A. Organization of the Supplementary Materials

The Supplementary Materials are organized as follows. We first derive the dynamics and learning rules for the weights and biases in the energy-based and prototypical settings of EP in Section B and Section C respectively. Then, we give more details about the scaling factor: chosen fixed or dynamical, in Section D. We discuss why the error signal has difficulties to flow in the system when the neurons have a binary activation function in Section E. We finally detail all the software implementations in Section F.

More precisely one can find:

- Experimental evidence showing the importance of the scaling factor (fixed in this section) for training systems with binary weights (Section D).
- The derivation of the learning rule for the scaling factor, together with a description of the results obtained with this training procedure, showing the acceleration that the learnt dynamical scaling factor provides (Section D).
- An empirical demonstration that the prediction is efficiently computed with an enlarged output layer for binary neural networks (Section E).
- Training curves and their flipping metric (defined in Eq. (6)) monitored over epochs (Section F).

B. Training Fully Connected Layers Networks with Equilibrium Propagation

In this section, we describe and define all the operations we used to train with EP a fully connected neural network recurrently connected through bidirectional synapses. We describe the dynamics and the underlying learning rules for the weights and the biases in the energy-based and prototypical settings. The units of the system are denoted $\mathrm{s} = \{h, y\}$ where h are the hidden units and y are the output units. The variable y is the one-hot encoded target vector. The inputs x are always clamped and are static.

B.1. Energy-Based Settings

In the energy based settings, we introduce an energy function for the network, that defines the neuron dynamics during the two phases of EP. We then derive the learning rules from the energy function.

B.1.1 Energy Function

We consider the following energy function [30]:

$$E(s, \rho(s), \theta = \{W, b\}) := \frac{1}{2}\sum_i s_i - \frac{1}{2}\sum_{i \neq j} W_{ij}\rho(s_i)\rho(s_i) - \sum_i b_i \rho(s_i) \tag{17}$$

where ρ is the activation function of the neurons, W_{ij} the weight connecting the unit s_i to s_j and reciprocally as synapses are symmetric for the system to converge and b_i the bias of unit s_i.

We also define ℓ the cost function describing how far are the output units of the system ($\hat{y}$) from their target state (y). We usually employ the mean squared error as a cost function with EP:

$$\ell(s, \mathrm{y}) = MSE(s, \mathrm{y}) := \frac{1}{2}\sum ||\mathrm{y} - \hat{y}||^2 \tag{18}$$

where y denotes a given target output.

B.1.2 Dynamics

The dynamics of neurons in the free phase evolve according to the energy function E:

$$\frac{ds}{dt} = -\frac{\partial E}{\partial s} \tag{19}$$

which translates for the neuron i and the energy function E defined in Eq.17 as:

$$\frac{ds_i}{dt} = -s_i + \rho'(s_i)(\sum_{j \neq i} W_{ij}\rho(s_j) + b) \tag{20}$$

The system eventually settles to a fixed steady state s_*.

During the nudged phase the dynamics differs from the free phase as the neurons now evolve to decrease the cost function ℓ:

$$\frac{ds}{dt} = -\frac{\partial E}{\partial s} - \beta\frac{\partial \ell}{\partial s} \tag{21}$$

which translates for the hidden unit h_i, the output unit $\hat{y}_i$ and the target y to:

$$\begin{cases} \frac{dh_i}{dt} = -h_i + \rho'(h_i)(\sum_{j \neq i} W_{ij}\rho(s_j) + b_i) \\ \frac{d\hat{y}_i}{dt} = -\hat{y}_i + \rho'(y_i)(\sum_{j \neq i} W_{ij}\rho(h_j) + b_i) + \beta \times (\mathrm{y_i} - \hat{y}_i) \end{cases} \tag{22}$$

The systems eventually reaches a second steady state denoted s_*^β.

B.1.3 Learning Rule

Scellier & Bengio [30] showed that the gradient of the loss $\mathcal{L}_*$ (defined in Eq. (...)) with respect to any parameter in the system can be approximated by the derivative of the energy function E with regard to the parameter evaluated at the two equilibrium points s_* and s_*^β:

$$-\frac{\partial \mathcal{L}_*}{\partial \theta} = \lim_{\beta \to 0} \frac{1}{\beta}\left(\frac{\partial E}{\partial \theta}(x, s_*^\beta, \theta) - \frac{\partial E}{\partial \theta}(x, s_*, \theta)\right) \tag{23}$$

In the energy-based settings, the resulting learning rules for the weights and biases are expressed as a function of the two steady states:

$$\begin{cases} \Delta W_{ij} = \frac{1}{\beta}(\rho(s_{i,*}^\beta)\rho(s_{j,*}^\beta) - \rho(s_{i,*})\rho(s_{j,*})) \\ \Delta b_i = \frac{1}{\beta}(\rho(s_{i,*}^\beta) - \rho(s_{i,*})) \end{cases} \tag{24}$$

B.2. Prototypical Settings

Ernoult *et al*. [8] introduced the prototypical settings for EP where the dynamics no longer derived from an energy function in a continuous-time setting but more generally from a scalar primitive in a discrete-time setting. As in Ernoult *et al*. [8], we chose a dynamics close to the one of conventional RNNs. We then write a primitive function from which the dynamics derives. Finally we obtain the learning rules from the primitive function.

B.2.1 Dynamics

We choose the same discrete time dynamics as in [8]:

$$\begin{cases} h_i^{t+1} = \rho(\sum_j W_{ij} s_j^t + b) \\ y_i^{t+1} = \rho(\sum_j W_{ij} h_j^t + b) + \beta \times (\mathrm{y_i} - \hat{y}_i) \text{ where } \beta = 0 \text{ during the free phase} \end{cases} \tag{25}$$

The nudge still derives from the MSE cost function as defined in Eq. 18. The system also sequentially settles to two fixed steady states s_* and s_*^β at the end of the free and the nudged phase respectively.

B.2.2 Primitive Function

We define the primitive function as the function from which the dynamics could derive:

$$s^{t+1} \approx \frac{\partial \Phi}{\partial s} \tag{26}$$

which gives, ignoring the activation function ρ:

$$\Phi = \frac{1}{2} s^T W s \tag{27}$$

B.2.3 Learning Rule

Similarly to the energy-based settings, we now compute the gradient of the primitive function with regard to a parameter of the system in order to perform optimization. The learning rule, expressed as a function of the two equilibrium points s_* and s_*^β, now reads:

$$\Delta\theta = \frac{1}{\beta}\left(\frac{\partial\Phi}{\partial\theta}(x, s_*^\beta, \theta) - \frac{\partial\Phi}{\partial\theta}(x, s_*, \theta)\right) \tag{28}$$

The learning rules for the weights and the biases read:

$$\begin{cases} \Delta W_{ij} = \frac{1}{\beta}(s_{i,*}^\beta s_{j,*}^\beta - s_{i,*}s_{j,*}) \\ \Delta b_i = \frac{1}{\beta}(s_{i,*}^\beta - s_{i,*}) \end{cases} \tag{29}$$

C. Training Convolutional Networks with Equilibrium Propagation

In this section, we describe and define all operations used to train with EP a convolutional neural network recurrently connected with symmetric synapses. We describe the dynamics and the underlying learning rules for the weights and the biases in the prototypical and the energy-based settings. We denote N^{conv} and N^{fc} the number of convolutional layers and fully connected layers in the convolutional system, and $N^{\text{tot}} = N^{\text{conv}} + N^{\text{fc}}$. The units of the system are denoted by s and listed from $s^0 = x$ the input to the output $s^{N^{\text{tot}}}$.

C.1. Operations involved in the convolutional system

We detail here the operations involved in the dynamics of a convolutional RNN in both the prototypical and the energy-based settings.

- The 2-D convolution between w with dimension $(C_{\text{in}}, C_{\text{out}}, F, F)$ and an input x of dimensions $(C_{\text{in}}, H_{\text{in}}, S_{\text{in}})$ and stride one is a tensor y of size $(C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ defined by:

$$y_{c,h,s} = (w \star x)_{c,h,s} = B_c + \sum_{i=0}^{C_{\text{in}}-1}\sum_{j=0}^{F-1}\sum_{k=0}^{F-1} w_{c,i,j,k}x_{i,j+h,k+s}, \tag{30}$$

  where B_c is a channel-wise bias.

- The 2-D transpose convolution of y by $\tilde{w}$ is then defined in this work as the gradient of the 2-D convolution with respect to its input:

$$(\tilde{w} \star y) = \frac{\partial(w \star x)}{\partial x} \cdot y \tag{31}$$

- The dot product "$\bullet$" generalized to pairs of tensors of same shape (C, H, S) writes:

$$a \bullet b = \sum_{c=0}^{C-1}\sum_{h=0}^{H-1}\sum_{w=0}^{S-1} a_{c,h,s}b_{c,h,s}. \tag{32}$$

- The pooling operation $\mathcal{P}$ with stride F and filter size F of x:

$$\mathcal{P}_F(x)_{c,h,s} = \max_{i,j\in[0,F-1]} \left\{x_{c,F(h-1)+1+i,F(s-1)+1+j}\right\}, \tag{33}$$

  with relative indices of maximums within each pooling zone given by:

$$\text{ind}_{\mathcal{P}}(x)_{c,h,s} = \operatorname*{argmax}_{i,j\in[0,F-1]} \left\{x_{c,F(h-1)+1+i,F(s-1)+1+j}\right\} = (i^*(x,h), j^*(x,s)). \tag{34}$$

- The unpooling operation $\mathcal{P}^{-1}$ of y with indices $\mathrm{ind}_{\mathcal{P}}(x)$ is then defined as:

$$\mathcal{P}^{-1}(y, \mathrm{ind}_{\mathcal{P}}(x))_{c,h,s} = \sum_{i,j} y_{c,i,j} \cdot \delta_{h,F(i-1)+1+i^*(x,h)} \cdot \delta_{sF(j-1)+1+j^*(x,s)}, \tag{35}$$

  which consists in filling a tensor with the same dimensions as x with the values of y at the indices $\mathrm{ind}_{\mathcal{P}}(x)$, and zeroes elsewhere. For notational convenience, we omit to write explicitly the dependence on the indices except when appropriate. We can also see unpooling as the gradient of the pooling operation with respect to its input.

- The flattening operation $\mathcal{F}$ is defined as reshaping a tensor of dimensions (C, H, S) to $(1, CHS)$. We denote by $\mathcal{F}^{-1}$ its inverse.

C.2. Prototypical Settings

C.2.1 Equations of the dynamics

We derive here the dynamics of the convolutional network with symmetric connections and with the mean square error as loss function in the prototypical settings. In this case, the dynamics reads:

$$\begin{cases} s_{t+1}^{n+1} = \rho\left(\mathcal{P}(w_{n+1} \star s_t^n) + \tilde{w}_{n+2} \star \mathcal{P}^{-1}(s_t^{n+2})\right), & \forall n \in [0, N^{\mathrm{conv}} - 2] \\ s_{t+1}^{N^{\mathrm{conv}}} = \rho\left(\mathcal{P}(w_{N^{\mathrm{conv}}} \star s_t^{N^{\mathrm{conv}}-1}) + \mathcal{F}^{-1}({w_{N^{\mathrm{conv}}+1}}^{\top} \cdot s_t^{N^{\mathrm{conv}}+1})\right), \\ s_{t+1}^{N^{\mathrm{conv}}+1} = \rho\left(w_{N^{\mathrm{conv}}+1} \cdot \mathcal{F}(s_t^{N^{\mathrm{conv}}}) + {w_{N^{\mathrm{conv}}+2}}^{\top} \cdot s_t^{N^{\mathrm{conv}}+2}\right), \\ s_{t+1}^{n+1} = \rho\left(w_{n+1} \cdot s_t^n + {w_{n+2}}^{\top} \cdot s_t^{n+2}\right), & \forall n \in [N^{\mathrm{conv}} + 1, N^{\mathrm{tot}} - 2] \\ s_{t+1}^{N^{\mathrm{tot}}} = \rho\left(w_{N^{\mathrm{tot}}} \cdot s_t^{N^{\mathrm{tot}}-1}\right) + \beta(\mathrm{y} - s^{N_t^{\mathrm{tot}}}), & \text{with } \beta = 0 \text{ during the first phase,} \end{cases} \tag{36}$$

where we take the convention $s^0 = x$, the input. In this case, we have $s^{N^{\mathrm{tot}}} = \hat{y}$, the output layer. Considering the function:

$$\begin{aligned} \Phi(x, s^1, \cdots, s^{N^{\mathrm{tot}}}) = & \sum_{n=N_{\mathrm{conv}}+2}^{N_{\mathrm{tot}}-1} {s^{n+1}}^{\top} \cdot w_n \cdot s^n + s^{N_{\mathrm{conv}}+1} \cdot w_{N_{\mathrm{conv}}+1} \cdot \mathcal{F}(s^{N_{\mathrm{conv}}}) \\ & + \sum_{n=1}^{N_{\mathrm{conv}}-1} s^{n+1} \bullet \mathcal{P}\left(w_{n+1} \star s^n\right) + s^1 \bullet \mathcal{P}\left(w_1 \star x\right), \end{aligned}$$

when ignoring the activation function, we have:

$$\forall n \in [1, N^{\mathrm{tot}}]: \quad s_t^n \approx \frac{\partial \Phi}{\partial s^n}. \tag{37}$$

C.2.2 Learning rules

We derive the learning from the primitive function with the help of Eq. 28. In the prototypical settings, the learning rules read:

$$\begin{cases} \Delta w_1 = \frac{1}{\beta}\left(\mathcal{P}^{-1}(s_*^{1,\beta}) \star x - \mathcal{P}^{-1}(s_*^1) \star x\right) \\ \forall n \in [1, N_{\mathrm{conv}} - 1]: \quad \Delta w_{n+1} = \frac{1}{\beta}\left(\mathcal{P}^{-1}(s_*^{n+1,\beta}) \star s_*^{n,\beta} - \mathcal{P}^{-1}(s_*^{n+1}) \star s_*^n\right) \\ \Delta w_{N_{\mathrm{conv}}+1} = \frac{1}{\beta}\left(s_*^{N_{\mathrm{conv}}+1,\beta} \cdot \mathcal{F}\left(s_*^{N_{\mathrm{conv}},\beta}\right)^{\top} - s_*^{N_{\mathrm{conv}}+1} \cdot \mathcal{F}\left(s_*^{N_{\mathrm{conv}}}\right)^{\top}\right) \\ \forall n \in [N_{\mathrm{conv}} + 2, N_{\mathrm{tot}} - 1]: \quad \Delta w_n = \frac{1}{\beta}\left(s_*^{n+1,\beta} \cdot {s_*^{n,\beta}}^{\top} - s_*^{n+1} \cdot {s_*^n}^{\top}\right) \end{cases} \tag{38}$$

C.3. Energy-Based Settings

C.3.1 Equations of the Dynamics

Inspired by the primitive function derived in the prototypical settings we define an energy function which applies to an energy-based convolutional system, and rely on the same operations defined above:

$$E(x, s^1, \cdots, s^{N^{\text{tot}}}) = \frac{1}{2}\sum_{n=1}^{N_{tot}}(s^n)^2 - \sum_{n=1}^{N_{\text{tot}}} b_n\rho(s^n) - \frac{1}{2}\sum_{n=N_{\text{conv}}+2}^{N_{tot}-1} \rho(s^{n+1})^T.w_n.\rho(s^n)$$
$$- \rho(s^{N_{\text{conv}}+1}) \cdot w_{N_{\text{conv}}+1} \cdot \mathcal{F}(\rho(s^{N_{\text{conv}}})) - \sum_{n=1}^{N_{\text{conv}}-1} \rho(s^{n+1}) \bullet \mathcal{P}\left(w_{n+1} \star \rho(s^n)\right) - \rho(s^1) \bullet \mathcal{P}\left(w_1 \star x\right)$$

The dynamics is then derived from this energy function with the help of Eq. 1:

$$\begin{cases} \frac{\partial s^1}{\partial t} = -s^1 + \frac{\partial \rho(s^1)}{\partial s^1} \times \left(\mathcal{P}(w_1 \star \rho(x)) + \tilde{w}_2 \star \mathcal{P}^{-1}(\rho(s^2))\right), \\ \frac{\partial s^{n+1}}{\partial t} = -s^{n+1} + \frac{\partial \rho(s^{n+1})}{\partial s^{n+1}} \times \left(\mathcal{P}(w_{n+1} \star \rho(s^n)) + \tilde{w}_{n+2} \star \mathcal{P}^{-1}(\rho(s^{n+2}))\right), \qquad \forall n \in [1, N^{\text{conv}} - 2] \\ \frac{\partial s^{N^{\text{conv}}}}{\partial t} = -s^{N^{\text{conv}}} + \frac{\partial \rho(s^{N^{\text{conv}}})}{\partial s^{N^{\text{conv}}}} \times \left(\mathcal{P}(w_{N^{\text{conv}}} \star \rho(s^{N^{\text{conv}}-1})) + \mathcal{F}^{-1}({w_{N^{\text{conv}}+1}}^\top \cdot \rho(s^{N^{\text{conv}}+1}))\right), \\ \frac{\partial s^{N^{\text{conv}}+1}}{\partial t} = -s^{N^{\text{conv}}+1} + \frac{\partial \rho(s^{N^{\text{conv}}+1})}{\partial s^{N^{\text{conv}}+1}} \times \left(w_{N^{\text{conv}}+1} \cdot \mathcal{F}(\rho(s^{N^{\text{conv}}})) + {w_{N^{\text{conv}}+2}}^\top \cdot \rho(s^{N^{\text{conv}}+2})\right), \\ \frac{\partial s^{n+1}}{\partial t} = -s^{n+1} + \frac{\partial \rho(s^{n+1})}{\partial s^{n+1}} \times \left(w_{n+1} \cdot \rho(s^n) + {w_{n+2}}^\top \cdot \rho(s^{n+2})\right), \qquad \forall n \in [N^{\text{conv}} + 1, N^{\text{tot}} - 2] \\ \frac{\partial s^{N^{\text{tot}}}}{\partial t} = -s^{N^{\text{tot}}} + \frac{\partial \rho(s^{N^{\text{tot}}})}{\partial s^{N^{\text{tot}}}} \times \left(w_{N^{\text{tot}}} \cdot \rho(s^{N^{\text{tot}}-1})\right) + \beta(\text{y} - s^{N^{\text{tot}}}), \quad \text{with } \beta = 0 \text{ during the first phase.} \end{cases} \tag{39}$$

where again we have $s^{N^{\text{tot}}} = \hat{y}$, the output layer.

C.3.2 Learning Rules

We derive the learning from the primitive function with the help of Eq. 23. In the energy-based settings, the learning rules read:

$$\begin{cases} \Delta w_1 = \frac{1}{\beta}\left(\mathcal{P}^{-1}(\rho(s_*^{1,\beta})) \star x - \mathcal{P}^{-1}(\rho(s_*^1)) \star x\right) \\ \forall n \in [1, N_{\text{conv}} - 1]: \quad \Delta w_{n+1} = \frac{1}{\beta}\left(\mathcal{P}^{-1}(\rho(s_*^{n+1,\beta})) \star \rho(s_*^{n,\beta}) - \mathcal{P}^{-1}(\rho(s_*^{n+1})) \star \rho(s_*^n)\right) \\ \Delta w_{N_{\text{conv}}+1} = \frac{1}{\beta}\left(\rho(s_*^{N_{\text{conv}}+1,\beta}) \cdot \mathcal{F}\left(\rho(s_*^{N_{\text{conv}},\beta})\right)^\top - \rho(s_*^{N_{\text{conv}}+1}) \cdot \mathcal{F}\left(\rho(s_*^{N_{\text{conv}}})\right)^\top\right) \\ \forall n \in [N_{\text{conv}} + 2, N_{\text{tot}} - 1]: \quad \Delta w_n = \frac{1}{\beta}\left(\rho(s_*^{n+1,\beta}) \cdot \rho(s_*^{n,\beta^\top}) - \rho(s_*^{n+1}) \cdot \rho(s_*^{n^\top})\right) \end{cases}, \tag{40}$$

One should notice that we only need to store the activation $\rho(s)$ of the neurons to compute the gradient for each parameter which turns out to be very interesting when the activation function ρ outputs binary values, as we do in Section 4.

D. A Scaling Factor for Equilibrium Propagation

In this section, we discuss in detail the scaling factor introduced in Section 3. We first describe the initialization of the scaling factor. We then show that a naive initialization for the scaling factor inspired by XNOR-Net leads to good performance, but that tuning more precisely the scaling factor can increase the accuracy. Finally we derive learning rules for the scaling factors allowing EP to optimize by itself the value of the scaling factors. We show that systems learning their scaling factors better fit the training set but also learn faster.

D.1. Fixed Scaling Factor

D.1.1 Initializing α value

Rastegari *et al.* [29] introduced a scaling factor to normalize the binary weights in convolutional architectures with real-valued activations. They obtained the value of the scaling factors by minimizing layer-wise the squared difference between the binary weight vector $\mathbf{B}$ and the real-valued weight vector $\mathbf{W}$, where $\mathbf{B}$ and $\mathbf{W}$ are vectors in $\mathbb{R}^n$ and $n = c \times f_1 \times f_2$ with c the number of input channels, and f_1 and f_2 the sizes of the filter kernel. The factor α scales $\mathbf{B}$ in the following way:

$$\mathbf{W} = \alpha \mathbf{B} \tag{41}$$

They found that the optimal solution is given by:

$$\alpha^* = \frac{||\mathbf{W}||_{l1}}{\dim(W)} \tag{42}$$

In [29], the real-valued weights $\mathbf{W}$ are updated after each backward pas, and the scaling factor α is re-computed at each forward pass.

For training systems with binary synapses through EP, we first use a scaling factor fixed at initialization. We describe in Alg. 3 how we initialize the binary weights and the corresponding scaling factors layer-wise:

Algorithm 3 Initialize the scaling factors (α) layer-wise

Input: Architecture: $\{\mathrm{N_{Layers}}, \mathrm{N_{Neuronsperlayer}}\}$.
Output: System having binary weights (W^b) and scaling factors (α) initialized

for each Layer **do**
  w^{init} = rand(N) —— w^{init} is a random full-precision matrix
  $\alpha = \frac{||w^{init}||_{l1}}{\dim(w^{init})}$
  $W^b = \alpha \times \mathrm{Sign}(w^{init})$
end for

The rand() function used in Alg. 3 stands for the native random initialization of Pytorch which is the Kaiming initialization [12].

D.1.2 Naive initialization vs. our initialization

We found that the use of scaling factors α as initialized with Alg. 3 is crucial to ensure successful training. In fact, if the synapses are initialized to low or too large, we face the vanishing gradient issue as the activation saturate at both 0 or 1. In order to show this effect we trained a system with a fully connected architecture comprising 1 hidden layer of 4096 neurons, with binary weights and full-precision activations, for 50 epochs on MNIST. We plot in Fig. 4 the test error obtained with Alg. 3 for different values of the fixed scaling factors The blue arrow indicate the value corresponding to an initialization of α with Alg. 3 (we took the averaged value of both scaling factors in the network to obtain a point on the plot).

We see in Fig. 4 that the test error is highly dependent on the value of the scaling factor. The figure shows that for values of α between about 0.012 and 0.025 the test error is at EP literature level. But even in this range it is not obvious to find the best value for α. Moreover, we found that choosing arbitrarily the value of α in deep architectures (fully connected architecture with 2 hidden layers or convolutional architectures) fails at ensuring successful training. We finally chose to initialize the scaling factors with the method of Alg. 3 as this method is architecture-agnostic and reduces the number of hyperparameters as we already have some to tune.

In the next section, we address the difficulty to select the best value of α in order to get the best testing accuracy by directly learning the scaling factor with the help of EP.

D.2. Learning the Scaling Factor with EP

Results in the previous subsection show that optimizing the value of α can give rise to enhanced performance. Here we show that this optimization can be achieved through EP. In the context of EP we can indeed derive a learning rule for any parameter in the primitive or energy function. In this section, the scaling factor is first initialized with the method described in

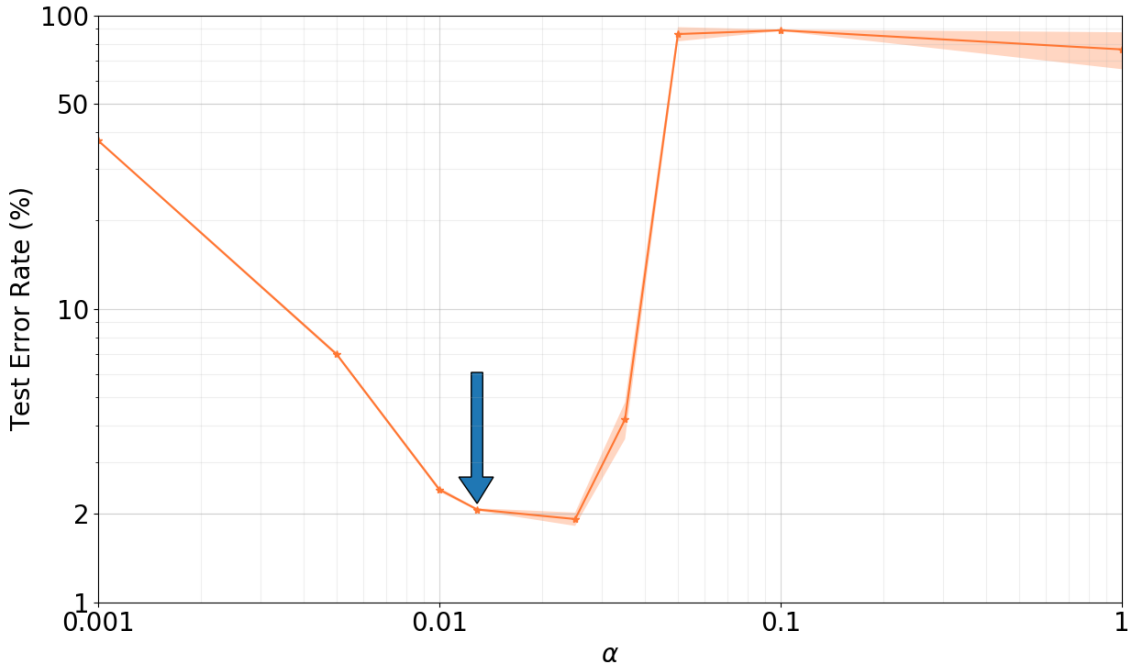

Figure 4: Mean test error $\pm$ standard deviation (computed with 3 trials each) of a 1 hidden layer fully connected neural network on MNIST as a function of the scaling factor α - All dots represent the test error of training performed with α being arbitrarily chosen - α initialized by the method described in D.1.1 is indicated by the blue arrow.

Alg. 3 and is then optimized with SGD with the gradient extracted by EP. For clarity, we decompose the binary weights W from $\pm\alpha$ to $\alpha \times w$ where $w = \pm 1$.

D.2.1 Learning Rules in the Prototypical settings

Fully connected layers architecture.
For a given fully connected layer, the scaling factor α can be introduced in the primitive function of the system as:

$$\Phi(s) = \frac{1}{2}\alpha \times s^T w s \tag{43}$$

Eq. 28 then indicates that the learning rule for the scaling factors in a fully connected architecture in the prototypical settings of EP is:

$$\Delta\alpha_{l,l+1} = \frac{1}{2\beta}\left((s_l^T w s_{l+1})^\beta - (s_l^T w s_{l+1}))^0\right) \tag{44}$$

where l denotes the index of a layer in the system.

Convolutional architecture:

The scaling factors in use for the classifier are updated with the gradient given by the learning rule stated above.

For convolutional layers, we use one scaling factor per output feature map which gives C_{out} scaling factors for a layer with C_{out} feature maps.

Thus for each channel in a convolutional layer c in C_{out} we can write:

$$\mathcal{P}\left(W_{n+1} \star s^n\right)_c = \alpha_c \times \mathcal{P}\left(w_{n+1} \star s^n\right)_c \tag{45}$$

where $W_{n+1} = \alpha_c \times w_{n+1}$ are the normalized weights for a channel and $w_{n+1} \in \{-1.1\}$. Following this observation, we can also rewrite a primitive function with α as we did for the fully connected architecture. From this primitive function, we can derive the learning rule for the scaling factors of the convolutional part which reads, channel-wisely:

$$\begin{cases} \forall n \in [1, N_{\text{conv}} - 1]: \quad \Delta\alpha_c^{n+1} = \frac{1}{\beta}((s_c^{n+1} \bullet \mathcal{P}\left(w_{n+1} \star s^n\right)_c)^\beta - (s_c^{n+1} \bullet \mathcal{P}\left(w_{n+1} \star s^n\right)_c)^0) \\ \Delta\alpha_c^1 = \frac{1}{\beta}((s_c^1 \bullet \mathcal{P}\left(w_1 \star x\right)_c)^\beta - (s_c^1 \bullet \mathcal{P}\left(w_1 \star x\right)_c)^0) \end{cases} \tag{46}$$

D.2.2 Learning Rules in the Energy-Based Settings

Fully connected layers architecture:

Similarly to the way we introduced α in the primitive function, we re-write the energy function of a fully connected layers architecture as a function of α:

$$E(s) = \frac{1}{2}\sum_i s_i - \frac{1}{2}\sum_{i \neq j} \alpha_{ij} w_{ij} \rho(s_i)\rho(s_i) - \sum_i b_i \rho(s_i) \tag{47}$$

Again, with the help of Eq. 23 we derive a learning rule for the scaling factors in a fully connected architecture in the energy-based settings of EP which reads as follow:

$$\Delta\alpha_{l,l+1} = \frac{1}{2\beta}\left((\rho(s_l^T) w \rho(s_{l+1}))^\beta - (\rho(s_l^T) w \rho(s_{l+1}))^0\right) \tag{48}$$

where l denotes the index of a layer in the system.

Convolutional architecture:

The scaling factors in use for the classifier are updated with the gradient given by the learning rule stated above.

In our convolutional networks, we use one scaling factor per feature map which gives C_{out} scaling factors for a layer with C_{out} feature maps. For each feature map, we have Eq. 45 verified and we can also easily derive the learning rule of the scaling factors of the convolutional layers which reads, channel-wise:

$$\begin{cases} \forall n \in [1, N_{\text{conv}} - 1]: \quad \Delta\alpha_c^{n+1} = \frac{1}{\beta}((\rho(s_c^{n+1}) \bullet \mathcal{P}\left(W_{n+1} \star \rho(s^n)_c\right))^\beta - (\rho(s_c^{n+1}) \bullet \mathcal{P}\left(W_{n+1} \star \rho(s^n)_c\right)^0)) \\ \Delta\alpha_c^1 = \frac{1}{\beta}((\rho(s_c^1) \bullet \mathcal{P}\left(W_1 \star x\right)_c)^\beta - (\rho(s_c^1) \bullet \mathcal{P}\left(W_1 \star x\right)_c)^0) \end{cases} \tag{49}$$

D.3. Benefits of Learning the Scaling Factor when the Synapses are Binary and the Activations Full-Precision

In the next Tables 3, 4 and 5, we report the training and test errors obtained with fixed and dynamical scaling factors on MNIST and CIFAR-10 with different architectures, at mid-training and at the end of the training.

Learning the scaling factor accelerates the training. In these tables, we show that the training times are accelerated when the scaling factor is dynamical instead of fixed after initialization. In particular for MNIST, the training is accelerated by a factor over two compared to the fixed scaling, both for fully connected and convolutional architectures.

For CIFAR-10 the acceleration is not as large as for MNIST but we struggled to fine-tune the learning rate for the scaling factors and thus better combinations could give larger acceleration.

Systems learning the scaling factors better fit the training set. Also in these tables we see that every trainings done with dynamical scaling factors always better fit the training set than trainings done with fixed scaling factors. We also see in these tables that every training done with dynamical scaling factors better fits the training set than with fixed scaling factors. Training errors on MNIST are improved by 0.8% and 0.15% for fully connected layers architectures having 1 and 2 hidden layers. The convolutional architecture trained on MNIST also gains 0.45% in terms of training error. Whereas the fully connected architectures sees the test error also improved alongside the training error, the convolutional architecture sees a slight degradation of the test error due tooverfitting.

The convolutional architecture trained on CIFAR-10 gains 1.1% training error.

Can learning the scaling factor reduce the memory requirements of the network? Finally, learning the scaling allows to train a fully connected architecture with 1 hidden layer of only 512 hidden neurons (the architecture usually trained by EP in the literature) with very low loss of accuracy: +0.2% testing error and +0.6% training error (see Table 3 and Fig. 1) whereas with a fixed scaling factor we have +2% testing error and +3% training error (see Table 3 and Fig. 8). However, we did not make this observation across all the architectures that we have studied and only rise it here as a curiosity.

Table 3: Mean Train and Test errors on MNIST (over 5 trials each) computed after 25 and 50 epochs for two shallow networks with one and two hidden layers with binary synapses trained with EqProp - We denote in the *Learn* α column if the scaling factor is learnt or not

		25 Epochs	50 Epochs
Architecture	Learn α	Test (Train)	Test (Train)
784-4096-10	✗	2.14 (0.92)	2.07 (0.77)
784-4096-10	✓	1.66 (0.03)	**1.7 (0)**
784-512-10	✓	2.45 (1.24)	**2.2 (0.7)**
784-4096(2)-10	✗	2.47 (0.4)	2.48 (0.15)
784-4096(2)-10	✓	2.27 (0.02)	**2.28 (0)**

Table 4: Mean Train and Test errors on MNIST (over 5 trials each) with a convolution network with binary synapses trained with EqProp - We denote in the *Learn* α column if the scaling factor is learnt or not

		25 Epochs	50 Epochs
Architecture	Learn α	Test (Train)	Test (Train)
1-32-64-(fc)	✗	0.92 (0.63)	0.84 (0.46)
1-32-64-(fc)	✓	0.79 (0.16)	**0.88 (0.047)**

Table 5: Mean Train and Test errors on CIFAR-10 (over 5 trials each) with a convolution network with binary synapses trained with EqProp - We denote in the *Learn* α column if the scaling factor is learnt or not

		100 Epochs	500 Epochs
Architecture	Learn α	Test (Train)	Test (Train)
3-68-128-256-256-(fc)	✗	18.4 (13.4)	16.8 (6.9)
3-68-128-256-256-(fc)	✓	17.6 (11.86)	**15.54 (5.54)**

E. Propagation of the Error Signal with Binary Activations

In this section we discuss how the binary activation function can cancel the error signal flowing from the nudged output neurons to the other upstream layers. We then explain how we can enhance the error signal in order to yield a nudging force strong enough to propagate throughout the system.

E.1. The Error Signal Flows Into the System if the First Hidden Layer is Sensitive to it

A layered architecture trained with EP makes the system sensitive to the error signal if and only if neurons between the output layer - where the error signal is applied - and a neuron of interest, are sensitive to the target. The first hidden layer is in this sense a bottleneck for the error signal. It often receives more forward signal from the other hidden layers than backward signal from the output layer which only has 10 neurons for MNIST and CIFAR-10. During the nudging phase of EP, only one neuron in the output layer is nudged to be 1, the others being nudged to 0. And this little change in the output layer in not sufficient for the first hidden layer to reach the criteria of good error signal Eq. 16. Therefore, the binary activations of the neurons in the first hidden layer do not change and the error signal is blocked.

Once the first hidden layer changes its binary activations, the error signal can flow through the network. In fact, it has more impact on the others hidden layers because it is often larger than the output layer and matches approximately the size of the others layers thus it is more likely to impact the binary activation of the next hidden layer. Augmenting the error signal is crucial to train systems with binary synapses and activations with EP.

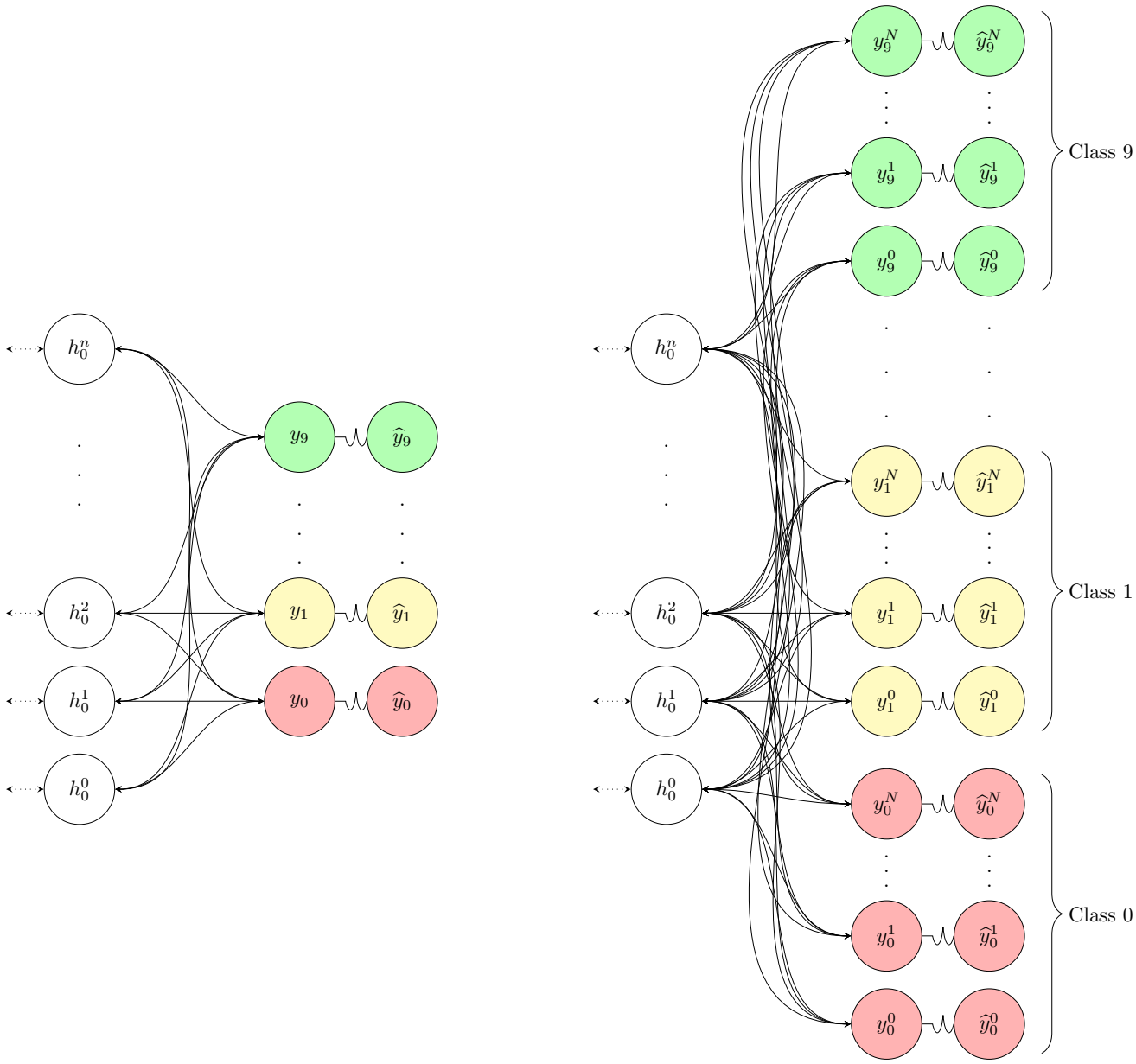

Figure 5: Left: Schematic of the classic output layer with one output neuron per class - compared to Right: the enlarged output layer where we have N_{perclass} output neurons per class - Hidden units are denoted by h, output units by y^x and the target units by $\widehat{y}^x$ - We represent the nudging of the output neurons by the corresponding target units with the small springs on the schematic - For simplicity we drew this schematic for datasets having 10 output classes but it can be applied to any dataset - Dashed arrows on the left hand of both networks indicate the bidirectional connections with the rest of the network

E.2. An Enlarged Output Layer for a Greater Error Signal

The scaling factors introduced in Section 3 to normalize the binary weights and not to saturate the activations show limitations with binary activations. Neurons with binary activations can sometimes indeed no longer propagate the error signal.

We enlarged the output layer to solve this issue as described in Fig. 5. This enlargement of the output layer makes the neurons having a binary activation again sensitive to the error signal and their activation can change during the nudging phase. This opens the path to training deeper architectures with binary activations and weights with EP. Our solution is similar in spirit to the augmented output layer used by [3].

E.3. Making Predictions with an Enlarged Output Layer

Usually an output layer has as many neurons as the number of classes in the dataset and the prediction is the *argmax* of the output layer.

But when we train systems having binary activations the output layer is augmented and it is not straightforward to make a prediction taking the *argmax* of the output layer. We describe here two methods to make a prediction with the enlarged output layer. We used both methods in our simulations and show they give similar accuracy in the end.

Making predictions by averaging each sub-class. This first method allow us to retrieve a situation similar to the classic output layer having one neuron per class. In fact we first average the internal state - or pre-activation - of each neuron belonging to a class which gives 10 averaged values and the prediction is taken as the *argmax* of these averaged values.

Making predictions with one neuron per sub-class. The first method we describe above to make the prediction could reveal to be computationally and time expensive and costly to realize on digital hardware. A second, more hardware-friendly, method is to look at the state of only one output neuron per class and take the argmax of these "single-neurons".

Comparison of the two methods. We want to see if the second method, which constitutes a great simplification of the prediction process, performs as well as the first method. For this purpose, we plot in Fig. 6 and Fig. 7 the difference of the training and the testing errors of two fully connected architectures with binary synapses and activations and 1 and 2 hidden layers trained on MNIST computed with the two methods described above. We see that for the network with 1 hidden layer, the difference between the two methods does not exceed 0.1% for both training and testing errors. For the network with 2 hidden layers, despite the fact that the difference starts at a high level with more than 0.7% of difference for the testing error and more than 1% for the training error, in the end of the training process the differences have decreased to almost 0%. Both figures show the effectiveness of the second method at making the predictions at a lower cost both computationally and in time than the first method.

F. Simulations Details - Hyperparameters and Training Curves

F.1. Binary Synapses

We detail in this section all settings and parameters used for the simulations for EP with binary synapses and full-precision activations (hardsigmoid activation function). We ran the simulations with PyTorch and speed them up on a GPU. The duration of the simulations runs from 30 mins for the shallow network to 5 days for the convolutional architecture on CIFAR-10.

For these simulations, we use the prototypical settings of EP for the sake of saving simulation time. The energy-based settings would perform the same way but such models are much longer to train.

We found that comparatively to full-precision models trained by EP, the error signal vanishes through the system and thus deep layers need a greater learning rate for the biases and greater γ for the weights. All hyperparameters are reported in Table 6.

The target is one-hot encoded and the prediction is computed by taking the argmax of the state of the output neurons. The output layer is designed in a way that we have one output neuron per class of the dataset. We initialize the binary weights taking the sign of randomly-initialized weights matrices.

We choose the sign of beta randomly at each mini-batch which is known to give better results [30, 20]. For all simulations we used mini-batches of size of 64 as we found it performs better.

All figures report the mean of the training and testing errors computed with 5 trials each $\pm$ 1 standard deviation.

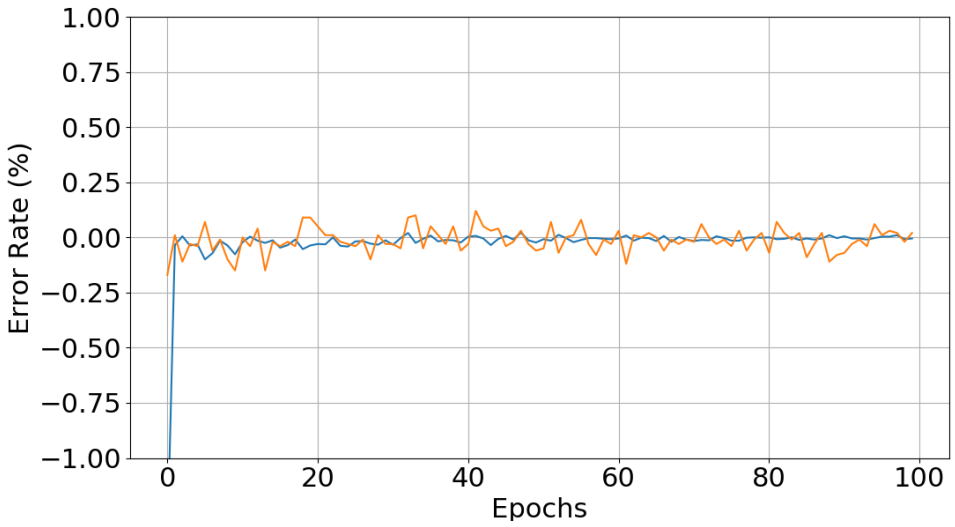

Figure 6: Difference of the train and test errors computed with the averaging method and the method with only one neuron per subclass for a system with one hidden layer of 8192 neurons and 100 output neurons

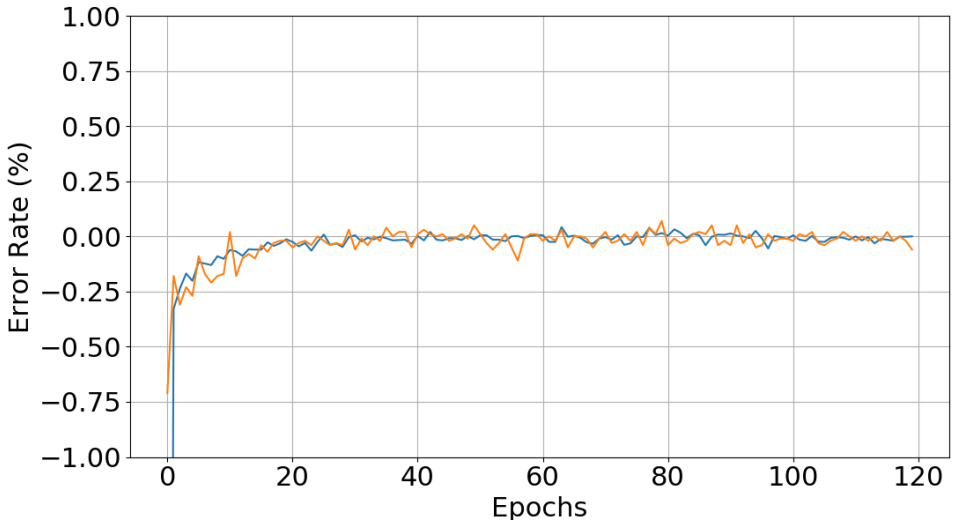

Figure 7: Difference of the train and test errors computed with the averaging method and the method with only one neuron per subclass for a system with two hidden layers of 8192 neurons and 8000 output neurons

MNIST - fully connected layer - 1 hidden layer. We train a network with a fully connected architecture and 1 hidden layer on MNIST. We first tuned the EP hyperparameters (T, K, β) making EP gradient estimates match those given by BPTT [8]. At the same time we tuned BOP hyperparameters in order to fit the flipping metric (Eq. 6) in the range leading to successful training as described in Section 3. We found that contrarily to Helwegen *et al*. [13], the flipping metric starts at high level (between 0 and $-4/-5$) and decreases over epochs to reach a region below -5.

We initialize one scaling factor per weight matrix with the method described in Alg. 3. When the scaling factors are learnt, we use the same learning rate for all scaling factors. Despite the fact that the learning rule for the scaling factors requires the

sign of the weights ± 1 for the computation, we found that using the scaled weights $\pm\alpha$ performs the same way so we used the scaled weight matrix to compute the gradient.

To reach an accuracy at levels of reported results in the literature with such architecture trained by EP on MNIST, we needed to increase by 8 the number of neurons in the hidden layer as shown in Fig. 8 when the scaling factors are fixed which justifies the architecture we trained: 784-4096-10.

We report all hyperparameters in Table 6. We initialize the biases with the native PyTorch random initialization and the state of the neurons to zero as it has proven to perform better.

We performed two sets of simulations:

- Simulations where the scaling factors were fixed which achieve accuracy (Table 1, Fig. 9) close to those reported in the EP literature: [8].
- Simulations where the scaling factors were learnt. We show that learning the scaling factors improves by a considerable margin the training -0.7% and the testing -0.4% errors: Fig. 9, Fig. 10 and Table 3. We link the better testing error to a better fit on the training set as the network seems to overfit a bit: the testing error starts to increase after 10 epochs which also highlights the fact that when we learn the scaling factors, we can use less neurons per hidden layer and still get accuracy close to those reported in the EP literature. We also report that the training is at least five times faster, as after 10 epochs the training and testing errors are below the levels obtained after 50 epochs with fixed scaling factors. Learning the scaling factors makes the flipping metric of BOP to decrease more quickly than when the scaling factors are fixed.

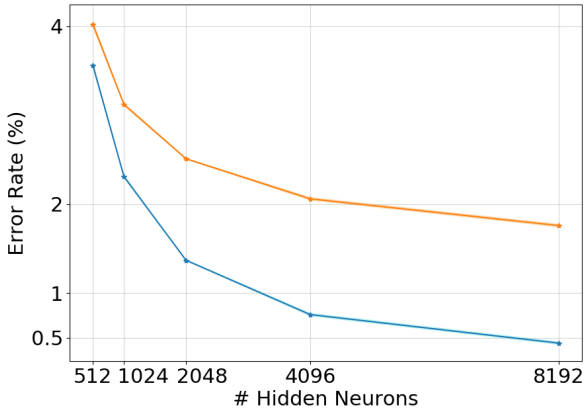

Figure 8: Averaged train (blue) and test (orange) errors on MNIST with a fully connected architecture with one hidden layers as a function of the number of hidden neurons - We average the errors over 5 trials and plot the average $\pm$ 1 standard deviation

MNIST - fully connected layer - 2 hidden layers. We train a network with a fully connected architecture which has 2 hidden layers on MNIST. We initially chose EP and BOP hyperparameters close to the hyperparameters chosen for training the network with one hidden layer network and then fine-tuned them to achieve the best accuracy. The metric of BOP (Eq. 6) also decreases over epochs to reach a level below -5 in the good range for BOP.

Again, we initialize with Alg. 3 one scaling factor per weight matrix which gives 3 scaling factors for this architecture. We also use the same learning rate for all scaling factors and the scaled weights for computing the gradient as done with the architecture which has 1 hidden layer.

We kept the same number of neurons (4096) in each hidden layer as for the architecture which has only 1 hidden layer.

We report all hyperparameters in Table 6. We initialize the weights with the native PyTorch random initialization and the state of the neurons to zero as it has proven to perform better.

We performed two sets of simulations:

- Simulations where the scaling factors were fixed, which achieve accuracy (Table 1, Fig. 11) close to those reported in the EP literature [8].

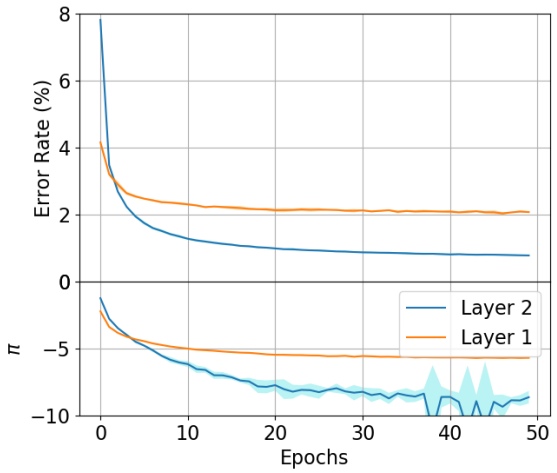

Figure 9: Top: Train (blue) and test (orange) error on MNIST with a fully connected architecture with one hidden layer of 4096 neurons trained with EP with binary synapses - The scaling factors are fixed - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output

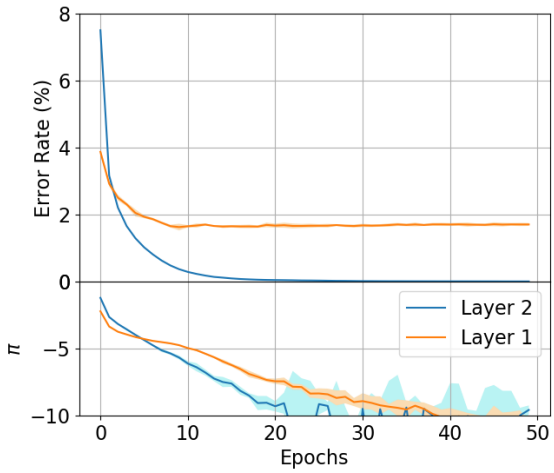

Figure 10: Top: Train (blue) and test (orange) error on MNIST with a fully connected architecture with one hidden layer of 4096 neurons trained with EP with binary synapses - The scaling factors are learnt - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output

- Simulations where the scaling factors were learnt. We show that learning the scaling factors improves a lot the fit by 0.15% on the train set and thus improves the testing accuracy by 0.2% in Fig. 11, Fig. 12 and Table 3. Finally learning the scaling factors also speed up the training by at least a factor 5.

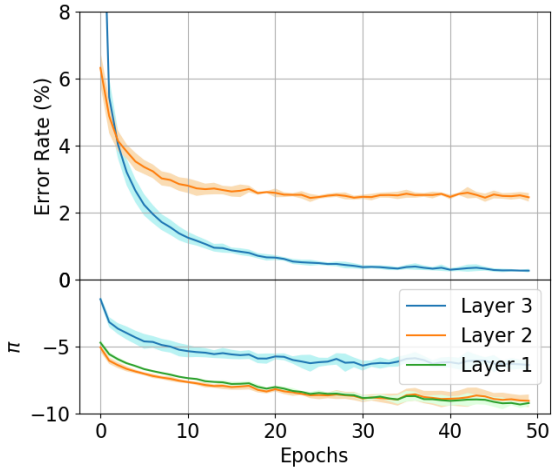

Figure 11: Top: Train (blue) and test (orange) error on MNIST with a fully connected architecture with two hidden layers of 4096 neurons trained with EP with binary synapses - The scaling factors are fixed - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output

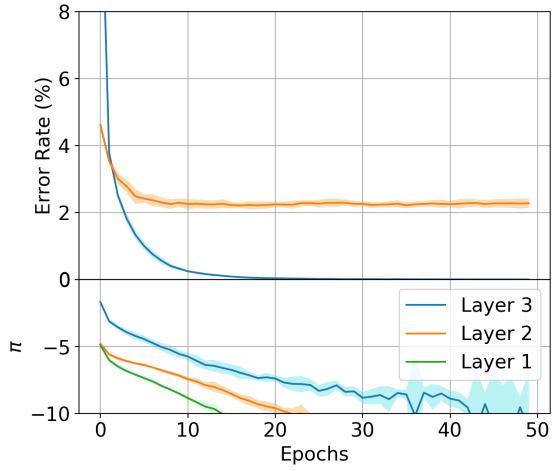

Figure 12: Top: Train (blue) and test (orange) error on MNIST with a fully connected architecture with two hidden layers of 4096 neurons trained with EP with binary synapses - The scaling factors are learnt - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output

MNIST - convolutional architecture: We train a convolutional network on MNIST. The architecture used consists in the following: 2 convolutional layers of respectively 32 and 64 channels. We use convolutional kernels of size 5×5, padding of 2 and a stride of 1. Each convolutional operation is followed by a 2 Max Pooling operation with a stride of 2. We flatten the output of the last convolutional layer to feed the output layer of 10 neurons.

We tuned EP hyperparameters (T,K,β) making EP gradient estimates match the gradient given by BPTT [8]. We tuned BOP hyperparameters to make the metric in the range below -5.

The scaling factors α are initialized channel-wise in each convolutional layer which gives 32 scaling factors for the first convolutional layer and 64 scaling factors for the second convolutional layer with the architecture used here. Again we use the scaled weights to compute the gradient of each scaling factor.

We performed two sets of simulations:

- Simulations where the scaling factors were fixed. We report accuracy slightly below the one reported with EP on MNIST with the same convolutional architecture in [8]: -0.4% for the training and -0.2% for the testing errors. Two things one: as underscored before, BOP seems to regularize the training with EP but also we used the sign of β randomly which is known to better estimate the gradient given by EP and thus improve the training.
- Simulations where the scaling factors were learnt. Learning the scaling factors allow the system to better fit the training set (-0.5% of training error). But this makes the system to overfit as the testing error increases to 0.88% after 50 epochs after having reached a minimum at 0.76% after 25 epochs. Learning the scaling factors also decreases more the flipping metric π as shown in Fig. 13, Fig. 14 and Table 4.

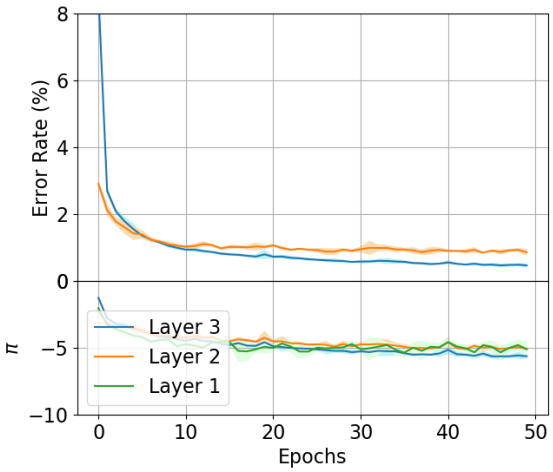

Figure 13: Top: Train (blue) and test (orange) error on MNIST with a convolutional architecture with 2 convolutional layers of respectively 32 and 64 channels trained with EP with binary synapses - The scaling factors are fixed - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output.

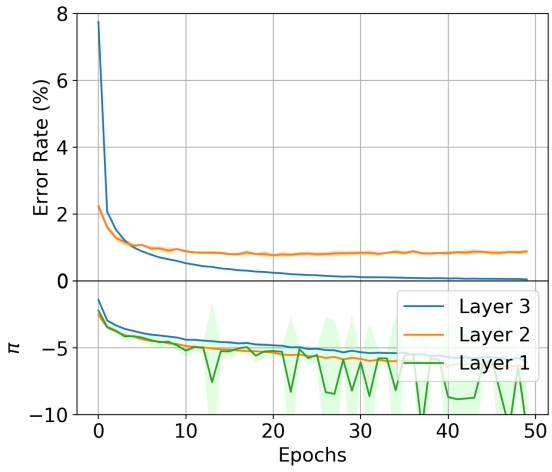

Figure 14: Top: Train (blue) and test (orange) error on MNIST with a convolutional architecture with 2 convolutional layers of respectively 32 and 64 channels trained with EP with binary synapses - The scaling factors are learnt - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output.

CIFAR-10 - convolutional architecture. We train a convolutional network on CIFAR-10. The architecture used consists in the following: 3-64-128-256-256-fc(10): 4 convolutional layers of respectively 64, 128, 256 and 256 channels, one output layer of 10 neurons. We use convolutional kernels of size 5×5, padding of 2 and a stride of 1. Each convolutional operation is followed by a 2 Max Pooling operation with a stride of 2. We flatten the output of the last convolutional layer to feed the output layer.

Because we used twice as less feature maps at each convolutional layer to speed up our training simulations compared to the original network Laborieux *et al.* [20], we used the available code [1] to run simulations with the same architecture as ours to

[1]The code is available at: https://github.com/Laborieux-Axel/Equilibrium-Propagation.

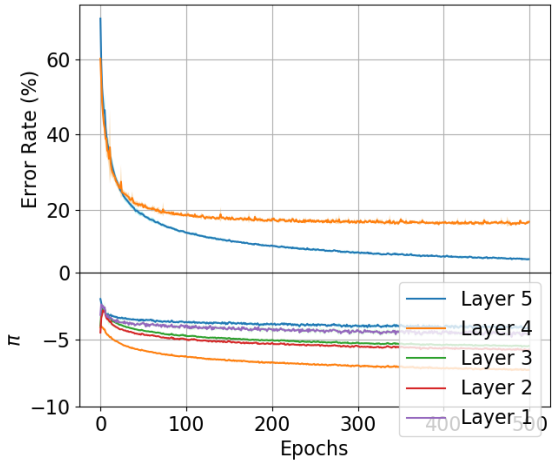

Figure 15: Top: Train (blue) and test (orange) error on CIFAR-10 with a convolutional architecture with 4 convolutional layers of respectively 64, 128, 256 and 256 channels trained with EP with binary synapses - The scaling factors are fixed - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output.

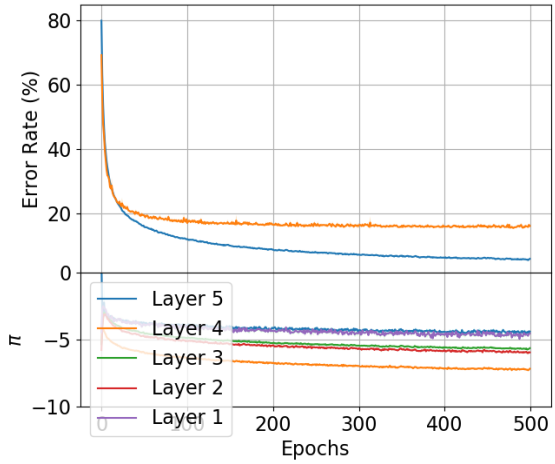

Figure 16: Top: Train (blue) and test (orange) error on CIFAR-10 with a convolutional architecture with 4 convolutional layers of respectively 64, 128, 256 and 256 channels trained with EP with binary synapses - The scaling factors are learnt - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output.

Table 6: Hyperparameters used for training systems with EP and binary synapses - $lrBias$ are the learning rates used for updating the biases with SGD and given from input to output layer - γ is layer-dependent and given from input to output layer

			EP			BOP		
Dataset	Method	Architecture	T	K	β	γ	τ	$lrBias$
MNIST	fc	784-4096-10	50	10	0.3	1e-4-1e-5	5e-7	0.05-0.025
MNIST	fc	784-4096(2)-10	250	10	0.3	2e-5-2e-5-5e-6	5e-7	0.2-0.1-0.05
MNSIT	conv	1-32-64-fc	150	10	0.3	5e-8	1e-8	0.1-0.05-0.025
CIFAR-10	conv	3-64-128-256(2)-fc	150	10	0.3	1e-7(2)-2e-7(2)-5e-8	1e-8	0.4-0.2-0.1-0.05-0.025

benchmark our technique.

We chose EP hyperparameters equal to those used for the convolutional architecture trained on MNIST as it has shown to work well. We tuned BOP hyperparameters to make the metric in the good range for BOP.

The scaling factors α are initialized channel-wise in each convolutional layer which for instance gives 64 scaling factors for the first convolutional layer with the architecture used here. Again we use the scaled weights to compute the gradient of each scaling factor.

We performed two sets of simulations:

- Simulations where the scaling factors were fixed which achieve accuracy (Table 1, Fig. 15) close to those reported in the EP literature ([20]).
- Simulations where the scaling factors were learnt. We show that learning the scaling factors improves a lot the fit by 1.4% on the train set and thus improves the testing accuracy by 1.2% in Fig. 15, Fig. 16 and Table 5.

We pre-processed CIFAR-10 with the following data augmentation and normalization techniques before feeding it to the system:

- Random Horizontal Flip with $p = 0.5$

- Random Crop with $padding = 4$
- Normalization with $\mu = (0.4914, 0.4822, 0.4465)$ and $\sigma = (0.247, 0.243, 0.261)$ for each rgb input channel.

F.2. Binary Synapses and Activations

We detail in this section all settings and parameters used for the simulations of EP with binary synapses and binary activations. We ran the simulations with PyTorch and sped them up on a GPU. For all simulations we used mini-batches of size of 64 as we found it performs better. The time of the simulations runs from 30 mins for the shallow network to 5 days for the convolutional architecture on CIFAR-10.

For the simulations of EP with binary synapses and binary activations we use the energy-based settings of EP with the rules derived in Section 4 such as the pseudo-derivative of the Heavyside step function and the enlarged output layer.

In this section, we explore how τ can be finely tuned layer-wise in order to give the best performance while having the same γ for all layers which could be more hardware friendly as we could use the same devices to store the momentum and only change the threshold layer-wise. All hyperparameters are reported in Table 6.

The target is one-hot encoded and then replicated N_{perclass} times to match the size of the output layer. We make the prediction with the two methods described in E.3. We initialize the binary weights taking the sign of randomly-initialized weights matrices (native random initialization of Pytorch which is the Uniform Kaiming initialisation).

The input data is kept full-precision thus the MAC operation for the first layer of each architecture is full-precision and also the gradient.

We choose the sign of beta randomly at each mini-batch which is known to give better results [30], [20] for all simulations except when training the network with the fully connected architecture and which has 2 hidden layers where we only used $\beta > 0$.

We used the Heaviside step function as the binary activation function as emphasised in Section 4. For defining the pseudo-derivative function $\hat{\rho}^{'}(s)$ (Eq. 14) we used $\sigma = 0.5$ despite using a binary activation.

All figures report the mean of the training and testing errors computed with 5 trials each $\pm$ 1 standard deviation.

MNIST - fully connected layer - 1 hidden layer: We train a network with a fully connected architecture and 1 hidden layer on MNIST.

We chose EP hyperparameters (T, K, β) close to those used for training the same architecture but with binary synapses and full-precision activations. At the same time we tuned BOP hyperparameters in order to fit the flipping metric in the range leading to successful training as described in Section 3.

We initialize one scaling factor per weight matrix with the method described in Alg. 3. Simulations with a learnt scaling factors gave results only for 1 hidden layer. When we deepened the network to be trained, learning the scaling factor does not behave well, which we think it is due to the nudging strategy (notably when $\beta < 0$) which does not give an accurate estimation of the gradient.

To reach an accuracy at the level of reported results in the literature with such architecture trained by EP on MNIST, we needed to increase by 16 the number of neurons in the hidden layer which gives the architecture we trained: 784-8192-100. We chose 100 output neurons as it is approximately the number of input units times the sparsity of MNIST data and 100 has shown to perform the best.

We report all hyperparameters in Table 7. We initialize the biases with the native PyTorch random initialization and the state of the neurons to one as it has proven to perform better.

We performed two sets of simulations:

- Simulations where the scaling factors were fixed which achieve accuracy (Table 2, Fig. 17) close to those reported in the EP literature: [8].
- Simulations where the scaling factors were learnt. We show that learning the scaling factors only improve a little bit the training error: -0.1% but not the testing error: Fig. 17, Fig. 18.

MNIST - fully connected layer - 2 hidden layers We train a network with a fully connected architecture and 1 hidden layer on MNIST.

We chose EP hyperparameters (T,K,β) close to those used for training the same architecture but with binary synapses and full-precision activations. At the same time we tuned BOP hyperparameters in order to fit the flipping metric in the range leading to successful trainings as described in Section 4.

We initialize one scaling factor per weight matrix with the method described in Alg. 3.

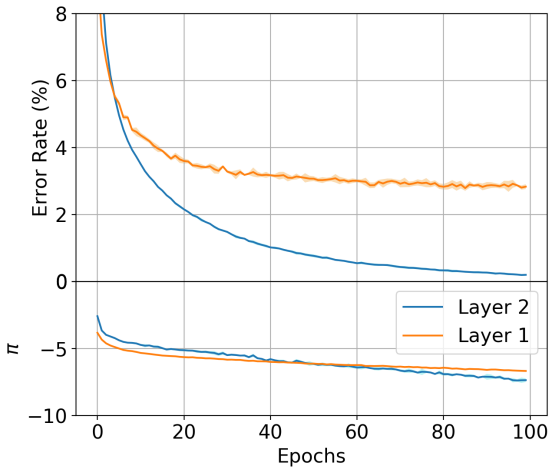

Figure 17: Top: Train (blue) and test (orange) error on MNIST with a fully connected architecture with one hidden layer of 8192 neurons trained with EP with binary synapses and binary activations - The scaling factors are fixed - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output.

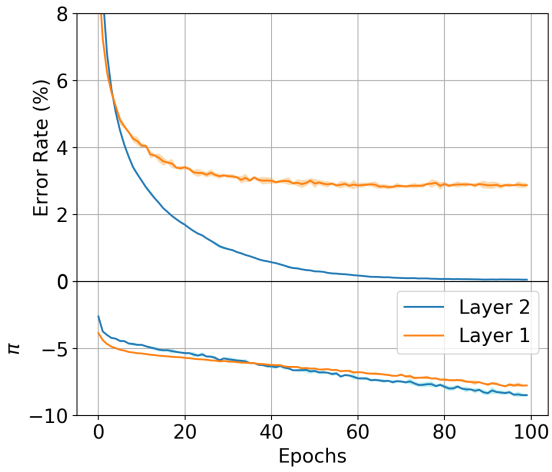

Figure 18: Top: Train (blue) and test (orange) error on MNIST with a fully connected architecture with one hidden layer of 8192 neurons trained with EP with binary synapses and binary activations - The scaling factor is learnt - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output.

We chose 6000 output neurons as it gives the best accuracy but also as it scales as the number of hidden neurons in the penultimate hidden layer times some sparsity in the layer.

We initially perform a nudging with the sign of β chosen randomly at each mini-batch. But when we nudge the system with $\beta < 0$, it appears that we should let the system evolve during K time steps with K very large (of the order of at least 500 time steps). Finally, we chose to nudge only using the sign of $\beta > 0$ despite the trainings perform less than if we used the sign of beta randomly. Monitoring the temporal evolution of some neurons in the network can also help at tuning EP hyperparameters.

To reach an accuracy at levels of reported results in the literature with such architecture trained by EP on MNIST, we used the following architecture we trained: 784-8192-8192-6000. We report all hyperparameters in Table 7. We initialize the biases with the native PyTorch random initialization and the state of the neurons to one as it has proven to perform better.

We report all hyperparameters in Table 7. We initialize the biases with the native PyTorch random initialization and the state of the neurons to one as it has proven to perform better.

MNIST - convolutional architecture We train a convolutional network on MNIST. The architecture used consists in the following: 2 convolutional layers of respectively 256 and 512 channels. We use convolutional kernels of size 5×5, padding of 1 and a stride of 1. Each convolutional operation is followed by a 3 Max Pooling operation with a stride of 3. We flatten the output of the last convolutional layer to feed the output layer of 700 neurons.

We tuned BOP hyperparameters to make the metric in the range below -5.

The scaling factors α are initialized channel-wise in each convolutional layer which gives 256 scaling factors for the first convolutional layer and 512 scaling factors for the second convolutional layer with the architecture used here.

Again, learning the scaling factors did not show better accuracy and could be also linked to the nudging strategy.

We initialize the biases at 0 and the state of the neurons to one as it has proven to perform better.

Finally, here we adopted another nudging implementation: although the nudging is usually performed by adding the derivative of the loss function with respect to the units of the output layer $+\beta(\mathrm{y} - \hat{y})$, we implemented a constant nudge: $+\beta(\mathrm{y} - \hat{y}_*)$, where $\hat{y}_*$ stands for the first steady state reached by the output units at the end of the first phase. This nudge has shown to perform better than the classic nudge.

We report all hyperparameters in Table 7. We initialize the biases with the native PyTorch random initialization and the state of the neurons to one as it has proven to perform better.

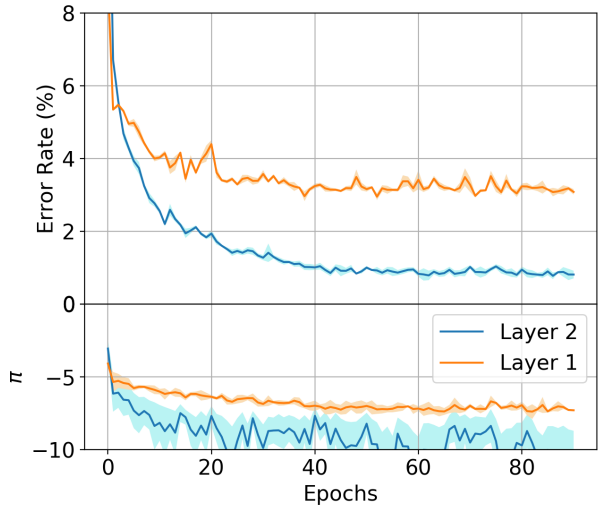

Figure 19: Top: Train (blue) and test (orange) error on MNIST with a fully connected architecture with two hidden layers of 8192 neurons trained with EP with binary synapses & binary neurons - The scaling factors are fixed - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output.

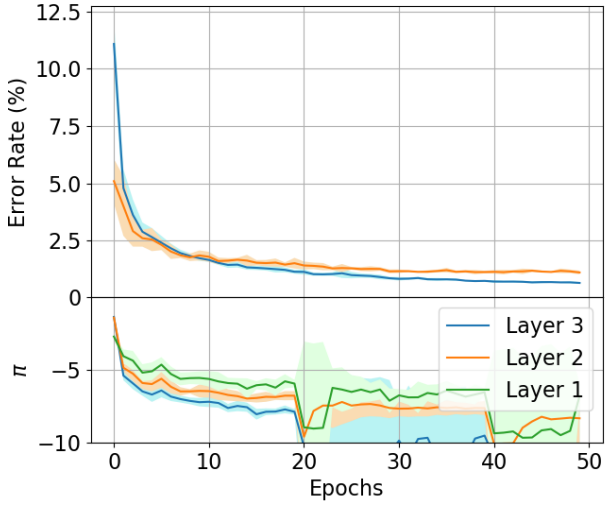

Figure 20: Top: Train (blue) and test (orange) error on MNIST with a convolutional architecture with 2 convolutional layers of respectively 256 and 512 channels trained with EP with binary synapses & binary neurons - The scaling factors are fixed - Down: metric of weights flipped during an epoch, given for each weights matrix from input to output.

Table 7: Hyperparameters used for training systems with EP and binary synapses with binary neurons - γ is layer-dependent and given from input to output layer when multiple values are given - γ has the same value for all layers when a single value is given.

			EP			BOP		
Dataset	Method	Architecture	T	K	β	γ	τ	$lrBias$
MNIST	fc	784-8192-100	20	10	2	2e-6	2.5e-7 - 2e-7	1e-7
MNIST	fc	784-8192(2)-8000	30	80	2	1e-6	2e-8 - 1e-8 - 5e-8	1e-6
MNIST	conv	1-256-512-1600(fc)	100	50	1	5e-8	8e-8 - 8e-8 - 2e-7	2e-6 - 5e-6 - 1e-5