Going Beyond Linear Transformers with Recurrent Fast Weight Programmers

Anonymous Author(s)
Affiliation
Address
email

Abstract

Transformers with linearised attention ("linear Transformers") have demonstrated the practical scalability and effectiveness of outer product-based Fast Weight Programmers (FWPs) from the '90s. However, the original FWP formulation is more general than the one of linear Transformers: a *slow* neural network (NN) continually reprograms the weights of a *fast* NN with *arbitrary* NN architecture. In existing linear Transformers, both NNs are feedforward and consist of a single layer. Here we explore new variations by adding recurrence to the slow and fast nets. We evaluate our novel recurrent FWPs (RFWPs) on two synthetic algorithmic tasks (code execution and sequential ListOps), Wikitext-103 language models, and on the Atari 2600 2D game environment. Our models exhibit properties of Transformers and RNNs. In the reinforcement learning setting, we report large improvements over the LSTM in several Atari games. Our code is publicly available.

1 Introduction

The Transformer [1] has arguably become the most popular neural network (NN) for processing text data. Its success on neural machine translation quickly transferred to other problems in natural language processing, such as language modelling [2, 3] or question answering [4]. Recently, it has also been applied in other domains such as image processing [5, 6] or mathematical problem solving [7, 8].

Conceptually, the Transformer is a deep feedforward NN that processes all elements of a sequence in parallel: unlike in recurrent NNs (RNNs), the computations of a layer for the entire sequence can be packed into one big matrix multiplication. This scales well with the number of parallel processors.

Despite the benefits of parallelisation, a major drawback of Transformers is their computational complexity in time and space which is quadratic in sequence length. Furthermore, in its auto-regressive version [1, 2], which is the focus of our work, the state size linearly increases with sequence length. This makes it prohibitive in auto-regressive settings dealing with very long or potentially infinite sequences—forcing practitioners to truncate the temporal context into a fixed-size window and ignoring dependencies going beyond that window. Although certain Transformer modifications try to address this issue [9, 10], reinforcement learning in partially observable environments [11, 12] is still dominated by RNNs such as the Long Short-Term Memory (LSTM; [13]).

To scale Transformers to longer sequences, recent works have proposed to linearise the softmax in the attention computation [14], reorganizing the latter in a sequential way. Such models include Katharopoulos et al.'s *Linear Transformer* (LT) [14] and Choromanski et al.'s *Performer* [15]. They enjoy time and space complexities linear in sequence length with states of constant size. While their performance on some tasks does not fully match the one of regular Transformers [16], several improvements have already been proposed [17, 18] (see our review in Sec. 2.2) which makes this Transformer family a promising alternative.

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

Here we go one step further in advancing linear Transformer variants as powerful sequence processing models, adopting the perspective of "Fast Weight Programmers" (FWPs) [19, 20, 21]. Recent work emphasised that linearised self-attention is equivalent to an outer product-based FWP from the '90s ([18]; reviewed in Sec. 2). Here we explore this connection further and develop more complex FWPs. The original FWP [19] is a two-NN system: a slow and a fast net, each with arbitrary architectures. The slow net learns to generate rapid context-dependent weight modifications for the fast net. In the case of existing linear Transformer variants, the slow and fast nets are simple one layer feedforward NNs. Here we augment them with recurrent connections to obtain recurrent FWPs (RFWPs). Recurrence enhances the model's theoretical power [22] and can help to solve tasks that naturally require recurrence as a part of the solution.

Our experiments on the language modelling dataset Wikitext-103 [23] show that our RFWPs are competitive compared to regular Transformers. We then study various properties of the proposed models on two synthetic algorithmic tasks: code execution [24] and sequential ListOps [25]. Finally, it is straightforward to apply our models to reinforcement learning problems as a drop-in replacement for LSTMs. Here our RFWPs obtain large improvements over LSTM baselines across many Atari 2600 2D game environments [26]. Although LSTM still works better in a few environments, we show that our RFWPs generally improve by scaling them up.

The main contribution of this work is twofold: (1) from the perspective of FWPs, we study alternative and potentially more powerful FWPs for sequence processing, demonstrating that NNs can easily learn to control NNs that are more complex than a single feedforward layer, and (2) from the perspective of Transformer models, our models augment linear Transformers with recurrence, addressing general limitations of existing auto-regressive Transformer models.

2 Background on Fast Weight Programmers (FWPs)

Here we review the general concept of FWPs, as well as two specific instances thereof: the linear Transformer [14, 15] and the Delta Net [18].

2.1 General Formulation

We refresh the concept of fast weight controllers or FWPs [19, 20] using modern notation in a sequence processing scenario. An FWP with trainable parameters $\boldsymbol{\theta}_{\text{slow}}$ sequentially transforms an input sequence $\{\boldsymbol{x}^{(i)}\}_{i=1}^{L}$ with $\boldsymbol{x}^{(i)} \in \mathbb{R}^{d_{\text{in}}}$ to an output sequence $\{\boldsymbol{y}^{(i)}\}_{i=1}^{L}$ with $\boldsymbol{y}^{(i)} \in \mathbb{R}^{d_{\text{out}}}$ as

$$\boldsymbol{\theta}_{\text{fast}}^{(i)}, \boldsymbol{q}^{(i)} = \texttt{SlowNet}\big(\{\boldsymbol{x}^{(j)}\}_{j=1}^{i}, \{\boldsymbol{y}^{(j)}\}_{j=0}^{i-1}, \{\boldsymbol{\theta}_{\text{fast}}^{(j)}\}_{j=0}^{i-1}; \boldsymbol{\theta}_{\text{slow}}\big) \tag{1}$$

$$\boldsymbol{y}^{(i)} = \texttt{FastNet}(\{\boldsymbol{q}^{(j)}\}_{j=1}^{i}, \{\boldsymbol{y}^{(j)}\}_{j=0}^{i-1}; \boldsymbol{\theta}_{\text{fast}}^{(i)}) \tag{2}$$

where $\boldsymbol{y}^{(0)}$ and $\boldsymbol{\theta}_{\text{fast}}^{(0)}$ are initial variables. In this two-NN system, the parameters $\boldsymbol{\theta}_{\text{fast}}^{(i)}$ of the NN `FastNet` are generated by another NN `SlowNet` at each time step i. The weights of the fast net are *fast* in the sense that they may rapidly change at every step of the sequence while the weights of the slow net $\boldsymbol{\theta}_{\text{slow}}$ are *slow* because they can only change due to gradient descent during training, remaining fixed afterwards[1]. Eq. 1 expresses a slow NN in its general form. The slow net can generate certain fast weight modifications in a way that reflects certain architectural choices for the slow and fast NNs. In addition to the fast weights $\boldsymbol{\theta}_{\text{fast}}^{(i)}$, the slow net also generates or *invents* an input $\boldsymbol{q}^{(i)}$ to be fed to the fast net (alternatively $\boldsymbol{q}^{(i)}$ can simply be $\boldsymbol{x}^{(i)}$). While the architectures of slow and fast nets are arbitrary, they are typically chosen to be differentiable such that the entire FWP can be trained in an end-to-end manner using gradient descent. By interpreting the weights of an NN as a program [27], the slow net effectively learns to control, or *program*, the fast NN. Thus, the slow net is a neural programmer of fast weights, and its parameter set $\boldsymbol{\theta}_{\text{slow}}$ is a compressed form of information used to produce potentially infinite variations of context-dependent fast weights.

In many settings it makes sense to generate the fast weights $\boldsymbol{\theta}_{\text{fast}}$ incrementally in an iterative fashion, where the `SlowNet` is further decomposed into two sub-parts:

$$\boldsymbol{z}^{(i)}, \boldsymbol{q}^{(i)} = \texttt{SlowSubnet}(\{\boldsymbol{x}^{(j)}\}_{j=1}^{i}, \{\boldsymbol{y}^{(j)}\}_{j=0}^{i-1}, \{\boldsymbol{\theta}_{\text{fast}}^{(j)}\}_{j=0}^{i-1}; \boldsymbol{\theta}_{\text{slow}}) \tag{3}$$

$$\boldsymbol{\theta}_{\text{fast}}^{(i)} = \texttt{UpdateRule}(\boldsymbol{\theta}_{\text{fast}}^{(i-1)}, \boldsymbol{z}^{(i)}) \tag{4}$$

[1]The fast net could also contain some additional slow weights; we ignore this possibility.

where `UpdateRule` takes the fast weights $\boldsymbol{\theta}_{\text{fast}}^{(i-1)}$ from the previous iteration to produce the new fast weights conditioned on $\boldsymbol{z}^{(i)}$. In the next section we'll review recent work from this perspective.

2.2 Linear Transformers as Fast Weight Programmers

In general, the dimension of the fast weights $\boldsymbol{\theta}_{\text{fast}}^{(i)}$ is too large to be conveniently parameterised by an NN. Instead, it was proposed [19] to perform a rank-one update via the outer product of two vectors generated by the slow net. Two recent models directly correspond to such outer product-based FWPs: linear Transformers [14] and the Delta Net [18].

Linear Transformer. The "linear Transformer" [14, 15] is a Transformer where the softmax in the attention is linearised. This is achieved by replacing the softmax with a kernel function ϕ, after which the self-attention can be rewritten as a basic outer product-based FWP [19, 18]. Previous works focused on different ϕ maps with properties such as increased capacity [18] or guaranteed approximation of the softmax in the limit [15, 17]. For our purposes, the particular choice of ϕ is irrelevant and we simply assume $\phi : \mathbb{R}^{d_{\text{key}}} \rightarrow \mathbb{R}^{d_{\text{key}}}$, simplifying our equations below by writing $\boldsymbol{k}, \boldsymbol{q}$ instead of $\phi(\boldsymbol{k}), \phi(\boldsymbol{q})$. Using otherwise the same notation, for each new input $\boldsymbol{x}^{(i)}$, the output $\boldsymbol{y}^{(i)}$ is obtained by:

$$\boldsymbol{k}^{(i)}, \boldsymbol{v}^{(i)}, \boldsymbol{q}^{(i)} = \boldsymbol{W}_k \boldsymbol{x}^{(i)}, \boldsymbol{W}_v \boldsymbol{x}^{(i)}, \boldsymbol{W}_q \boldsymbol{x}^{(i)} \tag{5}$$

$$\boldsymbol{W}^{(i)} = \boldsymbol{W}^{(i-1)} + \boldsymbol{v}^{(i)} \otimes \boldsymbol{k}^{(i)} \tag{6}$$

$$\boldsymbol{y}^{(i)} = \boldsymbol{W}^{(i)} \boldsymbol{q}^{(i)} \tag{7}$$

where the slow weight matrices $\boldsymbol{W}_k \in \mathbb{R}^{d_{\text{key}} \times d_{\text{in}}}$ and $\boldsymbol{W}_v \in \mathbb{R}^{d_{\text{value}} \times d_{\text{in}}}$ are used to obtain the *key* $\boldsymbol{k}^{(i)} \in \mathbb{R}^{d_{\text{key}}}$ and the *value* $\boldsymbol{v}^{(i)} \in \mathbb{R}^{d_{\text{value}}}$. The key and value vectors are used to generate new weights via the outer product $\boldsymbol{v}^{(i)} \otimes \boldsymbol{k}^{(i)} \in \mathbb{R}^{d_{\text{value}} \times d_{\text{key}}}$. A further simplification in the equations above is the omission of attention normalisation which has been experimentally shown to be unnecessary if the ϕ function produces normalised scores [18].

In Eq. 6, the previous fast weight matrix $\boldsymbol{W}^{(i-1)} \in \mathbb{R}^{d_{\text{value}} \times d_{\text{key}}}$ is updated to yield $\boldsymbol{W}^{(i)}$ by adding the update term $\boldsymbol{v}^{(i)} \otimes \boldsymbol{k}^{(i)}$. This corresponds to the *sum update rule*. Here the fast NN is a simple linear transformation as in Eq. 7 which takes as input the (query) vector $\boldsymbol{q}^{(i)} \in \mathbb{R}^{d_{\text{key}}}$ generated by the slow weights $\boldsymbol{W}_q \in \mathbb{R}^{d_{\text{key}} \times d_{\text{in}}}$. Hence, in linear Transformers, the previous Eq. 3 simplifies to: $\boldsymbol{z}^{(i)}, \boldsymbol{q}^{(i)} = \texttt{SlowSubnet}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}_{\text{slow}})$ with $\boldsymbol{z}^{(i)} = (\boldsymbol{k}^{(i)}, \boldsymbol{v}^{(i)})$.

Delta Net. The Delta Net [18] is obtained by replacing the sum update rule (Eq. 6) in the linear Transformer with the *delta update rule* [28]:

$$\boldsymbol{W}^{(i)} = \boldsymbol{W}^{(i-1)} + \beta^{(i)} (\boldsymbol{v}^{(i)} - \bar{\boldsymbol{v}}^{(i)}) \otimes \boldsymbol{k}^{(i)} \tag{8}$$

where $\beta^{(i)} \in \mathbb{R}$ is a fast parameter (learning rate) of the update rule generated by the slow net with weights $\boldsymbol{W}_\beta \in \mathbb{R}^{d_{\text{in}} \times 1}$ and the sigmoid function σ:

$$\beta^{(i)} = \sigma(\boldsymbol{W}_\beta \boldsymbol{x}^{(i)}) \tag{9}$$

and $\bar{\boldsymbol{v}}^{(i)} \in \mathbb{R}^{d_{\text{value}}}$ is generated as a function of the previous fast weights $\boldsymbol{W}^{(i-1)}$ and the key $\boldsymbol{k}^{(i)}$

$$\bar{\boldsymbol{v}}^{(i)} = \boldsymbol{W}^{(i-1)} \boldsymbol{k}^{(i)}. \tag{10}$$

Schlag et al. [18] introduce the Delta Net to address a memory capacity problem affecting linear Transformers with the sum update rule. The corresponding Eq. 3 is: $\boldsymbol{z}^{(i)}, \boldsymbol{q}^{(i)} = \texttt{SlowSubnet}(\boldsymbol{x}^{(i)}, \boldsymbol{W}^{(i-1)}; \boldsymbol{\theta}_{\text{slow}})$ with $\boldsymbol{z}^{(i)} = (\boldsymbol{k}^{(i)}, \boldsymbol{v}^{(i)}, \beta^{(i)}, \bar{\boldsymbol{v}}^{(i)})$. Thus, unlike linear Transformers, the `SlowNet` in the Delta Net takes the previous fast weights $\boldsymbol{W}^{(i-1)}$ into account to generate the new fast weight updates.

In summary, these models can be described as FWPs whose slow and fast net each consists of a single feedforward layer (Eqs. 5 and 7). In the next section we present several extensions.

Other Approaches. While our focus here is on outer product-based weight generation, which is an efficient method to handle high dimensional NN weights, there are also other approaches. For example, instead of generating a new weight matrix, Hypernetworks scale the rows of a slow weight matrix with a generated vector of appropriate size. In the broad sense of context-dependent weights, many concepts relate to FWPs, e.g. dynamic convolution [29, 30, 31], lambda networks [32], or dynamic plasticity [33, 34, 35].

3 Fast Weight Programmers with slow or fast RNNs

The original formulation of FWPs reviewed in Sec. 2.1 is more general than existing models presented in Sec. 2.2. In particular, both fast and slow networks in existing linear Transformers are one-layer feedforward networks (Eq. 7). Here we present FWPs with recurrent fast nets in Sec. 3.1 and FWPs with recurrent slow nets in Sec. 3.2.

3.1 Fast Network Extensions

In principle, any NN architecture can be made *fast*. Its fast weight version is obtained by replacing the networks weights with fast weights parameterised by an additional slow network. For example, consider a regular RNN layer with two weight matrices $\boldsymbol{W}$ and $\boldsymbol{R}$:

$$\boldsymbol{h}^{(i)} \quad = \quad \sigma(\boldsymbol{W}\boldsymbol{x}^{(i)} + \boldsymbol{R}\boldsymbol{h}^{(i-1)}) \tag{11}$$

A fast weight version can be obtained by replacing $\boldsymbol{W}$ and $\boldsymbol{R}$ with $\boldsymbol{W}^{(i)}$ and $\boldsymbol{R}^{(i)}$ which are controlled as in Eq. 8 with all necessary variables generated by a separate slow net at each time step i.

While this view illustrates the generality of FWPs, the angle under which we approach these models here is slightly different: we introduce recurrence as a way of augmenting existing linear Transformers.

Delta RNN. We obtain a fast weight RNN called **Delta RNN** by adding an additional recurrent term to the feedforward fast net of the FWP (Eq. 7):

$$\boldsymbol{y}^{(i)} \quad = \quad \boldsymbol{W}^{(i)}\boldsymbol{q}^{(i)} + \boldsymbol{R}^{(i)} f(\boldsymbol{y}^{(i-1)}) \tag{12}$$

where $\boldsymbol{R}^{(i)} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{out}}}$ is an additional fast weight matrix which introduces recurrent connections. It is also generated by the slow net using the delta update rule, similar to $\boldsymbol{W}^{(i)}$ in Eq. 8 but with additional slow weights. We apply an element-wise activation function f to the previous output of the fast network $\boldsymbol{y}^{(i-1)}$ to obtain the recurrent query. The choice of activation function is crucial here because, to achieve stable model behaviour, the elements in key and query vectors should be positive and sum up to one when the delta update rule is used [18]. We use the softmax function ($f = \mathrm{softmax}$ in Eq. 12) to satisfy these conditions.

Analogous to the Delta RNN, we also construct a **Delta LSTM** with six fast weight matrices. The exact equations can be found in Appendix A.

Alternative Feedforward Fast Nets. While the focus of this work is on RNNs, there are also interesting fast feedforward models to be used in Eq. 7 which might result in stronger feedforward baselines. For example, we can simply replace the one-layer transformation of Eq. 7 by a K-layer deep network:

$$\boldsymbol{h}_k^{(i)} \quad = \quad \boldsymbol{W}_k^{(i)} f(\boldsymbol{h}_{k-1}^{(i)}) \quad \text{for } \; k \in [1..K] \; \text{ with } \; \boldsymbol{h}_0^{(i)} = \boldsymbol{q}^{(i)} \tag{13}$$

$$\boldsymbol{y}^{(i)} \quad = \quad \boldsymbol{h}_K^{(i)} \tag{14}$$

where the slow network produces all K fast weights $\{\boldsymbol{W}_k^{(i)}\}_{k=1}^{K}$ and query $\boldsymbol{q}^{(i)}$ from a single input $\boldsymbol{x}^{(i)}$. In light of the capacity limitation in linear Transformers [18], this might introduce additional capacity without the need of larger representations, analogous to the trade-off in a multilayer perceptron (MLP) between narrow & deep versus shallow & wide. We refer to this class of models as **Delta MLPs**. Again, for stable model behaviour with the delta rule, we apply the softmax activation f to the vectors to be used as a query.

Another interesting approach is to use a Delta Net itself as a fast net, i.e., make the slow weights in the Delta Net fast (thus obtaining a **Delta Delta Net**). Such a model could in principle learn to improve the way of generating fast weights depending on the context. While we plan to investigate this potential in future work, we also include preliminary results of such a model in our language modelling experiments (Sec. 4.1).

We note that there are several previously proposed recurrent fast weight models. For example, Schmidhuber [21]'s FWP have been revisited by Ba et al. [36]. There, key and value vectors are not generated within the same time step, unlike in our models or in linear Transformers. Another related recurrent FWP is the Fast Weight Memory (FWM) [37]. The FWM fits neatly in the RFWP framework: the slow net is an LSTM and the fast net is a higher-order RNN. However, the FWM is a single pair of slow and fast nets, and a multi-layer version, as in the linear Transformer family, were not explored. Similarly, the Metalearned Neural Memory [35] uses an LSTM as its slow net and a deep MLP as its fast net but again limited to one pair. Others have investigated variants of RNNs with fast weights for toy synthetic retrieval tasks [38, 39]. In particular, Keller et al. [39] augment the LSTM with a fast weight matrix in the cell update. In contrast, we make all weights in the LSTM fast and, importantly, our model specifications build upon the successful Transformer architecture.

3.2 Slow Network Extensions

In linear Transformers, the slow network is purely feedforward (Eq. 5). It can be made recurrent at two different levels: within the slow network (i.e. the slow network computes weight updates based on its own previous outputs: e.g. key, value, query vectors) or via the fast network by taking the fast net's previous output as an input. In our preliminary experiments, we found the former to be sub-optimal (at least in language modelling experiments). So we focus on the latter approach: we make the slow net in the Delta Net dependent on the previous output of the fast network. We refer to this model as the **Recurrent Delta Net** (RDN).

Recurrent Delta Net. We obtain the RDN by modifying the generation of key, value, and query vectors in Eq. 5 as well as the learning rate in Eq. 9 for the delta update rule. We add additional slow weights ($\boldsymbol{R}_k$, $\boldsymbol{R}_v$, $\boldsymbol{R}_q$, and $\boldsymbol{R}_\beta$) for recurrent connections which connect the previous output of the fast net $\boldsymbol{y}^{(i-1)}$ (Eq. 7) to the new $\boldsymbol{k}^{(i)}$, $\boldsymbol{v}^{(i)}$, $\boldsymbol{q}^{(i)}$, and $\beta^{(i)}$ as follows:

$$\boldsymbol{k}^{(i)} = \boldsymbol{W}_k \boldsymbol{x}^{(i)} + \boldsymbol{R}_k \tanh(\boldsymbol{y}^{(i-1)}) \tag{15}$$

$$\boldsymbol{v}^{(i)} = \boldsymbol{W}_v \boldsymbol{x}^{(i)} + \boldsymbol{R}_v \tanh(\boldsymbol{y}^{(i-1)}) \tag{16}$$

$$\boldsymbol{q}^{(i)} = \boldsymbol{W}_q \boldsymbol{x}^{(i)} + \boldsymbol{R}_q \tanh(\boldsymbol{y}^{(i-1)}) \tag{17}$$

$$\beta^{(i)} = \sigma(\boldsymbol{W}_\beta \boldsymbol{x}^{(i)} + \boldsymbol{R}_\beta \tanh(\boldsymbol{y}^{(i-1)})) \tag{18}$$

While the rest of the model remains as in the Delta Net, with these simple extra recurrent connections the model becomes a proper RNN. The corresponding dependencies in Eq. 3 are: $\boldsymbol{z}^{(i)}, \boldsymbol{q}^{(i)} = \texttt{SlowSubnet}(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i-1)}, \boldsymbol{W}^{(i-1)}; \boldsymbol{\theta}_{\text{slow}})$ with $\boldsymbol{z}^{(i)} = (\boldsymbol{k}^{(i)}, \boldsymbol{v}^{(i)}, \beta^{(i)}, \bar{\boldsymbol{v}}^{(i)})$.

In contrast to regular RNNs, all these fast weight variants maintain the standard one-dimensional states and the two-dimensional fast weight states which both serve as additional memory. In our experiments (Sec. 4), we demonstrate that slow NNs can learn to control the weights of these rather complex RNN variants, and that the introduction of recurrence in the fast network introduces a crucial property that is missing in regular Transformers.

4 Experiments

We conduct experiments in four different settings. We start by evaluating all models on a language modelling task (Sec. 4.1) to obtain a performance overview and to discuss computational costs. Language modelling is an excellent task to evaluate sequence models. However, to highlight their different capabilities we evaluate our models also on algorithmic tasks. In fact, it is well-known that the actual capabilities of RNNs differ from one architecture to another [40]. We are interested in discussing such differences. With that goal in mind, we conduct experiments on two synthetic algorithmic tasks, code execution (Sec. 4.2) and sequential ListOps (Sec. 4.3), which are designed to compare elementary sequence processing abilities of models. Finally, we apply our models to reinforcement learning in 2D game environments (Sec. 4.4) as a replacement for LSTMs.

Table 1: WikiText-103 language model perplexity results with the common *small* setting [17, 18]. For each model, its name, corresponding slow and fast networks, and weight update rule (Update) is specified. All models are trained and evaluated on the span of 256 tokens except for the models in the last two rows (+ full context) which are trained and evaluated without context truncation. Training language models on Wikitext-103 is a resource-demanding experiment which is why we only provide results from a single run. Parameter count is in millions.

Name	Slow net	Update	Fast net	Valid	Test	#Prms
Transformer	-	-	-	33.0	34.1	44.0
Linear Transformer	Feedforward	sum	Linear	37.1	38.3	44.0
Delta Net		delta		34.1	35.2	44.0
Delta MLP	Feedforward	delta	Deep MLP	35.8	36.8	44.3
Delta Delta Net			Delta Net	34.0	35.2	44.6
Delta RNN			RNN	33.8	35.0	44.6
Delta LSTM			LSTM	**33.4**	**34.7**	47.3
RDN	Recurrent		Linear	34.1	35.2	44.1
Delta RNN	+ full context			**31.8**	**32.8**	44.6
RDN				32.5	33.6	44.1

4.1 Language Modelling

We first evaluate all discussed models on the generic language modelling task. This allows us to obtain a performance overview and review the computational efficiency of different models. We use the Wikitext-103 dataset [23] and follow the *small model setting* similar to those used in recent works by Peng et al. [17] and Schlag et al. [18]. This allows us to train and evaluate different models with a reasonable amount of compute on this resource-demanding language modelling task.

Perplexity results. See Table 1 which also serves as a tabular summary recapitulating different models described in Sec. 2 and 3 with various architectures for slow and fast nets, and the choice of update rule. The top block of Table 1 shows the performance of the baseline Transformer, Katharopoulos et al. [14]'s Linear Transformer, and Schlag et al. [18]'s Delta Net. The performance of models presented in Sec. 3 can be found in the middle block. First of all, Delta MLP performs worse than the baseline Delta Net despite a slight increase in parameter count (44.3 vs. 44.0 M). This supports the intuition that it is better to make the slow network aware of the outputs of intermediate layers to generate fast weights in a deep network, instead of generating fast weights for all layers at a time. In all other models, the performance never degrades with the proposed architectural augmentation. The Delta Delta Net yields limited improvements; we plan to study this model in depth in future work. With the same amount of parameters (44.6 M), Delta RNN yields greater improvements. Among the models presented here, the Delta LSTM variant exhibits the best performance. This shows that the slow network successfully controls the rather complex fast LSTM network, although it also requires more parameters (47.3 M) compared to other models. Finally, the benefits of recurrent connections added to the baseline Delta Net do not directly translate into practical improvements in language modelling as demonstrated by the performance of RDN compared to the baseline Delta Net. Importantly, given a constant memory size w.r.t. sequence length, it is straight-forward to train and evaluate our RNNs without context truncation (while still limiting the backpropagation span). Corresponding performances of Delta RNN and RDN are shown in the bottom part of Table 1: they are better than the regular Transformer with a limited context (256 tokens).

While language modelling is useful as a first sanity check (here for example, except for Delta Delta Net, all models achieve reasonable performance), the task is too generic to identify certain important aspects of the models, such as real benefits of recurrence. Before we move on to trickier reinforcement learning applications, Sec. 4.2 and 4.3 will focus on studying such aspects using synthetic algorithmic tasks.

Computational efficiency. What are the additional computational costs introduced to linear Transformers by the modifications described in Sec. 3? First of all, none of them affect the core complexity

of linear Transformers: they all have a constant space and linear time complexity w.r.t. sequence length. However, the per-time-step computational costs differ a lot from one model to another, as quantified here in terms of our implementation's training speed. All models are implemented using a custom CUDA kernel except the baseline Transformer for which we use regular PyTorch code. Training speeds of LT and Delta Net in Table 1 are 66 K and 63 K words per second respectively (vs. 33 K for the baseline Transformer). The most expensive model is the Delta LSTM. This fast LSTM with tied input-forget gates has 6 weight matrices, and each of these are manipulated by separate delta rules. The corresponding speed is 14 K words per second, which is prohibitively slow to scale for more experiments. In contrast, the speeds of Delta RNN and RDN remain reasonable: 41 K and 35 K words per second respectively. Therefore, the remaining experiments will focus on these two recurrent architectures which are promising and practical in terms of both performance and computational costs.

4.2 Code Execution Task: Learning to Maintain and Update Variable States

In code execution tasks [24], models are trained to sequentially read the input code provided as word-level text, and to predict the results of the corresponding code execution. We adopt the task setting from Fan et al. [41] with one conditional and three basic statements. We refer the readers to Appendix B for a precise description of the task. This code execution task requires models to maintain the values of multiple variables, which has been shown to be difficult for Transformers with only feedforward connections [41].

The left block of Table 2 shows the results. Following again Fan et al. [41], we control the task difficulty by modifying the number of variables (3 or 5). The model architectures are fixed: the LSTM has only one layer with 256 nodes and all Transformer variants have the same architecture with 4 layers with a hidden size of 256 using 16 heads and an inner feedforward layer size of 1024.

We first note that the LSTM is the best performer for both difficulty levels with the smallest performance drop through variable increase. In contrast to prior claims [41], the LSTM is clearly capable of storing the values of multiple variables in a single vector. With three variables, the regular Transformer already largely underperforms other models with a mutable memory: Delta Net, Delta RNN, and RDN. Linear Transformers completely fail at this task, likely due to the memory capacity problem pointed out by Schlag et al. [18]. By increasing number of variables to five, the baseline Transformers, Delta Net, and RDN become unstable as shown by high standard deviations w.r.t. the seed. The benefits of recurrent connections introduced in our RDN compared to the baseline Delta Net become more apparent (76.3 vs. 61.4%). In contrast, Delta RNN remains stable and gives the best performance (85.1%) among Transformer variants, which shows the benefits of recurrence and in particular the regular RNN architecture. To match the performance of LSTM on this task, however, these models need more layers (see Appendix B for additional results).

Table 2: **Test accuracies** (%) with standard deviations on **code execution** (Code Exec) and **sequential ListOps** (Seq ListOps). The difficulty of the task is controlled by the maximum number of possible variables (# variables) for code execution, and the list depth (10 or 15) for ListOps. For code execution with 5 variables, we report means over six seeds. In all other cases, the results are computed with three seeds. For more results, see Appendix B.

	Code Exec (# variables)		**Seq ListOps** (depth)	
	3	5	10	15
LSTM	**99.0** $\pm$ 0.1	**93.2** $\pm$ 6.1	**88.5** $\pm$ 2.9	24.4 $\pm$ 1.1
Transformer	71.8 $\pm$ 2.6	35.4 $\pm$ 28.2	79.1 $\pm$ 0.9	75.3 $\pm$ 0.4
Linear Transformer	0.0 $\pm$ 0.0	0.0 $\pm$ 0.0	64.0 $\pm$ 0.3	64.4 $\pm$ 0.4
Delta Net	90.7 $\pm$ 2.7	61.4 $\pm$ 20.0	**85.7** $\pm$ 1.8	77.6 $\pm$ 1.4
Delta RNN	90.8 $\pm$ 1.7	**85.1** $\pm$ 1.9	83.6 $\pm$ 1.2	78.0 $\pm$ 1.0
RDN	**92.6** $\pm$ 2.2	76.3 $\pm$ 17.6	83.2 $\pm$ 0.9	**79.2** $\pm$ 1.4

4.3 Sequential ListOps: Learning Hierarchical Structure and Computation

The ListOps task [25] requires list operation execution which is a typical test for hierarchical structure learning. We use a simple variant of ListOps whose detailed descriptions can be found in Appendix B. For example, the list `[MAX 6 1 [FIRST 2 3 ] 0 [MIN 4 7 1] ]` is of depth two and the expected output is 6. While early research comparing self-attention to RNNs [42] has shown some advantages of recurrence in hierarchical structure learning, more recent work [43] reports Transformers outperforming LSTMs on ListOps. According to Tay et al. [16], linear Transformer variants (LT and Performers) underperform other Transformer variants by a large margin on ListOps.

The right block of Table 2 shows results for two different depths: 10 and 15. The model architectures are identical to those used in the code execution task (Sec. 4.2). At depth 10, we find LSTM to perform best, while mutable memory Transformer variants (Delta Net, Delta RNN, and RDN) outperform the regular and linear Transformers. At depth 15, the LSTM's performance drops drastically (to 24.4%), while the differences between Transformer variants remain almost the same. We note that sequences are longer for the depth 15 problem (mean length of 185 tokens) than for the depth 10 version (mean length of 98 tokens). This might add extra difficulty for the small 256-dimensional LSTM. The performance differences between the baseline Delta Net and the proposed Delta RNN and RDN are rather small for this task. Importantly, introduction of recurrence does not hurt for this task requiring hierarchical structure learning, and our novel models outperform both regular and linear Transformers.

4.4 Reinforcement Learning in 2D Game Environments

We finally evaluate the performance of our models as a direct replacement for the LSTM in reinforcement learning settings. In fact, only a limited number of prior works have investigated Transformers for reinforcement learning. Parisotto et al. [11] evaluate them on the DMLab-30 [44, 45] and on Atari but in a multi-task setting [46]. Others [41, 12] use toy maze environments. In contrast to Parisotto et al. [11]'s work, which presents multi-task Atari as a side experiment, we study the Transformer family of models on the standard Atari 2600 setting [26, 47, 48] by training game specific agents.

Settings. We train an expert agent on each game separately with the Importance Weighted Actor-Learner Training Architecture (IMPALA) using the V-trace actor-critic setup [49] and entropy regularization [50] implemented in `Torchbeast` [51]. Our model follows the *large* architecture of Espeholt et al. [49] which consists of a 15-layer residual convolutional NN with one 256-node LSTM layer which we replace by either the RDN (Sec. 3.2) or the Delta RNN (Sec. 3.1). In line with the small LSTM used for Atari (only 1 layer with 256 hidden nodes) we also configure a small RDN: 2 layers with a hidden size of 128 using 4 heads, and a feedforward dimension of 512. We find this small model to perform already surprisingly well. For the rest, we use the same hyperparameters as Espeholt et al. [49] which can be found in Appendix C.

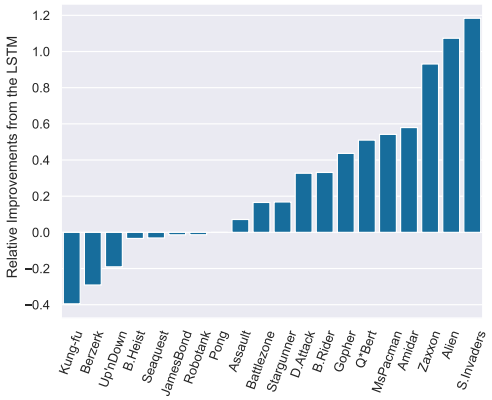

Figure 1: Rel. improvements in test scores obtained by 2-layer **RDN** compared to LSTM after **50 M** env. steps.

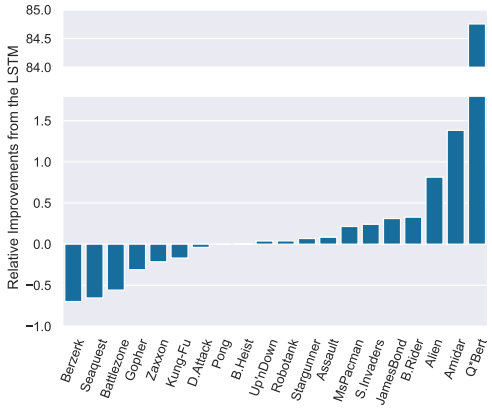

Figure 2: Rel. improvements in test scores obtained by 2-layer **RDN** compared to LSTM after **200 M** env. steps.

Main experiments. We evaluate our models in 20 environments. According to Mott et al. [52], in about half of them the LSTM outperforms the feedforward baselines — which we confirm in our setting with 50 M steps (see Appendix C). We report results at 50 M and 200 M environmental steps of training. As done by Nair et al. [53], we run the trained agent for 30 test episodes. Here we repeat this evaluation 5 times to report the average score with a standard deviation. The following analysis focuses on the RDN (Sec. 3.2). A similar study of the Delta RNN can be found in Appendix C.

Figure 1 shows relative improvements of RDN over LSTM after 50 M interactions. In 12 games, the RDN yields improvements over LSTM, whereas on 3 games, the LSTM performs better. In the remaining 5 games, both reach similar scores. The exact scores can be found in Appendix C. Interestingly, this trend does not directly extrapolate to the 200 M case, which is presented in Figure 2. With more training, the LSTM surpasses the performance of the RDN on *Battlezone*, *Gopher*, *Seaquest* and *Zaxxon*, while the RDN catches up on *Up'N Down* and *Kung-Fu Master*. Overall, there are 6 games on which LSTM clearly outperforms RDN at 200 M steps, wheras in 9 games the result is the opposite.

On a side note, some of the scores achieved by the RDN at 200 M step are excellent: a score of over 170 K and 980 K on *Space Invader* and *Q*Bert* respectively beats the state-of-the-art set by MuZero [54] and Agent57 [46]. However, a direct comparison is not fair as we train game-specific agents.

Experiments with larger models. Given the results above, a natural question to ask is whether a larger model size improves the RDN on games where the LSTM dominates. We focus on four such environements: *Battlezone*, *Berzerk*, *Gopher*, and *Seaquest* (See Fig. 2). We double the model size to 3.4 M parameters by increasing the number of layers to 4 and the hidden size to 256, with 8 heads. As shown in Table 3, larger RDN models reduce the gap to the LSTM (except in *Berzerk*). This indicates that further scaling RDN might be as promising as scaling regular Transformers in other domains.

Table 3: Performance of larger RDN on **games where the LSTM dominates** (200 M steps).

	Battlezone	Berzerk	Gopher	Seaquest
LSTM	24,873 ± 1,240	**1,150** ± 92	**124,914** ± 22,422	12,643 ± 1,627
RDN	10,980 ± 1,104	348 ± 17	86,008 ± 11,815	4,373 ± 504
RDN larger	**28,273** ± 5,333	346 ± 9	118,273 ± 14,872	**14,601** ± 712

5 Conclusion

We explore the connection between linear Transformers and Fast Weight Programmers and propose various new linear Transformer variants with recurrent connections. Our novel Recurrent Fast Weight Programmers (RFWPs) outperform previous linear and regular Transformers on a code execution task and significantly improve over Transformers in a sequential ListOps task. On Wikitext-103 in the "small" model setting, RFWPs perform competitively compared to previous best linear Transformer variants for truncated contexts and beat regular Transformers for full contexts. Our RFWPs can also be used as drop-in replacements for problems where RNNs are still dominant. In particular, we evaluate them in a reinforcement learning settings on 20 Atari 2600 environments. We obtain clear improvements over LSTMs across many environments with small models and demonstrate promising scaling properties for larger models. Our work highlights the usefulness of the FWP framework from the '90s and its connection to modern architectures, opening promising avenues for further research into new classes of recurrent Transformers.

Broader impact and limitations. Development of Transformers has resulted in many useful real world applications involving text, audio, and vision. Our work contributes in advancing linear attention variants thereof which are crucial for auto-regressive problems with long sequences because of their better time and space complexity. Our methods are evaluated on standard benchmarks which permit a direct comparison with similar work. However, applying such models to real world applications would require extra experiments and scaling up these models on larger data can be costly in terms of energy consumption. Also, training our models on biased data can result in learning undesirable properties.

References

[1] 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.

[2] Rami Al-Rfou, Dokook Choe, Noah Constant, Mandy Guo, and Llion Jones. Character-level language modeling with deeper self-attention. In *Proc. Conference on Artificial Intelligence (AAAI)*, pages 3159–3166, Honolulu, HI, USA, January 2019.

[3] Tom B Brown et al. Language models are few-shot learners. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Virtual only, December 2020.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proc. North American Chapter of the Association for Computational Linguistics on Human Language Technologies (NAACL-HLT)*, pages 4171–4186, Minneapolis, MN, USA, June 2019.

[5] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

[6] Xizhou Zhu, Weijie Su, Lewei Lu, Bin Li, Xiaogang Wang, and Jifeng Dai. Deformable DETR: Deformable transformers for end-to-end object detection. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

[7] David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. In *Int. Conf. on Learning Representations (ICLR)*, New Orleans, LA, USA, May 2019.

[8] Francois Charton, Amaury Hayat, and Guillaume Lample. Learning advanced mathematical computations from examples. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

[9] Zihang Dai, Zhilin Yang, Yiming Yang, William W Cohen, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proc. Association for Computational Linguistics (ACL)*, pages 2978–2988, Florence, Italy, July 2019.

[10] Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, Chloe Hillier, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, April 2020.

[11] Emilio Parisotto, Francis Song, Jack Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, et al. Stabilizing Transformers for reinforcement learning. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 7487–7498, Virtual only, July 2020.

[12] Emilio Parisotto and Ruslan Salakhutdinov. Efficient transformers in reinforcement learning using actor-learner distillation. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

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

[14] 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.

[15] 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.

[16] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena: A benchmark for efficient transformers. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 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] Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight memory systems. In *Proc. Int. Conf. on Machine Learning (ICML)*, Virtual only, July 2021.

[19] 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.

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

[21] 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.

[22] Michael Hahn. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171, 2020.

[23] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *Int. Conf. on Learning Representations (ICLR)*, Toulon, France, April 2017.

[24] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *Preprint arXiv:1410.4615*, 2014.

[25] Nikita Nangia and Samuel Bowman. ListOps: A diagnostic dataset for latent tree learning. In *Proc. North American Chapter of the Association for Computational Linguistics (NAACL): Student Research Workshop*, pages 92–99, New Orleans, LA, USA, June 2018.

[26] 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.

[27] 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, Institut für Informatik, Technische Universität München, 1990.

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

[29] Benjamin Klein, Lior Wolf, and Yehuda Afek. A dynamic convolutional layer for short rangeweather prediction. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 4840–4848, Boston, MA, USA, June 2015.

[30] Hyeonwoo Noh, Paul Hongsuck Seo, and Bohyung Han. Image question answering using convolutional neural network with dynamic parameter prediction. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 30–38, Las Vegas, NV, USA, 2016.

[31] Xu Jia, Bert De Brabandere, Tinne Tuytelaars, and Luc V Gool. Dynamic filter networks. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 667–675, Barcelona, Spain, 2016.

[32] Irwan Bello. Lambdanetworks: Modeling long-range interactions without attention. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

[33] 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.

[34] 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.

[35] Tsendsuren Munkhdalai, Alessandro Sordoni, Tong Wang, and Adam Trischler. Metalearned neural memory. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 13310–13321, Vancouver, Canada, December 2019.

[36] 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.

[37] Imanol Schlag, Tsendsuren Munkhdalai, and Jürgen Schmidhuber. Learning associative inference using fast weight memory. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

[38] Imanol Schlag and Jürgen Schmidhuber. Gated fast weights for on-the-fly neural program generation. In *NIPS Metalearning Workshop*, Long Beach, CA, USA, December 2017.

[39] T Anderson Keller, Sharath Nittur Sridhar, and Xin Wang. Fast weight long short-term memory. *Preprint arXiv:1804.06511*, 2018.

[40] Gail Weiss, Yoav Goldberg, and Eran Yahav. On the practical computational power of finite precision rnns for language recognition. In Iryna Gurevych and Yusuke Miyao, editors, *Proc. Association for Computational Linguistics (ACL)*, pages 740–745, Melbourne, Australia, July 2018.

[41] Angela Fan, Thibaut Lavril, Edouard Grave, Armand Joulin, and Sainbayar Sukhbaatar. Addressing some limitations of Transformers with feedback memory. *Preprint arXiv:2002.09402*, 2020.

[42] Ke Tran, Arianna Bisazza, and Christof Monz. The importance of being recurrent for modeling hierarchical structure. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4731–4736, Brussels, Belgium, October 2018.

[43] Kevin Lu, Aditya Grover, Pieter Abbeel, and Igor Mordatch. Pretrained transformers as universal computation engines. *Preprint arXiv:2103.05247*, 2021.

[44] Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *Preprint arXiv:1612.03801*, 2016.

[45] Joel Z Leibo, Cyprien de Masson d'Autume, Daniel Zoran, David Amos, Charles Beattie, Keith Anderson, Antonio García Castañeda, Manuel Sanchez, Simon Green, Audrunas Gruslys, et al. Psychlab: a psychology laboratory for deep reinforcement learning agents. *Preprint arXiv:1801.08116*, 2018.

[46] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, and Charles Blundell. Agent57: Outperforming the atari human benchmark. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 507–517, Virtual only, July 2020.

[47] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*, Lake Tahoe, NV, USA, December 2013.

[48] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[49] 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.

[50] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proc. Int. Conf. on Machine Learning (ICML)*, pages 1928–1937, New York City, NY, USA, June 2016.

[51] 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.

[52] Alexander Mott, Daniel Zoran, Mike Chrzanowski, Daan Wierstra, and Danilo Jimenez Rezende. Towards interpretable reinforcement learning using attention augmented agents. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pages 12329–12338, Vancouver, Canada, December 2019.

[53] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. In *Deep Learning Workshop, International Conference on Machine Learning (ICML)*, Lille, France, July 2015.

[54] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering Atari, Go, Chess and Shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

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

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] See discussion on computational cost in Sec. 4.1, as well as "Broader impact and limitations" paragraph in Conclusion.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See our "Broader impact and limitations" paragraph in Conclusion.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A] Our work is empirical.
   (b) Did you include complete proofs of all theoretical results? [N/A] Our work is empirical.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] Please check the supplemental material.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Please check the supplemental material.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] We provide mean and std for experiments which are not too expensive to be run multiple times.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] Please check the supplemental material.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]

(b) Did you mention the license of the assets? [No] The license of the used assets is discussed in the referenced documents.

(c) Did you include any new assets either in the supplemental material or as a URL? [Yes] We provide our code in the supplemental material.

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [No] We only run on default benchmarks or synthetic data that was not obtained from people.

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [No] This only applies to the Wikitext-103 dataset which is a standard dataset composed of Wikipedia articles. As such it follows the Wikipedia guidelines which do not contain personally identifiable information or offensive content.

5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]