Hopfield Networks is All You Need

Blog post

View on GitHub

This blog post explains the paper Hopfield Networks is All You Need and the corresponding new PyTorch Hopfield layer.

Table of Contents

Main contributions

We introduce a new energy function and a corresponding new update rule which is guaranteed to converge to a local minimum of the energy function.

The new energy function is a generalization (discrete states $\Rightarrow$ continuous states) of **modern Hopfield Networks** aka **Dense Associative Memories** introduced by Krotov and Hopfield and Demircigil et al. The new modern Hopfield Network with continuous states keeps the characteristics of its discrete counterparts:

- exponential storage capacity
- extremely fast convergence

Due to its continuous states this new modern Hopfield Network is differentiable and can be integrated into deep learning architectures. Typically patterns are retrieved after one update which is

compatible with activating the layers of deep networks. This enables an abundance of **new deep learning architectures**. Three useful types of Hopfield layers are provided.

Surprisingly, the new update rule is the attention mechanism of transformer networks introduced in Attention Is All You Need. We use these new insights to analyze transformer models in the paper.

What this blog post is about

This blog post is split into three parts. First, we make the transition from traditional Hopfield Networks towards **modern Hopfield Networks** and their generalization to continuous states through our **new energy function**. Second, the properties of our new energy function and the connection to the self-attention mechanism of transformer networks is shown. Finally, we introduce and explain a new PyTorch layer (Hopfield layer), which is built on the insights of our work. We show several practical use cases, i.e. Modern Hopfield Networks and Attention for Immune Repertoire Classification, Hopfield pooling, and associations of two sets.

From classical Hopfield Networks to self-attention

Associative memories are one of the earliest artificial neural models dating back to the 1960s and 1970s. Best known are Hopfield Networks, presented by John Hopfield in 1982. As the name suggests, the main purpose of associative memory networks is to associate an input with its most similar pattern. In other words, the purpose is to store and retrieve patterns. We start with a review of classical Hopfield Networks.

Hopfield Networks

The simplest associative memory is just a **sum of outer products** of the N patterns $\{\boldsymbol{x}_i\}_{i=1}^N$ that we want to store (Hebbian learning rule). In classical Hopfield Networks these patterns are polar (binary), i.e. $\boldsymbol{x}_i \in \{-1, 1\}^d$, where d is the length of the patterns. The corresponding weight matrix $\boldsymbol{W}$ is:

$$\boldsymbol{W} = \sum_i^N \boldsymbol{x}_i \boldsymbol{x}_i^T \ . \tag{1}$$

The weight matrix $\boldsymbol{W}$ stores the patterns, which can be retrieved starting with a **state pattern** $\boldsymbol{\xi}$.

Nomenclature

From now on we denote the N **stored patterns** as $\{\boldsymbol{x}_i\}_{i=1}^N$ and any **state pattern** or **state** as $\boldsymbol{\xi}$.

The basic **synchronuous update rule** is to repeatedly multiply the state pattern $\boldsymbol{\xi}$ with the weight matrix $\boldsymbol{W}$, subtract the bias and take the sign:

$$\boldsymbol{\xi}^{t+1} = \mathrm{sgn}(\boldsymbol{W}\boldsymbol{\xi}^t - \boldsymbol{b}) \ , \tag{2}$$

where $\boldsymbol{b} \in \mathbb{R}^d$ is a bias vector, which can be interpreted as threshold for every component. The **asynchronous update rule** performs this update only for one component of $\boldsymbol{\xi}$ and then selects the next component for update. Convergence is reached if $\boldsymbol{\xi}^{t+1} = \boldsymbol{\xi}^t$.

The asynchronous version of the update rule of Eq. (2) minimizes the **energy function** E:

$$\mathrm{E} = -\frac{1}{2}\boldsymbol{\xi}^T \boldsymbol{W}\boldsymbol{\xi} + \boldsymbol{\xi}^T \boldsymbol{b} = -\frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d} w_{ij}\xi_i\xi_j + \sum_{i=1}^{d} b_i\xi_i \ . \tag{3}$$

As derived in the papers of Bruck, Goles-Chacc et al. and the original Hopfield paper, the convergence properties are dependent on the structure of the weight matrix $\boldsymbol{W}$ and the method by which the nodes are updated:

- For asynchronous updates with $w_{ii} \geq 0$ and $w_{ij} = w_{ji}$, the updates converge to a stable state.
- For synchronous updates with $w_{ij} = w_{ji}$, the updates converge to a stable state or a limit cycle of length 2.

For the asynchronous update rule and symmetric weights, $\mathrm{E}(\boldsymbol{\xi}^{t+1}) \leq \mathrm{E}(\boldsymbol{\xi}^t)$ holds. When $\mathrm{E}(\boldsymbol{\xi}^{t+1}) = \mathrm{E}(\boldsymbol{\xi}^t)$ for the update of every component of $\boldsymbol{\xi}^t$, a local minimum in E is reached. All stored patterns $\{\boldsymbol{x}_i\}_{i=1}^N$ should be fixed points of the Hopfield Network, i.e.

$$\boldsymbol{x}_i = \mathrm{sgn}(\boldsymbol{W}\boldsymbol{x}_i - \boldsymbol{b}) \ . \tag{4}$$

They should even be local minima of E.

In the following example, no bias vector is used. This means that taking the inverse image, i.e. flipping all pixels at once, results in the same energy.

We start with an **illustrative example** of a Hopfield Network. **One input image** should first be stored and then be retrieved. The input image is:

Since an associative memory has **polar states and patterns** (or binary states and patterns), we convert the input image to a black and white image:

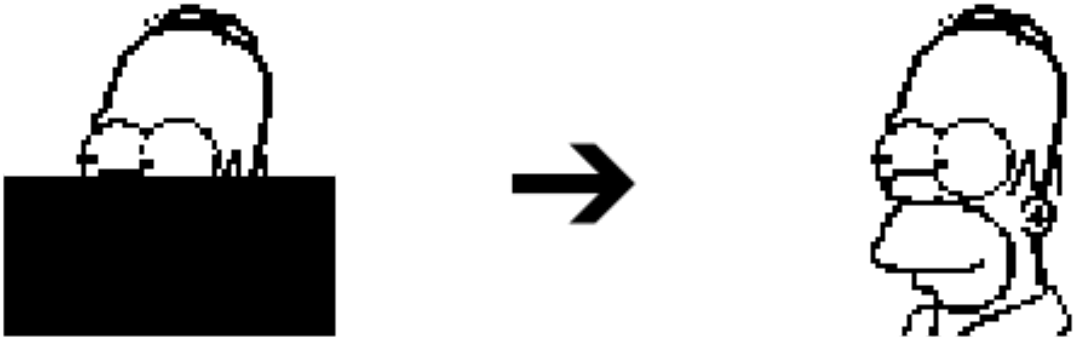
The **weight matrix** $\boldsymbol{W}$ is the outer product of this black and white image $\boldsymbol{x}_{\text{Homer}}$:

$$\boldsymbol{W} = \boldsymbol{x}_{\text{Homer}} \boldsymbol{x}_{\text{Homer}}^T \ , \qquad \boldsymbol{x}_{\text{Homer}} \in \{-1, 1\}^d \ , \tag{5}$$

where for this example $d = 64 \times 64$.

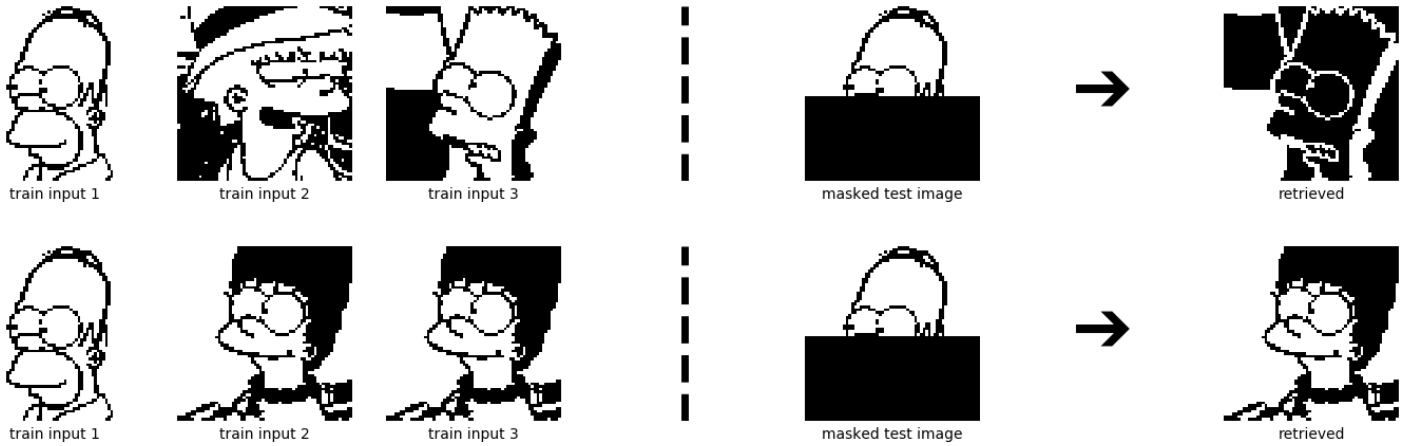
Can the original image be restored if half of the pixels are masked out? The masked image is:

which is our inital state $\boldsymbol{\xi}$. This inital state is updated via multiplication with the weight matrix $\boldsymbol{W}$. It takes one update until the original image is restored.



What happens if we store **more than one pattern**? The weight matrix is then built from the sum of outer products of **three stored patterns** (three input images):

$$\boldsymbol{W} = \sum_{i=1}^{3} \boldsymbol{x}_i \boldsymbol{x}_i^T \ , \qquad \boldsymbol{x}_i \in \{-1, 1\}^d \ . \tag{6}$$

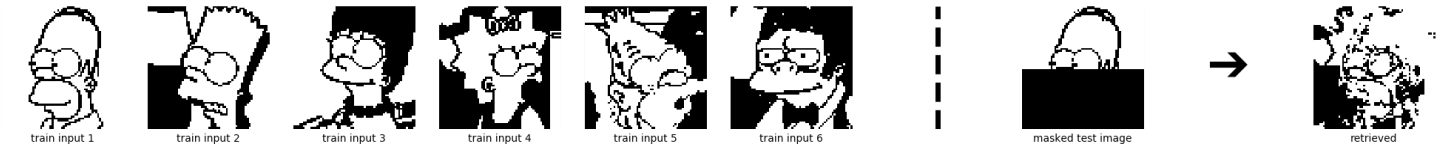


In this figure, the left hand side shows the three stored patterns, and the right hand side shows masked state patterns $\boldsymbol{\xi}$ together with the retrieved patterns $\boldsymbol{\xi}^{\text{new}}$.

Looking at the upper row of images might suggest that the retrieval process is no longer perfect. But there are two interesting facts to take into account:

- Masking the original images introduces many pixel values of -1. We therefore have the odd behavior that the inner product $\left\langle \boldsymbol{x}_{\text{Homer}}^{\text{masked}}, \boldsymbol{x}_{\text{Bart}} \right\rangle$ is larger than the inner product $\left\langle \boldsymbol{x}_{\text{Homer}}^{\text{masked}}, \boldsymbol{x}_{\text{Homer}} \right\rangle$.
- As stated above, if no bias vector is used, the inverse of the pattern, i.e. flipping all pixels at once, results in the same energy.

Although the retrieval of the upper image looks incorrect, it is de facto correct. However, for the lower row example, the retrieval is no longer correct. The weights of $2 \cdot \boldsymbol{x}_{\text{Marge}}$ have simply overwritten the weights of $\boldsymbol{x}_{\text{Homer}}$. For both examples, only the retrieval after the first update step is shown, but the results do not change when performing further update steps. The next figure shows the Hopfield Network retrieval for 6 patterns.



Clearly, retrieving the patterns is imperfect. One might suspect that the limited storage capacities of Hopfield Networks, see Amit et al. and Torres et al, is the problem. However, we show now that the storage capacity is not directly responsible for the imperfect retrieval. The storage capacity for **retrieval of patterns free of errors** is:

$$C \cong \frac{d}{2 \log(d)} \ , \tag{7}$$

where d is the dimension of the input.

The storage capacity for **retrieval of patterns with a small percentage of errors** is:

$$C \cong 0.14 d \ . \tag{8}$$

In the example, the storage capacity is $C \cong 0.14 d = 0.14 \cdot 64 \cdot 64 \sim 570$. Thus, insufficient storage capacity is not directly responsible for the retrieval errors. Instead, the example patterns are correlated, therefore the retrieval has errors.

Consequently, we need a model which **allows pulling apart close patterns**, such that (strongly) **correlated patterns can be distinguished**.

On storage capacity

The storage capacities stated in Eq. (7) and in Eq. (8) are derived for $w_{ii} = 0$. Recently, Folli et al. analyzed the storage capacity for Hopfield Networks with $w_{ii} \geq 0$. Also for $w_{ii} \geq 0$, a storage capacity of $C \cong 0.14 d$ for retrieval of patterns with a small percentage of errors was observed. The ratio C/d is often called **load parameter** and denoted by α. Folli et al. showed that there is a second regime with very large α, where the storage capacity is much higher, i.e. more fixed points exist. However, Rocci et al. and Gosti et al. reported that these fixed points for very large α are unstable and do not have an attraction basin.

Modern Hopfield Networks (aka Dense Associative Memories)

The storage capacity is a crucial characteristic of Hopfield Networks. **Modern Hopfield Networks** (aka Dense Associative Memories) introduce a new energy function instead of the energy in Eq. (3) to create a higher storage capacity. **Discrete** modern Hopfield Networks have been introduced first by Krotov and Hopfield and then generalized by Demircigil et al:

- Krotov and Hopfield introduced the energy function:

$$\mathrm{E} = -\sum_{i=1}^{N} F(\boldsymbol{x}_i^T \boldsymbol{\xi}) \ , \tag{9}$$

where F is an interaction function and N is again the number of stored patterns.

They choose a polynomial interaction function $F(z) = z^a$.

The storage capacity for **retrieval of patterns free of errors** is:

$$C \cong \frac{1}{2(2a-3)!!} \frac{d^{a-1}}{\log(d)} \ . \tag{10}$$

The storage capacity for **retrieval of patterns with a small percentage of errors** is:

$$C \cong \alpha_a d^{a-1} \ , \tag{11}$$

where α_a is a constant, which depends on an (arbitrary) threshold on the error probability.

For $a = 2$, the classical Hopfield model (Hopfield 1982) is obtained with the storage capacity of $C \cong 0.14d$ for retrieval of patterns with a small percentage of errors.

- Demircigil et al. extended the energy function by using an exponential interaction function $F(z) = \exp(z)$:

$$\mathrm{E} = -\sum_{i=1}^{N} \exp(\boldsymbol{x}_i^T \boldsymbol{\xi}) \ , \tag{12}$$

where N is again the number of stored patterns.

Eq. (12) can also be written as:

$$\mathrm{E} = -\exp\big(\mathrm{lse}(1, \boldsymbol{X}^T \boldsymbol{\xi})\big) \ , \tag{13}$$

where $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N)$ is the data matrix (matrix of stored patterns).

The *log-sum-exp function* (lse) is defined as:

$$\mathrm{lse}(\beta, \boldsymbol{z}) = \beta^{-1} \log\left(\sum_{l=1}^{N} \exp(\beta z_l)\right) . \tag{14}$$

This energy function leads to the storage capacity:

$$C \cong 2^{\frac{d}{2}} \ . \tag{15}$$

We now look at the **update rule**, which is valid for both Eq. (9) as well as Eq. (12). For polar patterns, i.e. $\boldsymbol{\xi} \in \{-1, 1\}^d$, we denote the l-th component by $\boldsymbol{\xi}[l]$. Using the energy function of Eq. (9) or Eq. (12), the update rule for the l-th component $\boldsymbol{\xi}[l]$ is described by the difference of the

energy of the current state $\boldsymbol{\xi}$ and the state with the component $\boldsymbol{\xi}[l]$ flipped. The component $\boldsymbol{\xi}[l]$ is updated to decrease the energy. The update rule is:

$$\boldsymbol{\xi}^{\text{new}}[l] = \text{sgn}\Big[- \text{E}\big(\boldsymbol{\xi}^{(l+)}\big) + \text{E}\big(\boldsymbol{\xi}^{(l-)}\big)\Big] \ , \tag{16}$$

which is (e.g. for Eq. (12))

$$\boldsymbol{\xi}^{\text{new}}[l] = \text{sgn}\Big[\sum_{i=1}^{N} \exp\big(\boldsymbol{x}_i^T \ \boldsymbol{\xi}^{(l+)}\big) - \sum_{i=1}^{N} \exp\big(\boldsymbol{x}_i^T \ \boldsymbol{\xi}^{(l-)}\big)\Big] \ , \tag{17}$$

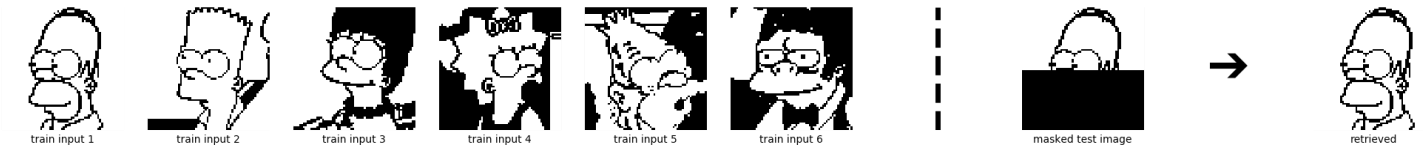
where $\boldsymbol{\xi}^{(l+)}[l] = 1$ and $\boldsymbol{\xi}^{(l-)}[l] = -1$ and $\boldsymbol{\xi}^{(l+)}[k] = \boldsymbol{\xi}^{(l-)}[k] = \boldsymbol{\xi}[k]$ for $k \neq l$.

In the paper of Demircigil et al., it is shown that the **update rule**, which minimizes the energy function of Eq. (12), converges with high probability after one (asynchronous) update of the current state $\boldsymbol{\xi}$. Note that one update of the current state $\boldsymbol{\xi}$ corresponds to d asynchronous update steps, i.e. one update for each of the d single components $\boldsymbol{\xi}[l]$ ($l = 1, \ldots, d$).

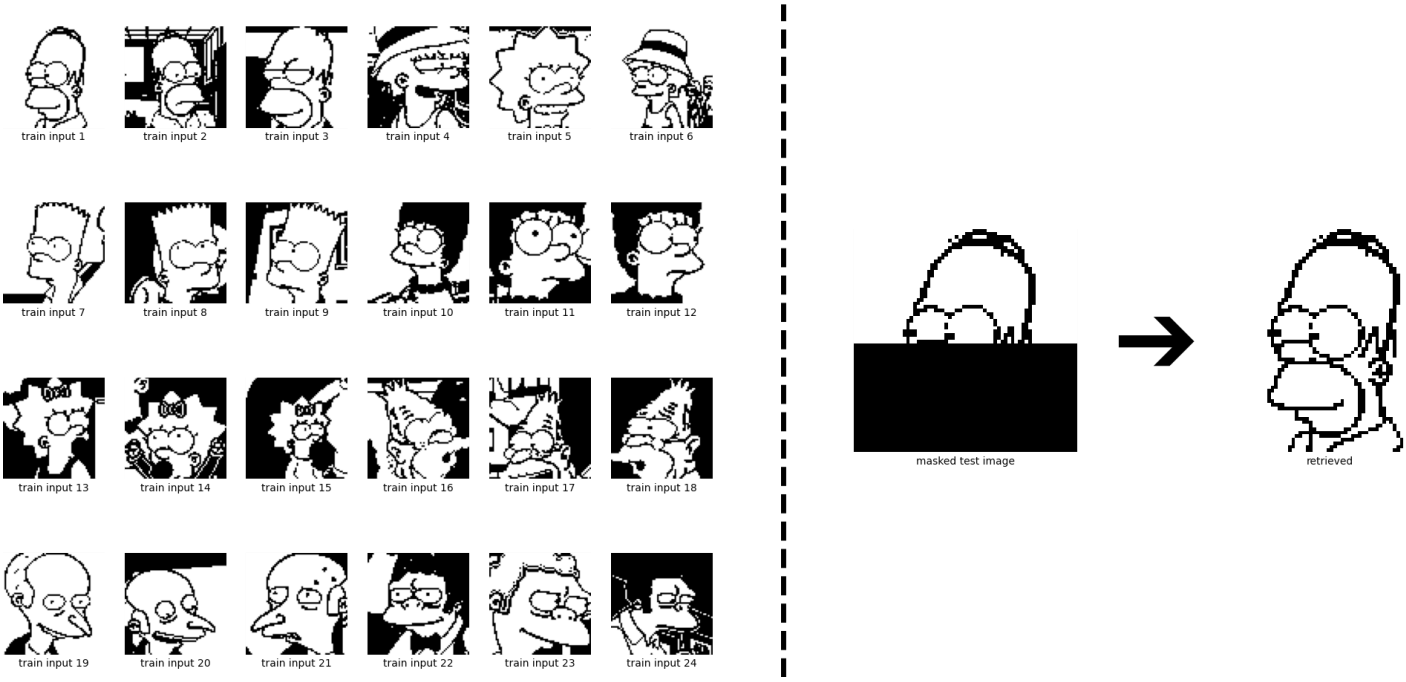
In contrast to classical Hopfield Networks, modern Hopfield Networks do not have a weight matrix as it is defined in Eq. (5). Instead, the energy function is the **sum of a function of the dot product** of every stored pattern $\boldsymbol{x}_i$ with the state pattern $\boldsymbol{\xi}$.

Using Eq. (17), we again try to retrieve Homer out of the 6 stored patterns. Compared to the classical Hopfield Network, it now works smoothly, not only for 6 patterns but also for many more:

- First we store the same 6 patterns as above:



- Next we increase the number of stored patterns to 24:



Compared to the traditional Hopfield Networks, the **increased storage capacity now allows pulling apart close patterns**. We are now able to distinguish (strongly) correlated patterns, and can retrieve one specific pattern out of many.

New energy function for continuous-valued patterns and states

We generalize the energy function of Eq. (13) to continuous-valued patterns. We use the logarithm of the negative energy Eq. (13) and add a quadratic term. The quadratic term ensures that the norm of the state $\boldsymbol{\xi}$ remains finite. The **new energy function** is defined as:

$$\mathrm{E} = -\mathrm{lse}\left(\beta, \boldsymbol{X}^T \boldsymbol{\xi}\right) + \frac{1}{2} \boldsymbol{\xi}^T \boldsymbol{\xi} + \beta^{-1} \log N + \frac{1}{2} M^2 \ , \tag{18}$$

which is constructed from N **continuous** stored patterns by the matrix $\boldsymbol{X} = \left(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\right)$, where M is the largest norm of all stored patterns.

According to the new paper of Krotov and Hopfield, the stored patterns $\boldsymbol{X}^T$ of our modern Hopfield Network can be viewed as weights from $\boldsymbol{\xi}$ to hidden units, while $\boldsymbol{X}$ can be viewed as weights from the hidden units to $\boldsymbol{\xi}$. With this interpretation we do not store patterns, but use only weights in our model as in the classical Hopfield Network.

The energy function of Eq. (18) allows deriving an update rule for a state pattern $\boldsymbol{\xi}$ by the **Concave-Convex-Procedure** (CCCP), which is described by Yuille and Rangarajan.

- the total energy $\mathrm{E}(\boldsymbol{\xi})$ is split into a convex and a concave term: $\mathrm{E}(\boldsymbol{\xi}) = \mathrm{E}_1(\boldsymbol{\xi}) + \mathrm{E}_2(\boldsymbol{\xi})$
- the term $\frac{1}{2} \boldsymbol{\xi}^T \boldsymbol{\xi} + C = \mathrm{E}_1(\boldsymbol{\xi})$ is convex (C is a constant independent of $\boldsymbol{\xi}$)
- the term $-\mathrm{lse}\left(\beta, \boldsymbol{X}^T \boldsymbol{\xi}\right) = \mathrm{E}_2(\boldsymbol{\xi})$ is concave (lse is convex since its Hessian is positive semi-definite, which is shown in the appendix of the paper)
- The CCCP applied to E is:

$$\nabla_{\boldsymbol{\xi}} \mathrm{E}_1(\boldsymbol{\xi}^{t+1}) = -\nabla_{\boldsymbol{\xi}} \mathrm{E}_2(\boldsymbol{\xi}^t) \tag{19}$$

$$\nabla_{\boldsymbol{\xi}} \left(\frac{1}{2} \boldsymbol{\xi}^T \boldsymbol{\xi} + C \right) (\boldsymbol{\xi}^{t+1}) = \nabla_{\boldsymbol{\xi}} \mathrm{lse}\left(\beta, \boldsymbol{X}^T \boldsymbol{\xi}^t\right) \tag{20}$$

$$\boldsymbol{\xi}^{t+1} = \boldsymbol{X} \mathrm{softmax}\left(\beta \boldsymbol{X}^T \boldsymbol{\xi}^t\right) \ , \tag{21}$$

where $\nabla_{\boldsymbol{\xi}} \mathrm{lse}\left(\beta, \boldsymbol{X}^T \boldsymbol{\xi}\right) = \boldsymbol{X} \mathrm{softmax}\left(\beta \boldsymbol{X}^T \boldsymbol{\xi}\right)$.

The update rule for a state pattern $\boldsymbol{\xi}$ therefore reads:

$$\boldsymbol{\xi}^{\mathrm{new}} = \boldsymbol{X} \mathrm{softmax}\left(\beta \boldsymbol{X}^T \boldsymbol{\xi}\right) \ . \tag{22}$$

Having applied the Concave-Convex-Procedure to obtain the update rule guarantees the monotonical decrease of the energy function.

The most important **properties of our new energy function** are:

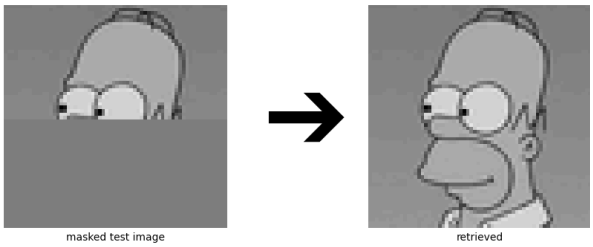
1. Global convergence to a local minimum (Theorem 2 in the paper)
2. Exponential storage capacity (Theorem 3 in the paper)

3. Convergence after one update step (Theorem 4 in the paper)

Exponential storage capacity and convergence after one update are inherited from Demircigil et al. Global convergence to a local minimum means that all limit points that are generated by the iteration of Eq. (22) are stationary points (local minima or saddle points) of the energy function of Eq. (18) (almost surely no maxima are found, saddle points were never encountered in any experiment).

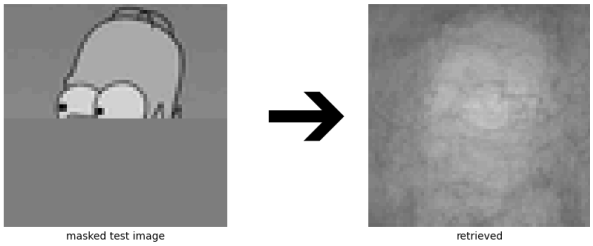
The new continuous energy function allows **extending our example to continuous patterns**. In the following, we are going to retrieve a continuous Homer out of many continuous stored patterns using Eq. (22). First we have to convert the input images into grey scale images:

Next, we conduct the same experiment as above, but now in continuous form:



We again see that Homer is perfectly retrieved. We have considered the case where the patterns are sufficiently different from each other, and consequently the iterate converges to a fixed point which is near to one of the stored patterns. However, if some stored patterns are similar to each other, then a **metastable state** near the similar patterns appears. Iterates that start near this metastable state or at one of the similar patterns converge to this metastable state. The **learning dynamics can**

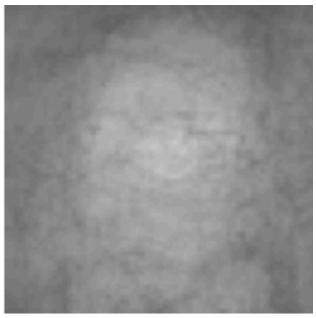
be controlled by the inverse temperature β, see Eq. (22). High values of β correspond to a low temperature and mean that the attraction basins of the individual patterns remain separated and it is unlikely that metastable states appear. Low values of β on the other hand correspond to a high temperature and the formation of metastable states becomes more likely. We now look at the same example, but instead of $\beta = 8$, we use $\beta = 0.5$. The retrieved state is now a superposition of multiple stored patterns.



To make this more explicit, we have a closer look how the results are changing if we retrieve with different values of β:

masked

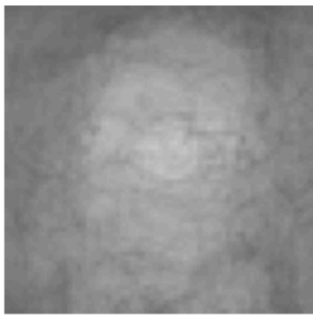
beta = 0.25



retrieved

masked

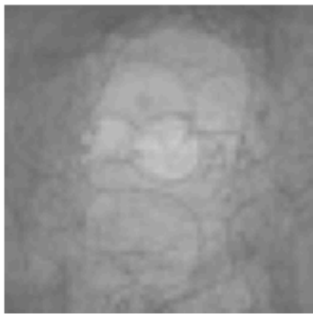
beta = 0.50



retrieved

masked

beta = 1.00



retrieved

masked

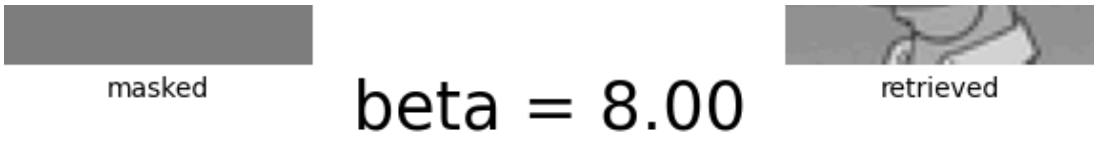
beta = 2.00

retrieved

masked

beta = 4.00

retrieved



The update of the new energy function is the self-attention of transformer networks

Starting with Eq. (21), and

1. generalizing the new update rule to multiple patterns at once,
2. mapping the patterns to an associative space,
3. projecting the result,

we arrive at the **(self-)attention of transformer networks**. Next, we will guide through these three steps.

For S state patterns $\boldsymbol{\Xi} = (\boldsymbol{\xi}_1, \ldots, \boldsymbol{\xi}_S)$, Eq. (21) can be generalized to:

$$\boldsymbol{\Xi}^{\text{new}} = \boldsymbol{X} \text{softmax}\big(\beta \boldsymbol{X}^T \boldsymbol{\Xi}\big) \ . \tag{23}$$

We first consider $\boldsymbol{X}^T$ as N *raw stored patterns* $\boldsymbol{Y} = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N)^T$, which are mapped to an associative space via $\boldsymbol{W}_K$, and $\boldsymbol{\Xi}^T$ as S *raw state patterns* $\boldsymbol{R} = (\boldsymbol{\xi}_1, \ldots, \boldsymbol{\xi}_S)^T$, which are mapped to an associative space via $\boldsymbol{W}_Q$.

Setting

$$\boldsymbol{Q} = \boldsymbol{\Xi}^T = \boldsymbol{R} \boldsymbol{W}_Q \ , \tag{24}$$

$$\boldsymbol{K} = \boldsymbol{X}^T = \boldsymbol{Y} \boldsymbol{W}_K \ , \tag{25}$$

$$\beta = \frac{1}{\sqrt{d_k}} \ , \tag{26}$$

we obtain:

$$\left(\boldsymbol{Q}^{\text{new}}\right)^T = \boldsymbol{K}^T \text{softmax}\left(\frac{1}{\sqrt{d_k}} \boldsymbol{K} \boldsymbol{Q}^T\right) \ . \tag{27}$$

In Eq. (24) and Eq. (25), $\boldsymbol{W}_Q$ and $\boldsymbol{W}_K$ are matrices which map the respective patterns into the associative space. Note that in Eq. (27), the softmax is applied column-wise to the matrix $\boldsymbol{K} \boldsymbol{Q}^T$.

Next, we simple transpose Eq. (27), which also means that the softmax is now applied row-wise to its transposed input $\boldsymbol{Q} \boldsymbol{K}^T$, and obtain:

$$\boldsymbol{Q}^{\text{new}} = \text{softmax}\left(\frac{1}{\sqrt{d_k}} \boldsymbol{Q} \boldsymbol{K}^T\right) \boldsymbol{K} \ . \tag{28}$$

Now, we only need to project $\boldsymbol{Q}^{\text{new}}$ via another projection matrix $\boldsymbol{W}_V$:

$$\boldsymbol{Z} = \boldsymbol{Q}^{\text{new}}\boldsymbol{W}_V = \text{softmax}\left(\frac{1}{\sqrt{d_k}}\boldsymbol{Q}\boldsymbol{K}^T\right)\boldsymbol{K}\boldsymbol{W}_V = \text{softmax}\left(\frac{1}{\sqrt{d_k}}\boldsymbol{Q}\boldsymbol{K}^T\right)\boldsymbol{V}\ , \quad (29)$$

and **voilà, we have obtained the transformer attention**. If the N raw stored patterns $\boldsymbol{Y} = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N)^T$ are used as raw state patterns $\boldsymbol{R}$, we obtain the **transformer self-attention**.

If we resubstitute our raw stored patterns $\boldsymbol{Y}$ and our raw state patterns $\boldsymbol{R}$, we can rewrite Eq. (29) as

$$\boldsymbol{Z} = \text{softmax}\big(\beta \cdot \boldsymbol{R}\boldsymbol{W}_Q\boldsymbol{W}_K^T\boldsymbol{Y}^T\big)\boldsymbol{Y}\boldsymbol{W}_K\boldsymbol{W}_V\ , \quad (30)$$

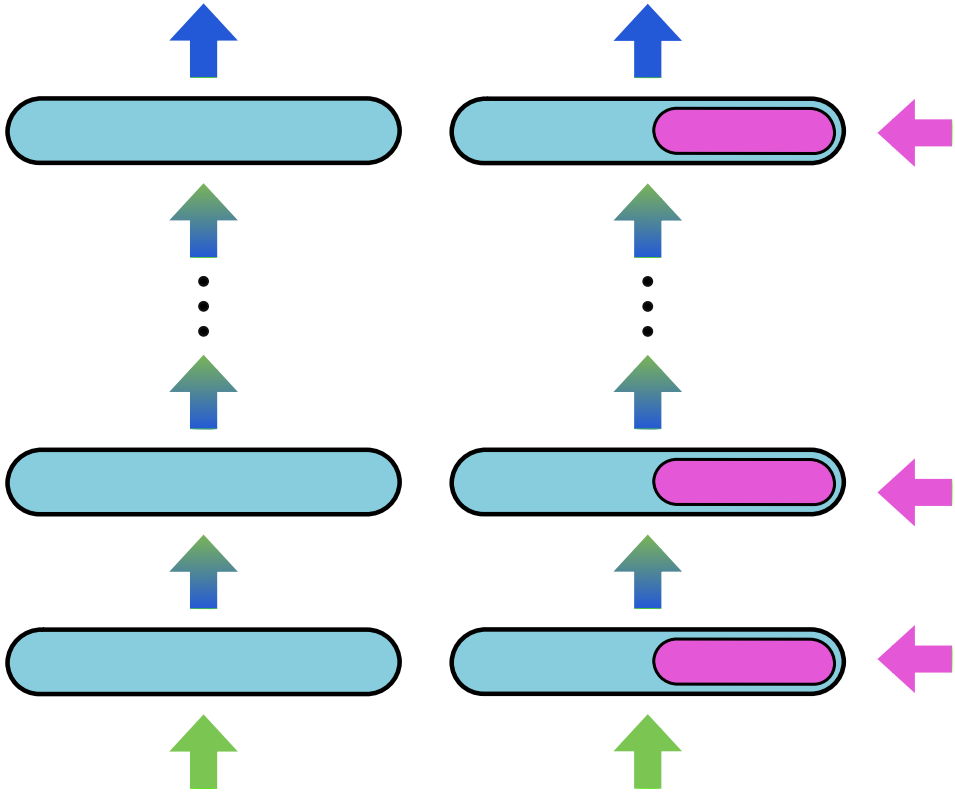
which is the fundament of our new PyTorch Hopfield layer.

Hopfield layers for Deep Learning architectures

The insights stemming from our work on modern Hopfield Networks allowed us to introduce new PyTorch Hopfield layers, which can be used as plug-in replacement for existing layers as well as for applications like multiple instance learning, set-based and permutation invariant learning, associative learning, and many more. We introduce three types of Hopfield layers:

- `Hopfield` for associating and processing two sets. Examples are the transformer attention, which associates keys and queries, and two point sets that have to be compared.
- `HopfieldPooling` for fixed pattern search, pooling operations, and memories like LSTMs or GRUs. The state (query) pattern is static and can be learned.
- `HopfieldLayer` for storing fixed patterns or learning internal prototypes. The stored (key) patterns are static and can be learned.

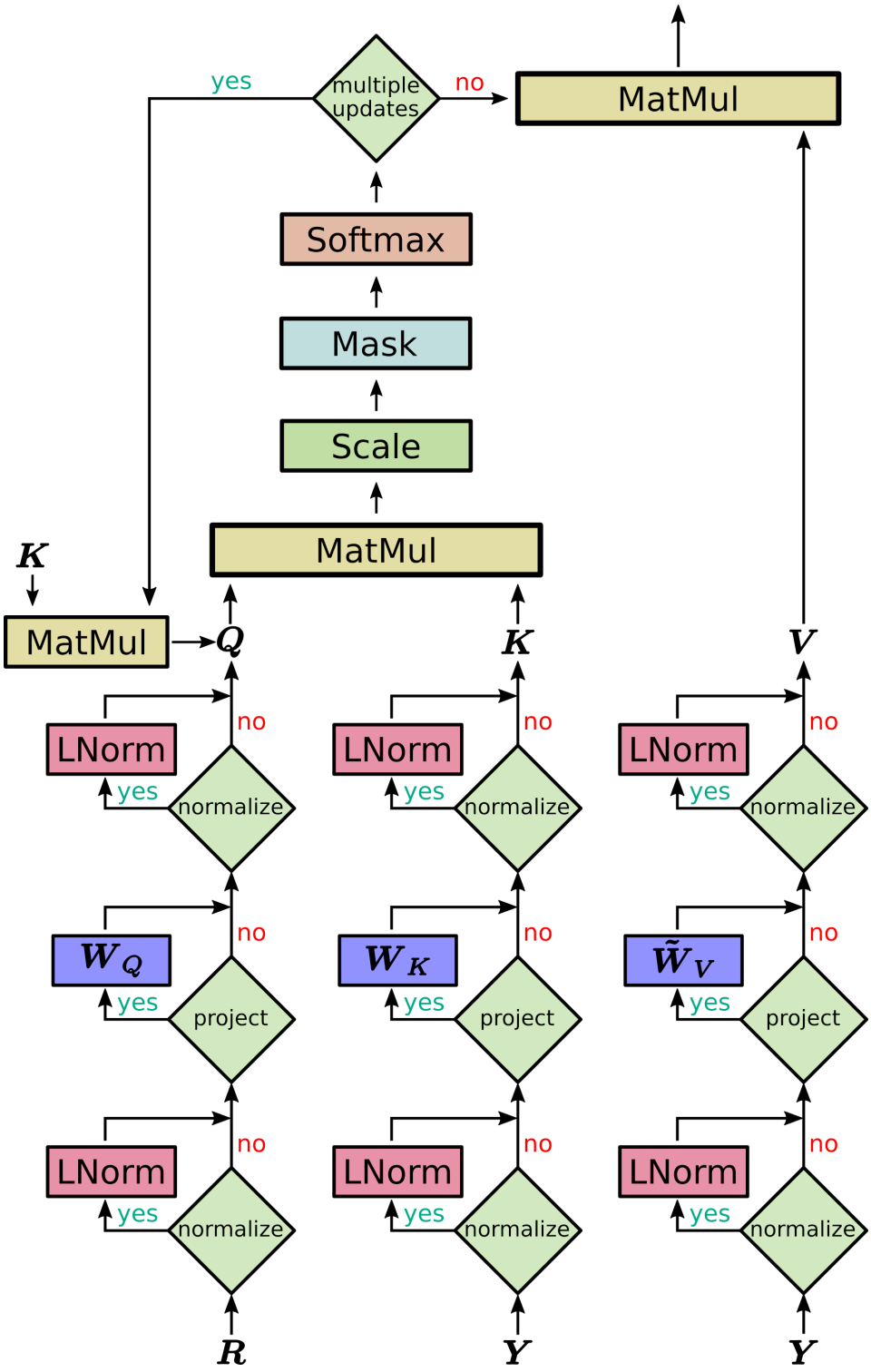
Due to their continuous nature Hopfield layers are differentiable and can be integrated into deep learning architectures to equip their layers with associative memories. On the left side of the Figure below a standard deep network is depicted. It propagates either a vector or a set of vectors from input to output. On the right side a deep network is depicted, where layers are equipped with associative memories via Hopfield layers. A detailed description of the layers is given below.



Additional functionalities of the new PyTorch Hopfield layers compared to the transformer (self-)attention layer are:

- **Association of two sets**
- **Variable** β that determines the kind of fixed points
- **Multiple updates** for precise fixed points
- **Dimension of the associative space** for controlling the storage capacity
- **Static patterns** for fixed pattern search
- **Pattern normalization** to control the fixed point dynamics by norm and shift of the patterns

A sketch of the new Hopfield layers is provided below.



Next, we introduce the underlying mechanisms of the implementation. Based on these underlying mechanisms, we give three examples on how to use the new Hopfield layers and how to utilize the principles of modern Hopfield Networks.

Layer *Hopfield*

In its most general form, the result patterns $\boldsymbol{Z}$ are a function of raw stored patterns $\boldsymbol{Y}$, raw state patterns $\boldsymbol{R}$, and projection matrices $\boldsymbol{W}_Q$, $\boldsymbol{W}_K$, $\boldsymbol{W}_V$:

$$\boldsymbol{Z} = f(\boldsymbol{Y}, \boldsymbol{R}, \boldsymbol{W}_Q, \boldsymbol{W}_K, \boldsymbol{W}_V) = \mathrm{softmax}\left(\beta \cdot \boldsymbol{R}\boldsymbol{W}_Q\boldsymbol{W}_K^T\boldsymbol{Y}^T\right)\boldsymbol{Y}\boldsymbol{W}_K\boldsymbol{W}_V \tag{31}$$

where we denote

$$\tilde{\boldsymbol{W}}_V = \boldsymbol{W}_K \boldsymbol{W}_V \ . \tag{32}$$

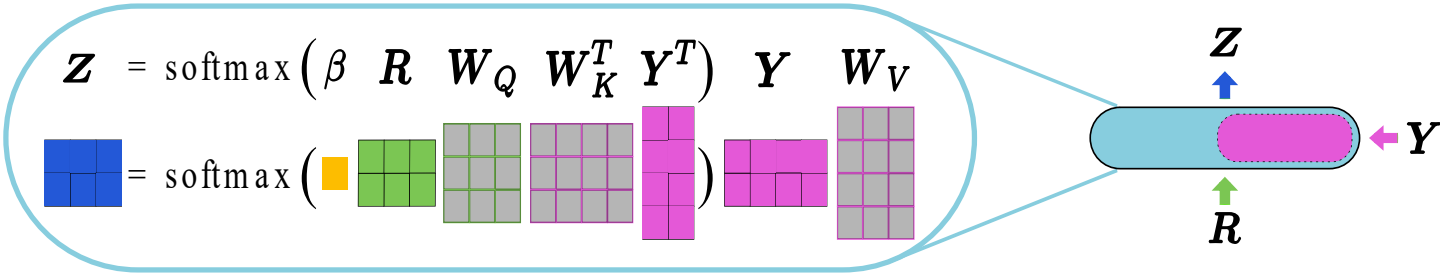
Here, the rank of $\tilde{\boldsymbol{W}}_V$ is limited by dimension constraints of the matrix product $\boldsymbol{W}_K \boldsymbol{W}_V$. To provide the Hopfield layer with more flexibility, the matrix product $\boldsymbol{W}_K \boldsymbol{W}_V$ can be replaced by one parameter matrix (flag in the code). In this case $\tilde{\boldsymbol{W}}_V$ is not the product from Eq. (32) but a stand-alone parameter matrix as in the original transformer setting.

In Eq. (31), the N raw stored patterns $\boldsymbol{Y} = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N)^T$ and the S raw state patterns $\boldsymbol{R} = (\boldsymbol{r}_1, \ldots, \boldsymbol{r}_S)^T$ are mapped to an associative space via the matrices $\boldsymbol{W}_K$ and $\boldsymbol{W}_Q$. We also allow **static state** and **static stored patterns**. A static pattern means that it does not depend on the network input, i.e. it is determined by the bias weights and remains constant across different network inputs.

Remark

For simplicity from now on we replace $\boldsymbol{W}_K \boldsymbol{W}_V$ by just $\boldsymbol{W}_V$.

An illustration of the matrices of Eq. (31) is shown below:



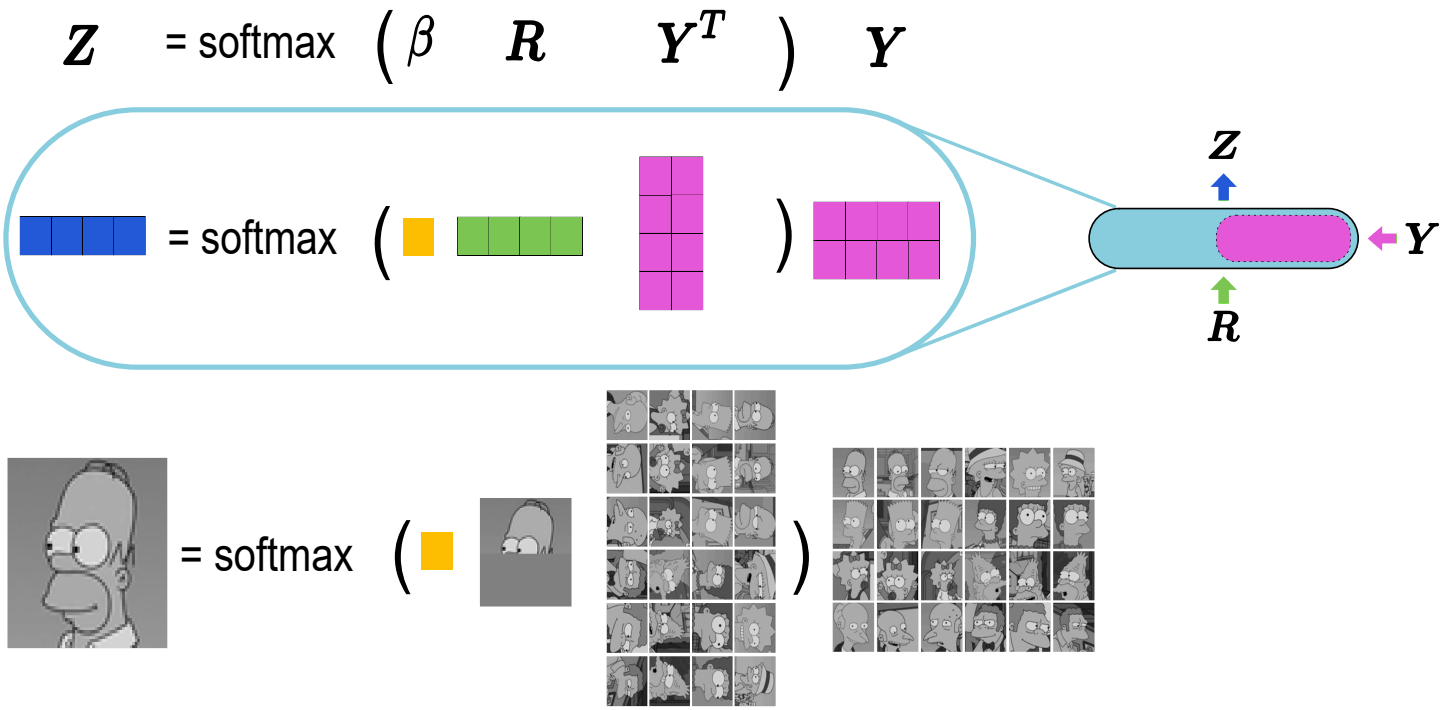
Note that in this simplified sketch $\boldsymbol{W}_V$ already contains the output projection.

It now depends on the underlying tasks which matrices are used. For example, the code for the above sketch would be the following:

```
hopfield = Hopfield(
    input_size=3,                       # R
    hidden_size=3,
    stored_pattern_size=4,              # Y
    pattern_projection_size=4,          # Y
    scaling=beta)

# tuple of stored_pattern, state_pattern, pattern_projection
hopfield((Y, R, Y))
```

Layer *HopfieldLayer*

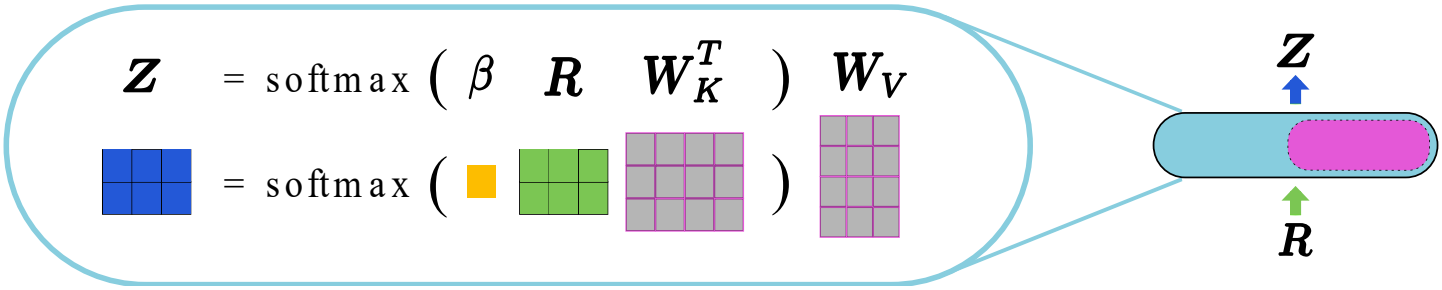
Of course we can also use the new Hopfield layer to solve the pattern retrieval task from above. For this task no trainable weights are needed.




```
hopfield = Hopfield(
    scaling=beta,

    # do not project layer input
    state_pattern_as_static=True,
    stored_pattern_as_static=True,
    pattern_projection_as_static=True,

    # do not pre-process layer input
    normalize_stored_pattern=False,
    normalize_stored_pattern_affine=False,
    normalize_state_pattern=False,
    normalize_state_pattern_affine=False,
    normalize_pattern_projection=False,
    normalize_pattern_projection_affine=False,

    # do not post-process layer output
    disable_out_projection=True)

# tuple of stored_pattern, state_pattern, pattern_projection
hopfield((Y, R, Y))
```

Hopfield Lookup via *HopfieldLayer*

A variant of our Hopfield-based modules is one which employs a **trainable but input independent lookup mechanism.** Internally, one or multiple **stored patterns and pattern projections are trained** (optionally in a non-shared manner), which in turn are used as a lookup mechanism independent of the input data.



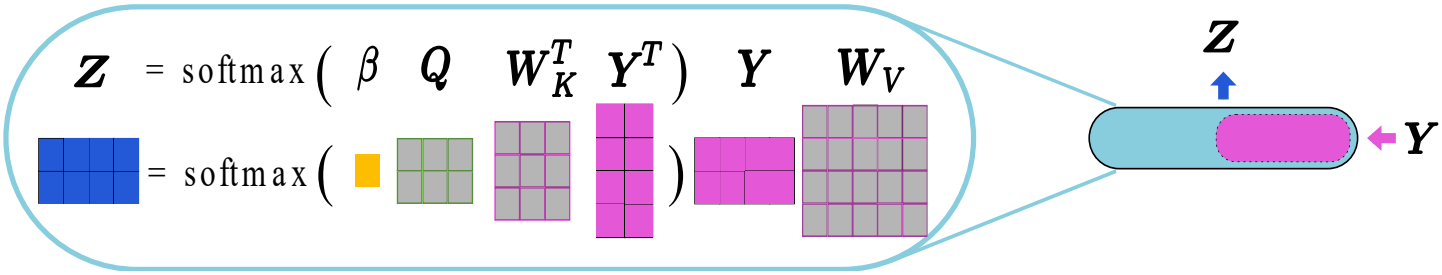

```
hopfield_lookup = HopfieldLayer(
    input_size=3,                           # R
    hidden_size=3,                          # W_K
    pattern_size=4,                         # W_V
    quantity=4,                             # W_K
    scaling=beta,
    stored_pattern_as_static=True,
    state_pattern_as_static=True)

# state pattern
hopfield_lookup(R)
```

This specialized variant of the Hopfield layer allows for a setting, where the training data is used as stored patterns, the new data as state pattern, and the training label to project the output of the Hopfield layer.

```
hopfield_lookup = HopfieldLayer(
    input_size=3,                           # R
    hidden_size=3,                          # W_K
    quantity=4,                             # W_K
    scaling=beta,
    lookup_weights_as_separated=True,
    lookup_targets_as_trainable=False,
    stored_pattern_as_static=True,
    state_pattern_as_static=True,
    pattern_projection_as_static=True)

# state pattern
hopfield_lookup(R)
```

Layer *HopfieldPooling*

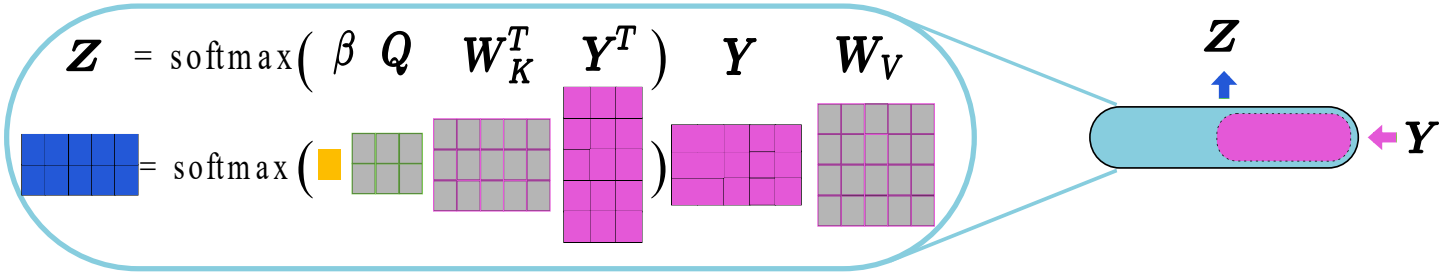
We consider the Hopfield layer as a pooling layer if only one static state pattern (query) exists. Then, it is de facto a **pooling over the sequence**. The static state pattern is considered as a **prototype pattern** and consequently learned in the Hopfield pooling layer. Below we give two examples of a Hopfield pooling over the stored patterns Y. Note that the pooling always operates over the token dimension (i.e. the sequence length), and not the token embedding dimension.




```
hopfield_pooling = HopfieldPooling(
    input_size=4,                          # Y
    hidden_size=3,                         # Q
    scaling=beta,
    quantity=2)                            # number of state patterns

# stored_pattern and pattern_projection
hopfield_pooling(Y)
```

The pooling over the sequence is de facto done over the token dimension of the stored patterns, i.e. $\boldsymbol{Y} \in \mathbb{R}^{(2\times4)} \Rightarrow \boldsymbol{Z} \in \mathbb{R}^{(2\times4)}$.

We show another example below, where the Hopfield pooling boils down to $\boldsymbol{Y} \in \mathbb{R}^{(3\times5)} \Rightarrow \boldsymbol{Z} \in \mathbb{R}^{(2\times5)}$:




```
hopfield_pooling = HopfieldPooling(
    input_siz=5,                           # Y
    hidden_size=3,                         # Q
    scaling=beta,
    quantity=2)                            # number of state patterns

# stored_pattern and pattern_projection
hopfield_pooling(Y)
```

DeepRC

One SOTA application of modern Hopfield Networks can be found in the paper Modern Hopfield Networks and Attention for Immune Repertoire Classification by Widrich et al. Here, the high storage capacity of modern Hopfield Networks is exploited to solve a challenging multiple instance learning (MIL) problem in computational biology called **immune repertoire classification**.

Biological background

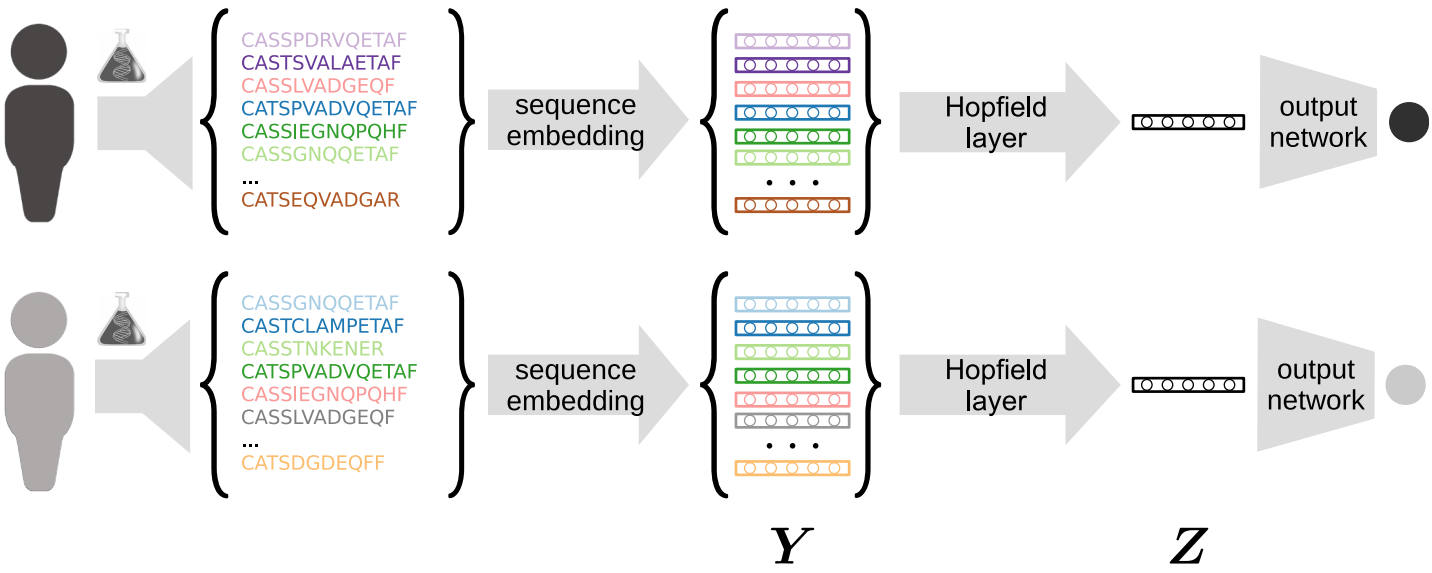
The immune repertoire of an individual consists of an immensely large number of immune repertoire receptors (and many other things). The task of these receptors, which can be represented as amino acid sequences with variable length and 20 unique letters, is to selectively bind to surface-structures of specific pathogens in order to combat them. Only a variable sub-sequence of the receptors might be responsible for this binding. Due to the large variety of pathogens, each human has about 10^7–10^8 unique immune receptors with low overlap across individuals and sampled from a potential diversity of $> 10^{14}$ receptors. However, only very few of these receptors bind to a single specific pathogen. This means that the immune repertoire of an individual that shows an immune response against a specific pathogen, e.g. a specific disease, should contain a few sequences that can bind to this specific pathogen. Turning this around, in order to classify such immune repertoires into those with and without immune response, one would have to find this variable sub-sequence that binds to the specific pathogen.

Consequently, the **classification of immune repertoires is extremely difficult**, since each immune repertoire contains a large amount of sequences as instances with only a very few of them indicating the correct class by carrying a certain variable sub-sequence. This is a prominent example of a **needle-in-a-haystack** problem and a strong challenge for machine learning methods.

Based on modern Hopfield networks, a method called **DeepRC** was designed, which consists of three parts:

- a sequence-embedding neural network to supply a fixed-sized sequence-representation (e.g. 1D-CNN or LSTM),
- a **Hopfield layer part** for sequence-attention, and
- an output neural network and/or fully connected output layer.

The following figure illustrates these 3 parts of DeepRC:

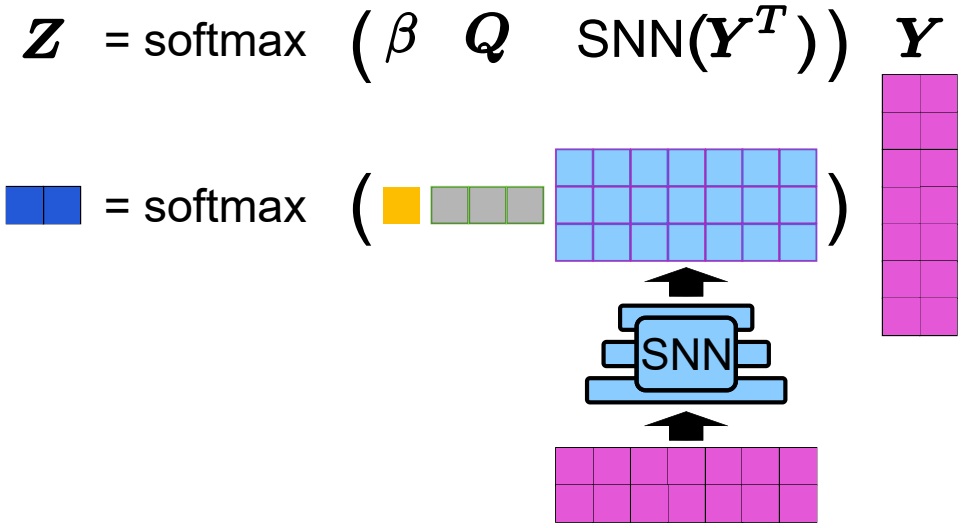


So far we have discussed two use cases of the Hopfield layer: (i) the default setting where the **input consists of stored patterns and state patterns** and (ii) the Hopfield pooling, where **a prototype pattern** is learned, which means that the vector Q is learned. For immune repertoire classification we have another use case. Now the **inputs** for the Hopfield layer are partly **obtained via neural networks**.

To be more precise, the three ingredients of the attention mechanism of **DeepRC** are:

- the output of the sequence-embedding neural network $\boldsymbol{Y}^T$ directly acts as values $\boldsymbol{V}$,
- a second neural network, e.g. a self-normalizing neural network (SNN), shares its first layers with the sequence-embedding neural network and outputs the keys $\boldsymbol{K}$, i.e. the stored patterns, and
- similar to the Hopfield pooling operation, the query vector $\boldsymbol{Q}$ is learned and represents the variable binding sub-sequence we are looking for.

The following sketch visualizes the **Hopfield layer part** of DeepRC:



It is to note that for immune repertoire classification the **number of instances is much larger than the number of features** (**~300k instances per repertoire**). To be more precise, the difference is a factor of 10^4 to 10^5. This is indicated in the sketch, where $\mathbf{Y}^T$ has more columns than rows. The complex SNN-based attention mechanism reduces this large number of instances, while keeping the complexity of the input to the output neural network low.

The pseudo-code for the Hopfield layer used in DeepRC is:

```
Y = EmbeddingNN(I)  # e.g. 1D-CNN
K = SNN(Y)
Q = StatePattern(size=3)

hopfield = Hopfield(
    scaling=beta,

    # do not project layer input
    state_pattern_as_static=True,
    stored_pattern_as_static=True,
    pattern_projection_as_static=True,

    # do not pre-process layer input
    normalize_stored_pattern=False,
    normalize_stored_pattern_affine=False,
    normalize_state_pattern=False,
    normalize_state_pattern_affine=False,
```

```
    normalize_pattern_projection=False,
    normalize_pattern_projection_affine=False,

    # do not post-process layer output
    disable_out_projection=True)

# tuple of stored_pattern, state_pattern, pattern_projection
hopfield((K, Q, Y))
```

Material

- Paper: Hopfield Networks is All You Need
- Githup repo: hopfield-layers
- Paper: Modern Hopfield Networks and Attention for Immune Repertoire Classification
- Github repo: DeepRC
- Yannic Kilcher's video on our two papers
- Blog post on Performers from a Hopfield point of view

For more information visit our homepage https://ml-jku.github.io/.

Correspondence

This blog post was written by Johannes Brandstetter: brandstetter[at]ml.jku.at

Contributions by Viet Tran, Bernhard Schäfl, Hubert Ramsauer, Johannes Lehner, Michael Widrich, Günter Klambauer and Sepp Hochreiter.
