# Self-Referential Meta Learning

**Anonymous Authors**

## Abstract

Meta Learning automates the search for learning algorithms. At the same time, it creates a dependency on human engineering on the meta-level, where meta learning algorithms need to be designed. In this paper, we investigate self-referential meta learning systems that modify themselves without the need for explicit meta optimization. We discuss the relationship of such systems to memory-based meta learning and show that self-referential neural networks require functionality to be reused in the form of parameter sharing. Finally, we propose Fitness Monotonic Execution (FME), a simple approach to avoid explicit meta optimization. A neural network self-modifies to solve bandit and classic control tasks, improves its self-modifications, and learns how to learn, purely by assigning more computational resources to better performing solutions. [1]

## 1. Introduction

Machine learning is the process of deriving models and behavior from data or environment interaction using human-engineered learning algorithms. Meta learning takes this process to the meta-level: Its goal is to derive the learning algorithms themselves automatically as well (Schmidhuber, 1987; Hochreiter et al., 2001; Wang et al., 2016; Duan et al., 2016; Finn et al., 2017; Flennerhag et al., 2019; Kirsch et al., 2019; Kirsch & Schmidhuber, 2020). Unfortunately, this creates a dependency on human engineering on the meta-level, where researchers now have to design meta learning algorithms. In this paper, we investigate methods for neural self-referential meta learning (Schmidhuber, 1992b; Schmidhuber et al., 1997). In particular, we seek a process of self-improvement that reduces our reliance on human engineering to the largest extent possible.

A central piece in the discussion of self-referential meta learning is the self-referential neural architecture (Schmidhuber, 1992b). Such an architecture allows the modification not just of some memory to improve future behavior, but also to modify all its own neural weights. This enables self-modification and self-improvement allowing the architecture to learn, meta-learn, meta-meta-learn, etc. We show that in order to construct systems that can change all their parameters (or variables more generally), functionality needs to be reused. This is done in the form of parameter (variable) sharing. We further demonstrate the representational equivalence between neural networks with memory and self-referential architectures while highlighting the benefit of self-referential architectures in the absence of meta optimization.

Finally, we propose Fitness Monotonic Execution (FME), a simple approach to avoid explicit meta optimization. Instead of proposing changes to the model or learning algorithm explicitly, all changes to the model are self-modifications and the resulting solutions are selected for execution more frequently the better their performance. We empirically demonstrate FME with a neural network that self-modifies to solve bandit and classic control tasks, improves its self-modifications, and learns how to learn.

## 2. Background

### 2.1. Self-referential Architectures

A key requirement for a self-improving system is that it can make self-modifications such that it can change its behavior and learning arbitrarily. One previously suggested way of achieving this is to bring all variables in a neural network under control of the network itself (Schmidhuber, 1992b). This is referred to as a self-referential neural architecture. Compare this to a conventional neural network where there is a subset of variables (called the weights or parameters) that are only updated by a fixed learning algorithm (such as backpropagation). This entails that part of the (meta-)learning behavior is fixed and needs to be defined by the researcher.

In contrast, self-referential architectures control all variables. This includes activations (conventionally updated by the neural network itself), weights (conventionally updated by a learning algorithm), meta weights, etc. In this section, we discuss necessary conditions on such a self-referential architecture and possible implementations.

[1] Code will be made available with the final version

**Notation** In the remainder of this paper, we denote external inputs to the neural network as $x \in \mathbb{R}^{N_x}$ (such as observations and rewards in Reinforcement Learning, or error signals in supervised learning), outputs as $y \in \mathbb{R}^{N_y}$ (e.g. actions in Reinforcement Learning), and the parameters of the neural network as $\theta$. Further, we denote time-varying variables (memory) as $h \in \mathbb{R}^{N_h}$ (such as the hidden state of an RNN). We summarize all variables in a neural network as $\phi = \{\theta, h, y\}$.

**Necessary Conditions** A self-referential architecture $\phi \leftarrow g_\phi(x)$ is described by a connected computational graph for the function $g$ that has variables $\phi = \{\theta, h, y\}$ (one node per scalar). The network controls all of its variables in the sense that any element(s) of $\phi$ can be changed through network actions at every iteration. This blurs the distinction between activations (memory) $h$ and weights $\theta$. Computational graphs that fulfill this definition are required to have a certain structure. At every iteration at least some variables need to be reused in multiple operations (node out-degree $> 1$). We refer to this as variable sharing, a generalization of weight sharing (Fukushima, 1979) extending beyond classical weights.

Consider a square dense weight matrix. It consists of $N^2$ weights and $N$ activations. While the activations are time-varying, the weights are source nodes in the computational graph and cannot be directly updated by actions of the network itself. To change that, we need to derive $N^2$ variables from $N$ time-varying variables. This can only be done by reusing some of the same $N$ time-varying variables in multiple operations generating the $N^2$ variables.

*Observation* 2.1. Variable sharing in self-referential systems. Assuming a connected computational graph, a self-referential architecture that updates all its variables $\phi \in \mathbb{R}^{N_\phi}$ in iteration $t$ needs to reuse elements of $\phi$ multiple times in the computational graph to generate $\phi_{t+1} \in \mathbb{R}^{N_\phi}$ from $\phi_t \in \mathbb{R}^{N_\phi}$.

*Proof.* As there are no more elements in $\phi_t$ than there are in $\phi_{t+1}$, any operation generating an element in $\phi_{t+1}$ that makes use of more than one element in $\phi_t$ needs to reuse an element already in use by a different operation. □

**Implementations** Under the previous constraints, various implementations for self-referential neural architectures are conceivable. Schmidhuber (1992b) assigns an address to each weight such that the network outputs can be used to attend to weights and both read and write their values. Instead of updating one weight at a time, the fast weight programmers of 1992-93 (Schmidhuber, 1992a; 1993) are networks that learn to generate key and value patterns to rapidly change many fast weights simultaneously. Outer products between activations (a type of sharing) are used to derive $M * N$ variables, $M, N \in \mathbb{N}$, from $M + N$ variables. This allows updating all the weights of a neural network layer by its own activations (Irie et al., 2021). Alternatively, a coordinate-wise mechanism may generate all updates continuously as a function of the weight address (D'Ambrosio & Stanley, 2007). Other works have used multiple RNNs with shared weights and messaging passing between those to increase the number of time-varying variables $h$ arbitrarily while keeping the number of parameters $\theta$ constant (Rosa et al., 2019; Kirsch & Schmidhuber, 2020). This allows the use of a subset of $h$ to update parameters $\theta$.

### 2.2. Expressivity of Memory and Self-referential Architectures

In this section, we show that self-referential architectures do not have a representational advantage over memory-based architectures when the free (initial) variables are meta-optimized using a human engineered learning algorithm.

We defined self-referential architectures $\phi, y \leftarrow g_\phi(x)$ as those that can update all their variables $\phi$ in the computational graph. Compare this to a memory architecture such as a recurrent neural network $h, y \leftarrow f_\theta(h, x)$ parameterized by $\theta$ where $h$ corresponds to its hidden state (memory). Can the self-referential architecture represent any functions that the memory architecture can not? A commonly used intuition (Schmidhuber, 1992b) is that self-referential architectures are self-modifying, in that they change their own weights, affecting not only their outputs and current weights but also future weight changes through $g_\phi$. These architectures can thus not only learn, but also meta-learn, meta-meta-learn, etc. While memory architectures do not update their weights, they are also self-modifying. Changes in memory $h$ affect the output directly, but also the effective function $f_\theta$ by modifying its input $h$, in turn determining future changes to $h$.

*Observation* 2.2. For any self-referential architecture $\phi, y \leftarrow g_\phi(x)$ and some initial $\phi_0$ we can find a memory architecture $h, y \leftarrow f_\theta(h, x)$, $\theta$, and initial $h_0$ such that for any sequence of $x_{1:T}$ we have $\hat{f}(x_{1:T}) = \hat{g}(x_{1:T})$ where $\hat{f}$ and $\hat{g}$ are the unrolls returning $y_{1:T}$ of $f$ and $g$ respectively.

*Proof.* We construct an emulator $f_\theta$ (a sufficiently large neural network parameterized by $\theta$) that stores $\phi$ in $h$ and set $\theta, h_0$ such that at each step $t \in \mathbb{N}$ it performs the same computation as $g_\phi$ by taking $\phi_t$ from $h_t$, computing $\phi_{t+1}$, and storing it in $h_{t+1}$. □

Furthermore, any memory-based architecture can be represented by a self-referential architecture where a subset of variables is updated by the identity function. In conclusion, the function class that can be represented by self-referential architectures is equivalent to memory architectures, given

sufficiently rich parameterizations. For both memory architectures and self-referential architectures the same question arises: How do we set the free variables $\theta$, $h_0$, or $\phi_0$? If these free variables are directly optimized (eg by following the gradient of some objective), from a representability perspective there is no advantage of self-referential architectures. In the following, we discuss how self-referential architectures are relevant in the absence of direct meta-optimization.

## 3. Method: Fitness Monotonic Execution

Both in the case of self-referential and memory architectures the free (initial) variables need to be found. Here we propose a method, *Fitness Monotonic Execution* (FME), that avoids explicit meta optimization of these free variables. Instead of modifying $\phi$ directly using a human-engineered learning algorithm, we simply select between different configurations of $\phi$ that are generated using self-modifications. In particular, through interactions with the environment we continuously add new solutions to a set of $\Phi = \{\phi_i\}$. Computation time is distributed across solutions monotonic in their performance, i.e. better performing solutions are executed longer (or are selected for execution more frequently). This can be formalized as a pmf $p(\phi)$ that assigns each solution $\phi \in \Phi$ a probability for being executed at any given time-step based on its average reward $\frac{R(\phi)}{\Delta t}$ relative to other solutions (where $\Delta t$ is the solution's total lifetime). To rank better performing solutions over worse ones, the pmf must be strictly monotonic. As a special case, $p$ may put all probability mass on the current best solution, greedily selecting for improvement. See Algorithm 1 for a full description.

Note that in this scheme, memory architectures are now inadequate. If there were any variables that are not modifiable, their value could not be determined through self-modifications. As we do not use any human-engineered optimization process, their value would be undefined. Thus, self-referential architectures are required.

**Algorithm 1** Fitness monotonic execution

**Require:** Initial solution(s) $\Phi = \{\phi_i\}$, self-referential architecture $g_\phi$, probability $p(\phi)$, an RL environment $E$
**while** forever **do**
  $\phi \sim p(\phi)$ where $\phi \in \Phi$ ▷ Sample next solution to execute, monotonic in its performance
  $\phi, y_{1:L} \leftarrow g_\phi^L(x_{1:L})$ ▷ Execute $g$ for $L$ steps with $x_{1:L}$ from the environment $E$ including a feedback signal
  $\Phi \leftarrow \Phi \cup \{\phi\}$ ▷ Add new $\phi$ to $\Phi$

**Least-recently-used Buffer** To limit the number of solutions we need to store, we implement Fitness Monotonic Execution with a least-recently-used (LRU) buffer. It consists of $m$ buckets where each bucket holds recent solutions in a specific performance range. Solutions from buckets with higher performance are sampled exponentially more frequently.

**Outer-product-based Architecture** For the self-referential neural network architecture, we chose an outer-product mechanism adapted from prior work (Irie et al., 2021). By applying a weight matrix $W_{t-1} \in \mathbb{R}^{N_x \times (N_y + 2N_x + 4)}$ to some input $x_t \in \mathbb{R}^{N_x}$ we generate the output $y_t \in \mathbb{R}^{N_y}$, key $k_t \in \mathbb{R}^{N_x}$, query $q_t \in \mathbb{R}^{N_x}$, and a learning rate $\beta_t \in \mathbb{R}^4$. Using an outer-product, the key and query generate an update to the weight matrix $W_{t-1}$, obtaining $W_t$:

$$y_t, k_t, q_t, \beta_t = W_{t-1}\psi(x_t) \tag{1}$$
$$\bar{v}_t = W_{t-1}\psi(k_t) \tag{2}$$
$$v_t = W_{t-1}\psi(q_t) \tag{3}$$
$$W_t = W_{t-1} + \sigma(\beta_t)(\psi(v_t) - \psi(\bar{v}_t)) \otimes \psi(k_t) \tag{4}$$

where $\psi$ is the tanh activation, $\sigma$ is the sigmoid function, and $\otimes$ is the outer product. The learning rate $\beta_t \in \mathbb{R}^4$ controls the rate of update to the four parts generating $y, k, q, \beta$. We stack multiple such self-referential layers.

## 4. Experiments

We empirically investigate several questions: Firstly, starting with a randomly initialized solution, can the network modify itself to solve a bandit task? We compare this to a hill climbing strategy. Secondly, how do self-modifications compare when solving a markov decision process? Thirdly, given a bandit task that is non-stationary, can the network learn to modify itself based on the reward it receives as input?

**Learning a Bandit Policy** The first question we investigate is whether a randomly initialized self-referential architecture is capable of making self-modifications that lead to a useful policy for a given task. We test this on a simple 2-armed bandit where one arm gives payouts (rewards) of 1 and the other 0. From Figure 1 (left) we observe that after around 40 self-modifications and selections a solution is found that always selects the arm with a higher payoff. We compare this to hill climbing with a variance-tuned Gaussian noise on the network parameters. We observe that in this simple environment, Fitness Monotonic Execution is as effective as hill climbing to find an optimal solution to this bandit problem.

**Cartpole** Next, we increase the difficulty of the policy to be found by running a self-referential network on the

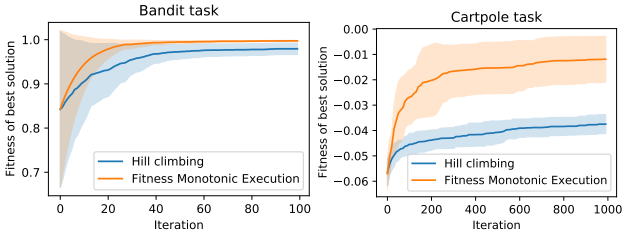


*Figure 1.* A randomly initialized self-referential architecture makes modifications to itself to solve a two-armed bandit problem (left). On a Cartpole task (right) the self-modifications not only directly improve the policy, but also improve future improvements, resulting in faster learning compared to hill climbing. The found policy balances the pole for about 100 steps. Standard deviations are shown for 5 seeds.

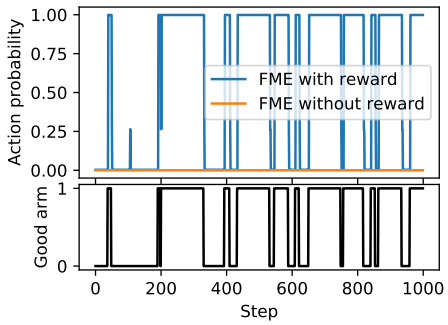


*Figure 2.* Given the reward as input, self-modifications enable adaptation to swapping of the good arm in a two-armed bandit.

CartPole task (Figure 1, right). We observe that reaching a good performing policy takes significantly more self-modifications and selections. At the same time, a simple hill climbing strategy (with tuned noise) fails at improving the policy at the same rate as the self-modifying architecture. This suggests that we are not only selecting for good policies but also strategies for self-modification that lead to policy improvement in the future.

**Meta Learning a Bandit Learning Algorithm** Given a non-stationary task, a good policy can not exhibit a fixed behavior but must adapt to changing rewards (learn). We test the capabilities of Fitness Monotonic Execution to adapt to a changing bandit task. In Figure 2 we swap the good and bad arm at random intervals. We further feed the reward as an input to the policy such that it can adapt its behavior based on the reward. We observe that Fitness Monotonic Execution leads to self-modifying policies that change their action (learn) in response to the reward they previously received. In contrast, if this reward is not fed as an input, the policy fails to adapt.

## 5. Discussion

**Limitations** This paper represents an initial discussion of self-referential systems that do not rely on fixed human-engineered meta-optimization. The empirical evaluation is still minimalistic at this time but should be a good starting point for larger experiments and improvements. Our intention is to incite further interest in this research direction. Semantically, it is an open question whether FME should be called a (minimalistic) meta optimizer after all. Usually, optimizers define how solutions are modified based on a feedback signal. In the case of FME, these modifications are self-generated.

**Broader impact** This paper is not directly concerned with applications of machine learning. Nevertheless, meta learning methods may pose additional challenges in the future. For instance, learning algorithms are now also subject to learning which means that learning itself becomes more difficult to interpret and to monitor for biases extracted from data.

**Self-modifying architecture** We described a meta learner that can self-modify all its variables including those that define the self-modifications, but its architecture is still hard-coded. In fitness monotonic execution, the self-modifications do not require differentiability. Thus, self-modifications can be extended to include architecture modifications $\phi, y, g \leftarrow g_\phi(x)$, such as adding or removing neurons and weights, changing operations, and (un-)sharing variables.

## 6. Conclusion

In this paper, we discussed self-referential systems that exhibit self-improvement while reducing the reliance on human engineering to the largest extent possible. In particular this means avoiding the use of human engineered learning algorithms on the meta level. We showed that in order to construct systems that can change all their parameters (or variables more generally), functionality needs to be reused. This is done in the form of parameter (variable) sharing. We further demonstrated the representational equivalence between neural networks with memory and self-referential architectures while highlighting the benefit of self-referential architectures in the absence of meta optimization. Finally, we proposed Fitness Monotonic Execution (FME), a simple approach to avoid explicit meta optimization. A neural network self-modifies to solve bandit and classic control tasks, improves its self-modifications, and learns how to learn, purely by assigning more computational resources to better performing solutions.

## References


D'Ambrosio, D. B. and Stanley, K. O. A novel generative encoding for exploiting neural network sensor and output geometry. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp. 974–981, 2007.

Duan, Y., Schulman, J., Chen, X., Bartlett, P. L., Sutskever, I., and Abbeel, P. $Rl^2$: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

Finn, C., Abbeel, P., and Levine, S. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pp. 1126–1135. PMLR, 2017.

Flennerhag, S., Rusu, A. A., Pascanu, R., Visin, F., Yin, H., and Hadsell, R. Meta-learning with warped gradient descent. *arXiv preprint arXiv:1909.00025*, 2019.

Fukushima, K. Neural network model for a mechanism of pattern recognition unaffected by shift in position-neocognitron. *IEICE Technical Report, A*, 62(10):658–665, 1979.

Hochreiter, S., Younger, A. S., and Conwell, P. R. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pp. 87–94. Springer, 2001.

Irie, K., Schlag, I., Csordás, R., and Schmidhuber, J. A modern self-referential weight matrix that learns to modify itself. In *Deep RL Workshop NeurIPS 2021*, 2021. URL https://openreview.net/forum?id=lVUGfLpNpCF.

Kirsch, L. and Schmidhuber, J. Meta learning backpropagation and improving it. *arXiv preprint arXiv:2012.14905*, 2020.

Kirsch, L., van Steenkiste, S., and Schmidhuber, J. Improving generalization in meta reinforcement learning using learned objectives. *arXiv preprint arXiv:1910.04098*, 2019.

Rosa, M., Afanasjeva, O., Andersson, S., Davidson, J., Guttenberg, N., Hlubuček, P., Poliak, M., Vítku, J., and Feyereisl, J. Badger: Learning to (learn [learning algorithms] through multi-agent communication). *arXiv preprint arXiv:1912.01513*, 2019.

Schmidhuber, J. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook*. PhD thesis, Technische Universität München, 1987.

Schmidhuber, J. Learning to control fast-weight memories: An alternative to recurrent nets. *Neural Computation*, 4 (1):131–139, 1992a.

Schmidhuber, J. Steps towards self-referential neural learning: A thought experiment. 1992b.

Schmidhuber, J. On decreasing the ratio between learning complexity and number of time-varying variables in fully recurrent nets. In *Proceedings of the International Conference on Artificial Neural Networks, Amsterdam*, pp. 460–463. Springer, 1993.

Schmidhuber, J., Zhao, J., and Wiering, M. Shifting inductive bias with success-story algorithm, adaptive levin search, and incremental self-improvement. *Machine Learning*, 28(1):105–130, 1997.

Wang, J. X., Kurth-Nelson, Z., Tirumala, D., Soyer, H., Leibo, J. Z., Munos, R., Blundell, C., Kumaran, D., and Botvinick, M. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

# A. Implementation Details

**Least-recently-used Buffer** We initialize a least-recently-used (LRU) buffer with a single randomly initialized neural network. The $m = 100$ buckets evenly cover the entire current performance range and each hold the $100$ most recent solutions. Solutions from buckets with higher performance are sampled exponentially more frequently. We use an exponential base of $e^{20}$. All layers are initialized from a truncated normal with a standard deviation of $\sigma = \frac{1}{\sqrt{N_x}}$.

**Architecture** We stack three self-referential layers with 32 hidden units.

**Sources of Randomness** To create a temporal tree of self-modifying solutions, randomness must be injected into the system. This randomness originates from the policy action sampling, non-deterministic environment steps, and potential external noise injection as an input to the policy. We found external noise injection not to improve the agent's performance when sufficient randomness originates from the policy and environment.