A. Update Rule Derivation

A.1. The Update Rule

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

$$\boldsymbol{W}^{(i)} = \boldsymbol{W}^{(i-1)} \underbrace{+\boldsymbol{v}^{(i)}_{\text{new}} \otimes \phi(\boldsymbol{k}^{(i)})}_{\text{write}} \underbrace{-\bar{\boldsymbol{v}}^{(i)} \otimes \phi(\boldsymbol{k}^{(i)})}_{\text{remove}} \tag{23}$$

$$= \boldsymbol{W}^{(i-1)} + \beta^{(i)}(\boldsymbol{v}^{(i)} - \bar{\boldsymbol{v}}^{(i)}) \otimes \phi(\boldsymbol{k}^{(i)}) \tag{24}$$

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

$$\boldsymbol{W}^{(i)} = \boldsymbol{W}^{(i-1)} + (\boldsymbol{v}^{(i)}_{\text{new}} - \bar{\boldsymbol{v}}^{(i)}) \otimes \phi(\boldsymbol{k}^{(i)}) \tag{38}$$

By using the definition of $\boldsymbol{v}^{(i)}_{\text{new}}$ from Eq. 22:

$$\boldsymbol{v}^{(i)}_{\text{new}} = \beta^{(i)}\boldsymbol{v}^{(i)} + (1-\beta^{(i)})\bar{\boldsymbol{v}}^{(i)} \tag{22}$$

we obtain:

$$\boldsymbol{v}^{(i)}_{\text{new}} - \bar{\boldsymbol{v}}^{(i)} = \beta^{(i)}\boldsymbol{v}^{(i)} + (1-\beta^{(i)})\bar{\boldsymbol{v}}^{(i)} - \bar{\boldsymbol{v}}^{(i)} \tag{39}$$

$$= \beta^{(i)}(\boldsymbol{v}^{(i)} - \bar{\boldsymbol{v}}^{(i)}) \tag{40}$$

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

A.2. Key Sum Normalisation

By considering one-hot vectors $\{\boldsymbol{e}^{(1)}, ..., \boldsymbol{e}^{(i)}, ..., \boldsymbol{e}^{(d_{\text{key}})}\}$ which form the Cartesian basis of $\mathbb{R}^{d_{\text{key}}}$, any matrix $\boldsymbol{W} \in \mathbb{R}^{d_{\text{value}} \times d_{\text{key}}}$ can be written as

$$\boldsymbol{W} = \sum_{i=1}^{d_{\text{key}}} \boldsymbol{w}^{(i)} \otimes \boldsymbol{e}^{(i)} \tag{41}$$

where $\{\boldsymbol{w}^{(1)}, ..., \boldsymbol{w}^{(i)}, ..., \boldsymbol{w}^{(d_{\text{key}})}\}$ are the column vectors of $\boldsymbol{W}$. In the context of associative memory, we can interpret this expression as a set of associations with fixed keys $\boldsymbol{e}^{(i)}$ and the associated values $\boldsymbol{w}^{(i)}$.

In this view, any update of $\boldsymbol{W}$ can be written as updates of each $\boldsymbol{w}^{(i)}$. This perspective allows us to derive the *sum normalisation* of Sec. 4.2. For that, we start by deriving the update of $\boldsymbol{w}^{(i)}$.

Given an arbitrary weight $\boldsymbol{W}$, we consider updating it to $\boldsymbol{W}'$ by adding a new association $(\boldsymbol{k}, \boldsymbol{v})$ using our update rule of Sec. 4.2 (where we omit β):

$$\bar{\boldsymbol{v}} = \boldsymbol{W}\boldsymbol{k} \tag{42}$$

$$\boldsymbol{W}' = \boldsymbol{W} + (\boldsymbol{v} - \bar{\boldsymbol{v}}) \otimes \boldsymbol{k} \tag{43}$$

By substituting $\boldsymbol{k}$ in Eq. 43 by its expression in the Cartesian basis $\boldsymbol{k} = \sum_{i=1}^{d_{\text{key}}} k_i \boldsymbol{e}^{(i)}$ with $k_i \in \mathbb{R}$, we obtain:

$$\boldsymbol{W}' = \boldsymbol{W} + (\boldsymbol{v} - \bar{\boldsymbol{v}}) \otimes \sum_{i=1}^{d_{\text{key}}} k_i \boldsymbol{e}^{(i)} \tag{44}$$

$$= \boldsymbol{W} + \sum_{i=1}^{d_{\text{key}}} k_i (\boldsymbol{v} - \bar{\boldsymbol{v}}) \otimes \boldsymbol{e}^{(i)} \tag{45}$$

Now by substituting $\boldsymbol{W}$ by its expression of Eq. 41:

$$\boldsymbol{W}' = \sum_{i=1}^{d_{\text{key}}} \boldsymbol{w}^{(i)} \otimes \boldsymbol{e}^{(i)} + \sum_{i=1}^{d_{\text{key}}} k_i (\boldsymbol{v} - \bar{\boldsymbol{v}}) \otimes \boldsymbol{e}^{(i)} \tag{46}$$

$$= \sum_{i=1}^{d_{\text{key}}} \left(\boldsymbol{w}^{(i)} + k_i (\boldsymbol{v} - \bar{\boldsymbol{v}})\right) \otimes \boldsymbol{e}^{(i)} \tag{47}$$

The column-wise update is thus:

$$\boldsymbol{w}'^{(i)} = \boldsymbol{w}^{(i)} + k_i (\boldsymbol{v} - \bar{\boldsymbol{v}}) \tag{48}$$

We can explicitly write down $\bar{\boldsymbol{v}}$ as:

$$\bar{\boldsymbol{v}} = \boldsymbol{W}\boldsymbol{k} = \boldsymbol{W} \sum_{j=1}^{d_{\text{key}}} k_j \boldsymbol{e}^{(j)} = \sum_{j=1}^{d_{\text{key}}} k_j \boldsymbol{w}^{(j)} \tag{49}$$

which we can substitute in Eq. 48 to obtain:

$$\boldsymbol{w}'^{(i)} = \boldsymbol{w}^{(i)} + k_i (\boldsymbol{v} - \sum_{j=1}^{d_{\text{key}}} k_j \boldsymbol{w}^{(j)}) \tag{50}$$

$$= \boldsymbol{w}^{(i)} + k_i \boldsymbol{v} - \sum_{j=1}^{d_{\text{key}}} k_i k_j \boldsymbol{w}^{(j)} \tag{51}$$

In Eq. 51, the *weight* k_i on the positive term $\boldsymbol{v}$ is in general not equal to the total weights on the negative terms $\sum_{j=1}^{d_{\text{key}}} k_i k_j$. We can force these weights to be balanced by introducing the normalisation: $\sum_{j=1}^{d_{\text{key}}} k_i k_j = k_i$.

If k_i is non zero, we obtain:

$$\sum_{j=1}^{d_{\text{key}}} k_j = 1$$

This corresponds to the *sum normalisation* we introduced in Sec. 4.2 □.

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

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

$$\boldsymbol{W}^{(i)} = (1-\beta^{(i)})\boldsymbol{W}^{(i-1)} + \beta^{(i)}\boldsymbol{v}^{(i)} \otimes \phi(\boldsymbol{k}^{(i)}) \tag{52}$$

which is motivated by the gating mechanism in recurrent neural networks (Hochreiter & Schmidhuber, 1997). In contrast, our update rule of Eq. 24

$$\boldsymbol{W}^{(i)} = \boldsymbol{W}^{(i-1)} + \beta^{(i)}(\boldsymbol{v}^{(i)} - \bar{\boldsymbol{v}}^{(i)}) \otimes \phi(\boldsymbol{k}^{(i)}) \quad (24)$$

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 $\boldsymbol{W}$ which is constructed by two associations $(\boldsymbol{k}_1, \boldsymbol{v}_1)$ and $(\boldsymbol{k}_2, \boldsymbol{v}_2)$, i.e.

$$\boldsymbol{W} = \boldsymbol{v}_1 \otimes \boldsymbol{k}_1 + \boldsymbol{v}_2 \otimes \boldsymbol{k}_2 \quad (53)$$

where we assume $\boldsymbol{k}_1$ and $\boldsymbol{k}_2$ to be orthonormal, and we omit ϕ. Now we consider updating $\boldsymbol{W}$ to $\boldsymbol{W}'$ by adding a new association $(\boldsymbol{k}_3, \boldsymbol{v}_3)$ where $\boldsymbol{k}_3 = \boldsymbol{k}_2$. Using Peng et al. (2021)'s update rule, we have:

$$\boldsymbol{W}' = (1-\beta)\boldsymbol{W} + \beta \boldsymbol{v}_3 \otimes \boldsymbol{k}_3$$

This rule thus updates the value associated with the key $\boldsymbol{k}_2 = \boldsymbol{k}_3$ to be a convex combination of the old and the new values $(1-\beta)\boldsymbol{v}_2 + \beta \boldsymbol{v}_3$:

$$\begin{aligned}\boldsymbol{W}'\boldsymbol{k}_3 &= (1-\beta)\boldsymbol{W}\boldsymbol{k}_3 + \beta \boldsymbol{v}_3 \\ &= (1-\beta)\boldsymbol{v}_2 + \beta \boldsymbol{v}_3\end{aligned}$$

However, it also modifies or in the worst case erases the value associated with the key $\boldsymbol{k}_1$:

$$\boldsymbol{W}'\boldsymbol{k}_1 = (1-\beta)\boldsymbol{W}\boldsymbol{k}_1 = (1-\beta)\boldsymbol{v}_1$$

In contrast, using our update rule, we have:

$$\boldsymbol{W}' = \boldsymbol{W} + \beta(\boldsymbol{v}_3 - \boldsymbol{v}_2) \otimes \boldsymbol{k}_3$$

since $\bar{\boldsymbol{v}} = \boldsymbol{W}\boldsymbol{k}_3 = \boldsymbol{W}\boldsymbol{k}_2 = \boldsymbol{v}_2$.
Our rule thus also updates the value associated with the key $\boldsymbol{k}_2 = \boldsymbol{k}_3$ to be a convex combination of the old and the new values $(1-\beta)\boldsymbol{v}_2 + \beta \boldsymbol{v}_3$:

$$\begin{aligned}\boldsymbol{W}'\boldsymbol{k}_3 &= \boldsymbol{W}\boldsymbol{k}_3 + \beta(\boldsymbol{v}_3 - \boldsymbol{v}_2) \\ &= \boldsymbol{v}_2 + \beta(\boldsymbol{v}_3 - \boldsymbol{v}_2) \\ &= (1-\beta)\boldsymbol{v}_2 + \beta \boldsymbol{v}_3\end{aligned}$$

while crucially, it keeps the value associated with $\boldsymbol{k}_1$ unmodified:

$$\boldsymbol{W}'\boldsymbol{k}_1 = \boldsymbol{W}\boldsymbol{k}_1 = \boldsymbol{v}_1$$

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-ν Implementation

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

```
import torch
from torch import cat
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] * nu, dim=-1)
  return x_repeat * x_rolled
```

Listing 1. Simple PyTorch implementation of DPFP-ν (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.

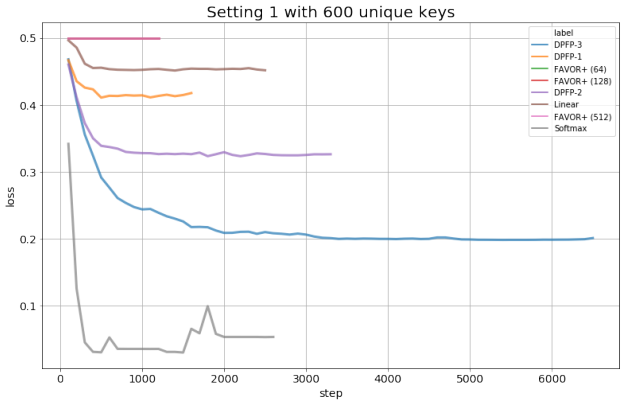

Figure 4. Training curves for setting 1 with 600 unique keys/values (sampled without replacement) as described in Sec. 6.1.1.

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.

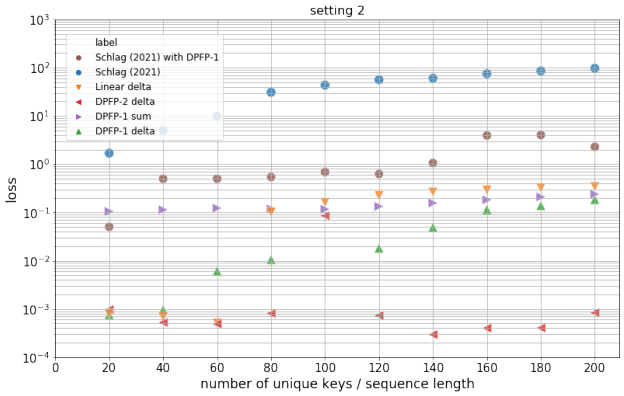

Figure 5. Final evaluation loss on synthetic setting 2 (with replacement) problems with the total number of unique associations ranging from 20 to 200. Each individual symbol is a model trained until convergence as described in Sec. 6.1.2. In all problems, with different sequence lengths and a different number of unique keys, our update rule outperforms all other approaches.

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_{dot} by setting $\nu = 2$ and $m = 32$ respectively. While this increases d_{dot} by a factor of two, it was not beneficial for this language modelling setting.

E. Details on Machine Translation Experiments

We implemented different ϕ functions in the FAIRSEQ tookit (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_{dot}; Table 6 provides the conversion from hyper-parameters m of Performers or ν in the DPFP to d_{dot}.

Table 5. WikiText-103 language model perplexity results showing effects of our update rule in non-overcapacity regime. The number of trainable parameters are almost the same for all models, up to the small difference introduced by gating in our update rule (16 K parameters). The *small* config is used, i.e. $D = 128$, $L = 256$ (40 M parameters). We set $m = 16$ for the Performers and $\nu = 1$ for the DPFP models, which result in $d_{\text{dot}} = 256$ for both cases. The model is thus not necessary in an overcapacity regime.

	Update Rule	*small* Valid	*small* Test
Transformer	-	33.0	34.1
Performer	sum	38.0	38.8
	delta	**36.0**	**37.0**
DPFP	sum	37.7	38.8
	delta	**33.9**	**35.0**

Table 6. Relation between dot product space dimension and the hyper-parameters in the Performer and our DPFP models. $d_{\text{key}} = 64$ in all our translation models.

d_{dot}	256	384	512
Performer m	128	192	256
DPFP ν	2	3	4

F. Details on Language Modelling Experiments

Implementation notes. All our implementations are based on PyTorch (Paszke et al., 2019). Our base language modelling code has been developed by using the public code by Dai et al. (2019) for Transformer-XL as a starting point. For ϕ functions, we ported the same implementation we used for our translation experiments. For the implementation of our update rule, we modified the CUDA kernel for the Linear Transformer made publicly available by Katharopoulos et al. (2020). We note that a custom implementation of

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) $\times$ fast weight matrix size which is per-head key dimension (here 32) $\times$ 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 $\times$ target segment length $\times$ (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.