A Universal Abstraction for Hierarchical Hopfield Networks

Anonymous Author(s)
Affiliation
Address
email

Abstract

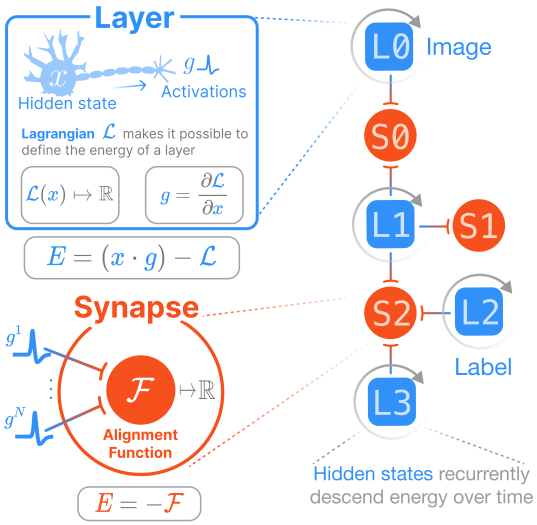
Conceptualized as Associative Memory, Hopfield Networks (HNs) are powerful models which describe neural network dynamics converging to a local minimum of an energy function. HNs are conventionally described by a neural network with two layers connected by a matrix of synaptic weights. However, it is not well known that the Hopfield framework generalizes to systems in which many neuron layers and synapses work together as a unified Hierarchical Associative Memory (HAM) model: a single network described by memory retrieval dynamics (convergence to a fixed point) and governed by a global energy function. In this work we introduce a universal abstraction for HAMs using the building blocks of neuron layers (nodes) and synapses (edges) connected within a hypergraph. We implement this abstraction as a software framework, written in JAX, whose autograd feature removes the need to derive update rules for the complicated energy-based dynamics. Our framework, called HAMUX (**HAM U**ser e**X**perience), enables anyone to build and train hierarchical HNs using familiar operations like convolutions and attention alongside activation functions like Softmaxes, ReLUs, and LayerNorms. HAMUX is a powerful tool to study HNs at scale, something that has never been possible before. We believe that HAMUX lays the groundwork for a new type of AI framework built around dynamical systems and energy-based associative memories.

1 Introduction

Non-linear ordinary differential equations (ODEs) are extensively used in modern AI architectures, leading to impressive results. An important subset of general ODEs are systems with an underlying global Lyapunov function, often called an energy function, which decreases in time as the non-linear dynamical system approaches the fixed point state. Paradigmatic examples of such systems are **Hopfield Networks** (HNs), introduced in 1982 by John Hopfield as models of associative memory retrieval [1, 2]. The core idea is that the non-linear (discrete or continuous) dynamical system, governed by the energy function, is designed to have multiple fixed points (memories) with substantial basins of attraction around them. Given an initial prompt (a query), the network picks one of the basins of attraction and follows the gradient of the energy function to "retrieve" the fixed point state stored at the bottom of that basin. There has been a resurgence of interest in HNs in the past few years thanks to novel results pertaining to their memory storage capacity, relationship to transformer's attention, and possible ways of integrating these ideas in a wide variety of deep learning architectures [3, 4, 5, 6, 7, 8].

Hopfield Networks are conventionally seen as shallow, two-layer systems, e.g., [9, 6, 7, 10]. However, this restriction of shallowness is one imposed by historical usage and not one of the energy-based equations themselves. It turns out that the same energy rules that govern the simple two-layer system generalize to HNs composed of any number of layers and synapses, an architecture called

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.




```
1 layers = [
2 0 IdentityLayer((3,32,32)),
3 1 ReluLayer((2000,)),
4 2 TanhLayer((10,)),
5 3 SoftmaxLayer((1000,)),
6 ]
7 synapses = [
8 0 ConvSynapse(kernel_size=(3,3)),
9 1 SelfAttentionSynapse(num_heads=4),
10 2 DenseSynapse()
11 ]
12 connections = [
13   (( 0 , 1 ), 0 ),
14   (( 1 ,), 1 ),
15   (( 1 , 2 , 3 ), 2 )
16 ]
17 ham = HAM(layers, synapses, connections)
```

$$E_{\text{HAM}} = \sum_{\text{layer}} E + \sum_{\text{synapse}} E$$

Figure 1: **Left:** the fundamental building blocks of our abstraction. Each *Layer* is given a Lagrangian function $\mathcal{L}$ that fully defines both its activations g and energy. Each *Synapse* has energy dependent only on a learnable alignment function that converts the activations of connected layers into a scalar. **Center:** an example HAM composed of four layers and three synapses. The system's energy function is the sum of the energies of its components. Layer states are recurrent in time. We use 0-based indexing to align the graphical representation with the matching Python code. **Right:** pseudocode illustrating how illustrating how a few lines of code can build a HAM. Each layer (lines 2-5) is assigned a Lagrangian (denoted by its familiar activation names e.g., identity, relu, tanh, softmax) and shape. Synapses (lines 8-10) are modeled after common network operations (e.g., convolution, attention, dense), which are all implemented in HAMUX and whose parameter shapes are fully defined by the connected layers. The hypergraph definition (lines 13-15) assembles the graph, one line per synapse.

the **Hierarchical Associative Memory** (HAM) [11]. The conventional HN is then a special limiting case of the HAM with two layers and one synapse.

The generalizability of the HN to the HAM is not common knowledge to the research community, and as such the behavior of the HAM as a "Hierarchical Hopfield Network" has not been well characterized or understood. In this work we propose a powerful abstraction and accompanying software framework called HAMUX that removes critical barriers that stand in the way of applying these networks at scale:

Barrier 1: There is no standardized terminology for the energy fundamentals of a HAM. Our framework proposes an abstraction that fully captures the behavior of any HN while being *modular* (i.e., it is easy to assemble deep HAMs connected to any number of signals), *generalizable* (i.e., it is possible to quickly propose novel operations and activation functions that satisfy energy constraints), and reminiscent of the *biological* inspiration for the original HN.

Barrier 2: The energy of a HAM grows increasingly complex with the number of components and connections. Our framework uses modern AI tooling (JAX [12]) to automatically calculate the gradient of the energy function for any given state and parameters, removing the need to manually derive complicated update rules.

Barrier 3: It can be challenging to conceptualize HAM architectures as modern machine learning pipelines. It requires the shift of perspective from the standard "classification" setup (take an input, pass it through the network, and return an output), to an "association" setup, where labels are just another attribute of a data point (data and labels are equally important and treated the same). With the examples released with HAMUX, we show how one can apply HAMs to conventional "classification" pipelines while reusing existing tools in the Deep Learning ecosystem to create custom energy components.

Our framework, though still in its infancy, aims to integrate HAMs into modern Deep Learning. All experiments in this paper were conducted using our framework.

2 The Abstraction

Hopfield Networks are recurrent networks whose behavior at time t is completely defined as a function of the neuron states. Our abstraction pivots on understanding how the individual energies of *neuron layers* and *synapses* (i.e., the "energy building blocks" of our abstraction) operate within the constraints of a *hypergraph*. Concretely, our HAM framework with N layers and K synapses is fully defined by a list of all **neuron layers** $\boldsymbol{\mathcal{X}} = \{\mathcal{X}^1, \mathcal{X}^2, \ldots, \mathcal{X}^N\}$, a list of all **synapses** $\boldsymbol{\mathcal{S}} = \{\mathcal{S}^1, \mathcal{S}^2, \ldots, \mathcal{S}^K\}$, and a list of connections specifying the **connection hypergraph** $\boldsymbol{\mathcal{G}} = \{\mathcal{G}^1, \mathcal{G}^2, ..., \mathcal{G}^K\}$. We emphasize that a HAM uses a hypergraph and not a normal graph: i.e., a synapse (edge) can operate on an arbitrary number of layers (nodes). In practice, we represent $\mathcal{G}^\kappa$ as the collection of integers $\big(\{\alpha, \beta, ...\}, \kappa\big)$ to specify that synapse $\mathcal{S}^\kappa$ operates on the activations of layers $\{\mathcal{X}^\alpha, \mathcal{X}^\beta, ...\}$. Figure 1 summarizes our abstraction. The system's total energy is defined as a sum of the energies of its components.

$$E_{\text{total}} = \sum_{\alpha=1}^{N} E^\alpha_{\text{layer}} + \sum_{\kappa=1}^{K} E^\kappa_{\text{synapse}} \tag{1}$$

2.1 Neuron Layers

A neuron layer $\mathcal{X}^\alpha$ is an assembly of D^α neurons each with a *hidden state* x^α_i governed by a *Lagrangian function* $\mathcal{L}^\alpha : \mathbb{R}^{D^\alpha} \mapsto \mathbb{R}$. All neurons share a scalar *time constant* τ^α that governs how quickly x^α_i will evolve in time. Each neuron can additionally have a *resting state* I^α_i that we call the bias in conventional deep learning. The choice of the Lagrangian $\mathcal{L}^\alpha$ fully defines the activation or the *gain function* $g^\alpha : \mathbb{R}^{D^\alpha} \mapsto \mathbb{R}^{D^\alpha}$ for a given neuron layer as $g^\alpha := \frac{\partial \mathcal{L}^\alpha}{\partial x^\alpha}$.

The most important rule of the neuron layer is the following: *the hidden state x^α_i is completely invisible to the rest of the network at any point in time*. The ONLY way a neuron can influence the rest of the network is via its activation g^α_i (i.e., all synapses must operate on activations).

The energy of neuron layer $\mathcal{X}^\alpha$ can be computed from its state x^α as follows:

$$E^\alpha_{\text{layer}} = \sum_i (x^\alpha_i - I^\alpha_i) g^\alpha_i - \mathcal{L}_\alpha(x^\alpha) \tag{2}$$

In practice, x^α_i need not be a scalar. For example, a layer of shape $\mathbb{R}^{D^\alpha \times H \times W}$ has D^α neurons whose states x^α_i and activations g^α_i are image patches in $\mathbb{R}^{H \times W}$.

2.2 Synapses

A *synapse* $\mathcal{S}^\kappa$ is a parameterized *alignment function* $\mathcal{F}^\kappa$ that transforms the activations $\{g^\alpha, g^\beta, \ldots\}$ of one or more layers $\{\mathcal{X}^\alpha, \mathcal{X}^\beta, \ldots\}$ into a meaningful scalar that represents the alignment of those layers. This is a novel construction of our abstraction as no previous work has considered anything beyond pairwise synapses. A synapse's energy is defined as the negative of its alignment function.

$$E^\kappa_{\text{synapse}} = -\mathcal{F}^\kappa, \quad \text{where} \quad F^\kappa(g^\alpha, g^\beta, \ldots) \mapsto \mathbb{R}. \tag{3}$$

2.3 The Update Rule

The update rule for x^α_i is the direct consequence of differentiating the total energy in Eq. 1. Incoming signals $\frac{\partial \mathcal{F}^\kappa}{\partial g^\alpha_i}$ into x^α_i can only come from connected synapses. See the derivations in [7, 11].

$$\tau^\alpha \frac{dx^\alpha_i}{dt} = -\frac{\partial E_{\text{total}}}{\partial g^\alpha_i} = \sum_{\kappa=1}^{K} \frac{\partial \mathcal{F}^\kappa}{\partial g^\alpha_i} + I^\alpha_i - x^\alpha_i \tag{4}$$

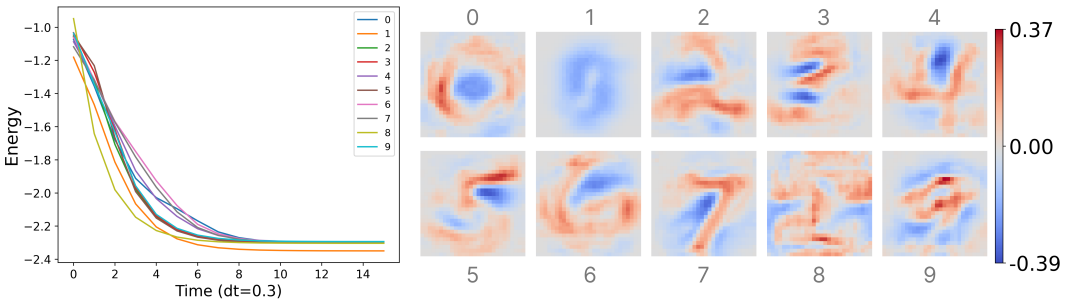


Figure 2: A HAM was trained on the MNIST classification task. After training, the image neurons of the trained HAM were initialized at random while the label neurons were clamped to one-hot encodings for each digit. The image neurons were then allowed to update their states so that the energy decreases in time (left) to the fixed point of the dynamics (in our experiments after 15 time steps). These fixed points are shown on the right as the $\tanh(\cdot)$ activations of the image layer. We use the HN *(SoftMax)* model from Table 1.

Table 1: Classification results reported as the best top-1 accuracy on the validation set at 100 (600) epochs.

Model	**Top-1 Val %**		**# of Params (M)**	
	MNIST	CIFAR10	MNIST	CIFAR10
HN (*ReLU*)	98.74 (99.03)	58.82 (66.23)	0.79	18.49
HN (*SoftMax*)	97.32 (97.99)	53.17 (58.89)	0.79	18.49
Conv HAM (*max pool*)	99.29 (99.51)	82.08 (86.39)	0.60	1.78

3 Classification and Observing the Dynamics

The problem of classification can be formulated in a HAM as an association problem: what one-hot encoded label is associated with a given collection of pixels? In this section we train several different architectures using HAMUX: the traditional HN in both the classical and modern paradigm where we concatenate flattened pixels to labels; and a novel HAM where we stack convolutional and pooling operations. All architectures behave under the global energy function and we do not use additional encoders, decoders, or classification heads. See Appendix A for technical details and discussion. The accuracy of our HAMs, shown in Table 1, stands in line with the accuracy of corresponding feedforward models on similar tasks.

Every system built with HAMUX is an associative memory. This means that training a HAM to "classify" images in one direction allows us to utilize the system in reverse — we can retrieve the memory most associated with a label by clamping the labels to a desired value over the dynamics. See Figure 2 for energy dynamics of a trained model with different clamped MNIST labels and Appendix B for details.

4 Conclusion

In this work, we have proposed a universal abstraction to describe the energy of general Hopfield Networks. Our abstraction proves particularly powerful for network design, allowing anyone to modularly construct deep HNs (HAMs) that can be applied to traditional machine learning tasks. At the same time, our abstraction generalizes the fundamental operations of synapses, giving us freedom to implement energy-constrained versions of convolutions, pooling, and even attention. We package our abstraction into a software framework called HAMUX that makes it trivial to implement complex HAMs. We believe that HAMUX gives HNs the representational power of modern Deep Learning by providing a framework that integrates modern AI tooling and advances into a regime that is governed by energy and whose cardinal function is association rather than prediction.

References

[1] J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, April 1982.

[2] J. J. Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences of the United States of America*, 81(10):3088–3092, May 1984.

[3] Dmitry Krotov and John J Hopfield. Dense associative memory for pattern recognition. *Advances in neural information processing systems*, 29, 2016.

[4] Mete Demircigil, Judith Heusel, Matthias Löwe, Sven Upgang, and Franck Vermet. On a Model of Associative Memory with Huge Storage Capacity. *Journal of Statistical Physics*, 168(2):288–299, July 2017.

[5] Dmitry Krotov and John Hopfield. Dense Associative Memory Is Robust to Adversarial Inputs. *Neural Computation*, 30(12):3151–3167, December 2018.

[6] Hubert Ramsauer, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Thomas Adler, Lukas Gruber, Markus Holzleitner, Milena Pavlović, Geir Kjetil Sandve, et al. Hopfield networks is all you need. *arXiv preprint arXiv:2008.02217*, 2020.

[7] Dmitry Krotov and John J Hopfield. Large associative memory problem in neurobiology and machine learning. In *International Conference on Learning Representations*, 2020.

[8] Michael Widrich, Bernhard Schäfl, Milena Pavlović, Hubert Ramsauer, Lukas Gruber, Markus Holzleitner, Johannes Brandstetter, Geir Kjetil Sandve, Victor Greiff, Sepp Hochreiter, et al. Modern hopfield networks and attention for immune repertoire classification. *Advances in Neural Information Processing Systems*, 33:18832–18845, 2020.

[9] Qun Liu and Supratik Mukhopadhyay. Unsupervised learning using pretrained cnn and associative memory bank. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 01–08. IEEE, 2018.

[10] Beren Millidge, Tommaso Salvatori, Yuhang Song, Thomas Lukasiewicz, and Rafal Bogacz. Universal Hopfield Networks: A General Framework for Single-Shot Associative Memory Models, June 2022.

[11] Dmitry Krotov. Hierarchical associative memory. *arXiv preprint arXiv:2107.06446*, 2021.

[12] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: Composable transformations of Python+NumPy programs, 2018.

[13] Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens, Hamza Merzic, Vladimir Mikulik, Tamara Norman, John Quan, George Papamakarios, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Luyu Wang, Wojciech Stokowiec, and Fabio Viola. The DeepMind JAX Ecosystem, 2020.

[14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] Ross Wightman. Pytorch image models. `https://github.com/rwightman/pytorch-image-models`, 2019.

[16] Maxence Ernoult, Julie Grollier, Damien Querlioz, Yoshua Bengio, and Benjamin Scellier. Updates of equilibrium prop match gradients of backprop through time in an rnn with static input. *Advances in neural information processing systems*, 32, 2019.

[17] Cristian Garcia. Treex, 2021.

Table 2: TIMM Data Augmentation configuration for the experiments

	MNIST	CIFAR
re_prob	0.1	0.2
hflip	0.0	0.5
vflip	0.0	0.0
scale	(0.9, 1.0)	(0.2, 1.0)
color_jitter	0.4	0.5
auto_augment	None	None
ratio	(0.75., 1.33)	(0.75, 1.33)

A Classification: Training and Architectures

We now provide the technical details for the variety of architectures that were implemented using HAMUX in section 3. All models were trained for 600 epochs using the ADAM optimizer with weight decay (using optax defaults: b1=0.9, b2=0.99, weight decay=1e-4 [13]) at a constant learning rate of 0.001 and batch size of 400. For consistency, we do not optimize hyperparameters for any particular architecture. We train the networks using simple backpropagation through time using the ADAM optimizer [14]. We load MNIST and CIFAR images with minor data augmentations from the popular `timm` [15] library. All experiments were run on an A100 GPU.

We provide example code for MNIST architectures; CIFAR10 versions of the architecture are almost identical, though they require different layer shapes for all layers related to images and patches. Depending on the depth of the HAM, we vary the number of recurrent timesteps along with our step size `dt` through time. This is to ensure that pixel information has enough time to propagate to a potentially distant label layer. All architectures and their training scripts will be released as demo models along with the framework.

A.1 Shallow Hopfield Network Configurations

We begin by implementing two different HNs where a visible layer, the concatenation of vectorized pixels and one-hot labels, is connected to a single hidden layer via a synaptic weight matrix. The difference in these two architectures is in the choice of activation function. The relu activation function defines the model as a continuous, recurrently applied Classical Hopfield Network (CHN) [2], whereas $\mathrm{softmax}$ activation function defines a Dense Associative Memory (DAM), also known as the Modern Hopfield Network (MHN) [3, 4, 6, 7]. The latter has been shown to have a higher storage capacity than the former; however, it is our experience in writing this paper that they are also harder to train using backpropagation-through-time. For this reason, in our HN *(Softmax)* architecture we additionally normalize each *memory* in the weights to have L2-norm equal to 1. Specifically, given our synaptic matrix connecting 1000 hidden units to 794 visible units (784 pixels + 10 labels), we enforce that the matrix consists of 1000 unit vectors each of dimension 794. This promotes diversifying of the weights during training and stability of the dynamics during longer inference runs.

To implement this architecture using HAMUX we define an image layer with the $\tanh$ activation function that is connected to a label layer with the $\mathrm{softmax}$ activation function. Functionally, the states of these two layers are concatenated together and their union is referred to as a single visible layer. The second "layer" of this two-layer HN is hidden inside the synapse (`DenseMatrixSynapseWithHiddenLayer`). So why do we put this layer within the synapse? In our abstraction, the states of each layer must be propagated through time — that is, a layer's state $x^\alpha(t)$ should depend on no other layer states at the same time t. The original definition of the HN defines the state of the hidden layer at time t as a function of the state of the visible layer at time t (which is equivalent to taking the limit as τ of this layer approaches 0). By including the Lagrangian function as a hidden layer within a synapse we propose a workaround that is mathematically consistent with the original definition of the HN (that is, states are independent of other states at time t).

```
layers = [
    TanhLayer((28,28,1)),
    SoftmaxLayer((10,)),
]
synapses = [
    DenseMatrixSynapseWithHiddenLayer(1000, hidden_lagrangian=LRelu()),
]
connections = [
    ((0, 1), 0),
]
ham = HAM(layers, synapses, connections)
```

Code 1: Shallow HN with ReLU, MNIST

A.2 Convolutional and Pooling HAMs

Dense matrix operations are only one kind of alignment function that describe layer-layer relationships. We introduce an architecture that additionally uses convolutions and pooling. We must first calculate the shape of each layer that serves as the output of a convolution or pooling operation. With this we can describe our 5-layer HAM as follows (layer 5 is hidden within our last synapse):

```
layers = [
    TanhLayer((28,28,1), tau=1.0),
    TanhLayer((7,7,64), tau=1.0),
    TanhLayer((2,2,128), tau=1.0),
    SoftmaxLayer((10,), tau=1.0),
]
synapses = [
    ConvSynapseWithPool(
        (4, 4),
        strides=(2, 2),
        padding=(2, 2),
        pool_window=(2, 2),
        pool_stride=(2, 2),
        pool_type="max",
    ),
    ConvSynapseWithPool(
        (3, 3),
        strides=(1, 1),
        padding=(0, 0),
        pool_window=(2, 2),
        pool_stride=(2, 2),
        pool_type="max",
    ),
    DenseMatrixSynapseWithHiddenLayer(1000, hidden_lagrangian=LRelu()),
]
connections = [
    ((0, 1), 0),
    ((1, 2), 1),
    ((2, 3), 2)
]
```

Code 2: Convolutional HAM with MaxPooling, MNIST

A.2.1 On the consequences of Max-Pooling

We make several architectural choices in this paper, not all of which have the most natural correspondence to energy. In particular, consider the max pooling operation which is implemented as part of our convolutional synapse. For each patch on which this operation is applied it will discard information from every element but one. This makes it unclear what the signal should be in the gradient of that synapse's energy for all non-maximum elements in the patch. The JAX autograd system (and hence HAMUX) uses the definition of max-pooling consistent with that proposed by [16] where the gradients of all non-maximum elements of the patch are zero.

If we consider the operation of max-pooling to be a competitive operation within a single neuron where the "winner takes all," we can easily consider a softer version of the maximum on each patch, e.g., "softmax pooling." We leave this for future work.

A.3 Example forward pass

Our HAMs are all fully dynamic systems through time, so it can be unnatural to consider them as prediction engines. Here we provide the forward pass that we used for our classification pipelines.

```python
import jax.numpy as jnp
import jax.tree_util as jtu

def simple_fwd(ham, x, depth, dt):
    """A simple version of the forward function for classification.

    Image layers are `layer[0]` and labels are `layer[-1]`
    """
    # Initialize hidden states given our data
    xs = ham.init_states(x.shape[0])
    xs[0] = jnp.array(x)

    # Masks allow us to clamp our visible data over time
    masks = jtu.tree_map(lambda x: jnp.ones_like(x), xs)
    masks[0] = jnp.zeros_like(masks[0])

    for i in range(depth):
        # Calculate update directions
        updates = ham.vupdates(xs)

        # Simple step down the energy function
        xs = ham.step(
            xs, updates, dt=dt, masks=masks
        )

    # Label layer has the softmax activation function
    # Use this to return probabilities
    return ham.layers[-1].g(xs[-1])
```

Code 3: An example of the forward function for classification

B Observing Dynamics

Using our trained HN-Softmax model from our classification experiments, we clamp the labels of our system to a desired value and let the dynamics of the system evolve around these signals. We display the images as our HAM sees them: as the $\tanh$ activations of our pixels (hence the negative values in Figure 2). To extract sharper memories (i.e., to prevent our system from choosing an incoherent superposition of memories at the limit of the dynamics), we decrease the temperature of the Softmax in our hidden layer by a factor of 10.

C About HAMUX: Software Details

The HAMUX software used for this paper is undergoing rapid API and tooling changes in an effort to incorporate more of the expected functionality present in a Machine Learning library. At the time of this writing, all components in HAMUX have been built using the excellent but unpopular `Treex` library [17], which we found to be the most robust and simple ML framework for building the unique constraints required HAMUX components. As part of the JAX ecosystem, any traditional tools for working with deep networks in JAX also work with HAMUX (e.g., it would be easy to consider alternative optimization procedures implemented in `Optax` [13] to descend the energy function). The software will be released prior to this workshop date.

D Examples of Lagrangian Functions

The choice of the Lagrangian function L_α for layer $\mathcal{X}^\alpha$ fully defines the activation or the *gain function* of that layer $g^\alpha : \mathbb{R}^{D_\alpha} \mapsto \mathbb{R}^{D_\alpha}$ as $g^\alpha := \frac{\partial \mathcal{L}_\alpha}{\partial x^\alpha}$ for a given neuron layer. These functions introduce non-linearites into our HAM. Sometimes these functions operate elementwise (e.g., ReLU, GeLU) whereas other activation functions include normalization effects that scale an individual

neuron's state x_i^α given the states of other neurons x^α in the same layer (e.g., SoftMax, LayerNorm). For behaved dynamics, we must choose a Lagrangian that is both convex and differentiable.

The following common activation functions in use today have easy parallels in the Lagrangian world.

ReLU The Lagrangian of the ReLU is

$$\mathcal{L} = \frac{1}{2} \sum_i \max(x_i, 0)^2,$$

where

$$g_i = \frac{\partial \mathcal{L}}{\partial x_i} = \max(x_i, 0).$$

SoftMax The Lagrangian of the SoftMax is also known as the LogSumExp. It proves beneficial to consider the case of a softmax with a (potentially learnable) inverse temperature scalar β (e.g., Transformer attention defaults to $\frac{1}{\sqrt{D_{\text{key}}}}$). The Lagrangian of this operation can be expressed as

$$\mathcal{L} = \frac{1}{\beta} \log \sum_i \exp\left(\beta x_i\right),$$

where

$$g_i = \frac{\partial \mathcal{L}}{\partial x_i} = \frac{\exp(\beta x_i)}{\sum\limits_j \exp(\beta x_j)}.$$

Identity The identity activation occurs with the Lagrangian

$$\mathcal{L} = \frac{1}{2} \sum_i x_i^2,$$

where

$$g_i = \frac{\partial \mathcal{L}}{\partial x_i} = x_i.$$

LayerNorm The LayerNorm activation modifies each neuron layer to have mean 0 and standard deviation 1, while optionally learning a scale γ on the standard deviation and a shift δ to the mean $\bar{x} = \frac{1}{D} \sum\limits_{k=1}^{D} x_k$. The corresponding Lagrangian is

$$L = D\gamma \sqrt{\frac{1}{D} \sum_j \left(x_j - \bar{x}\right)^2 + \varepsilon} \; + \; \sum_j \delta_j x_j,$$

where

$$g_i = \gamma \frac{x_i - \bar{x}}{\sqrt{\frac{1}{D} \sum\limits_j \left(x_j - \bar{x}\right)^2 + \varepsilon}} + \delta_i.$$