A Modern Self-Referential Weight Matrix That Learns to Modify Itself

Kazuki Irie[1] **Imanol Schlag**[1] **Róbert Csordás**[1] **Jürgen Schmidhuber**[1,2]
[1]The Swiss AI Lab, IDSIA, University of Lugano (USI) & SUPSI, Lugano, Switzerland
[2]King Abdullah University of Science and Technology (KAUST), Thuwal, Saudi Arabia
{kazuki, imanol, robert, juergen}@idsia.ch

Abstract

The weight matrix (WM) of a neural network (NN) is its program. The programs of many traditional NNs are learned through gradient descent in some error function, then remain fixed. The WM or program of a self-referential NN, however, can keep rapidly modifying all of itself during runtime. In principle, such NNs can meta-learn to learn, and meta-meta-learn to meta-learn to learn, and so on, in the sense of recursive self-improvement. Here we revisit such NNs, building upon recent successes of fast weight programmers (FWPs) and closely related linear Transformers. We propose a scalable self-referential WM (SRWM) that uses self-generated training patterns, outer products and the delta update rule to modify itself. We evaluate our SRWM in a multi-task reinforcement learning setting with procedurally generated ProcGen game environments. Our experiments demonstrate both practical applicability and competitive performance of the SRWM. Our code is public.[1]

1 Introduction

The program of a neural network (NN) is its weight matrix (WM) [1]. For example, with prediction tasks, starting from random values, an NN training procedure based on gradient descent might update the WM to minimize an error function that favors compression of given input-output observations [2]. The WM becomes permanent once training ends, and its usefulness is evaluated with respect to its generalisation capability on yet unseen data.

Many environments, however, continue to evolve after training has halted (e.g., [3, 4]), and the test setting may deviate from training in ways that exceed the NN's generalisation capability. Then human intervention might be required to re-train or fine-tune the model. To minimize such intervention, we consider NNs that can learn to update their own programs in the light of new experience. Especially in multi-task learning and meta-learning (learning to learn [5]), it may be useful to learn how to keep changing and fine-tuning the WM in a way that quickly adapts to new challenges [6].

In principle, a WM could learn by itself a way of executing rapid WM adaptations in task-dependent and context-dependent fashion through a generic mechanism for self-modification. Various self-modifying NNs have been proposed (see Sec. 5). We revisit the self-referential WM [7, 8, 9, 10] from the '90s in the light of modern techniques for updating and generating weights. In particular, we leverage mechanisms which are now well established in the context of Fast Weight Programmers (FWPs; reviewed in Sec. 2) [11, 12, 13]. FWPs have recently seen advancements in terms of performance and scalability, inspired by their formal equivalence [14] to linear variants [15, 16, 17] of the popular Transformer [18].

[1]https://github.com/IDSIA/modern-srwm

Deep RL Workshop, 35th Conference on Neural Information Processing Systems (NeurIPS 2021).

Here we derive a new type of self-modifying WM which naturally emerges as an extension to recent works on FWPs. Using ProcGen [19], we evaluate it in a multi-task reinforcement learning (RL) setting with procedurally generated game environments. We demonstrate both practical applicability and competitive performance of the proposed method.

2 Background on Fast Weight Programmers

Here we briefly review the essential components of fast weight programmers (FWPs) [11, 12, 13] which our model is built upon (Sec. 3). FWPs have a slow NN which can rapidly modify weights of another fast NN. The concept has seen a recent revival, in particular in light of its direct formal connection [14] to linear variants [15, 16, 17, 20] of the popular Transformer [18] when the weight generation is based on outer products between keys and values generated by the slow NN [11]. Recent work augmented the basic FWPs [11, 12] with an improved elementary programming instruction or update rule invoked by the slow NN to reprogram the fast NN, called *delta update rule* (akin to the delta rule by Widrow and Hoff [21]). The resulting DeltaNet [14] is a general purpose auto-regressive NN with linear complexity w.r.t. input sequence length, which transforms the input $\boldsymbol{x}_t \in \mathbb{R}^{d_{\text{in}}}$ to the output $\boldsymbol{y}_t \in \mathbb{R}^{d_{\text{out}}}$ as follows:

$$\boldsymbol{k}_t, \boldsymbol{v}_t, \boldsymbol{q}_t, \beta_t = \boldsymbol{W}_{\text{slow}} \boldsymbol{x}_t \tag{1}$$

$$\bar{\boldsymbol{v}}_t = \boldsymbol{W}_{t-1} \phi(\boldsymbol{k}_t) \tag{2}$$

$$\boldsymbol{W}_t = \boldsymbol{W}_{t-1} + \sigma(\beta_t)(\boldsymbol{v}_t - \bar{\boldsymbol{v}}_t) \otimes \phi(\boldsymbol{k}_t) \tag{3}$$

$$\boldsymbol{y}_t = \boldsymbol{W}_t \phi(\boldsymbol{q}_t) \tag{4}$$

where $\otimes$ denotes the outer product, σ is a sigmoid function, and ϕ is an element-wise activation function whose output elements are all positive and sum up to one (e.g. softmax). In Eq. 1, the input $\boldsymbol{x}_t$ is first projected to key $\boldsymbol{k}_t \in \mathbb{R}^{d_{\text{key}}}$, value $\boldsymbol{v}_t \in \mathbb{R}^{d_{\text{out}}}$, query $\boldsymbol{q}_t \in \mathbb{R}^{d_{\text{key}}}$ vectors and a scalar $\beta_t \in \mathbb{R}$ using a trainable weight matrix $\boldsymbol{W}_{\text{slow}} \in \mathbb{R}^{(d_{\text{value}}+2*d_{\text{key}}+1)\times d_{\text{in}}}$. The generated key vector $\boldsymbol{k}_t$ and a learning rate β_t (generated by the slow NN) are used to update the *fast weight matrix* $\boldsymbol{W}_{t-1}$ using the delta rule expressed in Eqs. 2-3. The fast weight matrix is typically initialized to zero i.e. $\boldsymbol{W}_0 = 0$. The final output $\boldsymbol{y}_t$ is obtained by querying the updated fast weight matrix $\boldsymbol{W}_t$ using a generated query vector $\boldsymbol{q}_t$ (Eq. 4). We note that the use of ϕ function for both writing to (Eq. 3) and reading from (Eqs. 2 and 4) the fast weights is crucial for stability when the delta rule is used [14]. In practice, we use the multi-head version [18] of the computation above, that is, after the projection (Eq. 1), the vectors $\boldsymbol{k}_t$, $\boldsymbol{v}_t$, $\boldsymbol{q}_t$ are split into equally sized H sub-vectors, and the operations in Eqs. 2-4 are conducted by H computation heads independently.

So the slow NN or the *programmer* (here a one-layer feedforward NN; Eq. 1) with *slow weights* $\boldsymbol{W}_{\text{slow}}$ learns by gradient descent to continuously modify or *program* the fast NN (here also a one-layer feedforward NN; Eq. 4) with fast weights $\boldsymbol{W}_t$ as it continually receives a stream of inputs. For further extensions of this concept to more complex slow and fast NN architectures such as recurrent NNs, we refer the readers to another recent study [20].

We note that FWPs are also of interest from the perspective of context-sensitive systems, since the fast WM is completely context-dependent: while processing some sequence, a continually changing custom fast NN is built on the fly.

Here we leverage this mechanism to design a new kind of FWP which programs itself. It can be naturally derived from the operations described above, resulting in a modern version of the self-referential weight matrix [7, 8, 9, 10] of the '90s.

3 A Modern Self-Referential Weight Matrix

Our *modern* self-referential weight matrix (SRWM) learns to train itself through self-invented key/value "training" patterns and learning rates, invoking sequences of elementary programming instructions based on outer products and the delta update rule, as in the recently proposed variants [14] of FWPs (Sec. 2).

Given an input $\boldsymbol{x}_t \in \mathbb{R}^{d_{\text{in}}}$ at time t, our SRWM $\boldsymbol{W}_{t-1} \in \mathbb{R}^{(d_{\text{out}}+2*d_{\text{in}}+1)\times d_{\text{in}}}$ produces four variables $[\boldsymbol{y}_t, \boldsymbol{q}_t, \boldsymbol{k}_t, \beta_t]$ where $\boldsymbol{y}_t \in \mathbb{R}^{d_{\text{out}}}$ is the output of this layer at the current time step, $\boldsymbol{q}_t \in \mathbb{R}^{d_{\text{in}}}$ and $\boldsymbol{k}_t \in \mathbb{R}^{d_{\text{in}}}$ are query and key vectors, and $\beta_t \in \mathbb{R}$ is the self-invented learning rate to be used by

the delta rule. In analogy to the terminology introduced by the original SRWM papers [7, 8, 9, 10], $\boldsymbol{k}_t \in \mathbb{R}^{d_{\text{in}}}$ is the *modifier*-key vector, representing the key whose current value in the SRWM has to be modified, and $\boldsymbol{q}_t \in \mathbb{R}^{d_{\text{in}}}$ is the *analyser*-query which is again fed to the SRWM to retrieve a new "value" vector to be associated with the modifier-key.

The overall dynamics can be expressed as simply as follows:

$$\boldsymbol{y}_t, \boldsymbol{k}_t, \boldsymbol{q}_t, \beta_t = \boldsymbol{W}_{t-1}\phi(\boldsymbol{x}_t) \tag{5}$$

$$\bar{\boldsymbol{v}}_t = \boldsymbol{W}_{t-1}\phi(\boldsymbol{k}_t) \tag{6}$$

$$\boldsymbol{v}_t = \boldsymbol{W}_{t-1}\phi(\boldsymbol{q}_t) \tag{7}$$

$$\boldsymbol{W}_t = \boldsymbol{W}_{t-1} + \sigma(\beta_t)(\boldsymbol{v}_t - \bar{\boldsymbol{v}}_t) \otimes \phi(\boldsymbol{k}_t) \tag{8}$$

where the value vectors have dimensions: $\boldsymbol{v}_t, \bar{\boldsymbol{v}}_t \in \mathbb{R}^{(d_{\text{out}}+2*d_{\text{in}}+1)}$. Figure 1 illustrates the model.

Importantly, the initial values of the SRWM $\boldsymbol{W}_0$ are the only parameters in this layer which are trained by gradient descent. In practice, we extend the output dimension of the matrix from "3D+1" $(d_{\text{out}} + 2 * d_{\text{in}} + 1)$ to "3D+4" $(d_{\text{out}} + 2 * d_{\text{in}} + 4)$ to generate four different learning rates $\beta_t \in \mathbb{R}^4$ to be used in Eq. 8 for the four sub-matrices of $\boldsymbol{W}_{t-1} = [\boldsymbol{W}^y_{t-1}, \boldsymbol{W}^q_{t-1}, \boldsymbol{W}^k_{t-1}, \boldsymbol{W}^\beta_{t-1}]$ used to produce $\boldsymbol{y}_t$, $\boldsymbol{q}_t$, $\boldsymbol{k}_t$, and β_t in Eq. 5. For efficient computation, we also make use of multi-head computation as is done in regular Transformers [18, 15]. Please refer to Appendix A for the full description.

The SRWM described above can potentially be used to replace any regular WM. Here we replace the WM of a simple feedforward layer. We consider two models which we can describe using the DeltaNet as a base model. Model (1) consists of the DeltaNet in which we replace Eqs. 1-4 by the corresponding SRWM equations Eqs. 5-8. In model (2), we replace the slow weight matrix (Eq. 1) in the DeltaNet by an SRWM (Eqs. 5-8). In the experimental section, we refer to the model (1) simply as "SRWM" and to the model (2) as "SR-Delta".

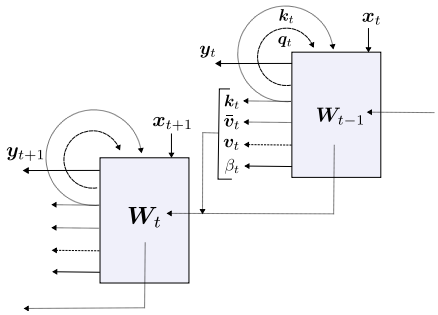

Figure 1: A "modern" self-referential weight matrix (SRWM).

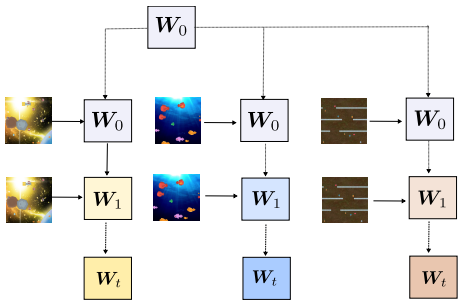

Figure 2: The initial weight matrix $\boldsymbol{W}_0$ is common to all tasks and episodes. The effective weight matrix is a function of task/episode specific input streams.

4 Experiments

4.1 Experimental Settings

We evaluate the proposed model in a multi-task reinforcement learning setting using procedurally generated game environments of ProcGen [19]. The corresponding setting is illustrated in Figure 2. Unlike other popular game environments such as Atari [22], ProcGen provides various procedurally different levels. This allows for creating clean train/test splits. Working with diverse levels is especially relevant to our setting, as we wish to build models which are adaptive to changes across game types, as well as diversity within the same game.

General Settings. In our main experiment, we jointly train on 6 environments, namely *Bigfish*, *Fruitbot*, *Maze*, *Leaper*, *Plunder*, and *Starpilot* in the **easy** distribution. We conduct distributed training using the standard IMPALA [23] architecture implemented in `Torchbeast` [24]. We use 48 actors (i.e. 8 actors per environment). All our models use the common *large* architecture of Espeholt

Table 1: ProcGen normalised aggregated scores (multiplied by 100) over **6 environments** (Bigfish, Fruitbot, Maze, Leaper, Plunder, and Starpilot) in the **easy distribution** and over **4 environments** (Dodgeball, Heist, Maze, Miner) in the **memory** distribution. The models are trained in a multi-task setting. Normalisation constants are taken from the original ProcGen paper (we use constants from the hard distribution for the memory distribution). Results are computed from 3 independent training runs for 300 M and 800 M steps each in the easy and memory distribution respectively. The test scores are averaged over 3 distinct sets of 200 fixed test levels (i.e. the mean/std computed from 9 data points). For further details, see tables in Appendix B where we provide scores obtained for each game. The number of trainable parameters are 626 K for feedforward baseline (FF), 959 K for Fake SR, 1.2 M for LSTM, 1.05 M for DeltaNet and 968 K for SRWM.

	easy dist. (6 env.)					memory dist. (4 env.)	
	FF	LSTM	Fake SR	DeltaNet	SRWM	DeltaNet	SR-Delta
Train	22.5 (2.6)	28.3 (1.4)	27.0 (1.8)	**35.0** (1.6)	34.6 (1.8)	51.8 (2.6)	**59.0** (2.1)
Test	16.4 (1.6)	15.7 (1.6)	15.3 (1.9)	18.6 (1.7)	**20.0** (1.8)	38.0 (4.1)	**38.5** (3.2)

et al. [23] which consists of a 15-layer residual convolutional vision model. They differ from each other by the "memory" module inserted between the vision stem and the output layer. In addition to our SRWM model, we train the baseline IMPALA feed-forward and LSTM [25, 26] models, as well as two additional baselines: the DeltaNet [14] and a "Fake SR" model which has exactly the same architecture as the SRWM model but from which we removed the self-modification mechanism. (i.e. we only keep the "y"-part in Eq. 5). We set the backpropagation span of 50 steps to train the self-modification, as well as LSTM and DeltaNet baselines. The LSTM model has 1 layer with 256 nodes as in the IMPALA baseline. Both DeltaNet and SRWM have two layers with a hidden size of 128, the same setting as the DeltaNet used for Atari in the previous work [20].

These 6 environments are known for not explicitly requiring "memory" to perform the task (we also confirm this trend by comparing our baseline feed-forward and LSTM RNN models). In principle, this allows us to evaluate the effect of self-modifications in isolation (even if it is difficult to completely dissociate self-modification from the concept of "memory"). In addition to the above setting using 6 environments in the easy distribution, we also conduct an extra experiment using 4 environments in the *memory* distribution (*Dodgeball*, *Heist*, *Maze*, *Miner*) to evaluate our models also in partially observable settings. In ProcGen memory distributions, the size of the world is increased and the observation is restricted to a small patch of space around the agent [19]. The backpropagation span is increased to 100 time steps for the memory distribution. In all cases, the memory states (including the weight changes for the SRWM) are only reset at episode boundaries for all stateful models (LSTM, DeltaNet, and SRWM). We train for a total of 300 M steps (ca. 50 M per environment) for joint training on 6 environments in the easy distribution, and 800 M steps (200 M steps per environment) for 4 environments in the memory distribution.

Train/Test split. Following Cobbe et al. [19], we use 200 levels (level ID 0 to 199) to train in the easy distribution. For evaluation, instead of randomly sampling the test levels as is commonly done for ProcGen, we consistently use the same set of 3 distinct test splits for all models. Each of our test splits contains 200 levels, respectively including levels 1000 to 1199, 1200 to 1399, and 1400 to 1599. We opt for using 3 test splits and report an average score to take into account the performance variability across the choice of test levels. The performance in the training set is computed using all 200 training levels. We train each model three times, and thus report training performance averaged over three runs, and test performance over 9 data points (3 test splits for 3 training runs). In the memory distribution setting, since there is no standard convention [19], we opt for training on 500 training levels (level ID 0 to 499) as recommended for the hard distribution. For testing, we use the exact same setting as in our easy distribution setting described above.

4.2 Results

Overall performance. Table 1 shows the aggregated normalised scores. First of all, comparing the LSTM and feed-forward baselines, we confirm Cobbe et al. [19]'s finding that the LSTM layer does not provide any improvements regarding the test performance (while some improvements are

obtained on the train set). The two fast weight models, DeltaNet and SRWM, clearly outperform the feedforward and LSTM baselines. The SRWM achieves a slightly better test score than the DeltaNet. The trend is slightly different in the memory distribution setting. While having a similar parameter count, the SRWM variant achieves a better training score than the DeltaNet baseline, while the test scores are rather close. Overall, the proposed SRWM based model variants achieve very competitive performance.

Comparison to expert models. We observed that the performance gains achieved by our SRWM over the baselines in the easy distribution are particularly large for two of the environments, *Bigfish* and *Starpilot*. Here we study these two cases in isolation. We compare the performance of multi-task agents presented above with expert agents trained specifically on one environment for 50 M steps. As shown in Table 2, the performance gap between the two agents tends to be larger in case of the model without self-modification, especially on *Bigfish*.

Table 2: Comparison between multi-task vs expert agent performance. Raw scores obtained in the easy distribution of ProcGen.

		Weight Update			
		No (Fake SRM)		Yes (SRM)	
Env	Split	Multi-6	Expert	Multi-6	Expert
Bigfish	Train	11.6 (5.7)	28.9 (0.9)	20.1 (2.4)	28.5 (1.2)
	Test	4.7 (2.4)	15.8 (1.7)	9.0 (2.0)	14.2 (2.0)
Starpilot	Train	55.0 (1.3)	59.8 (0.7)	61.3 (2.0)	64.0 (1.9)
	Test	49.6 (2.1)	52.9 (1.2)	54.6 (2.4)	57.3 (1.6)

Ablation on State Reset. The SRWM models presented above are trained by carrying over the weight modifications across entire episodes whose lengths are variable—often episodes are getting longer during training as the agent becomes better at the task. We were initially uncertain about the empirical stability of the dynamics described by Eqs. 5-8 in such a scenario. As an ablation study, we also trained and evaluated an SRWM agent by resetting the weight update every fixed time span whose length is the backpropagation span. We found such a model to fail in leveraging the SRWM mechanism. It obtained scores of 28.5 (1.2) and 16.1 (2.2) on the train and test splits respectively, which are similar to that of the baseline without self-modification (Table 1).

4.3 Discussion

Interpretability. As is often the case with NNs, interpreting the model behavior is not straightforward. Looking into the values of β_t in Eq. 8 which intuitively define the strength of the weight modification, we observe that the value of all four components of β_t varies between 0.50 to 0.65 depending on the input, rather than the full range of sigmoid values between 0 and 1. We find it difficult to derive any further interpretation beyond those statistics. Also β_t alone does not fully describe the behavior of Eq. 8, as it also depends on the actual values of key and query. We leave further analysis to future work on certain simpler supervised learning tasks.

Implementation/Limitation. Similar to recent works on fast weight programmers (FWPs) [14, 15, 20], our SRWM is implemented as a custom CUDA kernel. While this approach yields competitive computation time and memory-efficient custom backpropagation, its flexibility is limited. For instance, if we want to replace all weight matrices in the agent's vision module, a custom implementation for convolution would be required, even if we claim that in theory, the SRWM presented above could replace any regular weight matrix. Regarding speed, the feedforward and LSTM baselines process about 3,500 steps per second, while DeltaNet and SRWM do 2,300 and 1,700 steps per second respectively on a single P100 GPU.

5 Related Work

Original Self-Referential Weight Matrix. The *original* SRWM was proposed in the '90s as a framework for self-modifying recurrent NNs [7, 8, 9, 10]. Such an RNN has special output and input units to directly address and read and modify any of its own current weights through an index for each weight of its weight matrix (i.e., for a weight matrix with an input/output dimension N, the weight index ranges from 0 to $N^2 - 1$ which is encoded as a binary vector). In contrast, our self-modification is based on *key/value associations*, i.e., to encode a WM modification, our NN generates a key vector, a value vector, and a temporary learning rate which allows for the rapid modification of an entire rank at a time [11, 13]. This design is reinforced by the recent success of linear Transformers and fast weight programmers [15, 14, 20]. In this sense, our SRWM is a *modern* approach to self-modification, even if the use of outer products to parameterise fast weight generation itself is not (e.g. [11, 13]).

Other Self-Modifying Neural Networks. There are also more recent works on self-modifying NNs. Neuromodulated plasticity is a Hebbian-style self-modification [27, 28, 29, 30] which also makes use of outer products to generate a modulation term which is added to the base weights. The corresponding computations can also be interpreted as key/value/query association operations. However, the key, query, and value patterns are hard-coded to be one of the input/output pairs of the corresponding layer at each time step. While this circumvents the necessity to allocate parameters for generating those vectors, it is known that the resulting program can be simply expressed as a regular attention [13] over the past outputs [31]. In contrast, in our model, all these patterns are arbitrary as they are generated from learned transformations whose parameters are themselves self-modifying.

Hierarchical Fast Weight Programmers. As reviewed in Sec. 2, an FWP is an NN which learns to generate, update, and maintain weights of another NN. However, a typical FWP has a slow NN with a weight matrix that remains fixed after training. Previous work [20] has proposed to go one step further by parameterising the slow weights in the Delta Net with another FWP to obtain the DeltaDelta Net. However, such a hierarchy has no end, as the highest level programmer would still have a fixed weight matrix. In this work, we follow the spirit of early work [7, 8, 9, 10, 13] and collapse these potentially hierarchical meta-levels into one single self-referential weight matrix.

Fixed Weight Meta-RNNs. Learning learning dynamics using a fixed-weight NN (typically an RNN) has become a common approach [32, 33, 34, 35, 36, 37, 38, 39]. A truly self-referential weight matrix, however, would allow for modifying *all* of its own components. The only thing that's trained by gradient descent are the SWRM's *initial* weights at the beginning of each episode—all of them, however, may rapidly change during sequence processing, in a way that's driven by the SRWM itself.

Recursive Self-Improvements. Beyond the scope of NNs, the concept of self-modification is of general interest when considering autonomous, self-improving machines [5, 40, 41, 42, 43, 44]. In future work, we intend to use the proposed SRWM as a backbone for exploiting the benefits of recursive self-improvement.

6 Conclusion

We proposed a new type of self-referential weight matrix (SRWM) with a modern mechanism for self-modification. Our self-improving NNs learn to generate patterns of keys and values and learning rates, translating these patterns into rapid changes of their own weight matrix through sequential outer products and invocations of the delta update rule. We demonstrated that our SRWM is practical and performs well in a multi-task reinforcement learning setting using game environments procedurally generated by ProcGen. In future work, we plan to evaluate the SRWM also on supervised learning tasks such as few shot learning.

Acknowledgments

We would like to thank Karl Cobbe for answering some practical questions about ProcGen. Kazuki Irie wishes to thank Anand Gopalakrishnan for letting him know about ProcGen. This research was partially funded by ERC Advanced grant no: 742870, project AlgoRNN, and by Swiss National

Science Foundation grant no: 200021_192356, project NEUSYM. We thank NVIDIA Corporation for donating several DGX machines, and IBM for donating a Minsky machine.

References

[1] Jürgen Schmidhuber. Making the world differentiable: On using fully recurrent self-supervised neural networks for dynamic reinforcement learning and planning in non-stationary environments. Technical Report FKI-126-90, `http://people.idsia.ch/~juergen/FKI-126-90_(revised)bw_ocr.pdf`, Tech. Univ. Munich, 1990.

[2] Ray J Solomonoff. A formal theory of inductive inference. part i. *Information and control*, 7(1): 1–22, 1964.

[3] Angeliki Lazaridou, Adhiguna Kuncoro, Elena Gribovskaya, Devang Agrawal, Adam Liska, Tayfun Terzi, Mai Gimenez, Cyprien de Masson d'Autume, Sebastian Ruder, Dani Yogatama, et al. Pitfalls of static language modelling. *Preprint arXiv:2102.01951*, 2021.

[4] Zhiqiu Lin, Jia Shi, Deepak Pathak, and Deva Ramanan. The CLEAR Benchmark: Continual LEArning on Real-World Imagery. In *Conference on Neural Information Processing Systems (NeurIPS), Track on Datasets and Benchmarks*, Virtual only, December 2021.

[5] Jürgen Schmidhuber. Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook. Institut für Informatik, Technische Universität München, 1987. http://www.idsia.ch/~juergen/diploma.html.

[6] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 1126–1135, Sydney, Australia, August 2017.

[7] Jürgen Schmidhuber. Steps towards "self-referential" learning. Technical Report CU-CS-627-92, Dept. of Comp. Sci., University of Colorado at Boulder, November 1992.

[8] Jürgen Schmidhuber. An introspective network that can learn to run its own weight change algorithm. In *Proc. IEE Int. Conf. on Artificial Neural Networks*, pages 191–195, Brighton, UK, May 1993.

[9] Jürgen Schmidhuber. A self-referential weight matrix. In *Proc. Int. Conf. on Artificial Neural Networks (ICANN)*, pages 446–451, Amsterdam, Netherlands, September 1993.

[10] Jürgen Schmidhuber. A neural network that embeds its own meta-levels. In *Proc. IEEE Int. Conf. on Neural Networks (ICNN)*, San Francisco, CA, USA, March 1993.

[11] Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to recurrent nets. Technical Report FKI-147-91, Institut für Informatik, Technische Universität München, March 1991.

[12] Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.

[13] Jürgen Schmidhuber. Reducing the ratio between learning complexity and number of time varying variables in fully recurrent nets. In *International Conference on Artificial Neural Networks (ICANN)*, pages 460–463, Amsterdam, Netherlands, September 1993.

[14] Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear Transformers are secretly fast weight programmers. In *Proc. Int. Conf. on Machine Learning (ICML)*, Virtual only, July 2021.

[15] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *Proc. Int. Conf. on Machine Learning (ICML)*, Virtual only, July 2020.

[16] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, 2021.

[17] Hao Peng, Nikolaos Pappas, Dani Yogatama, Roy Schwartz, Noah A Smith, and Lingpeng Kong. Random feature attention. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, 2021.

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 5998–6008, Long Beach, CA, USA, December 2017.

[19] Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 2048–2056, Virtual only, July 2020.

[20] Kazuki Irie, Imanol Schlag, Róbert Csordás, and Jürgen Schmidhuber. Going beyond linear transformers with recurrent fast weight programmers. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Virtual only, 2021.

[21] Bernard Widrow and Marcian E Hoff. Adaptive switching circuits. In *Proc. IRE WESCON Convention Record*, pages 96–104, Los Angeles, CA, USA, August 1960.

[22] Marc G. Bellemare, Georg Ostrovski, Arthur Guez, Philip S. Thomas, and Rémi Munos. Increasing the action gap: New operators for reinforcement learning. In *Proc. AAAI Conf. on Artificial Intelligence*, pages 1476–1483, Phoenix, AZ, USA, February 2016. AAAI Press.

[23] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-RL with importance weighted actor-learner architectures. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 1406–1415, Stockholm, Sweden, July 2018.

[24] Heinrich Küttler, Nantas Nardelli, Thibaut Lavril, Marco Selvatici, Viswanath Sivakumar, Tim Rocktäschel, and Edward Grefenstette. Torchbeast: A PyTorch platform for distributed RL. *Preprint arXiv:1910.03552*, 2019.

[25] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with LSTM. *Neural computation*, 12(10):2451–2471, 2000.

[26] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

[27] Thomas Miconi, Kenneth Stanley, and Jeff Clune. Differentiable plasticity: training plastic neural networks with backpropagation. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 3559–3568, Stockholm, Sweden, July 2018.

[28] Thomas Miconi, Aditya Rawal, Jeff Clune, and Kenneth O. Stanley. Backpropamine: training self-modifying neural networks with differentiable neuromodulated plasticity. In *Int. Conf. on Learning Representations (ICLR)*, New Orleans, LA, USA, May 2019.

[29] Samuel Schmidgall. Adaptive reinforcement learning through evolving self-modifying neural networks. In *Proc. Genetic and Evolutionary Computation Conference (GECCO), Companion Volume*, pages 89–90, Cancún, Mexico, July 2020.

[30] Elias Najarro and Sebastian Risi. Meta-learning through hebbian plasticity in random networks. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Virtual only, December 2020.

[31] Jimmy Ba, Geoffrey E Hinton, Volodymyr Mnih, Joel Z Leibo, and Catalin Ionescu. Using fast weights to attend to the recent past. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 4331–4339, Barcelona, Spain, December 2016.

[32] Sepp Hochreiter, A. Steven Younger, and Peter R. Conwell. Learning to learn using gradient descent. In *Proc. Int. Conf. on Artificial Neural Networks (ICANN)*, volume 2130, pages 87–94, Vienna, Austria, August 2001.

[33] Neil E Cotter and Peter R Conwell. Fixed-weight networks can learn. In *Proc. Int. Joint Conf. on Neural Networks (IJCNN)*, pages 553–559, San Diego, CA, USA, June 1990.

[34] Neil E Cotter and Peter R Conwell. Learning algorithms and fixed dynamics. In *Proc. Int. Joint Conf. on Neural Networks (IJCNN)*, pages 799–801, Seattle, WA, USA, July 1991.

[35] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy P. Lillicrap. Meta-learning with memory-augmented neural networks. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 1842–1850, New York City, NY, USA, June 2016.

[36] Jane Wang, Zeb Kurth-Nelson, Hubert Soyer, Joel Z. Leibo, Dhruva Tirumala, Rémi Munos, Charles Blundell, Dharshan Kumaran, and Matt M. Botvinick. Learning to reinforcement learn. In *Proc. Annual Meeting of the Cognitive Science Society (CogSci)*, London, UK, July 2017.

[37] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. RL^2: Fast reinforcement learning via slow reinforcement learning. *Preprint arXiv:1611.02779*, 2016.

[38] Louis Kirsch and Jürgen Schmidhuber. Meta-learning backpropagation and improving it. NeurIPS Workshop on Meta-Learning, Virtual only, 2020.

[39] Mark Sandler, Max Vladymyrov, Andrey Zhmoginov, Nolan Miller, Tom Madams, Andrew Jackson, and Blaise Agüera y Arcas. Meta-learning bidirectional update rules. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 9288–9300, Virtual only, July 2021.

[40] Jürgen Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In *Artificial General Intelligence*. Springer, 2006.

[41] Pei Wang. The logic of intelligence. In *Artificial general intelligence*, pages 31–62. Springer, 2007.

[42] Eric Nivel and Kristinn R Thórisson. Self-programming: Operationalizing autonomy. In *Proc. Conf. on Artificial General Intelligence (AGI)*, pages 150–155, Arlington, VA, USA, March 2009.

[43] Bas R. Steunebrink, Kristinn R. Thórisson, and Jürgen Schmidhuber. Growing recursive self-improvers. In *Proc. Conf. on Artificial General Intelligence (AGI)*, pages 129–139, New York, NY, USA, July 2016.

[44] Pei Wang, Xiang Li, and Patrick Hammer. Self in NARS, an AGI system. *Frontiers in Robotics and AI*, 5:20, 2018.

A Model Details

Equations for the four-learning rate case. In Sec. 3, for the purpose of clarity, we presented the equations for our SRWM model in the case where we only have a single learning rate β_t. Here we provide a complete description of an SRWM with a separate self-inveted learning rate for each component. As we noted in Sec. 3, the SRWM can be split into sub-matrices: $\boldsymbol{W}_{t-1} = [\boldsymbol{W}^y_{t-1}, \boldsymbol{W}^q_{t-1}, \boldsymbol{W}^k_{t-1}, \boldsymbol{W}^\beta_{t-1}]$ according to the sub-components used to produce $\boldsymbol{y}_t$, $\boldsymbol{q}_t$, $\boldsymbol{k}_t$, and β_t in Eq. 5. In case where we use separate learning rates, we need separate equations to describe the update of each sub-matrices. For example, for the "y"-part $\boldsymbol{W}^y_{t-1}$, while keeping the same equation for the first projection (Eq. 5), the rest becomes:

$$\boldsymbol{y}^k_t = \boldsymbol{W}^y_{t-1}\phi(\boldsymbol{k}_t) \tag{9}$$

$$\boldsymbol{y}^q_t = \boldsymbol{W}^y_{t-1}\phi(\boldsymbol{q}_t) \tag{10}$$

$$\boldsymbol{W}^y_t = \boldsymbol{W}^y_{t-1} + \sigma(\beta_{y,t})(\boldsymbol{y}^q_t - \boldsymbol{y}^k_t) \otimes \phi(\boldsymbol{k}_t) \tag{11}$$

where $\boldsymbol{y}^k_t$ and $\boldsymbol{y}^q_t$ are the "y"-part of $\bar{\boldsymbol{v}}_t$ and $\boldsymbol{v}_t$ in Eq. 6 and 7 respectively, and $\beta_{y,t} \in \mathbb{R}$ is one of four learning rate dedicated to the "y"-part. The equations for other sub-matrices $\boldsymbol{W}^q_{t-1}$, $\boldsymbol{W}^k_{t-1}$, $\boldsymbol{W}^\beta_{t-1}$ are analogous.

Use of multiple heads. In this work, the SRWM was inserted between other layers with learned parameters with configurable dimensionalities. This allows us for an efficient computation using multiple heads for computation described as follows. Given a number of heads H which we use in the SRWM layer, we configured the model dimensions such that the input dimension to an SRWM layer d_{in} is divisible by H. The input is then split into H equally sized components, and each head executes separate SRWM operations (Eqs. 5-8) on one of the input components. In consequence, the SRWM with the same model hyper-parameters as the DeltaNet has less parameters than the DeltaNet. For example, if $d_{\text{in}} = d_{\text{key}}$, the common head dimension is $d = d_{\text{in}}//H$, the parameter shape of key projection in the SRWM is (H, d, d) while it is (d_{in}, d_{in}) = ($H * d$, $H * d$) for the DeltaNet. In case the input size of the SRWM layer is not configurable, this option has to be disabled and a single head version should be used.

B Extra Result Tables

Table 3: Performance on ProcGen game environments. Multi-task training in 6 environments in the **easy distribution**. The three variants of the SRWM are as follows: *True*: the SRWM model, *Fake*: the SRWM model without self-modification mechanism, and *Reset*: the SRWM trained and evaluated with weight update reset.

Env	Split	FF	LSTM	Delta	SRM		
					True	Fake	Reset
Bigfish	Train	8.3 (3.9)	6.5 (2.0)	19.6 (4.0)	**20.1** (2.4)	11.6 (5.7)	15.7 (2.8)
	Test	4.3 (2.3)	3.2 (1.1)	7.8 (1.5)	**9.0** (2.0)	4.7 (2.4)	5.8 (1.3)
Fruitbot	Train	**29.2** (0.2)	27.8 (0.5)	28.8 (0.9)	28.7 (0.2)	27.8 (1.3)	29.2 (0.2)
	Test	**25.6** (1.1)	24.8 (0.7)	24.5 (1.5)	25.5 (1.0)	24.6 (1.2)	25.2 (1.4)
Leaper	Train	3.3 (0.2)	3.3 (0.2)	**3.5** (0.4)	**3.5** (0.2)	3.3 (0.3)	3.4 (0.2)
	Test	3.4 (0.4)	**3.6** (0.4)	3.3 (0.4)	3.4 (0.4)	**3.6** (0.4)	3.5 (0.3)
Maze	Train	1.9 (0.3)	3.1 (0.7)	**3.8** (0.2)	3.6 (0.5)	3.2 (0.2)	2.9 (0.2)
	Test	1.4 (0.3)	1.6 (0.4)	1.7 (0.2)	**1.8** (0.5)	1.3 (0.3)	1.5 (0.3)
Plunder	Train	3.2 (0.2)	3.2 (0.4)	**3.3** (0.2)	3.1 (0.0)	3.1 (0.4)	3.1 (0.1)
	Test	3.2 (0.3)	2.9 (0.4)	**3.3** (0.2)	3.0 (0.2)	3.1 (0.5)	3.0 (0.2)
Starpilot	Train	57.6 (0.9)	56.0 (1.5)	60.3 (0.4)	**61.3** (2.0)	55.0 (1.3)	55.0 (1.9)
	Test	53.0 (1.7)	48.3 (2.0)	53.9 (2.4)	**54.6** (2.4)	49.6 (2.1)	48.6 (1.9)
Aggregated	Train	22.5 (2.6)	28.3 (1.4)	**35.0** (1.6)	27.0 (1.8)	34.6 (1.8)	28.5 (1.2)
	Test	16.4 (1.6)	15.7 (1.6)	18.6 (1.7)	**20.0** (1.8)	15.3 (1.9)	16.1 (2.2)

Table 4: Performance on ProcGen game environments. Multi-task training in 4 environments in the **memory distribution**.

Env	Split	DeltaNet	SRM-Delta
Dodgeball	Train	**7.1** (0.2)	**7.1** (0.6)
	Test	**6.4** (0.3)	6.2 (0.6)
Heist	Train	1.0 (0.3)	**1.5** (0.1)
	Test	0.8 (0.2)	**1.1** (0.3)
Maze	Train	5.3 (0.4)	**5.9** (0.2)
	Test	**3.3** (0.6)	**3.3** (0.4)
Miner	Train	32.3 (0.4)	**34.5** (0.8)
	Test	29.2 (1.1)	**29.4** (0.7)
Aggregated	Train	51.8 (2.6)	59.0(2.1)
	Test	38.0 (4.1)	38.5(3.2)