model (Widera et al., 2010).

Another potential advantage to pre-training for simplifying the fitness landscape is that more extensive computational effort can be expended during an early training phase, and then when evolution is applied to a specific problem, the evolutionary algorithm can run in fewer generations because more domain-specific information has been encoded into the fitness function. This may be of importance in some application where running a traditional evolutionary algorithm might be infeasible because of the need for a large population and many generations to escape local minima, whereas a smaller population and fewer generations might be needed for the simpler function. This is particularly important where the evolutionary algorithm needs to run in a time-constrained situation.

2 | DEEP LEARNED GUIDANCE FUNCTIONS

Fitness functions in evolutionary learning are provided as part of the problem definition. Typically, these are then used directly—individuals are evaluated using the fitness function, and operators in the search are used to avoid problems in the fitness landscape such as local minima. However, an alternative approach has been applied in both evolutionary learning (Szubert et al., 2013) and reinforcement learning (Erez & Smart, 2008), where the fitness function is *shaped* so that it more directly represents routes through the fitness landscape from an arbitrary point to the desired target.

A form of this called *Learned Guidance Functions* (LGFs) was introduced by (Johnson, 2018). The input to this is a search space and set of existing solution trajectories for the problem. For example, in the protein folding problem these would be sequences of points in the space of three-dimensional structures, going from a sequence to a completely folded structure. For an image denoising problem, this would be a sequence of images from a clean image to a very noisy one. These are an example of *True Distance Heuristics* (Sturtevant et al., 2009), but with a particular layered structure and the use of a predictive function to give the heuristic value rather than a look-up table.

These solution trajectories can be obtained in a number of ways. For some problems, we will have access to a set of already-solved examples. For others, we can construct artificial examples by starting from a solved state and carrying out a number of moves from that solved state to generate trajectories in reverse (a similar approach has been called *Autodidactic Iteration* in McAleer et al., 2018).

These trajectories can then be used to create a training set for a supervised learning problem. Each trajectory will consist of a number of states in the search space of the problem, each of which is paired with a number that is the number of steps away from the target that it took to get to that state. These pairs then become the training set: so, the task for the supervised learning problem is to build a model that takes an arbitrary state of the system and assigns a number predicting how many steps it will take to get to the target state. The LGF is the model learned from this supervised learning process.

This LGF then be used as a ranking function in an evolutionary algorithm. Take each member of the population, and apply the LGF to it. Then select the individuals that will form the parents of the next population from the lowest scoring ones on the LGF—these are the ones that are being predicted as being closest to the solution.

A similar approach has been taken in the recent papers by McAleer et al. (2018) and Agostinelli et al. (2019), though these are grounded in a reinforcement learning approach rather than a supervised learning approach. Compared with the work in this paper, their algorithm learns a mapping from points in the state space of the cube to a pair consisting of a value and a policy. The approach that we take in this paper can be seen as the equivalent of learning just the value function. Indeed, the *greedy* experiment in McAleer et al. (2018, fig. 5a) demonstrates a similar percentage-solved behaviour to the experiments in this paper. However, their full approach, which combines value function approximation and policy learning, and then uses an Monte Carlo Tree Search (Browne et al., 2012) approach, continues to work for problems of much higher complexity that the experiments in this paper, but at the cost of much longer search times (around 10 min as shown by Figure 5b and the description on Page 6 of McAleer et al., 2018) compared to a few seconds for our approach. In summary, their approach is scalable to larger problems (more scrambles), but at the cost of increased computation time, whereas the approach in this paper offers a quicker solution approach to simpler problem instances but fails for larger problems.

2.1 | Formalization

Now we formalize this idea. Consider a search space S consisting of a set of points, which is the node set of a directed graph M_S, which represent the possible moves (mutations) from each point in the search space. Identify one or more of these as goal states; these might be the only goal states, or they might represent a sample of the class of states that the eventual problem is trying to solve.

Now take a set of trajectories $T = \{T_1, T_2, \ldots T_{n_T}\}$, where each trajectory is a set of points in S, that is, $T_i = \left[s_1, s_2, \ldots, s_{n_{T_i}}\right]$, where in each of these cases s_1 is a goal state, and where each adjacent pair (s_i, s_{i+1}) are joined by an edge in M_s. Now create a new set X consisting of the pairs (s_i, i) for all the s_i in all members of T.

X can now be used as a training set for a supervised learning algorithm. The trained model from that supervised learning algorithm, $L : S \to \mathbb{Z}^{\geq 0}$, is a function that takes a set in the search space and predicts how many moves are needed to get to the goal state. This function will be used as an alternative kind of fitness function in the experiments below.

3 | EXAMPLE: APPLYING DEEP LGFS TO THE RUBIK'S CUBE

In Johnson (2018) we applied the LGF to the problem of unscrambling the Rubik's Cube. We used a number of classifiers from the *scikit-learn* library (scikit-learn, n.d.) to implement LGFs, and demonstrated that (1) the LGF can learn to recognize the number of turns that have been made to a cube to a decent level of accuracy; and, (2) that this LGF can then be used to unscramble particular states of the cube in a sensible number of moves. Unscrambling is not one of the non-oracular problems, because the goal state is known, but it has a complex fitness landscape with many local minima, and so is a good test for these kind of algorithms.

The search space C consists of all possible configurations of coloured facelets on the six faces of the cube, each of which has a 3×3 set of facelets. The move set M is notated by a list of twelve 90° moves, $\{F, B, R, L, U, D, F', B', R', L', U', D'\}$ (Singmaster, 1981), which are functions from C to C. We use the notation $m(c)$ to denote the application of move $m \in M$ to the cube c, returning the new state of the cube.

An earlier paper by the author (Johnson, 2018) applied a number of learning algorithms to the problem of learning LGFs for the Rubik's cube, with random forests demonstrating itself to be the best approach. That paper did not use any deep learning (Goodfellow et al., 2017) approaches—in this paper, we extend the work by using deep learning.

3.1 | Constructing the LGF

The LGF for this problem is constructed as follows (pseudocode in Algorithm 1). For n_s iterations, start with a solved cube and make $n_{\ell-1}$ moves. Each time a move is made (and in the initial state), the pair consisting of the current state and the number of moves made to get to that state is added to the training set. This is illustrated in Figure 1.

Algorithm 1

Training set construction for the Rubik's cube

1: **procedure** ConstructTrainingSetRubik (n_s, n_m)
2: **let** $M = \{F, B, R, L, U, D, F', B', R', L', U', D'\}$
3: **let** $X = \emptyset$
4: **for** $s \in [0, ..., n_s - 1]$ **do**
5: **let** c be a new cube in the solved state
6: for $\ell \in [0, ..., n_m - 1]$ **do**
7: **let** $X = X \cup \{(c, \ell)\}$
8: **let** m= random element from M
9: **let** $C = m(c)$
10: **end for**
11: **end for**
12: **return** T
13: **end procedure**

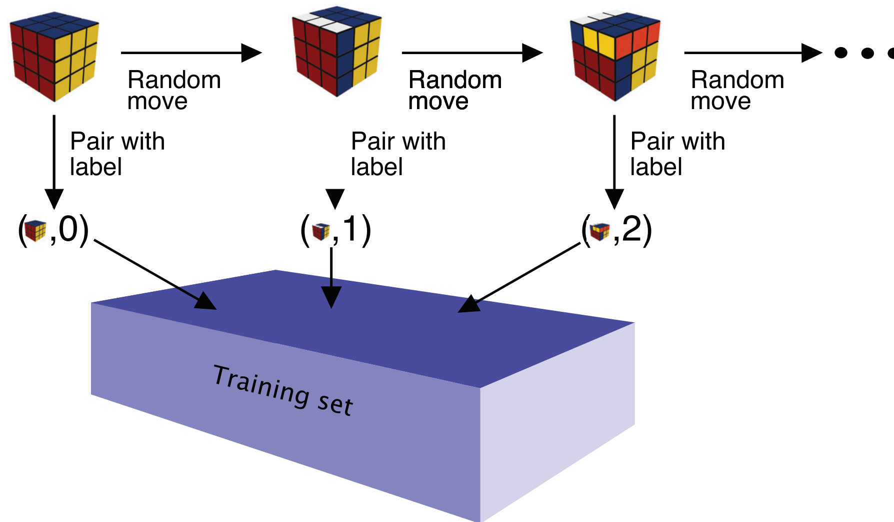


FIGURE 1 Construction of the training set (modified from Johnson, 2018)

The LGF is then constructed from this training set by applying a supervised learning algorithm, specifically a deep neural network implemented in the *Keras* framework (Keras, n.d.) on TensorFlow (TensorFlow, n.d.). The specific network used is illustrated in Figure 2. This is a fairly standard deep learning network, with dropout (Srivastava et al., 2014) used to encourage generalization and prevent overfitting. The categorical crossentropy function was used for the loss function, and the Adam optimizer was applied. Future work will apply meta-learning of parameters and network shape to optimize the model produced (Hutter et al., 2019).

Once an LGF is learned, it can be applied to the task at hand, which is to take a scrambled state of the cube and move through the search space with the aim of finding the solved state. This is done using a variant on evolution strategies (ES). The initial state of the cube is duplicated to fill the population. Then, in each generation a number of mutants are generated by making a random move for each member of the population. Any solutions that are predicted by the LGF to be closer to the solution than the current one are placed in an intermediate population pool, and a new generation created by uniform random sampling with replacement from this pool to bring the population up to full size. This is repeated until one of three states occurs: (1) the solution is found; (2) none of the mutants produce any improvement, in which case the algorithm is restarted;

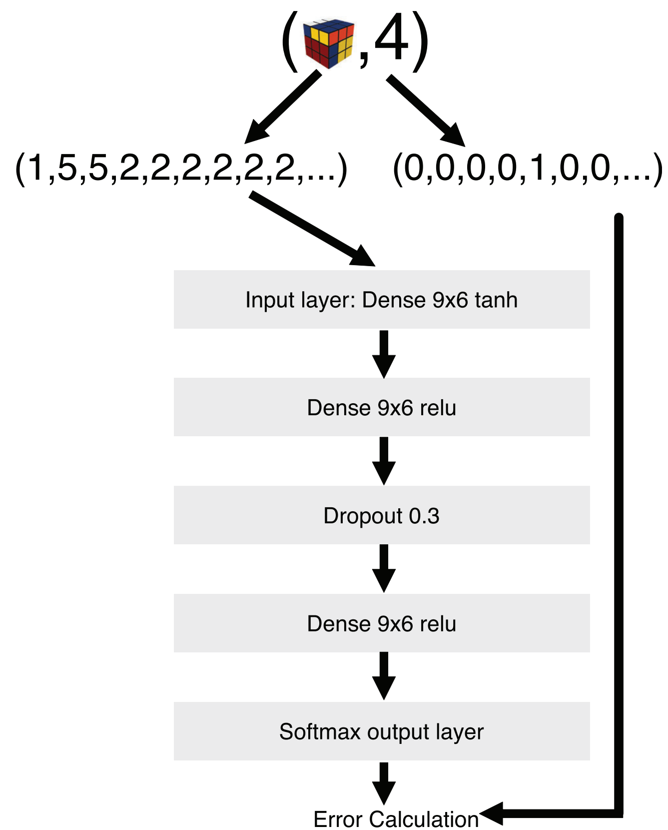


FIGURE 2 Keras deep learning network used for training

(3) a user-set limit on the number of generations is reached (in the experiments below, this is 100 generations), in which case a fail-state is returned.

This is summarized in pseudocode in Algorithm 2, where the inputs are: n_ℓ, the problem size (number of scrambling twists given); n_p, the population size; θ the maximum number of generations; and, L the LGF function used. In the results tables, this is referred to as *Deep LGF + ES* (ES refers to *Evolution Strategies*).

When Algorithm 2 discovers a solution, it is then easy to reconstruct a path through the various population iterations to create a solution trajectory. Each population consists of solutions that are mutations (moves) from the previous population, so by starting from the solution, identifying which cube it was a mutation of, and so on back to the original scrambled state, we can construct the trajectory (Figure 3).

A second approach is to use the same fitness function, but to remove the population-based aspect of it. So, instead of having a population of cube states, we start with a single cube state, and then at each iteration generate all possible 12 moves from that state, and use the deep network to predict how many scrambles it is from solved. If one of these predictions is closer to the solved state, then that move is made. Otherwise, a random (neutral) move is made. This is summarized in pseudocode in Algorithm 3, with the input parameters being a subset of those in Algorithm 2. In the results tables, this is referred to as *Deep LGF + Hillclimbing*.

3.2 | Sources of error

Note that if a perfect LGF existed for a problem, we could solve the problem in a minimal number of steps. Starting from an arbitrary scrambled state, we can examine all possible moves from that state. At least one of these will be closer in terms of number of moves to the target state, and so we can move the state of the system to the state which is closest, and repeat until we reach the target state.

In practice, there are two forms of error. The first is in the formation of the training set for the problem. A particular sequence of scrambling moves of length n might, nonetheless, end up with the cube in a state which could have been reached using fewer moves. A simple example of this is where one move is followed by a move which is the inverse of that move (this is explored in more detail by Johnson (2018)). The second is where the model makes the wrong prediction. For these reasons, the fitness landscape created by a real LGF will still have local minima.

Algorithm 2

Evolutionary scrambling/unscrambling algorithm for the Rubik's cube (Deep LGF + ES)

1: **procedure** ES-LGF-Unscramble n_ℓ, n_p, θ, L
2: **let** $M = \{F, B, R, L, U, D, F', B', R', L', U', D'\}$
3: **let** c be a new cube in the solved state
4: **for** $\ell = 0$; $\ell < n_\ell$; $\ell = \ell + 1$ **do**
5: **let** m= random element from M
6: **let** $c = m(c)$
7: **end for**
8: ▷ c now in scrambled state
9: **let** $\ell = n_\ell$
10: **let** $P = n_p$ copies of c
11: **for** $t = 0$; $t < \theta$; $t = t + 1$ **do**
12: **let** $P' = \emptyset$
13: **for** $p \in P$ **do** m= random element from M
14: **let** $P' = P' \cup m(p)$
15: **if** $m(p)$ is the solved state **then**
16: **return** $m(p)$ ▷ Problem Solved
17: **end if**
18: **end for**
19: **let** $P'' = \emptyset$
20: **for** $p \in P'$ **do**
21: **if** $L(p) < \ell$ **then**
22: **let** $P'' = P'' \cup p$
23: **end if**
24: **end for**
25: if $P'' = = \emptyset$**then**
26: **let** $P = n_p$ copies of c ▷ Reinitialise
27: **let** $\ell = n_\ell$
28: **else**
29: **let** $P = \emptyset$
30: **for** $n = 0$; $n < n_p$; $n = n + 1$ **do**
31: $P = P \cup$ random member of P''
32: **end for**
33: **let** $\ell = \ell - 1$
34: **end if**
35: **end for**
36: **return** null ▷ Timed out
37: **end procedure**

4 | EXPERIMENTS AND RESULTS

The experiments were carried out as follows. For each pair $(n_\ell, n_m) \in [2, 13] \times [3, 13]$ where $n_\ell \leq n_m$, Algorithm 2 was run 100 times with the following parameters:

1. Size of problem $n_\ell = n_\ell$
2. Population size $n_p = 100$
3. Maximum number of generations $\theta = 100$

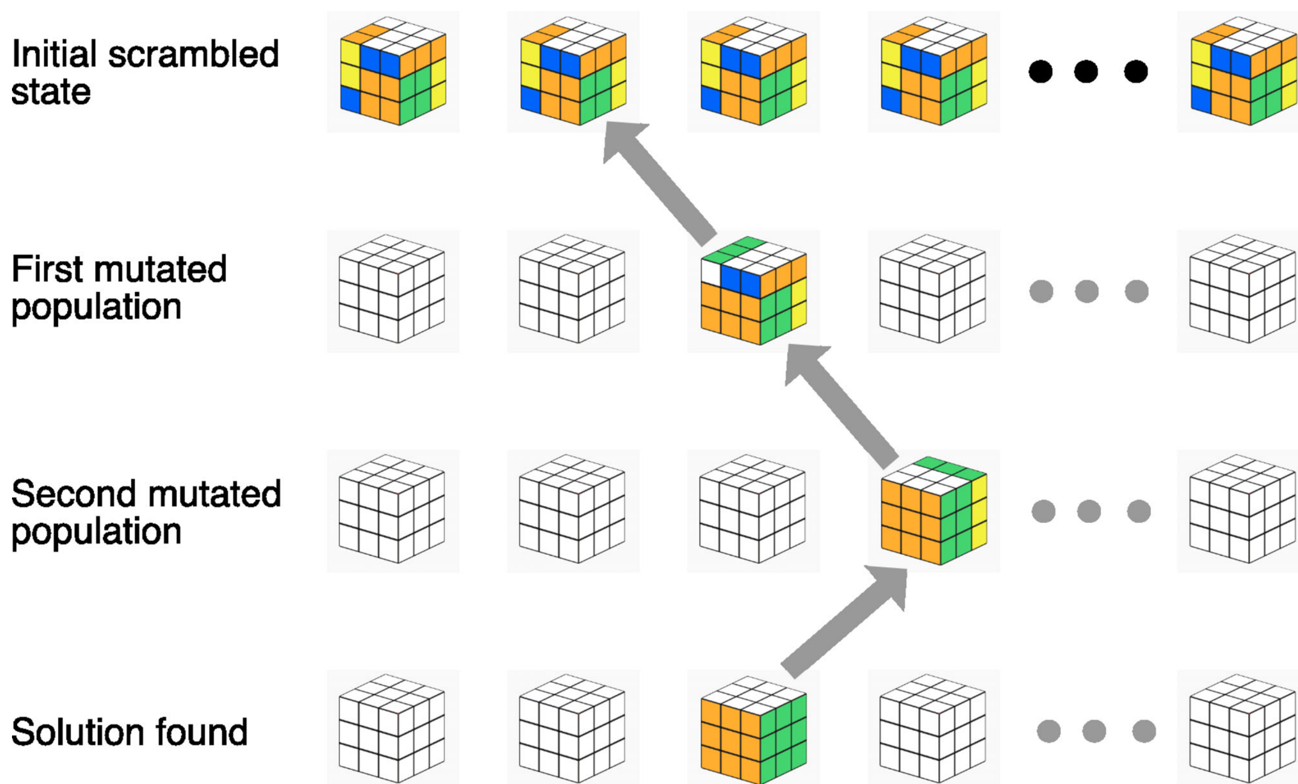


FIGURE 3 Reconstructing a single trajectory from the evolutionary search once the solution has been found

4. LGF function used L is the result of running Algorithm 1 with trajectory length n_m and 100,000 trajectories, then using that as the training set for the Keras network in Figure 2 with 50 epochs.

The total time to run all of these experiments was under 3 h, not including time to train the models (training time was between 11–59 s per epoch depending on the size of the model).

Results for the unscrambling experiments are presented in two tables. Table 1 shows for each (n_ℓ, n_m) pair the percentage of times that the problem was solved. Table 2 shows the number of generations taken by successful algorithms to unscramble the cube.

There are a number of observations. Firstly, for the smaller problem sizes, a solution to the problem is frequently found; for problems below size 9, at least half of the attempts are successful, and it is very reliable for small problem sizes. Secondly, the size of the model makes little difference—using a larger model than the problem size is of little value. Thirdly, the number of generations needed is small for the lower problem sizes, but increases for large problem sizes; this may be an effect of more re-initialisations needing to be carried out.

Fourthly, note that some of the average lengths in Table 2 are shorter than the problem size. This is because the problems were constructed by scrambling randomly n_ℓ times, but no check was made to ensure that the resulting state could not be solved in less than n_ℓ moves; indeed, doing such a check is rather complex. Therefore, the starting state for some runs may contain a problem that can be solved in fewer than n_ℓ moves.

Tables 3 and 4 present the results for the Deep LGF combined with hillclimbing. The total time to run all of these experiments was considerably than the previous experiments, at around 52 h, not including time to train the models (the models were reused from the previous experiment).

The results in terms of percentages solved are overall worse for the hillclimbing approach rather than the evolutionary approach. The lengths to solution are larger in the hillclimbing process for simpler problems, but are in many cases better for more complex problems.

Tables 5 and 6 compare the results of the experiments in this paper (a preliminary version of which as presented by Johnson (2019)) to two experiments in a previous paper (Johnson, 2018). The main experiments in the current paper (Deep LGF + ES) varied from the experiments in this earlier paper (Random Forest LGF + Hillclimbing) in two main ways. Firstly, the models were trained using a random forest classifier (the implementation in the *scikit-learn* package (scikit-learn, n.d.)). The tables give the results for models trained on examples with up to 13 moves. Secondly, the unscrambling in the earlier paper was based on only the hill-climbing approach, whereas in this paper the evolution strategy approach is also used.

Algorithm 3

Hillclimbing scrambling/unscrambling algorithm for the Rubik's cube (Deep LGF + Hillclimbing)

```
1: procedure Hillclmbing-LGF-Unscramble n_ℓ, θ, L
2:    let M = {F, B, R, L, U, D, F′, B′, R′, L′, U′, D′}
3:    let c be a new cube in the solved state
4:    for ℓ = 0; ℓ < n_ℓ; ℓ = ℓ + 1 do
5:        let m= random element from M
6:        let c = m(c)
7:    end for
8:                                              ▷ c now in scrambled state
9:    let ℓ = n_ℓ
10:   for t = 0; t < θ; t = t + 1 do
11:       let C = ∅
12:       for m in M do
13:           let C = C ∪ m(c)
14:           if m(c) is the solved state then
15:               return m(p)                   ▷ Problem Solved
16:           end if
17:       end for
18:       for c′ ∈ C do
19:           let S = False
20:           if L(c′) < ℓ then
21:               let c = c′
22:               let S = True
23:               break
24:           end if
25:       end for
26:       if S = = False then                   ▷ Neutral Move
27:           let m= random element from M
28:           let c = m(c)
29:       else
30:           let ℓ = L(c′)
31:       end if
32:   end for
33:   return null                               ▷ Timed out
34: end procedure
```

These tables also contain a comparison with a baseline experiment also described in detail in the earlier paper (Johnson, 2018) (Error + Hillclimbing). This also uses a simple hill-climbing method, but the choice of moves is made by choosing the move that maximizes the number of correct facelets. This is a traditional error-based fitness function.

It is notable that the percentage of successes in the *Deep LGF + ES* approach is considerably higher than the *Random Forest LGF + Hillclimbing* approach. However, the length of the solutions found by the new approach is much larger for larger problem sizes. This may well reflect the use of reinitialisation in the latter approach; in the earlier paper, a search that did not terminate was considered a failure. The evolutionary approach using the Deep LGF is largely a better performer than the hillclimbing approach. All of these new methods clearly outperform the traditional error-based fitness measure, demonstrating the value of this pre-training step.

TABLE 1 Percentage of times unscrambling problem of each size was solved using a model of each size

		Size of problem											
		2	3	4	5	6	7	8	9	10	11	12	13
Size of model	**3**	100	100	-	-	-	-	-	-	-	-	-	-
	4	100	100	100	-	-	-	-	-	-	-	-	-
	5	100	100	100	98	-	-	-	-	-	-	-	-
	6	100	100	100	89	92	-	-	-	-	-	-	-
	7	100	100	100	94	80	70	-	-	-	-	-	-
	8	100	100	100	93	82	71	61	-	-	-	-	-
	9	100	100	100	92	76	57	52	47	-	-	-	-
	10	100	100	100	97	85	75	63	60	38	-	-	-
	11	100	100	99	92	83	73	73	52	37	21	-	-
	12	100	100	100	97	89	73	71	56	37	22	15	-
	13	100	100	100	90	77	61	63	50	37	22	17	6

Note: Results from 100 runs, Deep LGF + ES.

TABLE 2 Average number of generations needed to solve problem of each size using trained model of each size

		Size of problem											
		2	3	4	5	6	7	8	9	10	11	12	13
Size of model	**3**	1.0	1.7	-	-	-	-	-	-	-	-	-	-
	4	1.0	1.7	2.6	-	-	-	-	-	-	-	-	-
	5	1.0	1.7	2.8	4.4	-	-	-	-	-	-	-	-
	6	1.0	1.6	2.7	4.3	8.4	-	-	-	-	-	-	-
	7	1.0	1.7	2.8	4.0	6.7	8.1	-	-	-	-	-	-
	8	1.0	1.8	2.8	4.8	7.0	11.8	18.1	-	-	-	-	-
	9	1.0	1.7	2.7	4.1	5.9	7.8	10.0	23.4	-	-	-	-
	10	1.0	1.7	2.8	3.6	6.2	12.9	13.3	20.2	28.8	-	-	-
	11	1.0	1.7	2.7	4.0	6.8	10.8	16.7	21.9	23.5	34.3	-	-
	12	1.0	1.8	2.8	4.2	7.3	11.1	14.7	20.5	30.4	31.4	30.5	-
	13	1.0	1.7	2.8	4.1	5.4	10.5	11.4	16.7	19.2	33.6	25.4	69.8

Note: Includes successful solutions only, and includes restarts. Deep LGF + ES.

TABLE 3 Percentage of times unscrambling problem of each size was solved using a model of each size

		Size of problem											
		2	3	4	5	6	7	8	9	10	11	12	13
Size of model	**3**	100	-	-	-	-	-	-	-	-	-	-	-
	4	100	99	-	-	-	-	-	-	-	-	-	-
	5	100	94	58	-	-	-	-	-	-	-	-	-
	6	100	95	70	45	-	-	-	-	-	-	-	-
	7	100	94	75	45	32	-	-	-	-	-	-	-
	8	100	97	77	46	34	28	-	-	-	-	-	-
	9	96	93	73	47	39	21	17	-	-	-	-	-
	10	100	90	84	55	44	24	12	8	-	-	-	-
	11	100	93	81	55	43	24	20	9	7	-	-	-
	12	100	91	72	43	32	20	18	9	12	2	-	-
	13	100	91	72	56	34	18	12	11	3	5	3	-

Note: Results from 100 runs, Deep LGF + Hillclimbing.

TABLE 4 Average number of generations needed to solve problem of each size using trained model of each size

		Size of problem											
		2	3	4	5	6	7	8	9	10	11	12	13
Size of model	**3**	8.4	-	-	-	-	-	-	-	-	-	-	-
	4	6.9	8.8	-	-	-	-	-	-	-	-	-	-
	5	7.5	8.1	12.0	-	-	-	-	-	-	-	-	-
	6	10.7	7.3	11.4	8.7	-	-	-	-	-	-	-	-
	7	9.0	6.7	14.7	8.4	16.6	-	-	-	-	-	-	-
	8	8.0	7.8	14.5	12.8	13.8	9.7	-	-	-	-	-	-
	9	8.3	8.6	14.7	11.6	16.4	13.5	21.9	-	-	-	-	-
	10	7.8	8.6	12.1	11.7	14.0	11.5	17.7	15.0	-	-	-	-
	11	8.9	7.6	14.7	10.7	13.0	9.2	12.1	11.3	9.7	-	-	-
	12	8.6	6.8	13.1	9.3	13.2	13.9	12.5	8.7	12.3	11.5	-	-
	13	9.2	5.6	14.1	13.0	11.9	17.4	17.6	21.8	14.0	10.8	42.0	-

Note: Includes successful solutions only, and includes restarts. Deep LGF + Hillclimbing.

TABLE 5 Comparison of four models: Deep learning of LGF with hillclimbing, deep learning of LGF with evolution strategies, random forest learning of LGF and hillclimbing, and error-based fitness with hillclimbing

	Size of problem											
	2	3	4	5	6	7	8	9	10	11	12	13
Deep LGF + Hillclimbing (this paper)	100	91	72	56	34	18	12	11	3	5	3	-
Deep LGF + ES (this paper and Johnson, 2019**)**	100	100	100	90	77	61	63	50	37	22	17	6
RF LGF + Hillclimbing (Johnson, 2018**)**	100	100	98	75	62	45	20	17	11	9	7	0
Error + Hillclimbing (Johnson, 2018**)**	62	33	24	10	4	3	2	0	0	0	1	0

Note: Percentage of runs that found the solution.

TABLE 6 Comparison of four models: Deep learning of LGF with hillclimbing, deep learning of LGF with evolution strategies, random forest learning of LGF and hillclimbing, and error-based fitness with hillclimbing

	Size of problem											
	2	3	4	5	6	7	8	9	10	11	12	13
Deep LGF + Hillclimbing (this paper)	9.2	5.6	14.1	13.0	11.9	17.4	17.6	21.8	14.0	10.8	42.0	-
Deep LGF + ES (this paper and Johnson, 2019**)**	1.0	1.7	2.8	4.1	5.4	10.5	11.4	16.7	19.2	33.6	25.4	69.8
RF LGF + Hillclimbing (Johnson, 2018**)**	1.8	2.6	3.3	3.9	4.2	4.6	4.8	5.5	4.9	4.6	4.9	-
Error + Hillclimbing (Johnson, 2018**)**	2.1	2.2	2.3	2.8	4.5	3.7	2.0	-	-	-	4.0	-

Note: Average length to solution for successful runs. Note that the average lengths for the approaches with low success percentages are likely to be distorted by those approaches only solving simpler instances (i.e., ones where the scrambling process has undone earlier scrambles).

5 | WHAT IS BEING LEARNED BY THE DEEP NETWORK?

What is being learned by the network? Is it learning the structure of solutions, or relationships between fixed values in the network? This next set of experiments will investigate this.

Recent work by Geirhos et al. (2020) that demonstrates that deep learning learns so-called "short-cuts" to classifying a particular class of items in the training set. For example, a deep learning network will misclassify an image of a cow when presented on a beach rather than a field, whereas a human would not make the same mistake (Beery et al., 2018), and an object recognition task will prioritize the position of an object in the visual field over the content of the objects in that field (Geirhos et al., 2020). Deep learning demonstrates cognitive biases in learning (Geirhos et al., 2019), that might not always align with human cognitive biases.

TABLE 7 Results of applying a model trained on cubes generated by anticlockwise scrambles to test sets based on clockwise and anticlockwise scrambles

	Size of problem							
	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Tested on clockwise	100	100	97	99	69	31	14	14
Tested on anti-clockwise	16	0	11	4	22	31	15	26

Note: Percentages of correct predictions.

We would expect an intelligent solution to a problem such as learning an LGF for the Rubik's Cube to learn the structure of the problem, rather than learning relationships between colours at specific positions. It seems reasonable that a system that has learned *tout court* that a cube with four sides each with a single row of miscoloured faces is likely to represent a single scramble has found a more intelligent and generalisable solution to the problem than one that has learned that relationship based on lines of specific colours, and is unable to generalize when those colours are permuted.

In this next set of experiments, we will explore these ideas in the kinds of models used in this paper. This will be done by creating a restricted training set, T_a based solely on anticlockwise scrambles. This will then be tested on two test sets: one generated using anticlockwise scrambles, and generated using clockwise scrambles.

The hypothesis being tested is that the deep learning system will not be able to develop a structural understanding from the restricted training set. If it did, it should be able to transfer that understanding from the anticlockwise training to the clockwise test.

In detail, a solved cube c was generated, and the pair (c, 0) added to the (initially empty) training set T_a. This was then scrambled 10 times (n = 1, …, 10) using just anticlockwise scrambles, and after each of these scrambles the pair (c, n) added to T_a. This process was repeated 100,000 times to produce the training set. A model M was then generated using T_a and the network from earlier (Figure 2), again trained over 50 epochs.

The testing was as following. For each of n = 2, …, 9, a testing set was created of 100 cubes generated using random anticlockwise turns only, as was a second testing set in the same way except that clockwise turns were created. Predictions were then made using M, and a count made of how often the model made a correct prediction. The results are presented in Table 7.

Notably, the performance is overall considerably worse for the clockwise test set, though this evens out for more complex problems (7,8,9 scrambles) where the overall performance is lower. This demonstrates, certainly for the regime seen at smaller problem size, that the learning does not generalize from the anticlockwise to the clockwise.

This points towards alternative approaches based on representations that have a more explicit representation of these structural relationships. For example, it would be interesting to explore whether a representation that contains explicit representations of relationships between facelets on the cube, such as program code manipulated with an approach such as genetic programming (GP) (Poli et al., 2008), would make this kind of generalization more possible.

More generally, an *interpretable* representation (Murdoch et al., 2019) such as those generated by GP (at least for small program sizes) would not only be useful for human interpretability, but would also expose the key meaningful ideas used in classification. This would then provide a representation that could be manipulated by another AI system, facilitating transfer learning (Pan & Yang, 2010) to other related problems.

6 | OTHER RELATED WORK

There are similarities between this approach and the idea of a learned value function in reinforcement learning (Sutton & Barto, 2018). However, the reinforcement learning approach calculates this by working back from a reward that has been found during search. In this paper, there is an explicit training set based on trajectories formed by making moves back from a successful state (similar to the approach taken by McAleer et al., 2018). It would be interesting to see if this approach of backwards synthesis of trajectories could be applied to the learning of value functions in reinforcement learning. It is notable that the idea of learning from a rich set of behaviour trajectories—rather than just from a scalar reward function—is becoming more prominent in machine learning, for example in the work by Bojarski et al. (2016) on self-driving cars which learn from examples of human driving.

In the metaheuristics literature the idea of learning from a set of already-solved problems is explored in the idea of *target analysis* (Glover & Greenberg, 1989). This takes a set of solved problems from a problem class, and uses those known solutions to set the parameters of a metaheuristic. This is different to our approach in that it still relies on the metaheuristic operators to avoid local optima in the landscape, whereas the approach in this paper uses those already-solved problems to construct a new landscape based on a metric which is designed to have fewer such local optima. The idea of learning a metric from a large set of examples is explored in the literature on metric learning (Kulis, 2013), and it would be interesting to explore further connections between metric learning and the idea of constructing new fitness functions.

It should be noted that there are algorithms specifically for solving the Rubik's Cube, as summarized in the book by Slocum et al. (2011). However, comparisons with these are less relevant to this paper, which was using the Rubik's Cube as an example to see whether a learning algorithm could learn naïvely from it. The importance of these methods that can learn without explicit human knowledge has been emphasized as an important route towards artificial general intelligence (Silver et al., 2017). More widely, this approach points towards a human-like approach where the

machine learns concepts such as similarity from existing solutions. There is an interesting continuum of learning methods from much contemporary machine learning that attempts to generalize from examples (whether labelled examples as in supervised learning or reward as in reinforcement learning), through the approaches in this paper that learn from solution trajectories, through to expert systems where the inputs data explicitly attempts to articulate what aspects of successful solutions should be used by the machine.

7 | SUMMARY AND FUTURE WORK

We have developed the idea of deep learning for learned guidance functions, and shown how these can then be used as fitness drivers in evolutionary computation. This has been applied to a case study of solving a Rubik's Cube, and shown to have a advantages in terms of frequency of finding a solution and the size of the models needed when compared to a random forest-based LGF; however, the number of generations needed is, for more complex problems, larger. Compared to the work by Agostinelli et al. (2019), the results for smaller problems are comparable but quicker to compute, but the combination of policy and value learning approach in that paper allows reliable solution of more complex problems compared to the approach in this paper that relies solely on value function approximation.

There are a number of areas for future work. Firstly, there is much of scope for optimizing the deep learning system using automated machine learning approaches both to optimize the parameters and the structure of the system (Hutter et al., 2019). Secondly, there are a number of further experiments that would investigate the behaviour further: investigating the frequency of and impact of the reinitialisation in this method, using measures of landscape smoothness to understand the effect of the LGF on the landscape, and experimenting with different population sizes. Finally, there are a large number of other problems to which this approach could be applied, for example, protein folding, and de-noising of audio and video files, audio transcription (Souto-Rico et al., 2020), etc.

CONFLICT OF INTEREST

The author declares that there is no conflict of interest that could be perceived as prejudicing the impartiality of the research reported.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request. The code that supports the findings of this study is available in the supplementary material of this article.

ORCID

Colin G. Johnson https://orcid.org/0000-0002-9236-6581

REFERENCES

Agostinelli, F., McAleer, S., Shmakov, A., & Baldi, P. (2019). Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, *1*, 356–363. https://doi.org/10.1038/s42256-019-0070-z

Asselmeyer, T., Ebeling, W., & Rosé, H. (1996). Smoothing representation of fitness landscapes — The genotype-phenotype map of evolution. *Biosystems*, *39*(1), 63–76.

Beery, S., Van Horn, G., & Perona, P. (2018, October). *Recognition in terra incognita*. Paper presented at Proceedings of the European Conference on Computer Vision (pp. 456–473).

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., & Zieba, K. (2016). End to end learning for self-driving cars. *CoRR*, *abs/1604.07316*. Retrieved from http://arxiv.org/abs/1604.07316

Browne, C., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., & Colton, S. (2012). A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(1), 1–43. https://doi.org/10.1109/TCIAIG.2012.2186810

Culberson, J., & Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, *14*(3), 318–334.

Dobson, C. M. (2003). Protein folding and misfolding. *Nature*, *426*, 884–890.

Erez, T., & Smart, W. D. (2008, September). *What does shaping mean for computational reinforcement learning?* Paper presented at 2008 7th IEEE international conference on Development and learning (pp. 215–219).

Geirhos, R., Jacobsen, J.-H., Michaelis, C., Zemel, R., Brendel, W., Bethge, M., & Wichmann, F. A. (2020). *Shortcut learning in deep neural networks*.

Geirhos, R., Rubisch, P., Michaelis, C., Bethge, M., Wichmann, F. A., & Brendel, W. (2019, February). *Imagenet-trained cnns are biased towards texture; increasing shape bias improves accuracy and robustness*. Paper presented at Proceedings of the International Conference on Learning Representations.

Glover, F., & Greenberg, H. (1989). New approaches for heuristic search: A bilateral linkage with artificial intelligence. *European Journal of Operational Research*, *39*(2), 119–130.

Goodfellow, I., Bengio, Y., & Courville, A. (2017). *Deep learning*. MIT Press.

Grefenstette, J. (1986). Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, *16*, 122–128.

Hopgood, A. A., & Mierzejewska, A. (2009). Transform ranking: A new method of fitness scaling in genetic algorithms. In M. Bramer, M. Petridis, & F. Coenen (Eds.), *Research and development in intelligent systems XXV* (pp. 349–354). Springer.

Hutter, F., Kotthoff, L., & Vanschoren, J. (Eds.). (2019). *Automated machine learning: Methods, systems, challenges*. Springer.

Johnson, C. G. (2018). Solving the Rubik's cube with learned guidance functions. In *Proceedings of the 2018 IEEE Symposium Series in Computational Intelligence*. IEEE Press.

Johnson, C. G. (2019, December). Stepwise evolutionary learning using deep learned guidance functions. In M. Bramer & M. Petridis (Eds.), *Artificial Intelligence XXXVI: 39th SGAI International Conference on Artificial Intelligence, AI 2019, Cambridge, UK, Proceedings* (pp. 50–62). Springer.

Keras. (n.d.). *The python deep learning library*. Retrieved from http://keras.io/

Krawiec, K. (2016). *Behavioural program synthesis with genetic programming*. Springer.

Krawiec, K., Swan, J., & O'Reilly, U.-M. (2016). Behavioral program synthesis: Insights and prospects. In R. Riolo, B. Worzel, M. Kotanchek, & A. Kordon (Eds.), *Genetic programming theory and practice XIII* (pp. 169–183). Springer.

Kreinovich, V., Quintana, C., & Fuentes, O. (1993). Genetic algorithms: What fitness scaling is optimal? *Cybernetics and Systems*, *24*, 9–26.

Kulis, B. (2013). Metric learning: A survey. *Foundations and Trends® in. Machine Learning*, *5*(4), 287–364. https://doi.org/10.1561/2200000019

McAleer, S., Agostinelli, F., Shmakov, A., & Baldi, P. (2018, May). Solving the Rubik's cube without human knowledge. *ArXiv e-prints*. Retrieved from https://arxiv.org/abs/1805.07470

Mehdi Samadi, J. S., Felner, A., & Schaeffer, J. (2008). Learning from multiple heuristics. In D. Fox & C. P. Gomes (Eds.), *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence - Volume 1*, (pp. 357–362). California, CA: Association for the Advancement of Artificial Intelligence.

Moraglio, A., Krawiec, K., & Johnson, C. G. (2012). Geometric semantic genetic programming. In C. Coello Coello, V. Cutello, K. Deb, S. Forrest, G. Nicosia, M Pavone. (Eds.), *Parallel problem solving from nature—PPSN XII* (pp. 21–31). Springer.

Murdoch, W. J., Singh, C., Kumbier, K., Abbasi-Asl, R., & Yu, B. (2019). Definitions, methods, and applications in interpretable machine learning. *Proceedings of the National Academy of Sciences of the United States of America*, *116*(44), 22071–22080. https://doi.org/10.1073/pnas.1900654116

Pan, S., & Yang, Q. (2010). A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, *22*(10), 1345–1359.

Poli, R., Langdon, W. B., & McPhee, N. F. (2008). *A field guide to genetic programming*. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk (With contributions by J. R. Koza).

scikit-learn. (n.d.). Machine learning in python. Retrieved from http://scikit-learn.org/

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., & Hassabis, D. (2017). Mastering the game of go without human knowledge. *Nature*, *550*(7676), 354–359. https://doi.org/10.1038/nature24270

Singmaster, D. (1981). *Notes on Rubik's magic cube*. Enslow Publishing.

Slocum, J., Singmaster, D., Gebhardt, D., Huang, W. -H., & Hellings, G. (2011). *The cube: The ultimate guide to the world's best-selling puzzle*. Black Dog and Leventhal.

Souto-Rico, M., González-Carrasco, I., López-Cuadrado, J.-L., & Ruíz-Mezcua, B. (2020). A new system for automatic analysis and quality adjustment in audiovisual subtitled-based contents by means of genetic algorithms. *Expert Systems*, *37*, e12512. https://doi.org/10.1111/exsy.12512.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*(1), 1929–1958.

Sturtevant, N. R., Felner, A., Barrer, M., Schaeffer, J., & Burch, N. (2009). Memory-based heuristics for explicit state spaces. In *Proceedings of the 21st International Joint Conference on Artifical Intelligence* (pp. 609–614). Morgan Kaufmann Publishers Inc.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (2nd ed.). MIT Press.

Szubert, M., Jaśkowski, W., Liskowski, P., & Krawiec, K. (2013). Shaping fitness function for evolutionary learning of game strategies. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation* (pp. 1149–1156). ACM.

Tensorflow. (n.d.). *An open source machine learning framework for everyone*. Retrieved from http://www.tensorflow.org/

Vanneschi, L., Castelli, M., Manzoni, L., & Silva, S. (2013). A new implementation of geometric semantic GP and its application to problems in pharmacokinetics. In K. Krawiec et al. (Eds.), *Genetic programming* (pp. 205–216). Springer Berlin Heidelberg.

Ware, J. M., Wilson, I. D., & Ware, J. A. (2003). A knowledge based genetic algorithm approach to automating cartographic generalisation. In A. Macintosh, R. Ellis, & F. Coenen (Eds.), *Applications and innovations in intelligent systems X* (pp. 33–49). Springer.

Widera, P., Garibaldi, J. M., & Krasnogor, N. (2010). GP challenge: Evolving energy function for protein structure prediction. *Genetic Programming and Evolvable Machines*, *11*(1), 61–88

AUTHOR BIOGRAPHY

Colin G. Johnson is an Associate Professor in the School of Computer Science at the University of Nottingham, UK. He received his PhD from the University of Kent in 2003. His research interests are in artificial intelligence and machine learning, including work in genetic programming and neural networks, and the application of those techniques to a wide variety of topics in the natural science, engineering, and the digital humanities.

SUPPORTING INFORMATION

Additional supporting information may be found online in the Supporting Information section at the end of this article.

How to cite this article: Johnson CG. Solving the Rubik's cube with stepwise deep learning. *Expert Systems*. 2021;e12665. https://doi.org/10.1111/exsy.12665