A. Update Rule Derivation

A.1. The Update Rule

Here we provide the intermediate steps from Eq. 23 to Eq. 24.

\[ W^{(i)} = W^{(i-1)} + \beta^{(i)} v^{(i)} - \bar{v}^{(i)} \otimes \phi(k^{(i)}) \] (23)

By grouping the last two terms, Eq. 23 becomes:

\[ W^{(i)} = W^{(i-1)} + \beta^{(i)} (v^{(i)} - \bar{v}^{(i)}) \otimes \phi(k^{(i)}) \] (24)

By using the definition of \( v^{(i)} \) from Eq. 22:

\[ v^{(i)} = \beta^{(i)} v^{(i)} + (1 - \beta^{(i)}) \bar{v}^{(i)} \] (22)

we obtain:

\[ v^{(i)} - \bar{v}^{(i)} = \beta^{(i)} v^{(i)} + (1 - \beta^{(i)}) \bar{v}^{(i)} - \bar{v}^{(i)} = \beta^{(i)} (v^{(i)} - \bar{v}^{(i)}) \] (39)

By substituting this expression to Eq. 38, we obtain Eq. 24:

Now by substituting \( W \) by its expression of Eq. 41:

\[ W^{(i)} = \sum_{i=1}^{d_{\text{key}}} w^{(i)} \otimes e^{(i)} + \sum_{i=1}^{d_{\text{key}}} k_i (v - \bar{v}) \otimes e^{(i)} \] (46)

which we can substitute in Eq. 48 to obtain:

\[ w^{(i)} = w^{(i)} + k_i (v - \bar{v}) \] (49)

B. Formal comparison to Peng et al. (2021)

Concurrently to our work, Peng et al. (2021) proposed the following update rule:

\[ W^{(i)} = (1 - \beta^{(i)}) W^{(i-1)} + \beta^{(i)} v^{(i)} \otimes \phi(k^{(i)}) \] (52)
which is motivated by the gating mechanism in recurrent neural networks (Hochreiter & Schmidhuber, 1997). In contrast, our update rule of Eq. (24)

\[ W^{(i)} = W^{(i-1)} + \beta^{(i)}(v^{(i)} - \bar{v}^{(i)}) \otimes \phi(k^{(i)}) \]  

(24)

is driven by an associative memory perspective, relates to the famous error-correcting delta rule, and offers a crucial property.

To illustrate a similarity and a crucial difference between the two update rules, we consider a fast weight matrix \( W \) which is constructed by two associations \((k_1, v_1)\) and \((k_2, v_2)\), i.e.

\[ W = v_1 \otimes k_1 + v_2 \otimes k_2 \]  

(53)

where we assume \( k_1 \) and \( k_2 \) to be orthonormal, and we omit \( \phi \). Now we consider updating \( W \) to \( W' \) by adding a new association \((k_3, v_3)\) where \( k_3 = k_2 \). Using Peng et al. (2021)’s update rule, we have:

\[ W' = (1 - \beta)W + \beta v_3 \otimes k_3 \]

This rule thus updates the value associated with the key \( k_2 = k_3 \) to be a convex combination of the old and the new values \((1 - \beta)v_2 + \beta v_3\):

\[ W'k_3 = (1 - \beta)Wk_3 + \beta v_3 = (1 - \beta)v_2 + \beta v_3 \]

However, it also modifies or in the worst case erases the value associated with the key \( k_1 \):

\[ W'k_1 = (1 - \beta)Wk_1 = (1 - \beta)v_1 \]

In contrast, using our update rule, we have:

\[ W' = W + \beta(v_3 - v_2) \otimes k_3 \]

since \( \bar{v} = Wk_3 = Wk_2 = v_2 \). Our rule thus also updates the value associated with the key \( k_2 = k_3 \) to be a convex combination of the old and the new values \((1 - \beta)v_2 + \beta v_3\):

\[ W'k_3 = Wk_3 + \beta(v_3 - v_2) = v_2 + \beta(v_3 - v_2) = (1 - \beta)v_2 + \beta v_3 \]

while crucially, it keeps the value associated with \( k_1 \) unmodified:

\[ W'k_1 = Wk_1 = v_1 \]

Our update rule thus differs from Peng et al. (2021)’s one on this property of updating associations while keeping other “unrelated” ones intact in an associative memory.

### C. DPFP-\( \nu \) Implementation

Listing 1 is a simple PyTorch implementation of DPFP-\( \nu \) (Eq. 37) which consist of two concatenations followed by one element-wise multiplication.

```python
import torch
from torch.nn.functional import relu as r

def dpfp(x, nu=1):
    x = cat([r(x), r(-x)], dim=-1)
    x Rolled = cat([x.roll(shifts=j, dims=-1) for j in range(1, nu+1)], dim=-1)
    x repeat = cat([x Repeat * nu, dim=-1])
    return x repeat * x rolled
```

Listing 1. Simple PyTorch implementation of DPFP-\( \nu \) (Eq. 37).

### D. Additional Experimental Results

In this section, we provide additional experimental results which we could not include in the main paper because of space limitations.

#### D.1. Synthetic Task Setting 1

Figure 4 shows learning curves for the synthetic setting 1 (without replacement) with 600 unique keys and values. The scripts used to generate such figures can be found in our GitHub repository.

![Setting 1 with 600 unique keys](image)

Figure 4. Training curves for setting 1 with 600 unique keys/values (sampled without replacement) as described in Sec. 6.1.1.

#### D.2. Synthetic Task Setting 2

Figure 5 is a capacity plot for setting 2 with an increasing number of unique keys and queries (analogous to Figure 2 of setting 1 apart from the log-scale of the y-axis). We did not include FAVOR+ in this plot, because its combination with our update rule resulted in not-a-number in this setting.
Linear Transformers Are Secretly Fast Weight Programmers

D.3. Language Modelling

In Sec. 6.3, we evaluated our update rule when the model is under overcapacity regime. Here we present an extra language modelling experiment which evaluate the benefits of our update rule in non-overcapacity scenarios. This also allows us to include DPFP in the evaluation. We train both, Performer and DPFP, in the small setting \( (D = 128, L = 256) \) with \( m = 16 \) and \( \nu = 1 \), resulting in \( d_{\text{dot}} = 256 \) for both cases. Table 5 shows the perplexity results. First we observe that the Performer and DPFP baseline models with the sum update rule do not outperform the Linear Transformer baseline from Table 2. In fact, language modelling might be less affected by the capacity issue than the synthetic retrieval task, as it might not require the exact retrieval. Second we observe that our update rule improves both variants of linear attention over the sum update-rule baselines even in this condition. This indicates the general benefits of our update rule in Fast Weight Programmers. We note that the improvement is larger for the DPFP model than for the Performer. This is similar to Table 2 where our update rule improves the deterministic Linear Transformers more than the Performers. Finally, we note that we also tried the DPFP and Performer models with an increased \( d_{\text{dot}} \) by setting \( \nu = 2 \) and \( m = 32 \) respectively. While this increases \( d_{\text{dot}} \) by a factor of two, it was not beneficial for this language modelling setting.

E. Details on Machine Translation Experiments

We implemented different \( \phi \) functions in the FAIRSEQ toolkit (Ott et al., 2019). The Transformer architecture used in the experiment is the one referred to as big in the original Transformer paper (Vaswani et al., 2017): the model has 6 layers each in the encoder and the decoder, with a hidden layer size of 1024 with 16 attention heads, 4096-dimensional feed-forward layers, using 32 K byte-pair encoding sub-word units (Sennrich et al., 2016). FAIRSEQ provides a training configuration for the corresponding model (Ott et al., 2018), which we adapted for our infrastructure. We trained our models on three GPUs using a batch size of up to 3584 tokens per GPU and accumulating gradients over 16 batches for 45 epochs, and selected the best model based on the validation BLEU score. In Table 1, we directly report BLEU for different values of \( d_{\text{dot}} \); Table 6 provides the conversion from hyper-parameters \( m \) of Performers or \( \nu \) in the DPFP to \( d_{\text{dot}} \).

<table>
<thead>
<tr>
<th>Update Rule</th>
<th>small Valid</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer</td>
<td>-</td>
<td>33.0</td>
</tr>
<tr>
<td>Performer</td>
<td>sum</td>
<td>38.0</td>
</tr>
<tr>
<td></td>
<td>delta</td>
<td>36.0</td>
</tr>
<tr>
<td>DPFP</td>
<td>sum</td>
<td>37.7</td>
</tr>
<tr>
<td></td>
<td>delta</td>
<td>33.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( d_{\text{dot}} )</th>
<th>256</th>
<th>384</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performer ( m )</td>
<td>128</td>
<td>192</td>
<td>256</td>
</tr>
<tr>
<td>DPFP ( \nu )</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
the backward pass for fast weights is crucial for language modelling. A naive backward computation generated by automatic differentiation would store the fast weights for each time step, which can quickly hit the GPU memory limit. The custom implementation ensures that we need to store only one set of weights by recomputing the fast weights needed for computing the gradients for each time step in the backward pass (which still remains time-efficient as the operations involved in the computation of our fast weights are rather inexpensive).

**Experimental details.** Here we provide extra experimental details to complement the descriptions of Sec. 6.3. For the small and medium configurations, we use batch sizes of 96 and 56 sequences, respectively, and train for about 120 and 70 epochs. In both settings, we apply 10% dropout (Hanson, 1990; Srivastava et al., 2014), and train using the Adam optimiser (Kingma & Ba, 2014) with an initial learning rate of 0.00025 and 2000 learning rate warm-up steps. For further details, we refer the readers to our code. For experiments with Transformer-XL (Table 4), we train it with the same backpropagation span as our models (i.e. 384 words in the medium configuration). The model is trained with memory and target segment lengths of 384. The models with different state sizes in Table 4 are obtained by using different Transformer-XL memory segment lengths at evaluation time. The models with state sizes of 1.05 M, 2.10 M, and 6.29 M are obtained by using memory and target lengths of 64, 128, and 384, respectively. The model with a state size of 0.13 M uses a memory length of 15 and a target length of 1. Like for other models, a batch size of 1 is used for evaluating the Transformer XL. The state sizes in Table 4 are computed as follows. The per-layer state size of the Linear Transformer and the Delta Net are: number of heads (here 8) × fast weight matrix size which is per-head key dimension (here 32) × per-head value dimension (here 32). This yields a total size of 8,192. The per-layer state size of the Transformer XL is: memory segment length × target segment length × (total key dimension, here 256 + total value dimension, here 256). We obtain the total state size we report in Table 4 by multiplying the per-layer state size by the number of layers which is 16 for all our models.