Combining high-performance hardware, cloud computing, and deep learning frameworks to accelerate physical simulations: probing the Hopfield network

Vaibhav S. Vavilala*

Department of Computer Science, Columbia University, New York, NY 10027

Abstract

The synthesis of high-performance computing (particularly graphics processing units), cloud computing services (like Google Colab), and high-level deep learning frameworks (such as PyTorch) has powered the burgeoning field of artificial intelligence. While these technologies are popular in the computer science discipline, the physics community is less aware of how such innovations, freely available online, can improve research and education. In this tutorial, we take the Hopfield network as an example to show how the confluence of these fields can dramatically accelerate physics-based computer simulations and remove technical barriers in implementing such programs, thereby making physics experimentation and education faster and more accessible. To do so, we introduce the cloud, the GPU, and AI frameworks that can be easily repurposed for physics simulation. We then introduce the Hopfield network and explain how to produce large-scale simulations and visualizations for free in the cloud with very little code (fully self-contained in the text). Finally, we suggest programming exercises throughout the paper, geared towards advanced undergraduate students studying physics, biophysics, or computer science.

I. INTRODUCTION

Computation is critical to calculate physical properties of modeled systems. Some experiments are impossible to perform but can be simulated if the theory known and sufficient computational tools are available. Concepts that feel abstract to students can become tangible when simulated on a computer. Although computation is a component of most undergraduate physics curricula, it is commonly under-emphasized, and educators must keep pace with the rapidly improving technology to produce the most capable students.[1] In this paper, we discuss innovations that have improved accessibility to numerical computations (via the cloud), and accelerated the performance of large-scale simulations (via the graphics processing unit).

Frequently the bottleneck of numerical computations involves matrix operations. Such examples include matrix diagonalization to solve the Schrödinger Equation and solving systems of linear or differential equations to perform Finite Element Analysis. These computations are generally of $O(N^3)$ complexity (i.e. scaling the dimensions of the input matrix by $10\times$ results in a $1000\times$ performance hit). In the absence of high-performance hardware, most numerical computations cannot scale well, which slows if not entirely precluding simulations beyond a certain size. To perform simulations, students commonly use scripting languages such as MATLAB or Python in the classroom and laboratory. Using a personal machine involves an up-front cost of programming environment setup, and the speed of simulations is limited by the hardware (typically two or four CPUs). Workstations with high-performance hardware are available only in the most well-funded labs. To help address limitations in the computational aspects of current physics curricula, we show that the software and hardware innovations that have accompanied the rise of deep learning can dramatically enhance the runtime performance and accessibility of computational physics. We start with a brief introduction to deep learning.

The past decade has seen a meteoric rise in artificial intelligence research, and the bulk of this progress is from deep learning - which aims to solve complex problems by constructing computational models that learn from a high volume of data.[2] For example, deep learning models can classify thousands of objects in images with very little error, synthesize speech from text, and detect anomalies in financial transactions to stop fraud. Although the key algorithm for training deep learning models - backpropagation - has been known since 1986,[3]

it was only in the past decade or so that researchers recognized the value of collecting vast amounts of data, and corporations such as NVIDIA created hardware that could feasibly train large-scale deep learning models via the graphics processing unit (GPU).[4] From personal computing to industrial data centers, most computers hold between 2 - 96 CPU cores. A single GPU has embedded within it thousands of processing cores - which, although each GPU core individually is slower than a CPU core, the sheer quantity of cores in a GPU enables it to dwarf the performance of multiple CPU cores in tasks that can be parallelized. For example, when performing matrix multiplication, each element of the product can be computed independently and in parallel. Hence for large matrices, matrix multiplication can be performed significantly faster on a GPU than on a CPU. In fact, several problems can benefit from the GPU such as backpropagation (in which gradients of millions of variables can be computed in parallel), rendering[5] (in which thousands of pixel colors on a screen can be computed in parallel), and the simulation of natural phenomena like fluids,[6] cloth,[7] & hair[8] (which are commonly GPU-accelerated in the film and games industries). A typical simulation will involve the GPU(s) and CPU(s) working together, whereby general-purpose instructions (like loading data or plotting graphs) run on the CPU, and parallelizable, compute-intensive parts of the application (like matrix calculations) run on the GPU.

While the GPU enjoys several use-cases, initially only researchers with a computer science background could access its benefits, as strong knowledge of a low-level programming language like CUDA would be required. With the open-sourcing of deep learning frameworks such as PyTorch[9] and TensorFlow,[10] physicists no longer face such programming language barriers. These frameworks allow developers to code in a high-level scripting language (usually Python) and abstract away low-level CUDA function calls. The mathematical underpinnings of most physics simulations including sampling, linear algebra, numerical optimization, signal processing, and differentiation are implemented in these frameworks in an aggressively optimized manner, using MATLAB/NumPy-like syntax that is intuitive to non-programmers. Additionally, these frameworks are supported by extensive function manuals and numerous high-quality courses such as fast.ai. The physicist, as a result, reaps the benefits of the rapidly improving technology and can remain focused on the scientific aspects of the simulation, instead of implementation details.

Furthermore, the advantage of using frameworks that garner strong adoption like PyTorch

and TensorFlow is that with greater usage, bugs are caught sooner, and robust documentation & support are further justified. Crucially, with more users, online programming help communities like StackOverflow become filled with questions & solutions to common (and uncommon) programming errors, aiding physics students and researchers in rapidly fixing their own bugs by benefitting from crowdsourced knowledge.

Yet obstacles remain - GPUs cost hundreds to thousands of dollars, often impractical for students and schools in developing countries. Further, installing an optimized deep learning framework with GPU support is non-trivial and hardware-specific, often requiring several days to complete even for experts. Although most personal computers on sale today ship with a GPU, they are typically limited in memory, and the user still runs into environment setup challenges. Cloud computing solves these problems. With high-end GPUs and pre-built deep learning environments available over the Internet, anyone can write simulations on their personal computer, remotely execute their code on a cloud-based machine, and visualize the results in real-time.

In the remainder of this tutorial, we first show how to set up a GPU instance in the cloud, pre-loaded with PyTorch. We then introduce the foundations of the Hopfield neural network (HNN) - including its theoretical roots in condensed matter physics and its applications in AI. From there, we guide the reader through simulating the HNN on the GPU with few lines of code, showing the drastic improvement in performance as compared with the CPU. We conclude with suggested programming exercises to reproduce famous results of the HNN.

II. COLAB & PYTORCH ENVIRONMENT SETUP

For this article, we take Google Colab[11] as our (currently free) cloud provider of choice, although Kaggle, Azure Notebooks, Paperspace Gradient, and Amazon Sagemaker are among the alternatives we are aware of. We use the PyTorch deep learning library here, and note that TensorFlow is an outstanding alternative. CuPy[12] is a package for purely numerical computing that is also worth considering. Our hardware of choice are NVIDIA GPUs, as these chips benefit from the strongest support in the deep learning community. We note that certain matrix operations can be further accelerated with Tensor Processing Units[13] (TPUs), freely available on Colab, but such hardware is outside the scope of this paper.

To access Colab, navigate to https://colab.research.google.com. Sign in with a Google

account, and then click *connect* to request an instance. To obtain a GPU, click *Runtime* → *Change Instance Type* → *GPU* for Hardware Accelerator. In our experience, GPUs are available immediately upon request. The user then has access to a pre-built Python environment with PyTorch, TensorFlow, and common scientific computing packages like NumPy, SciPy, and Matplotlib pre-installed. Additional packages from GitHub or the Python Package Index can easily be installed in-browser to augment the pre-built environment. For example, a biophysics student studying protein interactions may wish to use the pypdb package. Installing this package is as simple as `!pip install pypdb`. Within Colab, the user can write Python code in-browser using a robust Integrated Development Environment (IDE) with tab completion and code formatting. All code is automatically backed-up in Google Drive, which allows for easy code sharing/collaboration and the ability to work from any machine connected to the Internet. We note that although the details of how to access a particular cloud provider will change over time and based on which service provider is used, the concept of leveraging this trifecta of technologies (HPC, Cloud, AI) will become increasingly important in performing computer simulations in the years to come. We further note that the present article focuses on using GPU-accelerated numerical libraries contained in deep learning frameworks, as opposed to using deep learning itself in physics research. Although AI algorithms have started to enjoy a symbiotic relationship with data-heavy physical simulations,[14–16] such applications are beyond the scope of this paper.

We can verify the environment setup with the following example. In Code Sample 1, we first import the PyTorch and timing packages. We then construct a large random matrix $A_{16000 \times 16000}$ sampled from a uniform distribution $A_{ij} \in [0, 1]$ on the specified hardware. Note that the `device` variable in line 3 defines whether a tensor should be stored on the GPU (`device="cuda"`) or CPU (`device="cpu"`). We finally compute the matrix inverse in line 5:

Code Sample 1: Test PyTorch environment (matrix inverse)

```
1  import torch, time
2  start_time = time.time() #Begin timer
3  device='cuda' #'cpu' for CPU
4  A=torch.rand(16000,16000, device=device)
5  iA = A.inverse()
6  print('execution time: ' + str(time.time() - start_time))
```

The code executes in 2.98 seconds on the GPU and 53.4 seconds on the CPU. To show the ease of performing matrix multiplication, we take an example use-case where a modeled linear system has more unknowns than observations, and a least-squares solution is desired. We construct a large random matrix $A_{8000 \times 10000}$ sampled from a uniform distribution $A_{ij} \in [0, 1]$ on the specified hardware (GPU or CPU), and initialize a random vector $b_{8000 \times 1}$. To solve the underdetermined linear system $Ax = b$, we use the Moore-Penrose pseudoinverse in line 13, $x = (A^\top A)^{-1} A^\top b$.

```
Code Sample 2: Test PyTorch environment (pseudoinverse)
7   import torch, time
8   start_time = time.time() #Begin timer
9   device='cuda' #'cpu' for CPU
10  A=torch.rand(8000,10000, device=device)
11  b=torch.rand(8000,1,device=device)
12  A_t = torch.t(A) #precompute matrix transpose
13  x=torch.matmul(torch.matmul(A_t,A).inverse(),torch.matmul(A_t,b))
14  print('execution time: ' + str(time.time() - start_time))
```

On the GPU, the code executes in 1.46 seconds. On the CPU, it requires 45.1 seconds. We encourage the reader to reproduce our timings, and observe that as matrix inversion and matrix multiplication are of $O(N^3)$ complexity, the timings scale as such. These brief examples show the order of magnitude superior performance of the GPU in scientific computing, and its ease of access. As an exercise, generate a random square matrix sampled from a Gaussian distribution and diagonalize the matrix. Plot how the performance scales with the input size, and compare the CPU vs. GPU timings. You can use the PyTorch documentation to obtain the syntax and the Matplotlib package for visualization.

To illustrate the value of the GPU in studying physical systems, we take the Hopfield network as a specific example. In the next section, we introduce the theory behind the HNN.

III.   THE HOPFIELD NETWORK

The Hopfield neural network is a two-state information processing model first described in 1982.[17] The dynamical system exhibits numerous physical properties relevant to the study of spin glasses (disordered magnets),[18] biological neural networks,[19] and computer science (including circuit design,[20] the traveling salesman problem,[21] image segmentation,[22] and character recognition[23]).
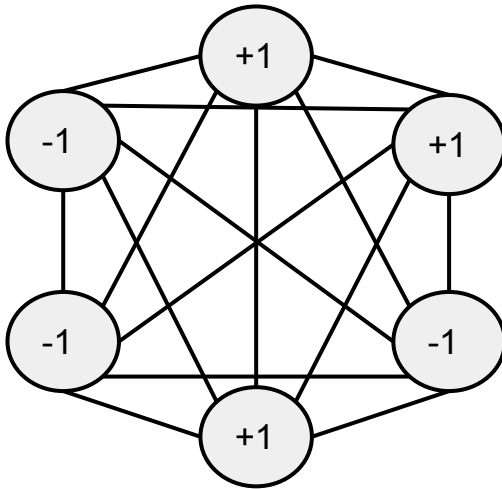

FIG. 1: A fully-connected Hopfield neural network with $N = 6$ and instantaneous state $S = \{+1, +1, -1, +1, -1, -1\}$, starting from the top and indexing clockwise. The synaptic weights are stored in the 6×6 matrix J_{ij}.

The network consists of N fully-connected neurons that can store N-tuples of ± 1's (shown in Fig. 1). Each pair of neurons has an associated weight stored in a symmetric synaptic weight matrix J_{ij} with $i, j \in 1...N$. The binary state of a neuron represented by S_i is mapped onto a classical Ising spin,[24] where $S_i = +1$ (-1) represents a neuron that is firing (at rest). In the binary representation, such a neuron fires when its potential exceeds a threshold U_i that is independent of the state S_i of the neuron. The state of the neuron at time $t+1$ is determined solely by the sum total of post-synaptic potential contributions from all other neurons at time t. This assumption, where the time evolution of a neural state

is determined by the local field produced by other neurons, allows us to associate a classical Hamiltonian[24] (or energy functional) consistent with the discrete, asynchronous time evolution of the neural network. The neural network is, thus, mapped onto an Ising model with long-ranged, generically frustrated interaction. Spin glass approaches have been highly fruitful in investigating the properties of such Hopfield neural networks near criticality.[18] In particular, they have quantified the critical memory loading $\alpha_c \sim 0.144$, such that when p patterns are imprinted on a network with N neurons, the network has no faithful retrieval for $\alpha = p/N > \alpha_c$ whereas when $\alpha < \alpha_c$, the memory retrieval is accompanied by a small fraction of error (for example, no more than 1.5% of bits flipped).[18]

We consider a network of N two-state neurons $S_i = \pm 1$ trained with $p = \alpha N$ random patterns ξ_k^μ where $\mu = 1, \ldots, p$, $k = 1, \ldots, N$, and $N \gg 1$. The symmetric synaptic matrix is created from the p quenched patterns,

$$J_{ij} = \frac{1}{N} \sum_{\mu=1}^{p} \xi_i^\mu \xi_j^\mu = J_{ji} \qquad i \neq j. \tag{1}$$

With symmetric neural interconnections, a stable state is ultimately reached.[25] Further, we note that the neural interconnections are considered fixed after training, and $J_{ii} = 0$ in the traditional interpretation. However, previous work has shown that the hysteretic (or self-interaction) terms enhance retrieval quality, especially in the presence of stochastic noise.[26–29] Hysteresis is a property found in biological neurons (via a refractory period after a neuron fires) and is inherent in many physical, engineering, and economic systems. Thus, in this paper we set $J_{ii} = \lambda\alpha$ to probe the effects of self-action, with $\lambda = 0$ representing the traditional model.

We compute the simulated memory capacity by flipping some fraction of bits less than

a Hamming distance $N/2$ away from each imprinted pattern, time evolving the corrupted probe vectors through the network until some convergence criterion, and then measuring the quality of recall. The Hamming distance is defined as the number of different bits between two patterns. The zero-temperature network dynamics are as follows:

$$S_i(t+1) = \text{sign}\left[\sum_{j\neq i}^{N} J_{ij}S_j(t)\right].$$ (2)

The network evolving under this deterministic update rule behaves as a thermodynamical system in such a way as to minimize an overall energy measure defined over the whole network.[30] These low energy states are called attractor states. When $\alpha < \alpha_c$, the imprinted patterns are the attractors. Above criticality, non-imprinted local minima, called spurious memories, also become dynamically stable states. The basin of attraction is defined as the maximal fraction of bits that can be flipped such that the probe vector still relaxes to its intended imprint within a small fraction of error.[19] At low memory loading, the basin of attraction of imprinted patterns is very high, near $N/2$, and beyond criticality α_c, the basin of attraction vanishes. We let m_0 be the normalized dot-product between an imprinted pattern and its corrupted probe vector, and let m_f represent the final overlap between a time-evolved probe vector and its intended imprint. Therefore, an imprint with $0.1N$ bits flipped would have an overlap of $m_0 = 0.8$. We note that asynchronous update refers to each neuron updating serially (in random order, per time step), and synchronous update refers to all neurons updated at once per time step.

Despite the simplicity of the Hopfield network, considerable computational power is inherent in the system. We find numerous interesting properties like Hebbian learning, associative recall (whereby similar to the human brain, whole memories can be recovered from parts of

them), and robustness to thermal and synaptic noise.[19]

In the next section, we demonstrate how to simulate the Hopfield network on the GPU with a large system size $(N = 32K)$ and very few lines of code.

IV. IMPLEMENTING SIMULATIONS ON THE GPU

Here we show with code how to implement a Hopfield network simulation. After setting up a PyTorch environment in Google Colab as specified in the Introduction, we start by defining functions to construct the set of imprinted memories, the synaptic weight matrix (as defined in eq. 1), and probe vectors perturbed from the imprints.

Code Sample 3: Initialize synaptic weight matrix

```
15  def initSynapticMatrix(N, V, L):
16      """Constructs the synaptic weight matrix
17
18      Args:
19        N:         number of neurons
20        V:         N x p matrix containing set of imprinted memories
21        L:         the diagonal term
22      Result:
23        J:         the N x N synaptic matrix
24      """
25
26      J=(1.0/N)*torch.matmul(V,torch.t(V))
27      # set the diagonal self-action terms with L
28      J.as_strided([N],[N + 1]).copy_(torch.diag(J)*L)
29      return J
```

Code Sample 4: Perturb probe vectors

```
30  def flipBits(m0, N, p, device, probe):
31      """Randomly negates elements of the input.
32
33      Args:
34        m0:      desired dot product overlap after flipping bits
35        N:       number of neurons
36        p:       number of patterns
37        device:  cpu or gpu
```

```
38        probe:  N x p probe matrix whose elements should be flipped
39     """
40
41     #corrupt (1-m0)*N/2 random bits per pattern
42     nFlip=round((1-m0)*N/2) #How many bits to flip for overlap m0
43     if nFlip > 0:
44         #random sample indices from every vector
45         y=torch.multinomial(torch.ones(p,N,device=device),nFlip)
46         #corresponding index to bit flip
47         r=torch.arange(0,p,1,device=device).expand(nFlip,p)
48         #flip bits in probe array
49         probe[y.reshape(-1),torch.t(r).reshape(-1)]*=-1
```

Code Sample 5: Time evolve network

```
50  def evolveNetwork(N, probe, probe_new, J, V, t_type, dotp_evol, num_conv,
        device):
51      """Time evolves the Hopfield network.
52
53      Args:
54        N:         number of neurons
55        probe:     N x p probe matrix of imprints
56        probe_new: N x p probe matrix of imprints after one time step
57        J:         N x N synapic weight matrix
58        V:         N x p matrix containing set of imprinted memories
59        t_type:    torch.FloatTensor if CPU, torch.cuda.FloatTensor if GPU
60        dotp_evol: max_steps+1 x p matrix storing each pattern's dot product
                 overlap per time step
61        num_conv:  1 x max_steps+1 matrix storing the number of converged
                 states per time step
62        device:    CPU or GPU
63      """
64      #keep track of non-converged pattern indices
65      nidx = torch.arange(0,p,1,device=device)
66      for i in range(1,max_steps+1):
67          probe_new[:,nidx] = torch.matmul(J,probe[:,nidx])
68          #add noise to zero elements and take sign function
69          probe_new[:,nidx] = torch.sign(probe_new[:,nidx]+(probe_new[:,nidx
                 ]==0).type(t_type)*(2*torch.rand(N,len(nidx),device=device)-1))
70          dotp_evol[i,:]=torch.sum(probe_new*V,dim=0)
71          nidx = nidx[torch.sum(probe_new[:,nidx]*probe[:,nidx],dim=0)!=N]
72          num_conv[0,i]=nidx.nelement()
73          if nidx.nelement() == 0:
74            dotp_evol[i+1:,:] = dotp_evol[i,:]
75            num_conv[0,i+1:] = num_conv[0,i]
```

```
76          print('converging early: ' + str(i) + ' tsteps to converge')
77          break
78       probe=probe_new.clone()
```

We then import key packages in line 79 and specify that our code should run on the GPU. The same code can be run on the GPU or CPU by simply switching the `dst` variable between `"cuda"` and `"cpu"` in line 80. In subsequent lines, we define the system parameters including the network size N, memory loading α, initial overlap m_0, and self-interaction term λ.

```
Code Sample 6: Define system parameters
79  import torch, numpy as np, time
80  dst = 'cuda'   #cpu for cpu-only mode
81  t_type=torch.FloatTensor
82  if dst=='cuda':
83      torch.cuda.empty_cache() #frees memory for large matrices
84      t_type = torch.cuda.FloatTensor
85  device = torch.device(dst)
86  start_time = time.time() #Begin timer
87  #System parameters
88  N_list=[1000,4000,16000]
89  alpha=0.24 #memory loading
90  L=1 #diagonal coefficient lambda
91  m0=0.9 #initial overlap
92  max_steps = 100 #max tsteps
93  dotp_lists = [] #track the evolution of the overlap m over time
94  num_conv_lists = [] #track the number of converged states over time
```

Using these parameters, we are ready to construct the set of imprinted memories (line 100), the synaptic weight matrix (line 103, as defined in eq. 1), and probe vectors perturbed from the imprints (line 107). Finally, the probe vectors are repeatedly updated according to eq. 2 until either convergence or the max time steps are reached (line 121).

```
Code Sample 7: Execute simulation
95  for N in N_list:
96      print('Running simulation for N=' + str(N))
97      p=int(alpha*N) #number of imprints
```
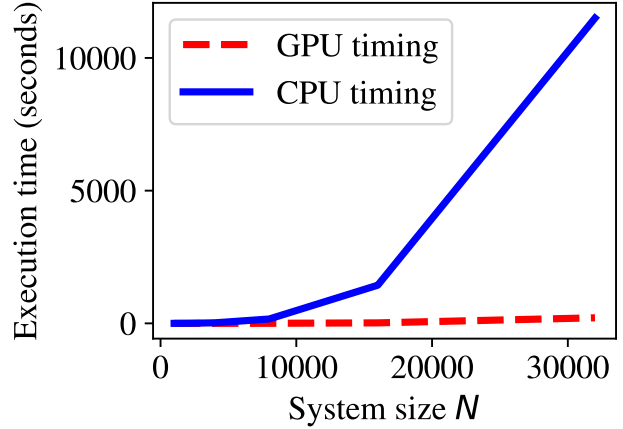
```
98
99        #Construct set of patterns
100       V=2*torch.round(torch.rand(N,p, device=device))-1
101
102       #Construct synaptic matrix including diagonal terms
103       J = initSynapticMatrix(N,V,L)
104
105       #corrupt patterns such that initial overlap is m0
106       probe = V.clone() #Construct a probe matrix
107       flipBits(m0, N, p, device, probe)
108
109       #Initialize variables for storing simulation data
110       probe_new = probe.clone()
111
112       #store time evolution of dot-products
113       dotp_evol = torch.zeros(max_steps+1,p,device=device)
114       dotp_evol[0,:] = torch.sum(probe*V,dim=0) # time t=0
115
116       #keep track of how many states have not converged over time
117       num_conv = torch.zeros(1,max_steps+1,device=device)
118       num_conv[0,0]=p #At time t=0, p states have not converged
119
120       #main neural updating loop: time evolve network until convergence
121       evolveNetwork(N, probe, probe_new, J, V, t_type, dotp_evol, num_conv,
              device)
122
123       #store the dot product evolutions and number of converged states
124       dotp_lists.append((dotp_evol/N).cpu())
125       num_conv_lists.append((p-num_conv.cpu())/p)
126
127 print('execution time: ' + str(time.time() - start_time))
```

With our code complete, we are ready to demonstrate the advantages of using the GPU over CPU. In fig. 2 we compare timings for a simulation with $(\alpha, \lambda, m_0, max\_steps) = (0.12, 0, 0.8, 100)$. For CPU timings, we use two Intel Xeon CPUs @ 2.20GHz. For GPU timings, we use one NVIDIA Tesla T4 GPU with 15GB allocated RAM. In the author's experience, such a GPU has consistently been available in the USA. For large N, the GPU-accelerated simulations execute 50-80 times faster than the CPU mode. As the bottleneck of Hopfield network simulation is matrix multiplication, we observe that the asymptotic

N	GPU (sec)	CPU (sec)
$1K$	6.75×10^{-2}	3.76×10^{-1}
$2K$	1.33×10^{-1}	2.54
$4K$	4.19×10^{-1}	2.07×10^1
$8K$	2.41	1.64×10^2
$16K$	1.86×10^1	1.44×10^3
$32K$	2.09×10^2	$1.15 \times 10^{4}*$

(a) table comparing timings

(b) plot of the timings

FIG. 2: Comparing the performance of Hopfield network simulation on the GPU vs. CPU. For large N, the GPU version executes 50-80 times faster than the CPU. Note that the CPU timing for $N = 32K$ is an estimate based on cubic scaling and not actually simulated.

complexity scales $O(N^3)$ with the input size.

V.   VISUALIZATIONS & EXTENSIONS

We now execute large-scale simulations and plot the results in figs. 3 and 4. In fig. 3, we visualize the network with zero self-coupling terms ($\lambda = 0$), $\alpha \in \{0.13, 0.15\}$, $N \in \{1k, 4k, 16k\}$, and initial overlap $m_0 = 0.9$. To study network recall quality, we can plot the probability distribution of overlaps $P(m_f)$, shown in figs. 3a and 3b. Below criticality, the weight at $m = 1$ increases with increasing N. Above criticality, the weight at $m = 1$ decreases with N and we instead observe a two-peak structure with weight emerging near $m = 0.35$. We conclude that for $\lambda = 0$, $0.13 < \alpha_c < 0.15$. In figs. 3c and 3d we plot how the fraction of converged states evolves over time. Near criticality, a vanishingly small number of states truly converge and states instead relax into 2-cycles, a known result accompanying synchronous update. We leave it as an exercise for the reader to show that with asynchronous update, the 2-cycle behavior is eliminated while the memory capacity remains the same.

In addition to asynchronous update, there are numerous avenues to further investigate the Hopfield network, such as modifying the synaptic matrix (with disorder, non-local weights, or dilution), asymmetric neural updating rules, and even using the Hopfield network to solve problems in another domain. The extension we show for this paper is probing the effects of non-zero self-action ($\lambda \neq 0$). We do so in line 90 of the code. In fig. 4, we observe that $\lambda = 1$ produces useful recall as high as $\alpha = 0.21$, and its performance degrades more gracefully in response to loading exceeding criticality ($\alpha = 0.24$, figs. 4a and 4b). However, with the introduction of self-coupling, the convergence time appears to increase (figs. 4c and 4d).[28]

(a) $P(m)$ histogram with $\alpha = 0.13 < \alpha_c$ (b) $P(m)$ histogram with $\alpha = 0.15 > \alpha_c$

(c) pattern convergence with $\alpha = 0.13 < \alpha_c$ (d) pattern convergence with $\alpha = 0.15 > \alpha_c$

(e) dot-product evolution with $\alpha = 0.13 < \alpha_c$, $p = \alpha N \in \{130, 520, 2080\}$

(f) dot-product evolution with $\alpha = 0.15 > \alpha_c$, $p = \alpha N \in \{150, 600, 2400\}$
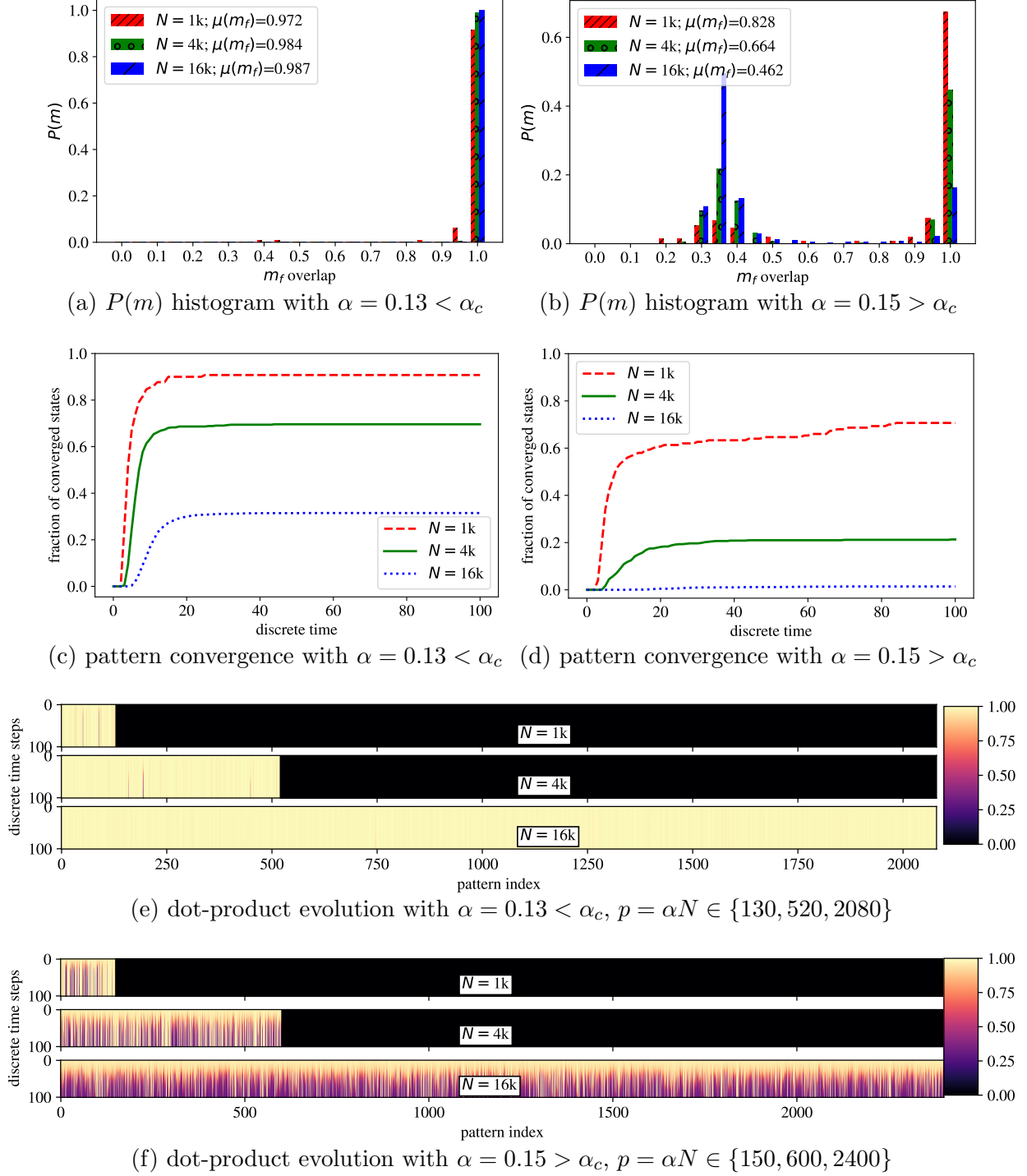
FIG. 3: Dynamics of the Hopfield network with zero self-coupling terms ($\lambda = 0$), $N \in \{1k, 4k, 16k\}$, and initial overlap $m_0 = 0.9$. In (a),(c), and (e) we show dynamics for $\alpha = 0.13$. In (b),(d), and (f) we show $\alpha = 0.15$. We plot the probability distribution of overlaps $P(m_f)$ with mean $\mu(m_f)$ after 100 time steps in (a) and (b). Below criticality, the overlap at the $m = 1$ weight increases with N, suggesting the memory loading is below criticality α_c. In (b), the loading at $m = 1$ decreases with N, implying that $\alpha > \alpha_c$. In (c) and (d) we observe that with increasing N and synchronous update, patterns do not converge to a steady state and instead fluctuate in 2-cycles. In (e) and (f) we show how the overlap m (represented by the colorbar) evolves over time.

16

(a) $P(m)$ histogram with $\alpha = 0.21 < \alpha_c$

(b) $P(m)$ histogram with $\alpha = 0.24 > \alpha_c$

(c) pattern convergence with $\alpha = 0.21 < \alpha_c$

(d) pattern convergence with $\alpha = 0.24 > \alpha_c$

(e) dot-product evolution with $\alpha = 0.21 < \alpha_c$, $p = \alpha N \in \{210, 840, 3360\}$

(f) dot-product evolution with $\alpha = 0.24 > \alpha_c$, $p = \alpha N \in \{240, 960, 3840\}$
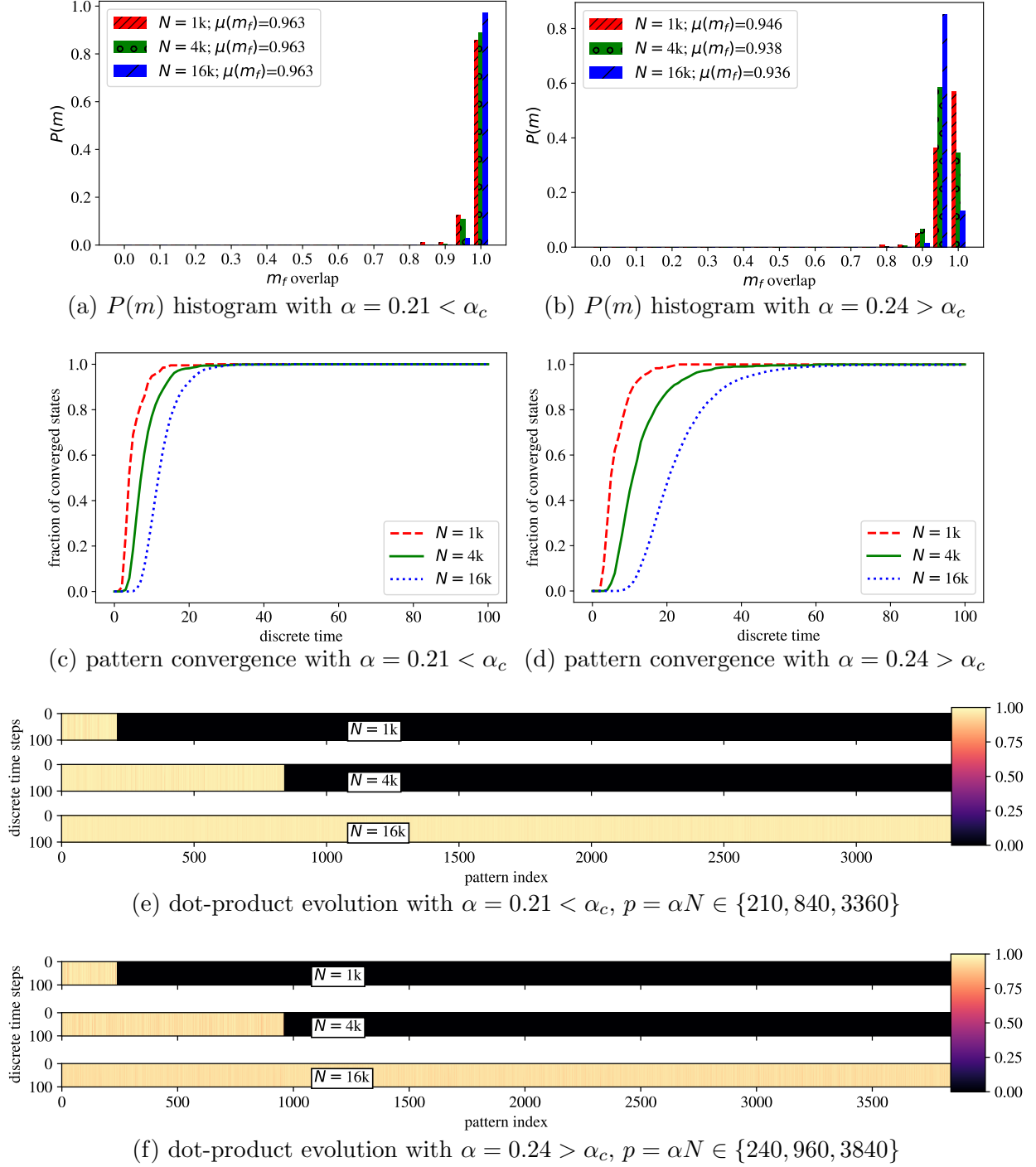
FIG. 4: Dynamics of the Hopfield network with self-coupling ($\lambda = 1$), $N \in \{1k, 4k, 16k\}$, and initial overlap $m_0 = 0.9$. In (a),(c), and (e) we show dynamics for $\alpha = 0.21$. In (b),(d), and (f) we show $\alpha = 0.24$. We plot the probability distribution of overlaps $P(m_f)$ with mean $\mu(m_f)$ after 100 time steps in (a) and (b). Below criticality, the overlap at the $m = 1$ weight increases with N, suggesting the memory loading is below criticality α_c. In (b), the loading at $m = 1$ decreases with N, implying that $\alpha > \alpha_c$. In (c) and (d) we observe that with synchronous update, all patterns converge but the rate slows with increasing N. In (e) and (f) we show how the overlap m evolves over time.

17

VI. SUGGESTED EXERCISES

1. Implement asynchronous update by introducing a parameter $0 < k < 1$ such that at every time step, kN neurons are updated. When $k = 1/N$, this is purely asynchronous update. When $k = 1$, we have purely synchronous update. For all other k, we have hybrid updating, which enjoys the massive parallelism inherent in synchronous update, while avoiding 2-cycles.

2. In the large-N limit, the maximal number of memories stored such that all are recalled perfectly[24] is $p < N/4log(N)$. Derive this result with theory using a signal-to-noise analysis[19,24,31] and test it with simulation.

3. Stochastic noise is typically implemented by using a probabilistic update rule[18,24] that modifies the time evolution of each neuron as follows:

$$h_i = \sum_{j \neq i}^{N} J_{ij} S_j(t) \tag{3}$$

$$P_{S_i}(+1; t+1) = (1 + \tanh(\beta h_i))/2 \tag{4}$$

$$P_{S_i}(-1; t+1) = 1 - P_{S_i}(+1; t+1) \tag{5}$$

where $\beta = 1/k_B T$ tunes the strength of noise, T is the absolute temperature, and k_B is the Boltzmann constant. $\beta = 0$ encodes totally random dynamics, and $T = 0$ encodes the usual deterministic update. Produce a $T - \alpha$ phase diagram to show the effect of these two parameters on m_f for varying m_0 and large N.

VII. CONCLUSION

In this tutorial, we have suggested the use of cloud computing, GPUs, and deep learning frameworks to accelerate large-scale physical simulations and make high-performance computing accessible to students and researchers. We demonstrated this by performing a simulation of the Hopfield network with large system size ($N = 32K$) and realized a GPU acceleration exceeding $50\times$ as compared with CPU-only simulations – using only free cloud resources. Our hope is that the rapid pace of development in the computer science discipline can enable physicists to work faster, and help educators remove barriers between their students and participation in research. We encourage the reader to modify the example code and implement their own physical simulations.

Appendix: Plotting code

To aid student learning, we show the reader the source code to reproduce the fraction of converged states figures (3c, 3d, 4c, 4d). The following code can be appended to the simulation code for instant viewing of figures in the browser. We use the Matplotlib package for creating publication-quality figures, and note that line 152 shows how to download any file (here a pdf image) from the cloud machine onto a personal computer. Implementations of the remaining figures look similar and we encourage students to reproduce them.

All source code in this paper can be found in this pre-populated Colab notebook: `https://colab.research.google.com/drive/1bS9V5GDzfeKe3Pu_yza8t66KjUNpMcFM`

Code Sample 8: Plot fraction of converged states

```
128  import matplotlib, matplotlib.pyplot as plt
129  from google.colab import files
130  font = {'family' : 'STIXGeneral',
131         'weight' : 'normal',
```

```
132          'size'   : 14}
133 matplotlib.rc('font', **font)
134 fig = plt.figure()
135 fig.set_size_inches(6, 4)
136 ax = plt.axes()
137 xticks = np.arange(0,max_steps+1,1)
138 ax.plot(xticks,num_conv_lists[0].numpy()[0,:],'--r',label=r'$N=$'+str(int(
        N_list[0]/1000))+'k',linewidth=2);
139 ax.plot(xticks,num_conv_lists[1].numpy()[0,:],'-g',label=r'$N=$'+str(int(
        N_list[1]/1000))+'k',linewidth=2);
140 ax.plot(xticks,num_conv_lists[2].numpy()[0,:],':b',label=r'$N=$'+str(int(
        N_list[2]/1000))+'k',linewidth=2);
141 plt.xlabel(r'discrete time')
142 plt.ylabel(r'fraction of converged states')
143 leg = plt.legend()
144 yticks = np.arange(0,1.001,0.2)
145 plt.yticks(yticks,[str(y)[0:3] for y in yticks])
146 leg_lines = leg.get_lines()
147 plt.setp(leg_lines, linewidth=2)
148 plt.tight_layout()
149 plt.show()
150 fname='frac_lambda0_alpha014.pdf'
151 fig.savefig(fname, dpi=300)
152 files.download(fname)
```

ACKNOWLEDGMENTS

---

[*] vsv2109@columbia.edu

[1] M. D. Caballero and S. J. Pollock, American Journal of Physics **82**, 231 (2014), https://doi.org/10.1119/1.4837437.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (The MIT Press, 2016).

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Nature **323**, 533 (1986).

[4] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, in *Proceedings of the 30th International Conference on Machine Learning*, Proceedings of Machine Learning Research, Vol. 28, edited by S. Dasgupta and D. McAllester (PMLR, Atlanta, Georgia, USA, 2013) pp. 1337–1345.

[5] T. Akenine-Moller, E. Haines, and N. Hoffman, *Real-Time Rendering*, 3rd ed. (A. K. Peters, Ltd., Natick, MA, USA, 2008).

[6] B. Kim, V. C. Azevedo, N. Thuerey, T. Kim, M. Gross, and B. Solenthaler, Computer Graphics Forum **38**, 59 (2019), https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13619.

[7] L. Kavan, D. Gerszewski, A. W. Bargteil, and P.-P. Sloan, in *ACM Transactions on Graphics (TOG)*, Vol. 30 (ACM, 2011) p. 93.

[8] S. Tariq and L. Bavoil, in *SIGGRAPH '08* (2008).

[9] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, in *Advances in Neural Information Processing Systems 32*, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Curran Associates, Inc., 2019) pp. 8024–8035.

[10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, arXiv preprint arXiv:1603.04467 (2016).

[11] T. Carneiro, R. V. Medeiros Da Nbrega, T. Nepomuceno, G. Bian, V. H. C. De Albuquerque, and P. P. R. Filho, IEEE Access **6**, 61677 (2018).

[12] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, in *of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing*

Systems (NIPS) (2017).

[13] N. Jouppi, C. Young, N. Patil, and D. Patterson, IEEE Micro **38**, 10 (2018).

[14] P. Baldi, P. Sadowski, and D. Whiteson, Nature communications **5**, 4308 (2014).

[15] P. Mehta, M. Bukov, C.-H. Wang, A. G. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, Physics Reports **810**, 1 (2019), a high-bias, low-variance introduction to Machine Learning for physicists.

[16] K. Schtt, P. Kessel, M. Gastegger, K. Nicoli, A. Tkatchenko, and K.-R. Mller, Journal of chemical theory and computation **15**, 448 (2018).

[17] J. J. Hopfield, Proceedings of the National Academy of Sciences **79**, 2554 (1982).

[18] D. J. Amit, H. Gutfreund, and H. Sompolinsky, Annals of Physics **173**, 30 (1987).

[19] Y. Bar-Yam, *Dynamics of complex systems* (Addison-Wesley, 1997).

[20] Y. Li, Z. Tang, G. Xia, and R. Wang, Circuits and Systems I: Regular Papers, IEEE Transactions on **52**, 200 (2005).

[21] Y. Li, Z. Tang, G. Xia, R. L. Wang, and X. Xu, in *SICE 2004 Annual Conference*, Vol. 2 (2004) pp. 999–1004 vol. 2.

[22] F. Kazemi, M.-R. Akbarzadeh-T, S. Rahati, and H. Rajabi, in *Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference on* (2008) pp. 001855–001860.

[23] W. Widodo, R. A. Priambodo, and B. P. Adhi, IOP Conference Series: Materials Science and Engineering **434**, 012034 (2018).

[24] D. J. Amit, *Modeling brain function* (Cambridge University Press, 1989).

[25] R. McEliece, E. Posner, E. Rodemich, and S. Venkatesh, IEEE Transactions on Information Theory **33**, 461 (1987).

[26] K. Gopalsamy and P. Liu, Nonlinear Analysis: Real World Applications **8**, 375 (2007).

[27] M. P. Singh, Phys. Rev. E **64**, 051912 (2001).

[28] Y. Tsuboshita and M. Okada, journal of the Physical Society of Japan **79**, 024002 (2010).

[29] S. Bharitkar and J. M. Mendel, IEEE Transactions on neural networks **11**, 879 (2000).

[30] J. L. McClelland and D. E. Rumelhart, Behavior Research Methods, Instruments, & Computers **20**, 263 (1988).

[31] J. Hertz, A. Krogh, and R. G. Palmer, *Introduction to the theory of neural computation* (Addison-Wesley Pub. Co., 1991).