Stepwise Evolutionary Learning using Deep Learned Guidance Functions

Colin G. Johnson[0000−0002−9236−6581]

[1] School of Computing, University of Kent, Canterbury, Kent, UK
C.G.Johnson@kent.ac.uk
[2] IASH, University of Edinburgh, Edinburgh, UK Colin.Johnson@ed.ac.uk

Abstract. This paper explores how Learned Guidance Functions (LGFs)—a pre-training method used to smooth search landscapes—can be used as a fitness function for evolutionary algorithms. A new form of LGF is introduced, based on deep neural network learning, and it is shown how this can be used as a fitness function. This is applied to a test problem: unscrambling the Rubik's Cube. Comparisons are made with a previous LGF approach based on random forests, and with a baseline approach based on traditional error-based fitness.

1 Introduction

The aim of this paper is to present a new kind of fitness function in evolutionary 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. This new kind of fitness is based on *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.

The fitness function is one of the key components of evolutionary algorithms. Typically, a fitness function is either a domain-specific *loss function*, measuring how far a particular population member is from being a solution, or a ranking function that allows the comparison of two population members, returning the fittest. This is one of the powerful aspects of evolutionary algorithms—we can specify a problem by giving a single, simple function that allows us to choose between population members.

There are problems with such an approach. Most obviously, such functions typically have many local minima. Typically, this is seen as an intrinsic part of the problem, to be solved by the search process. 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. The focus of this work is typically on improvements to the search process. However, another strand of work is focused on transformations to the fitness function itself. This has a long history in the evolutionary computation, typifed by work on *fitness scaling* [11,17,12,30]. The principle aim of fitness

scaling prevent premature convergence of the search algorithm, by composing a scaling function with the fitness function that doesn't 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) [21] have which attempts to reconfigure the problem so that a much simpler search process such as hillclimbing can be used. The "intelligence" in these approaches is in this initial phase of reconfiguring the problem. However, these approaches have sometimes traded off this simplicity of search against another kind of complexity; 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 [29]. This idea of 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 [4].

Another form of smoothing the search landscape is in the form of *pattern databases* [6]. These consist of patterns in the search space such that the patterns have 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. This has a similar idea of transforming the search space to the above work, but it is different because the pattern databases are produced based on domain knowledge. Some work has used learning methods to generalise from pattern databases—for example, by using neural networks to learn how pattern databases can be combined [20]

Another important source of inspiration is the view that traditional fitness functions take a very narrow view of the problem; whilst 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 [15,16]. 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 [7]. 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 [31].

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.

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 [28] and reinforcement learning [8], 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 [14]. 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* [26], 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 [19]).

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 a 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.

2.1 Formalisation

Now we formalise 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 $T = \{T_1, T_2, ...T_{n_T}\}$, where each trajectory is a set of points in S, i.e. $T_i = [s_1, s_2, ..., s_{n_{T_i}}]$, 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 [14] we applied the LGF to the problem of unscrambling the Rubik's Cube. We used a number of classifiers from the *scikit-learn* library [2] to implement LGFs, and demonstrated that (1) the LGF can learn to recognise 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, 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, $\{F, B, R, L, U, D, F', B', R', L', U', D'\}$ [23], which are functions from C to C.

This paper presents two new aspects compared to the previous one. Firstly, we introduced a new approach to learning the LGFs, based on deep learning [10]. Secondly, we apply a population-based approach to this problem, based around an evolution strategy, rather than the hillclimbing approach used in the previous paper.

3.1 Constructing the LGF

The LGF for this problem is constructed as follows (pseudocode in 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 = ∅
4:     for s ∈ [0, ..., n_s − 1] do
5:         let c be a new cube in the solved state
6:         for ℓ ∈ [0, ..., n_m − 1] do
7:             let X = X ∪ {(c, ℓ)}
8:             let m = random element from M
9:             let C = m(c)
10:        end for
11:    end for
12:    return T
13: end procedure
```

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 [1] on TensorFlow [3]. The specific network used is illustrated in Figure 2. This is a fairly standard deep learning network, with dropout [25] used to encourage generalisation 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 optimise the model produced [13].

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. 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; (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.

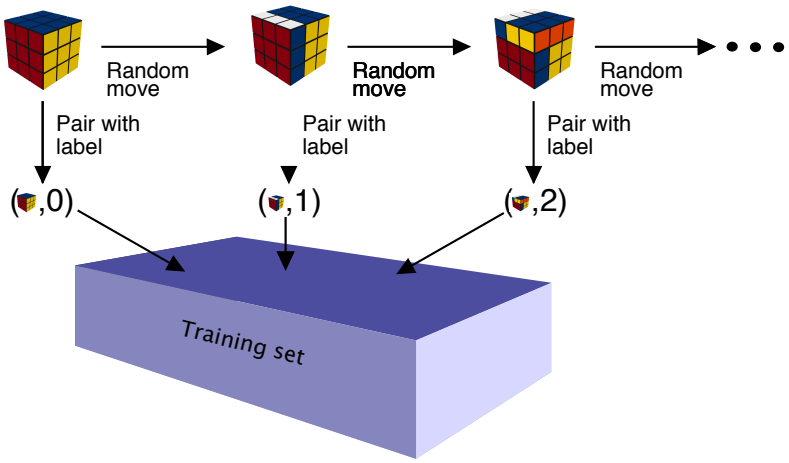

Fig. 1. Construction of the training set (modified from [14]).

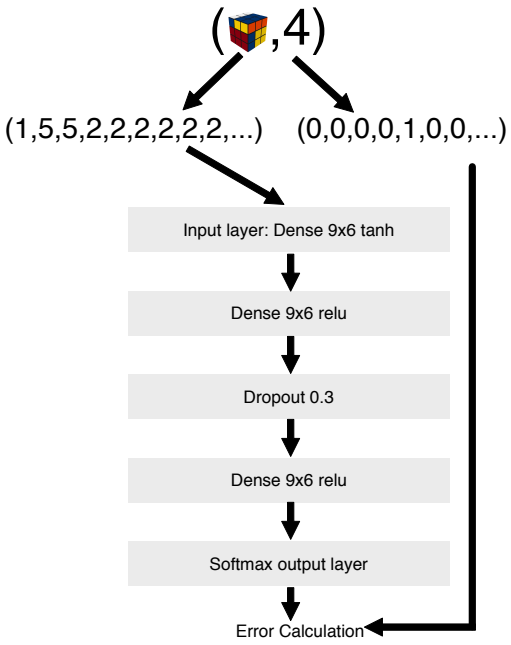

Fig. 2. Keras deep learning network used for training.

This is summarised in pseudocode in 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.

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 in [14]). 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.

4 Experiments and Results

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

- Size of problem $n_\ell = n_\ell$
- Population size $n_p = 100$
- Maximum number of generations $\theta = 100$
- 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 three hours, not including time to train the models (training time was between 11s–59s 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

Algorithm 2 Scrambling/unscrambling algorithm for the Rubik's cube

```
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 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:    let P = n_p copies of c
11:    for t = 0; t < θ; t = t + 1 do
12:        let P′ = ∅
13:        for p ∈ P do m = random element from M
14:            let P′ = P′ ∪ m(p)
15:            if m(p) is the solved state then
16:                return p                          ▷ Problem Solved
17:            end if
18:        end for
19:        let P″ = ∅
20:        for p ∈ P′ do
21:            if L(p) < ℓ then
22:                let P″ = P″ ∪ p
23:            end if
24:        end for
25:        if P″ == ∅ then
26:            let P = n_p copies of c               ▷ Reinitialise
27:            let ℓ = n_ℓ
28:        else
29:            let P = ∅
30:            for n = 0; n < n_p; n = n + 1 do
31:                P = P∪ random member of P″
32:            end for
33:            let ℓ = ℓ − 1
34:        end if
35:    end for
36:    return null                                   ▷ Timed out
37: end procedure
```

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.

Table 1. Percentage of times unscrambling problem of each size was solved using a model of each size. Results from 100 runs.

Size of Model	Size of Problem											
	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
2	100	-	-	-	-	-	-	-	-	-	-	-
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

Table 3 and Table 4 compare the results to two experiments in a previous paper [14]. 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 [2]). 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 a simple hill-climbing approach rather than the ES used in this paper.

These tables also contain a comparison with a baseline experiment also described in detail in the earlier paper [14] (Error + Hillclimbing). This also uses a simple hill-climbing method, but the choice of moves is made by choosing the move that maximises the number of correct facelets. This is more similar to 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

Table 2. Average number of generations needed to solve problem of each size using trained model of each size. Includes successful solutions only, and includes restarts.

		Size of Problem											
		2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Size of Model	**2**	1.0	-	-	-	-	-	-	-	-	-	-	-
	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

approach; in the earlier paper, a search that did not terminate was considered a failure. Both methods clearly outperform the traditional error-based fitness measure, demonstrating the value of this pre-training step.

Table 3. Comparison of three models: deep learning of LGF with evolution strategies, random forest learning of LGF and hillclimbing, and error-based fitness with hillclimbing. Percentage of runs that found the solution.

	Size of Problem										
	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Deep LGF + ES (this paper)	100	100	100	90	77	61	63	50	37	22	17	6
RF LGF + Hillclimbing [14]	100	100	98	75	62	45	20	17	11	9	7	0
Error + Hillclimbing [14]	62	33	24	10	4	3	2	0	0	0	1	0

5 Related Work

There are similarities between this approach and the idea of a learned value function in reinforcement learning [27]. However, the reinforcement learning approach calculates this by starting from a point in the space and working back from later successes, whereas the approach in this paper constructs trajectories by making moves back from a successful state (similar to the approach taken by McAleer et al. [19]). It would be interesting to see if this approach of backwards synthesis of trajectories could be applied to the learning of value functions in

Table 4. Comparison of three models: deep learning of LGF with evolution strategies, random forest learning of LGF and hillclimbing, and error-based fitness with hillclimbing. Average length to solution for successful runs.

	Size of Problem											
	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Deep LGF + ES (this paper)	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 [14]	1.8	2.6	3.3	3.9	4.2	4.6	4.8	5.5	4.9	4.6	4.9	-
Error + Hillclimbing [14]	2.1	2.2	2.3	2.8	4.5	3.7	2.0	-	-	-	4.0	-

reinforcement learning. It is notable that the idea of learning from a rich set of behaviour trajectories—rather than just from a single measure of quality—is becoming more prominent in machine learning, for example in the work by Bojarski et al. [5] 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* [9]. 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 [18], 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 summarised in the book by Slocum et al. [24]. 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 emphasised as an important route towards artificial general intelligence [22].

6 Summary and Future Work

We have introduced 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. It would be interesting to explore the comparative impact of the deep learning aspects and the evolutionary computation aspects by doing more experiments that use these two separately.

There are a number of areas for future work. Firstly, there is much of scope for optimising the deep learning system using automated machine learning approaches both to optimise the parameters and the structure of the system[13]. 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, e.g. protein folding, and de-noising of audio and video files.

References

1. Keras: The python deep learning library, http://keras.io/ (visited January 2019)
2. scikit-learn: Machine learning in python, http://scikit-learn.org/ (visited January 2019)
3. Tensorflow: An open source machine learning framework for everyone, http://www.tensorflow.org/ (visited January 2019)
4. Asselmeyer, T., Ebeling, W., Rosé, H.: Smoothing representation of fitness landscapes — the genotype-phenotype map of evolution. Biosystems **39**(1), 63–76 (1996)
5. Bojarski, M., et al.: End to end learning for self-driving cars. CoRR **abs/1604.07316** (2016), `http://arxiv.org/abs/1604.07316`
6. Culberson, J., Schaeffer, J.: Pattern databases. Computational Intelligence **14**(3), 318–334 (1998)
7. Dobson, C.M.: Protein folding and misfolding. Nature **426**, 884–890 (2003)
8. Erez, T., Smart, W.D.: What does shaping mean for computational reinforcement learning? In: 2008 7th IEEE International Conference on Development and Learning. pp. 215–219 (2008)
9. Glover, F., Greenberg, H.: New approaches for heuristic search: A bilateral linkage with artificial intelligence. European Journal of Operational Research **39**(2), 119–130 (1989)
10. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2017)
11. Grefenstette, J.: Optimization of control parameters for genetic algorithms. IEEE Transactions on Systems, Man, and Cybernetics **16**, 122–128 (1986)
12. Hopgood, A.A., Mierzejewska, A.: Transform ranking: a new method of fitness scaling in genetic algorithms. In: Bramer, M., Petridis, M., Coenen, F. (eds.) Research and Development in Intelligent Systems XXV. pp. 349–354. Springer (2009)
13. Hutter, F., Kotthoff, L., Vanschoren, J.: AutoML: Methods, Systems, Challenges (2019), book in preparation. Current draft at https://www.automl.org/book/ (visited July 2019)
14. Johnson, C.G.: Solving the Rubik's Cube with learned guidance functions. In: Proceedings of the 2018 IEEE Symposium Series in Computational Intelligence. IEEE Press (2018)
15. Krawiec, K.: Behavioural Program Synthesis with Genetic Programming. Springer (2016)
16. Krawiec, K., Swan, J., O'Reilly, U.M.: Behavioral program synthesis: Insights and prospects. In: Riolo, R., Worzel, B., Kotanchek, M., Kordon, A. (eds.) Genetic Programming Theory and Practice XIII. pp. 169–183. Springer (2016)

17. Kreinovich, V., Quintana, C., Fuentes, O.: Genetic algorithms: What fitness scaling is optimal? Cybernetics and Systems **24**, 9–26 (1993)
18. Kulis, B.: Metric learning: A survey. Foundations and Trends® in Machine Learning **5**(4), 287–364 (2013). https://doi.org/10.1561/2200000019, http://dx.doi.org/10.1561/2200000019
19. McAleer, S., Agostinelli, F., Shmakov, A., Baldi, P.: Solving the Rubik's Cube Without Human Knowledge. ArXiv e-prints (May 2018)
20. Mehdi Samadi, Ariel Felner, J.S.: Learning from multiple heuristics. In: Proceedings of Association for the Advancement of Artificial Intelligence (AAAI-08). pp. 357–362 (2008)
21. Moraglio, A., Krawiec, K., Johnson, C.G.: Geometric semantic genetic programming. In: Coello Coello, C.A., et al. (eds.) Parallel Problem Solving from Nature - PPSN XII: 12th International Conference, Taormina, Italy, September 1-5, 2012, Proceedings, Part I. pp. 21–31. Springer Berlin Heidelberg (2012)
22. Silver, D., et al.: Mastering the game of go without human knowledge. Nature **550**(7676), 354–359 (2017), http://dx.doi.org/10.1038/nature24270
23. Singmaster, D.: Notes on Rubik's Magic Cube. Enslow Publishing, Hillside, NJ (1981)
24. Slocum, J., et al.: The Cube: The Ultimate Guide to the World's Best-Selling Puzzle. Black Dog and Leventhal (2011)
25. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research **15**(1), 1929–1958 (2014)
26. Sturtevant, N.R., Felner, A., Barrer, M., Schaeffer, J., Burch, N.: Memory-based heuristics for explicit state spaces. In: Proceedings of the 21st International Joint Conference on Artifical Intelligence. pp. 609–614. IJCAI'09, Morgan Kaufmann Publishers Inc. (2009)
27. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press (1998)
28. Szubert, M., Jaśkowski, W., Liskowski, P., Krawiec, K.: 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 (2013)
29. Vanneschi, L., Castelli, M., Manzoni, L., Silva, S.: A new implementation of geometric semantic gp and its application to problems in pharmacokinetics. In: Krawiec, K., et al. (eds.) Genetic Programming. pp. 205–216. Springer Berlin Heidelberg (2013)
30. Ware, J.M., Wilson, I.D., Ware, J.A.: A knowledge based genetic algorithm approach to automating cartographic generalisation. In: Macintosh, A., Ellis, R., Coenen, F. (eds.) Applications and Innovations in Intelligent Systems X. pp. 33–49. Springer (2003)
31. Widera, P., Garibaldi, J.M., Krasnogor, N.: Gp challenge: evolving energy function for protein structure prediction. Genetic Programming and Evolvable Machines **11**(1), 61–88 (2010)