



Formalized Hopfield Networks and Boltzmann Machines

Matteo Cipollina^{1†}, Michail Karatarakis^{2†}, and Freek Wiedijk²

¹ Axiomatic AI, Barcelona, Spain
matteo.cipollina@yahoo.it

² Radboud University, Nijmegen, The Netherlands
michail.karatarakis@ru.nl, freek@cs.ru.nl

Abstract

Neural networks are widely used, yet their analysis and verification remain challenging. We present a Lean 4 formalization covering both deterministic and stochastic models. We first formalize Hopfield networks – recurrent networks that store patterns as stable states – and prove their convergence, and the correctness of Hebbian learning, the rule that updates parameters to encode patterns. We then turn to stochastic networks, whose probabilistic updates converge to a stationary distribution: we formalize the dynamics and learning of Boltzmann machines and prove their ergodicity – convergence to a *unique* stationary distribution – via a new formalization of the Perron–Frobenius theorem.

1 Introduction

Neural networks underpin most current practice in machine learning. In October 2024, John Hopfield and Geoffrey Hinton were awarded the Nobel Prize in Physics for their work in machine learning, particularly in the development of artificial neural networks [63]. The prize recognizes models whose mathematics comes directly from statistical mechanics.

One of Hopfield’s most influential achievements is the Hopfield network [34], introduced in 1982. Hopfield networks model associative memory, storing and recalling patterns from partial or noisy input, and were originally inspired by the Ising model of magnetism [35]. A key feature is their convergence property, where the network reaches a stable state corresponding to a stored pattern. Building on Hopfield networks, Boltzmann machines were introduced by Ackley, Hinton and Sejnowski [3] in 1985 as a stochastic generalization. In Boltzmann machines, neuron updates are probabilistic rather than deterministic, allowing the network to sample from probability distributions over states instead of converging to a single stable state.

In 2020, Ramsauer et al. showed that the update rule of continuous Hopfield networks is equivalent to the attention mechanism used in Transformers [55]. These models also have a concrete computational role: minimizing a Hopfield energy is, up to an affine change of variables, quadratic unconstrained binary optimization (QUBO) [45, 27], the native input format of quantum annealers. The same energy descent is realized directly in analog neuromorphic hardware [28, 39], while stochastic Ising machines realize Boltzmann sampling over discrete

[†]Equal contribution, authors listed alphabetically.

spin states [18, 51] – optimization and sampling carried out in the physical device rather than a general-purpose processor. A machine-checked account of these dynamics therefore gives a specification against which such hardware and the algorithms running on it could be verified.

Mainstream neural-network verification targets feedforward networks, where the network is a fixed function and the properties are local: typically robustness of a single input-output map on a bounded region [14, 17, 22, 24, 70, 41, 42]. In contrast, the formalization of recurrent networks and their learning algorithms themselves remains relatively unexplored. Energy-based recurrent models demand global statements of a different kind: that asynchronous updates terminate, that the induced Markov chain is ergodic, and that it converges to a unique stationary distribution – theorems of probability theory and statistical mechanics, not of bound propagation.

This paper addresses that gap, and the project was inspired by a suggestion from Britt Anderson in the Lean Zulip chat [6]. We formalize both deterministic Hopfield networks and stochastic Boltzmann machines, giving a rigorous foundation for reasoning about their dynamics and learning. Engaging with these models also led to a general framework treating feedforward and recurrent architectures uniformly.

Contributions and overview. To our knowledge, this is the first formalization of Hopfield networks and Boltzmann machines in any proof assistant. The development comprises around 16,000 lines of new Lean code (excluding blank lines and comments), of which roughly 7,100 lines comprise reusable mathematical infrastructure (probability and Perron–Frobenius) and 9,000 comprise the neural-network development; Table 1 gives the per-component split. The proofs use standard Lean and `mathlib` tactics and automation throughout. We work in Lean 4 because `mathlib` offers a single unified hierarchy of probability, algebra, analysis, and topology, all used together in the proofs below; aiming to contribute back, we follow its philosophy of maximal generality rather than ad hoc formalization. We also bridge to the statistical-mechanics part of `Physlib`, a `mathlib`-style physics library, and lay groundwork for `Optlib` [40] (a library of first-order optimization and convergence theorems). Every component in Table 1 is new; the table records its size, computability and ties to existing Lean libraries. Most definitions in §4 – §4.2 are `noncomputable`, relying on `mathlib`’s nonconstructive reals, so they yield no executable code. The clickable symbol  links to the corresponding source, and a public repository accompanies the paper . We conclude with related work (§5) and future directions (§6).

Component	Lines	Comp.	Reuse	Upstreaming
Unified framework for feedforward and recurrent networks (§2). The organizing <code>Quiver</code> core was <i>discovered</i> , not designed: adopted for path manipulation in the Perron–Frobenius proof, it proved the right abstraction for the whole framework.	2,400	yes	<code>mathlib</code> : <code>Quiver</code> , linear algebra, analysis	<code>Physlib</code> (PR under review)
Discrete symmetric Hopfield networks: convergence and Hebbian learning (§3). Formalization surfaced an implicit <i>fairness</i> condition on the update order, required for convergence yet omitted from textbook statements (Theorem 3.1).	2,000	yes	<code>mathlib</code> : <code>Matrix</code> , <code>Vector</code>	<code>Physlib</code> (PR under review)
Probabilistic infrastructure: reversibility, invariance of Markov kernels, Gibbs sampling (§4).	1,700	no	<code>mathlib</code> : <code>Kernel</code> , PMF, linear algebra	<code>mathlib</code> (ongoing)
Perron–Frobenius for irreducible/stochastic matrices (§4.1); first in Lean 4, via an analytic Collatz–Wielandt argument.	5,400	no	<code>mathlib</code> : spectrum, topology, <code>Quiver</code> paths	<code>mathlib</code> (PR in prep.)

Boltzmann machines: convergence to the Boltzmann distribution, plus single-pattern Boltzmann learning and CD- k (§4.2, §4.3).	4,600	no	Physlib: CanonicalEnsemble, Temperature; mathlib: probability, analysis	Physlib (PR in prep.); Optlib (submission in prep.)
---	-------	----	---	---

Table 1: Contributions. “Comp.” records whether the component yields executable code; “Reuse” lists the existing library components it builds on; “Upstreaming” gives the status of contributing it back.

2 General Neural Networks

A fundamental design decision in formalizing neural networks is the choice of the underlying representation. Neural networks are commonly represented as directed graphs, categorized as feedforward (acyclic) or recurrent (cyclic) and typically treated as distinct mathematical objects in the literature (e.g., Ch. 5 vs. Ch. 8, [37]). Modern deep learning frameworks, such as TensorFlow [1] and PyTorch [54], predominantly adopt a sequential model, in which networks are represented as a sequence of layers – typically a list paired with an abstract table of activation functions – or, more generally, as a directed acyclic graph (DAG) of tensor operations, i.e., a computational graph. This sequential approach is also reflected in the verification literature (see §5). In contrast, the definition we give below is general enough to accommodate both architectures and integrate with Physlib (§4.4). Feedforward networks arise as a special case (see SequentialArch, Sequential.toNeuralNetwork). This work is guided by the theory in Chapter 4 of [37] and builds on mathlib’s graph theory.

Let $G = (U, C)$ be a directed graph consisting of a finite set U of vertices (or nodes) and a finite set $C \subseteq U \times U$ of edges. An edge $e = (u, v) \in C$ is said to be *directed* from the vertex u to the vertex v . The predecessors and successors of a vertex u in a directed graph $G = (U, C)$ are defined as $\text{pred}(u) = \{v \in U \mid (v, u) \in C\}$ and $\text{succ}(u) = \{v \in U \mid (u, v) \in C\}$ respectively.

Definition 2.1 (Def. 4.3, [37]). *An (artificial) neural network is a directed graph $G = (U, C)$, where neurons $u \in U$ are connected by directed edges $c \in C$ (connections). The neuron set is partitioned as $U = U_{\text{in}} \cup U_{\text{out}} \cup U_{\text{hidden}}$, with $U_{\text{in}}, U_{\text{out}} \neq \emptyset$ and $U_{\text{hidden}} \cap (U_{\text{in}} \cup U_{\text{out}}) = \emptyset$. Each connection $(v, u) \in C$ has a weight w_{uv} , and each neuron u has real-valued quantities: network input net_u , activation act_u , and output out_u . Input neurons $u \in U_{\text{in}}$ also have a fourth quantity, the external input ext_u . Each neuron u is associated with the following functions:*

$$f_{\text{net}}^{(u)} : \mathbb{R}^{2|\text{pred}(u)| + \kappa_1(u)} \rightarrow \mathbb{R}, \quad f_{\text{act}}^{(u)} : \mathbb{R}^{1 + \kappa_2(u)} \rightarrow \mathbb{R} \quad \text{and} \quad f_{\text{out}}^{(u)} : \mathbb{R} \rightarrow \mathbb{R}.$$

These functions compute net_u , act_u , and out_u , where $\kappa_1(u)$ and $\kappa_2(u)$ count the number of parameters of those functions, which can depend on the neurons.

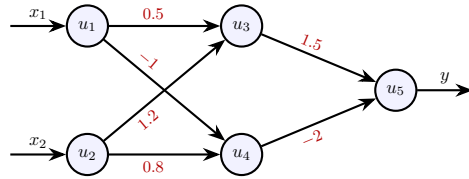


Figure 1: A feedforward network with external inputs x_1, x_2 and output y , and no loops; arrow labels are connection weights.

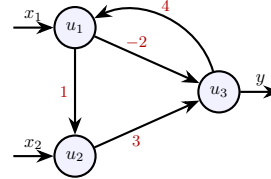


Figure 2: A recurrent network, same conventions as Fig. 1. The feedback edge $u_3 \rightarrow u_1$ lets information cycle back.

In the index of a weight w_{uv} , the neuron receiving the connection is listed first, following the “row first, then column” convention of the adjacency matrix, with matrix-vector equations written using column vectors and standard matrix-vector multiplication rules. This structure groups all incoming weights to a neuron within the same row, represented compactly as the matrix $W = (w_{uv})_{u,v \in U}$. The new activation act'_u of a neuron u is computed as follows:

$$\text{act}'_u = f_{\text{act}}^{(u)}(f_{\text{net}}^{(u)}(w_{uv_1}, \dots, w_{uv_{\text{pred}(u)}}), f_{\text{out}}^{(v_1)}(\text{act}_{v_1}), \dots, f_{\text{out}}^{(v_{\text{pred}(u)})}(\text{act}_{v_{\text{pred}(u)}}), \sigma^{(u)}), \theta^{(u)})$$

where $\sigma^{(u)} = (\sigma_1^{(u)}, \dots, \sigma_{\kappa_1(u)}^{(u)})$ and $\theta^{(u)} = (\theta_1^{(u)}, \dots, \theta_{\kappa_2(u)}^{(u)})$ are the input parameter vectors.

The following diagram shows the structure of a generalized neuron abstracting away specific model assumptions. Here $\mathbf{act} = (\text{act}_{v_1}, \dots, \text{act}_{v_{|U|}})$ and $\mathbf{out} = (\text{out}_{v_1}, \dots, \text{out}_{v_{|U|}})$ collect the activations and outputs of all neurons, and v_i ranges over the predecessors of u .

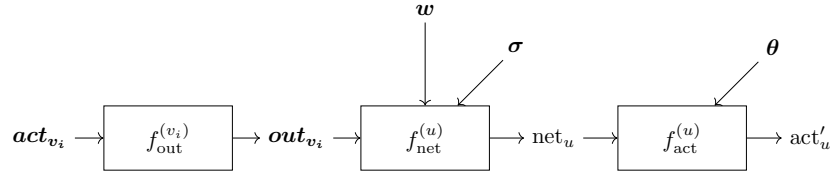


Figure 3: Structure of a generalized neuron.

In our formalization, we separate a neural network into its architecture, defined by its graph structure `NeuralNetwork` (see Listing 1), and parameters `Params` (see Listing 2), controlling dynamics like input processing. This separation allows flexibility, enabling different functions, weights, or learning rules without redefining the architecture. For brevity, we omit well-formedness conditions, which are fully specified in the repository.

```

universe uR uU uζ v
structure NeuralNetwork (R : Type uR) (U : Type uU) (ζ : Type uζ) [Zero R]
  extends Quiver.{v, uU} U where
    (Ui Uo Uh : Set U)
    (hUi : Ui ≠ ∅)
    (hUo : Uo ≠ ∅)
    (hU : Set.univ = (Ui ∪ Uo ∪ Uh))
    (hhio : Uh ∩ (Ui ∪ Uo) = ∅)
    (κ1 κ2 : U → ℕ)
    (fnet : ∀ u : U, (U → R) → (U → R) → Vector R (κ1 u) → R)
    (fact : ∀ u : U, ζ → R → Vector R (κ2 u) → ζ)
    (fout : ∀ _ : U, ζ → R)
    (m : ζ → R)
    (pact : ζ → Prop)
    (pw : ∀ (u v : U), (u → v) → Prop)
    (pwMat : Matrix U U Prop := fun u v => ∃ f : (u → v), pw u v f)
    (pm : Matrix U U R → Prop)

```

Listing 1: Definition of neural network structure.

```

variable {R U ζ : Type} [Zero R]
structure Params (NN : NeuralNetwork R U ζ) where
  (h_arrows : ∀ u v (f : NN.Hom u v), NN.pw u v f)
  (w : Matrix U U R) -- the weight matrix W
  (σ : ∀ u : U, Vector R (NN.κ1 u)) -- the input vector

```

```

( $\theta$  :  $\forall u : U$ , Vector R (NN. $\kappa 2$  u)) -- the threshold vector
(hw :  $\forall u v$ ,  $\neg$  NN.pwMat u v  $\rightarrow$  w u v = 0)
(hw' : NN.pm w)

```

Listing 2: Definition of parameters.

To ensure flexibility and generality across network structures, we used an arbitrary type R for weights, with `[Zero R]` indicating the existence of a zero element in R , instead of restricting ourselves to real numbers. Similarly, we use U for the neurons. However, standard definitions of neural networks model activations simply as real numbers, giving them the same type as the weights. Since activations may be discrete whereas weights may be any real number, we instead parameterize the network over a separate abstract activation type ζ , using the embedding ($m : \zeta \rightarrow R$) to interpret these states as elements of the weight type. This design unifies architectures via the `TwoStateNeuralNetwork` typeclass, which abstracts over concrete encodings such as $\{-1, +1\}$, $\{0, 1\}$, or custom types like `Signum` and ensures compatibility with the `Physlib` library (see §4.4). The construction is also parameterized by an interpreter (`actMap :  $\zeta \rightarrow R \rightarrow R$`), which assigns a concrete function $R \rightarrow R$ to each abstract descriptor. This decoupled design allows the architecture to remain abstract while supporting heterogeneous activations – such as ReLU for hidden layers and sigmoid for output – within a single network instance.

Design decisions We initially modeled neural networks using `mathlib`’s `Digraph` structure, which defines a graph as a binary relation on vertices :

```

structure Digraph (V : Type*) where Adj : V  $\rightarrow$  V  $\rightarrow$  Prop

```

In the textbook definition 2.1, function arguments and summations for a neuron u are generally indexed running over the set $\text{pred}(u)$. In Lean that could correspond to passing around adjacency proof objects everywhere. For example, the weight matrix could have been written as ($w : \forall u v : U$, Adj v u $\rightarrow$ R). Similarly, the type of $f_{\text{net}}^{(u)} : \mathbb{R}^{2|\text{pred}(u)|+\kappa_1(u)} \rightarrow \mathbb{R}$ in Lean would have been :

```

(fnet :  $\forall u : U$ , ( $\forall v : U$ , Adj v u  $\rightarrow$  R)  $\rightarrow$  ( $\forall v : U$ , Adj v u  $\rightarrow$  R)  $\rightarrow$ 
  Vector R ( $\kappa 1$  u)  $\rightarrow$  R)

```

This would require us to track adjacency relations explicitly, complicating the formalization. To simplify our formalization, we decided not to do this. Instead we used the convention that all weights for neurons that are not adjacent are zero. Therefore we used a matrix ($w : \text{Matrix } U \ U \ R$) directly and the function formalizing $f_{\text{net}}^{(u)}$ had type

```

(fnet :  $\forall u : U$ , (U  $\rightarrow$  R)  $\rightarrow$  (U  $\rightarrow$  R)  $\rightarrow$  Vector R ( $\kappa 1$  u)  $\rightarrow$  R)

```

This may be less efficient for sparse networks needing precise connection control, and can make the functions harder to implement. Adjacency could be supplied as extra ($U \rightarrow \text{Bool}$) arguments, but our applications did not need this, so it is not part of our formalization. Furthermore, whilst a propositional representation suffices for simple connectivity, the formalization of the Perron–Frobenius theorem in §4.1 necessitated a transition to `mathlib`’s `Quiver` type. A quiver on a vertex type V assigns a type $a \rightarrow b$ of directed arrows to every pair of vertices ($a \ b : V$).

```

class Quiver (V : Type u) where Hom : V  $\rightarrow$  V  $\rightarrow$  Type v

```

This generalizes `Digraph V`, where the arrow $\text{Hom} : a \rightarrow b$ is a proposition and not a type. By adopting `Quiver` as our foundation, connections inhabit a `Type`, enabling the representation

of multigraphs with multiple distinct edges between vertices. Furthermore, **Quiver** benefits from a significantly more mature API in **mathlib** and provides the extensibility required to model complex neural architectures beyond those explored in this work.

Operation of neural networks. Computation proceeds in two phases. In the *input phase*, each input neuron is set to its external input and all other neurons to 0, and the output function is applied to these activations so that every neuron produces an output. In the *work phase*, neurons repeatedly recompute their activations – and hence their outputs – in an order that is not fixed: either simultaneously (synchronous update), using the previous outputs, or one at a time (asynchronous update), using the most recently computed ones. Different update schemes suit different network types.

Each neuron is either *firing* or *not firing*; the two states are encoded numerically and enforced by an admissibility predicate on activations. The general convention uses 1 (firing) and 0 (not firing), so a state is a vector $s = (\text{act}_{u_1}, \dots, \text{act}_{u_n})^\top \in \{0, 1\}^n$, whereas Hopfield networks use ± 1 ($\text{act}_u \in \{-1, 1\}$). The **State** structure stores the activations ($\text{act} : U \rightarrow \zeta$) together with a proof that each is admissible. Functions **State.fout** and **State.fnet** operate on these stored activations, playing the role of **NeuralNetwork.fout** and **NeuralNetwork.fnet** for the current state, while **State.Up** updates a single chosen neuron u via **NeuralNetwork.fact**, leaving all other activations unchanged.

In practice, a stopping criterion – based on updates, energy, or convergence measures – ends the iteration. Stability is the property **State.isStable**: a state s is stable if applying **State.Up** to any neuron leaves that neuron’s activation unchanged. Given an update sequence ($\text{useq} : \mathbb{N} \rightarrow U$), **seqStates** constructs the resulting sequence of states recursively – the first is the initial state s , and each subsequent one updates the neuron useq names at that step. The **workPhase** function instead takes an initial state extu and a finite list uOrder , folding **State.Up** over it (**List.foldl**) to perform one update per list entry, in order, and return the resulting state.

As a sanity check we validate the framework on the three-neuron recurrent network of Fig. 2 (p. 42, [37]): under one update order the work phase reaches a stable all-zero state, while another order oscillates indefinitely, so the output can depend not only on the input but also on the number of updates – behavior precluded in Hopfield networks, which always converge under asynchronous updates.

3 Hopfield networks

Before Hopfield’s work, neural networks were primarily feedforward models, with unidirectional information flow. Hopfield networks popularized recurrent connections, enabling feedback between neurons, allowing them to model dynamic systems and handle more complex tasks. We follow the mathematical background in Chapter 8 of [37].

Definition 3.1. A Hopfield network is a neural network with graph $G = (U, C)$ as described in the previous section, that satisfies the following conditions: $U_{\text{hidden}} = \emptyset$, and $U_{\text{in}} = U_{\text{out}} = U$, $C = U \times U - \{(u, u) \mid u \in U\}$, i.e., no self-connections. The connection weights are symmetric, i.e., $\forall u, v \in U$, we have $w_{uv} = w_{vu}$ when $u \neq v$. The activation of each neuron is either 1 or -1 depending on the input. There are no loops, i.e., no neuron receives its own output as input: $w_{uu} = 0$ for all $u \in U$. The neuron functions for the Hopfield network are defined for all $u \in U$ as:

$$f_{\text{net}}^{(u)}(w_u, \text{in}_u) = \sum_{v \in U \setminus \{u\}} w_{uv} \text{out}_v, \quad f_{\text{act}}^{(u)}(\text{net}_u, \theta_u) = \begin{cases} 1 & \text{if } \text{net}_u \geq \theta_u, \\ -1 & \text{otherwise,} \end{cases} \quad f_{\text{out}}^{(u)}(\text{act}_u) = \text{act}_u.$$

Here θ_u is the firing threshold: increasing θ_u makes activation more difficult. This is consistent with the energy convention below, where the threshold contribution is $+\theta_u \text{act}_u$.

We define the **HopfieldNetwork** as an instance of **NeuralNetwork** over a finite type U and a strictly ordered ring R . The network topology is fully connected excluding self-loops, formalized by defining the edge type $\text{Hom } u \ v$ as $\text{PLift } (u \neq v)$. The sets of input U_i and output U_o neurons coincide with the universal set Set.univ , while the set of hidden neurons U_h is empty.

As the three-neuron example of §2 shows, updates can cause a recurrent network to oscillate between activation states. However, asynchronous updates (one neuron at a time) in a Hopfield network always reach a stable state, preventing oscillation.

Theorem 3.1 (Convergence Theorem for Hopfield networks: Theorem 8.1, [37]). *If the activations of the neurons of a Hopfield network are updated asynchronously, a stable state is reached in a finite number of steps (**HopfieldNet_convergence_fair**). If the neurons are traversed in an arbitrary, but fixed cyclic fashion, at most $n \cdot 2^n$ steps (updates of individual neurons) are needed, where n is the number of neurons of the network (**HopfieldNet_convergence_cyclic**).*

This theorem is proved by defining an energy function that assigns a real value to each state of the Hopfield network, which decreases or remains constant with each transition. When energy remains unchanged, we show that a state can never be revisited. Since there are only finitely many states, this ensures the network will eventually reach a stable state.

The energy function E of a Hopfield network with n neurons $u_1, \dots, u_n$ is

$$E = -\frac{1}{2} \sum_{\substack{u,v \in U \\ u \neq v}} w_{uv} \text{act}_u \text{act}_v + \sum_{u \in U} \theta_u \text{act}_u. \quad (1)$$

The factor $\frac{1}{2}$ compensates for the symmetry of the weights, as each term in the first sum appears twice. We show that energy cannot increase during a state transition when a neuron is updated. In an asynchronous update, only one neuron's activation, u , changes from $\text{act}_u^{(\text{old})}$ to $\text{act}_u^{(\text{new})}$. The energy difference between the old and new states consists only of terms containing act_u , with all other terms canceling out. The factor $\frac{1}{2}$ vanishes due to the symmetry of the weights, causing each term to occur twice. Then we can extract the new and old activation states of neuron u from the sums, yielding the energy difference (**energy_diff_formula**):

$$\Delta E = E^{(\text{new})} - E^{(\text{old})} = (\text{act}_u^{(\text{old})} - \text{act}_u^{(\text{new})}) \underbrace{\left(\sum_{v \in U - \{u\}} w_{uv} \text{act}_v - \theta_u \right)}_{\text{net}_u}.$$

We consider two cases. If $\text{net}_u < \theta_u$ and the activation changed, then $\text{act}_u^{(\text{new})} = -1$ and $\text{act}_u^{(\text{old})} = 1$, so $\Delta E < 0$. If $\text{net}_u \geq \theta_u$, then $\text{act}_u^{(\text{old})} = -1$ and $\text{act}_u^{(\text{new})} = 1$, yielding $\Delta E \leq 0$.

If a state transition reduces energy, the original state cannot be revisited, as this would require an energy increase. In the second case, where energy remains constant, the transition leads to more $+1$ activations, as the updated neuron's activation changes from -1 to 1 . Thus, the original state cannot be revisited in this case either. The theorem **energy_lt_zero_or_pluses_increase** captures this idea.

Formalization insight Each state transition reduces the number of reachable states. Since the total number of states is finite, the network will eventually reach a stable state. However,

this convergence is only guaranteed if every neuron is eventually updated. If some neurons are excluded, the network may fail to stabilize. This highlights a logical gap in the standard literature, where [Theorem 3.1](#) frequently omits the necessary hypothesis of “fairness”. Without this constraint, an update schedule may permanently exclude neurons that need to change, causing the energy to stop decreasing even though the network has not yet reached a stable state. To prevent this, we define “fairness” [↗](#) in the update sequence as follows:

```
def fair (useq : ℕ → U) : Prop := ∀ u n, ∃ m ≥ n, useq m = u
```

Similarly, a fixed cyclic update order naturally ensures that every neuron is updated. We define a cyclic update sequence and prove its inherent fairness using the lemma `cycl_fair` [↗](#) as a sanity check.

```
def cyclic [Fintype U] (useq : ℕ → U) : Prop :=
  (∀ u : U, ∃ i, useq i = u) ∧
  (∀ i j, i % card U = j % card U → useq i = useq j)
```

We have shown that asynchronous updates, when applied fairly, always lead the Hopfield network to a stable state, forming the basis for its use in associative memory – a kind of memory that is addressed by its contents.

3.1 Hopfield Networks as Associative Memory

A *pattern* $p = (\text{act}_{u_1}, \dots, \text{act}_{u_n})^\top \in \{-1, 1\}^n$ ($n \geq 2$) is a vector of neuron activations – a possible network state. Unlike classical memory, which needs the exact state vector, associative memory recovers a stored pattern from partial, noisy, or corrupted input. Hopfield networks realize this by storing patterns as stable states, using the Hebbian algorithm [\[31\]](#) to set the weights and thresholds. Formally, storing p means $S(Wp - \theta) = p$, where W is the weight matrix, $\theta = (\theta_{u_1}, \dots, \theta_{u_n})$ the threshold vector, and $S : \mathbb{R}^n \rightarrow \{-1, 1\}^n$ maps $x \mapsto y$ with $y_i = 1$ if $x_i \geq 0$ and -1 otherwise.

Given a choice of thresholds θ , the goal is to find a suitable weight matrix W such that the stored patterns correspond to stable states. For example, in the case of storing a single pattern p with all thresholds chosen to be zero, it clearly suffices if $Wp = cp$ with $c \in \mathbb{R}_+$.

When storing multiple patterns $p_1, \dots, p_m$ with $m < n$, the weight matrix W_i is determined for each pattern p_i . The overall weight matrix W is then the sum of these individual matrices:

$$W = \sum_{i=1}^m W_i = \sum_{i=1}^m p_i p_i^\top - mI \quad \text{↗} \tag{2}$$

where I is the $n \times n$ identity matrix. Since $p_i^\top p_j = 0$ for $i \neq j$ and $p_i^\top p_i = n$, the inner product of orthogonal vectors is zero, while the inner product of a vector with itself equals its squared length.

The patterns p_i and p_j to be stored can be pairwise orthogonal (that is, if $p_i^\top p_j = 0$ for $i \neq j$) (see `patterns_pairwise_orthogonal`), or non-orthogonal (see `patterns_pairwise_non_orthogonal`). The weight matrix W for an arbitrary pattern p_j , $j \in \{1, \dots, m\}$ is given by:

In practice, patterns are rarely orthogonal, introducing disturbance terms that affect stability. As m increases, the system becomes more susceptible to these disturbances (as $(n - m)$ shrinks), preventing the network from achieving its theoretical storage capacity.

Orthogonal case $Wp_j = \sum_{i=1}^m p_i (p_i^\top p_j) - mp_j$ $= (n - m)p_j$	Non-orthogonal case $Wp_j = (n - m)p_j + \underbrace{\sum_{\substack{i=1 \\ i \neq j}}^m p_i (p_i^\top p_j)}_{\text{disturbance term}}$
--	---

Table 2: Stability analysis of the Hebbian weight matrix acting on a stored pattern p_j .

In Lean, the Hebbian learning rule is the function `Hebbian` , which maps m patterns (`ps : Fin m → (HopfieldNetwork R U).State`) to network parameters `Params (HopfieldNetwork R U)` with zero thresholds and the weight matrix of Equation (2).

As a sanity check, we instantiate a four-neuron Hopfield network with two orthogonal patterns, store them via Hebbian learning, and demonstrate convergence to a stable stored pattern during the work phase .

4 Stochastic networks

Hopfield networks are described by an energy function: asynchronous updates drive the system to stable states, and Hebbian learning encodes patterns as such states. This deterministic model is mathematically clean but restrictive; we generalize to stochastic updates, where a neuron changes its activation at random with a probability set by the resulting energy change. This links the energy to a probability distribution and recasts the dynamics as a Markov Chain Monte Carlo (MCMC) method – sampling from a distribution by constructing a Markov chain whose long-run behavior reflects it. In the stochastic setting, “stability” no longer means a fixed stable state but convergence to a stationary distribution, formalized by reversibility (detailed balance).

A complete MCMC theory has two parts: a measure-theoretic model and a convergence theory. The model is the *Markov kernel*, a measurable function giving transition probabilities over arbitrary state spaces; on a finite state space the kernel becomes a stochastic matrix, and convergence is governed by the Perron–Frobenius theorem and the fundamental theorem of Markov chains (Thms. 6.2.2/6.2.5, pp. 236–237, [56]). We assume familiarity with Markov kernels, single-site update kernels and basic measure-theoretic probability, following [56, 36], which underlie the `mathlib` constructions.

On a finite state space (sigma-algebra $\mathcal{A} = \mathcal{P}(\mathcal{X})$), a Markov kernel is a *stochastic matrix* (pp. 48–52, [59]) with non-negative entries; we adopt the *column-stochastic* convention, so each column sums to one and the entry A_{ij} is the probability of transitioning from state j to state i .

Intuitively, a single-site update kernel “resamples” only one coordinate of the state vector while keeping the others fixed, which underlies neuron updates in stochastic finite-state systems such as Boltzmann machines. To generate samples from a distribution π , we define single-site Gibbs updates (p. 285, [56]).

Definition 4.1 (Algorithm A.31, p. 301, [56]). *Let $\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_n$ be a finite product state space, and let π be a target probability measure on $\mathcal{X}$. For each site $i \in \{1, \dots, n\}$, the Gibbs update kernel K_i is a single-site kernel defined by*

$$K_i(x, A) = \sum_{x'_i \in \mathcal{X}_i : (x_1, \dots, x_{i-1}, x'_i, x_{i+1}, \dots, x_n) \in A} \pi(x'_i \mid x \setminus i),$$

for all measurable sets $A \subseteq \mathcal{X}$, where $x \setminus i$ denotes all coordinates of x except the i -th.

Mathematically, Gibbs sampling is a special case of the Metropolis–Hastings (MH) algorithm [30], in which one proposes a move and accepts it with a probability that ensures invariance of the target distribution. In Gibbs sampling, the proposal is the exact conditional distribution of a single coordinate given the others, for which the MH acceptance probability is always 1. The `TwoState.gibbsUpdate` [↗](#) function formalizes the Gibbs kernel for the `TwoStateNeuralNetwork` class using `mathlib`’s PMF type. It instantiates the abstract kernel K_i with the *Boltzmann distribution* below as the target measure π .

$$\pi(\mathbf{act}) = \frac{1}{c} \exp\left(-\frac{E(\mathbf{act})}{kT}\right) \quad (3)$$

where $\mathbf{act}$ is the network state, E is the energy of the network, $c = \sum_{\mathbf{act}} \exp(-E(\mathbf{act})/kT)$ ensures the probabilities of all possible states equal one, $T > 0$ is a real number denoting the temperature of the system, and k is Boltzmann’s constant¹. We adopt the common simplification in machine learning by replacing kT with T , omitting Boltzmann’s constant.

The structure of the random process is defined constructively using Lean’s discrete `ProbabilityMassFunction` (PMF) monad, consistent with the main choice of structuring our neural network in a linearly ordered field.

We now make the notion of *reversibility* precise. A Markov kernel K_i is *reversible* with respect to a probability measure π if it satisfies the detailed balance condition (Def. 6.2.1, p. 235, [56]). Intuitively, at equilibrium the “probability flow” between any two states s and s' is symmetric: the probability of being in s times the transition probability to s' equals the flow back from s' to s . If each single-site kernel K_i is reversible with respect to π , so is the *random-scan kernel* $K_{\text{rand}}(x, A) = \frac{1}{n} \sum_{i=1}^n K_i(x, A)$.

A probability measure π is *invariant* for a kernel K if applying K leaves it unchanged: a system distributed as π stays distributed as π after any number of steps. Reversibility implies invariance (`IsReversible.invariant` [↗](#), Theorem 6.2.2, [56]): for all $A \subseteq \mathcal{X}$, $\pi(A) = \int_{\mathcal{X}} K(x, A) d\pi(x)$, so π is stationary for both the single-site updates K_i and their uniform mixture K_{rand} .

4.1 Perron–Frobenius theorem

Although `mathlib` provides the infrastructure for Markov kernels, it lacked a formal convergence theory for matrices; we supply the first Lean 4 formalization of the Perron–Frobenius theorem. Reversibility makes π invariant but guarantees neither convergence to π from an arbitrary starting state nor its uniqueness – the stronger property of *ergodicity*. For a finite state space, a Markov chain is ergodic (Theorem 6.2, [56]) iff it is *irreducible* (every state reaches every other) and *aperiodic* (no trapping in deterministic cycles). Convergence of finite Markov chains is then formalized via the Perron–Frobenius theorem for non-negative matrices (Theorem 1.5, [59]). Following Chapter 1 of [59], we assume familiarity with real-variable and matrix theory.

A non-negative matrix A is defined as *irreducible* if for every pair of indices (i, j) , there exists a power k such that the (i, j) -th entry of A^k is positive ([59], Def. 1.6).

Theorem 4.1 (`pft_irreducible`). [↗](#) *Suppose T is an $n \times n$ irreducible non-negative matrix. Then there exists an eigenvalue r such that:*

- (i) r is real, and $r > 0$;
- (ii) r can be associated with strictly positive left and right eigenvectors [↗](#);

¹ $k \approx 1.38 \cdot 10^{-23} \text{ J/K}$, that is, the unit is Joule (energy) divided by Kelvin (temperature).

- (iii) $r \geq |\lambda|$ for any eigenvalue λ of T [\[59\]](#);
- (iv) the eigenvectors associated with r are unique up to constant multiples [\[59\]](#);
- (v) if $0 \leq B \leq T$ and β is an eigenvalue of B , then $|\beta| \leq r$. Moreover, $|\beta| = r$ implies $B = T$.
- (vi) r is a simple root of the characteristic equation of T [\[59\]](#).

For an irreducible non-negative matrix, the theorem yields a positive real eigenvalue r (the Perron root) with a unique strictly positive eigenvector (up to scaling); for a non-negative column-stochastic matrix (column sums one), $r = 1$, and the corresponding right Perron eigenvector, normalized to lie in the standard simplex, is the unique stationary distribution. It further separates aperiodic from periodic matrices: if an irreducible matrix is aperiodic (or *primitive*), r is the unique eigenvalue of maximum modulus (p. 14, Theorem 1.1, [\[59\]](#), `pft_primitive`), whereas if it is periodic with period $d > 1$ there are exactly d such eigenvalues (p. 34, Theorem 1.7, [\[59\]](#)).

This is essential for ergodicity: aperiodicity ensures convergence to a unique stationary distribution, while periodicity yields cyclic behavior.

The Perron–Frobenius theorem is based on studying matrix powers, whose entries encode the number (or total weight) of fixed-length paths in the associated graph. This correspondence is central to *irreducibility*: it is defined algebraically by the positivity of matrix powers ($(A^k)_{ij} > 0$), yet reasoned about combinatorially via strong connectivity of the graph – the property that every ordered pair (u, v) is joined by a (possibly empty) directed path. Formally bridging these algebraic and combinatorial views posed technical challenges, requiring a robust API for path manipulation.

Design decisions We require directed edges, ruling out `SimpleGraph` [\[59\]](#), and multiple edges, ruling out `Digraph`, whose relational adjacency ([§2](#)) abstracts away the identity of individual edges and cannot support path construction and manipulation. `Graph` [\[59\]](#), similarly, is undirected. Our solution was to build on `mathlib`’s `Quiver` (directed multigraph), which explicitly models paths as sequences of specific, identifiable arrows ($a \rightarrow b$). We also realized that `Quiver` was not merely a tool for the proofs, but the correct abstraction for our neural networks themselves (see [§2](#)). We then define a function `Matrix.toQuiver` [\[59\]](#) that maps a matrix A to a `Quiver`, where an edge $i \rightarrow j$ exists if and only if $A_{ij} > 0$. The lemma `pow_apply_pos_iff_nonempty_path` [\[59\]](#) proves that a positive entry in A^k is equivalent to the existence of a path of length k . This allows us to prove essential properties, such as the strict positivity of eigenvectors of irreducible matrices (`eigenvector_is_positive_of_irreducible`) [\[59\]](#).

The analytic part of the proof (see p. 15 and Theorem 1.1, [\[59\]](#)) centers on the Collatz–Wielandt function:

Definition 4.2. *Let T be a non-negative $n \times n$ matrix. The Perron root r of T can be characterized by the Collatz–Wielandt formula:*

$$r = \sup_{\substack{\mathbf{x} \geq 0 \\ \mathbf{x} \neq \mathbf{0}}} \left(\min_{x_i > 0} \frac{(T\mathbf{x})_i}{x_i} \right) \quad \text{[59]}.$$

The main challenge is that this function is discontinuous where vector components vanish. We addressed this by proving it upper-semicontinuous on the standard simplex (non-negative vectors summing to 1), so a generalized Extreme Value Theorem (`exists_isMaxOn_of_upperSemicontinuousOn`) [\[59\]](#) yields a maximum (`exists_maximizer`) [\[59\]](#), which we show to be the Perron eigenvector (`maximizer_satisfies_le_mulVec`) [\[59\]](#). This lemma in turn gives the

Perron–Frobenius theorem for stochastic matrices (pp. 33–34, [59], `exists_positive_eigen_vector_of_irreducible_stochastic` [↗](#)) – the main ingredient for the ergodicity of Boltzmann machines.

4.2 Boltzmann machines

Boltzmann machines are a canonical example of stochastic networks. The energy function is identical to that of Hopfield networks (Equation (1)) and ensures that Gibbs updates satisfy detailed balance with respect to the Boltzmann distribution. We describe this construction following (Section 8.6, [37]) and [25]. Standard presentations divide neurons into *visible* neurons, which are held fixed to a pattern during the positive phase, and *hidden* neurons, which remain free to update and allow a richer distribution [37, 3].

Design choices. We keep the Hopfield-style topology of §3 ($U_{\text{in}} = U_{\text{out}} = U$, $U_{\text{hidden}} = \emptyset$): every neuron is visible and patterns are full activation states. We adopt the binary $\{0, 1\}$ convention standard for Boltzmann machines (`TwoState.ZeroOne` [↗](#)) rather than the $\{-1, +1\}$ bipolar states of Hopfield networks.

Following §2, we focus on a single neuron u and how changing its state affects the energy. For clarity we write updates as probabilities rather than kernels: for a state $\mathbf{act}$ and neuron u , $\mathbb{P}(act_u = act_u^{(\text{new})})$ abbreviates the conditional probability $\mathbb{P}(act_u = act_u^{(\text{new})} \mid \mathbf{act}_{\setminus u})$, where $\mathbf{act}_{\setminus u}$ is the activations of all neurons except u . We now consider the difference in energy between the two possible states of u , $act_u = 0$ and $act_u = 1$, while keeping the states of all other neurons fixed. This difference is

$$\Delta E_u = E_{act_u=1} - E_{act_u=0} = \theta_u - \sum_{v \in U \setminus \{u\}} w_{uv} act_v = \theta_u - \text{net}_u. \quad (4)$$

Writing the energies in terms of the Boltzmann distribution (Equation (3)) yields $\Delta E_u = -T \ln(\mathbb{P}(act_u = 1)) + T \ln(\mathbb{P}(act_u = 0))$, since the probability of u having activation 0 and the probability of it having activation 1 must sum to 1 (since there are only these two possible states). Solving for the conditional probability $\mathbb{P}(act_u = 1)$ of neuron u being active gives

$$\mathbb{P}(act_u = 1) = \frac{1}{1 + e^{\Delta E_u/T}}, \quad (5)$$

a logistic function of the (scaled) energy difference between its active and inactive states. The neuron activation is then set to 1 with the probability from Equation (5) and to 0 otherwise. Concretely, this can be implemented by sampling a random number x uniformly from the interval $[0, 1]$ and updating the neuron according to

$$act_u^{(\text{new})} = 1 \text{ if } x \leq \mathbb{P}(act_u = 1), \text{ else } 0. \quad (6)$$

It can be shown that the single-site Gibbs update `gibbsUpdateSingleNeuron` [↗](#) satisfies detailed balance by construction. Let $\mathbf{act}_{[u \rightarrow act_u^{(\text{new})}]}$ denote the state obtained by replacing a randomly selected neuron u 's activation in $\mathbf{act}$ with $act_u^{(\text{new})}$, leaving all others unchanged.

Theorem 4.2 (detailed_balance). [↗](#) *The single-site Gibbs update satisfies detailed balance with respect to the Boltzmann distribution π :*

$$\pi(\mathbf{act}) \mathbb{P}(act_u = act_u^{(\text{new})}) = \pi(\mathbf{act}_{[u \rightarrow act_u^{(\text{new})}]}) \mathbb{P}(act_u^{(\text{new})} = act_u).$$

Detailed balance makes π stationary for the Gibbs sampler but is local: it does not rule out other stationary distributions. We therefore establish the global property of *ergodicity*: the following theorem shows the sampler has a unique stationary state and converges to it, via the Perron–Frobenius theorem (see §4.1) for stochastic matrices.

Theorem 4.3 (`randomScan_ergodicUniqueInvariant`). *Let K denote the random-scan Gibbs kernel for a two-state neural network, and let M_K be its corresponding column-stochastic transition matrix. Let π be the Boltzmann distribution. The random-scan Gibbs sampler satisfies the following four properties:*

- (i) *The kernel K is reversible with respect to the Boltzmann distribution π .*
- (ii) *Every network state $\mathbf{act}$ has strictly positive self-transition probability, $(M_K)_{\mathbf{act},\mathbf{act}} > 0$, and hence the Markov chain is aperiodic.*
- (iii) *The transition matrix M_K is irreducible.*
- (iv) *There exists a unique probability vector $\mathbf{v}$ in the standard simplex such that $M_K \mathbf{v} = \mathbf{v}$.*

By (i) and the fact that reversibility implies invariance, established above, π is stationary for K . Under the finite-state identification of distributions with probability vectors and our column-stochastic convention, the probability vector corresponding to π is therefore a fixed point of M_K . Since this vector lies in the standard simplex, the uniqueness statement in (iv) implies that it equals $\mathbf{v}$. Hence π is the unique stationary distribution of the random-scan Gibbs sampler.

This construction generalizes Hopfield dynamics: updates are stochastic at high temperature (neurons may change their activation even when energy increases), but as $T \rightarrow 0$ the single-site Gibbs update (Equation (5)) becomes deterministic, recovering the asynchronous Hopfield dynamics.

Lemma 4.4 (`gibbs_update_tends_to_zero_temp_limit`). *As the inverse temperature goes to infinity ($1/T \rightarrow \infty$), the probability mass function of the single-site Gibbs update converges pointwise to the deterministic zero-temperature limit kernel.*

After sufficiently many Gibbs updates, the network reaches *thermal equilibrium*: each state’s probability depends only on its energy, not the initial state. It is natural to ask whether W and θ can be adjusted so the energy function’s distribution matches a given *sample of patterns*.

4.3 Boltzmann learning

Where Hebbian learning (§3) stores patterns as stable states in one shot, with retrieval by convergence (Theorem 3.1), Boltzmann learning instead chooses W and θ so that the stationary distribution approximates that of a given sample of patterns [3]; a *stochastic associative memory* assigning high probability to states resembling the stored ones. This succeeds only insofar as the sample is compatible with some Boltzmann distribution. The ergodicity result of §4.2 is a prerequisite: it guarantees the unique stationary distribution that makes the objective well posed.

A standard approach to achieve this task is to measure the difference between two distributions and perform *gradient descent* to minimize that measure (e.g. the Kullback–Leibler divergence [38]). However, we do not formalize this distribution-level objective or its derivation. Instead, the gradient result below is specialized to a single presented pattern $\mathbf{act}^{(0)}$:

learning should make the model assign it high probability $\pi(\mathbf{act}^{(0)})$, i.e. minimize the *negative log-likelihood*

$$\ell(W, \theta) = -\log \pi(\mathbf{act}^{(0)}) = \frac{1}{T} E(\mathbf{act}^{(0)}) + \log c \quad \color{blue}{\square}, \quad (7)$$

the model’s “surprise” at $\mathbf{act}^{(0)}$ (**negLogLik**). The partition function c couples all states, so the gradient of $\log c$ is an equilibrium average under π ; this is what produces the positive-negative phase structure below. We prove the corresponding gradient identity for this single-pattern objective in **Theorem 4.5**.

Standard Boltzmann learning is commonly described in terms of two phases [37, 3]: in the *positive phase*, statistics are averaged over the training distribution; in the *negative phase*, all neurons are updated freely until equilibrium. In each phase one collects statistics on activations: let $\mathbb{P}_u^+$ be the probability that neuron u is active in the positive phase, $\mathbb{P}_u^-$ the same in the negative phase, and $\mathbb{P}_{uv}^+$, $\mathbb{P}_{uv}^-$ the probabilities that neurons u and v are both active simultaneously in the positive and negative phase respectively. The learning update is [3]:

$$\Delta w_{uv} = \eta (\mathbb{P}_{uv}^+ - \mathbb{P}_{uv}^-), \quad \Delta \theta_u = -\eta (\mathbb{P}_u^+ - \mathbb{P}_u^-), \quad (8)$$

with learning rate $\eta > 0$. In our formalization, the positive phase is specialized to a point mass at a single presented pattern $\mathbf{act}^{(0)}$, and **Theorem 4.5** proves the corresponding single-pattern gradient result. Intuitively, if neuron u is active more often when a training example is presented than when the network runs freely, its equilibrium probability is too low; decreasing θ_u makes activation more likely. Similarly, if two neurons are active simultaneously more often under a presented example than under free running, their connection weight should increase, making them more likely to fire together.

The positive-negative update direction (8), **learningDirection** $\color{blue}{\square}$, is packaged as a verified **BM.LearningRule** $\color{blue}{\square}$: the positive phase is the point mass at the fixed pattern (**positivePhaseAtData** $\color{blue}{\square}$) and the negative phase is the Boltzmann equilibrium π (**negativePhaseMeasure** $\color{blue}{\square}$), which **Theorem 4.3** identifies as the unique stationary distribution, making the negative-phase averages well defined (**negativePhaseMeasure_is_gibbs_stationary** $\color{blue}{\square}$).

For the single pattern $\mathbf{act}^{(0)}$ the positive phase concentrates on it, so $\mathbb{P}_{uv}^+ = \text{act}_u^{(0)} \text{act}_v^{(0)}$ and $\mathbb{P}_u^+ = \text{act}_u^{(0)}$, while the negative-phase statistics are the equilibrium averages $\mathbb{P}_{uv}^- = \mathbb{E}_\pi[\text{act}_u \text{act}_v]$ and $\mathbb{P}_u^- = \mathbb{E}_\pi[\text{act}_u]$ (for $\{0, 1\}$ activations an activation probability equals its expectation). With these, **learningDirection** is exactly the gradient of ℓ from (7):

Theorem 4.5 (**hasGradientAt_negLogLik_scaled**). $\color{blue}{\square}$ For inverse temperature $\beta = 1/T$, the update of (8) is gradient descent on ℓ ; for weights,

$$\frac{\partial}{\partial w_{uv}} \log \pi(\mathbf{act}^{(0)}) = \beta (\text{act}_u^{(0)} \text{act}_v^{(0)} - \mathbb{E}_\pi[\text{act}_u \text{act}_v]) \quad (9)$$

Beyond the gradient, we prove the single-pattern loss is convex (**convexOn_negLogLik** $\color{blue}{\square}$) and that a step along **learningDirection** strictly decreases it when the data and model statistics differ (**negLogLik_descent_step_scaled** $\color{blue}{\square}$). Exact negative-phase statistics require long Gibbs runs; for large networks equilibrium is slow, which makes full Boltzmann learning costly [37]. A standard remedy is *contrastive divergence* (CD- k), which replaces equilibrium averages by statistics after k full Gibbs sweeps from the presented pattern; we formalize it and prove it convergent to the exact learning direction as $k \rightarrow \infty$ (**cdLearningDirection_tendsto_learningDirection** $\color{blue}{\square}$). We omit the construction here, as it is a practical approximation rather than part of the core development, and refer the reader to the repository.

4.4 Integration with Physlib and Optlib

We interpret the network energy as a Hamiltonian H – exactly the network energy for two-state networks – so stochastic updates sample a *canonical ensemble* (a system in thermal equilibrium at $T > 0$ favoring lower-energy states; p. 251, [20]); the `IsHamiltonian` typeclass [↗](#) links `NeuralNetwork` to `CanonicalEnsemble` [↗](#), embedding our networks in Physlib’s statistical mechanics.

The learning side connects to `Optlib` [40] the same way: we prove and package the loss and gradient of the single-pattern Gibbs negative log-likelihood ([Theorem 4.5](#), `convexOn_negLogLik`), with the fixed-step update `gradientDescentFixStep` [↗](#), in `OptlibBridge.Objective`. We do not import `Optlib`; the bridge records only the expected interface, so a verified multi-step procedure can later reuse its convergence theorems without reproving gradient descent.

5 Related Work

Beyond the related work in §1 and §2, the closest to ours in convergence is Murphy et al. [50], who formalized and proved convergence of the one-layer perceptron and its averaged variant in Rocq. Their convergence proof establishes a finite bound on the number of misclassifications made during perceptron training on linearly separable data, whereas our Hopfield convergence proof uses the energy function and the finiteness of the state space to show that the network state stabilizes under asynchronous updates. Otherwise, machine learning formalizations remain sparse: PAC learnability for decision stumps in Lean [71, 62], the expressiveness of deep learning in Isabelle/HOL [9], AI planning via a SAT encoder [2], and, in Rocq, Dvoretzky’s convergence theorem, value/policy iteration [73, 72], expectation semantics, and support vector machines [10].

Formalizations inspired by the sequential perspective include the sequential models in Isabelle/HOL [15, 16] and the `kernel` representation in Rocq’s MLCERT [8]. This approach suits verification goals that closely reflect real-world implementations, where interoperability with external frameworks matters. Prior Isabelle/HOL work [15] used dual encodings – one graph-based for proofs, one for computation – enabling formal reasoning alongside on-demand execution in frameworks such as TensorFlow, and, inspired by it, a Rocq development [5] incorporated external MILP and SMT solvers for similar objectives. For interoperability, [8] relied on an unverified Python script to import models, whereas [15] proved correctness for each TensorFlow import. The sequential model also suits localized computations such as tensor operations or backpropagation correctness (Certigrad [58]); and in proof assistants enforcing provable termination, such as Rocq, feedforward networks can be defined inductively and evaluated by structural recursion, which ensures termination [8]. However, the sequential layer paradigm struggles with arbitrary topologies, especially the cycles of recurrent networks, and even general computational graphs (DAGs) often need complex unfolding or fixed-point semantics to faithfully model true recurrence. Concurrently, TorchLean [26] unifies specification, execution, and verification of neural networks in Lean 4; its Hopfield case study builds on the present work [19], complementing our focus on global recurrent/stochastic dynamics.

Probability theory has been formalized across several proof assistants – Doob’s martingale convergence in Lean [74], limit theorems and hidden Markov chains in HOL4 [52, 43], discrete probability in Mizar [53], Shannon’s theorems and randomized-concurrency logics in Rocq [4, 61], and measure-theoretic results including the central limit theorem and the law of large numbers in Isabelle/HOL [7, 23, 33]. Closest to our Markov-chain infrastructure is Hölzl’s

Isabelle/HOL formalization of Markov kernels via the Giry monad and the Ionescu–Tulcea extension theorem [32]; in contrast, we target finite-state kernels with reversibility and a Perron–Frobenius–based convergence theory aimed at the ergodicity of the dynamics.

The most extensive prior formalization of Perron–Frobenius, by Thiemann et al. [64] in Isabelle/HOL, is topological, establishing the Perron eigenvector via Brouwer’s fixed-point theorem. Ours is, to our knowledge, the first analytic formalization, via the Collatz–Wielandt formula (4.2): we prove its upper-semicontinuity, identify its maximizer on the standard simplex with the Perron eigenvector, and give novel proofs of uniqueness and spectral dominance from the phase alignment of complex eigenvectors, distinct from the topological method. Where Thiemann et al. bridge separate matrix libraries [29, 65], we work in `mathlib`’s unified linear algebra, graph theory, and analysis, enabling a novel characterization of irreducibility via quiver strong connectivity (§4.1). Finally, their development targets the full peripheral spectrum and reducible matrices for program analysis, whereas ours is a foundation for Markov-chain ergodicity and MCMC.

Closer to the natural sciences, a growing body of work formalizes physics in proof assistants. The largest Lean 4 effort, `Physlib` [68], spans electromagnetism, quantum mechanics, and statistical mechanics, with index notation [66] and Wick’s theorem [67], and even uncovered an error in the published two-Higgs-doublet-model literature [69]; Lean-QuantumInfo [46] underpins a machine-checked generalized quantum Stein’s lemma [48]. Beyond Lean, chemical thermodynamics and adsorption have been formalized [11], and relativity in Isabelle/HOL [60, 57] and Rocq [44].

6 Conclusion and future work

This paper presents a multigraph-based framework for neural networks spanning recurrent and feedforward architectures, showing that proof assistants can faithfully capture the interplay of their combinatorial, probabilistic, and thermodynamic aspects, and establishing a verified bridge between statistical mechanics, machine learning, and formal mathematics – to our knowledge, the first to be mechanized. We outline several directions it opens below.

On the probabilistic side, a general MCMC library would extend our Markov chain formalization. This requires completing the Perron–Frobenius theorem for general non-negative matrices and the Fundamental Theorem of Markov Chains in `mathlib`, providing a foundation for advanced convergence analysis [49] and for verifying sampling algorithms beyond the single-site Gibbs updates formalized here, such as Metropolis–Hastings [30] and simulated annealing (p. 142, Example 4.2.4, [56]).

On the learning side, we formalize Boltzmann learning on fully visible $\{0, 1\}$ networks with a simplified positive phase and single-pattern gradients (§4.3). Natural extensions include visible/hidden architectures (only hidden neurons equilibrate, visible pattern fixed), a Kullback–Leibler derivation linking phase statistics to gradient descent on the data distribution, and restricted Boltzmann machines. Completing the `OptlibBridge` (§4.4) would yield machine-checked multi-step convergence for the verified learning direction; deeper `Physlib` integration would let us formalize other convergence properties across deterministic and stochastic settings.

On the computational side, our formalization of discrete Hopfield networks is executable but inefficient; still, it can serve as a target specification for verifying a realistic implementation by refinement, in Lean 4 or a more hardware-oriented framework outside of Lean. Making it fully computable would require computable reals such as Rocq’s CoRN [21], alongside Coquelicot [12] for the analysis and Flocq [13] for floating point; in Lean, Meiburg’s computable reals [47] are a promising option.

Acknowledgments

We thank the anonymous reviewers for their constructive suggestions; Britt Anderson for proposing this project; and Niels van der Weide for his valuable feedback on earlier versions of this manuscript.

References

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI’16, page 265–283, USA, 2016. USENIX Association.
- [2] Mohammad Abdulaziz and Friedrich Kurz. Verified SAT-Based AI Planning. *Archive of Formal Proofs*, October 2020. https://isa-afp.org/entries/Verified_SAT_Based_AI_Planning.html, Formal proof development.
- [3] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. A Learning Algorithm for Boltzmann Machines. *Cognitive Science*, 9(1):147–169, 1985.
- [4] Reynald Affeldt and Manabu Hagiwara. Formalization of Shannon’s Theorems in SSReflect-Coq. In *3rd Conference on Interactive Theorem Proving (ITP 2012)*, volume 7406 of *Lecture Notes in Computer Science*, pages 233–249. Springer, 2012.
- [5] Andrei Aleksandrov and Kim Völlinger. Formalizing Piecewise Affine Activation Functions of Neural Networks in Coq. In *NASA Formal Methods: 15th International Symposium, NFM 2023, Houston, TX, USA, May 16–18, 2023, Proceedings*, page 62–78, Berlin, Heidelberg, 2023. Springer-Verlag.
- [6] Britt Anderson. Computational Neuroscience in Lean?, September 2024. Zulip message.
- [7] Jeremy Avigad, Johannes Hölzl, and Luke Serafin. A Formally Verified Proof of the Central Limit Theorem. *Journal of Automated Reasoning*, 59(4):389–423, 2017.
- [8] Alexander Bagnall and Gordon Stewart. Certifying the True Error: Machine Learning in Coq with Verified Generalization Guarantees. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):2662–2669, Jul. 2019.
- [9] Alexander Bentkamp, Jasmin Christian Blanchette, and Dietrich Klakow. A Formal Proof of the Expressiveness of Deep Learning. *Journal of Automated Reasoning*, 63:347–368, 2019.
- [10] Sooraj Bhat. *Syntactic Foundations for Machine Learning*. PhD thesis, Georgia Institute of Technology, 2013.
- [11] Maxwell P. Bobbin, Samiha Sharlin, Parivash Feyzishendi, An Hong Dang, Catherine M. Wraback, and Tyler R. Josephson. Formalizing chemical physics using the Lean theorem prover. *Digital Discovery*, 3:264–280, 2024.
- [12] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A User-Friendly Library of Real Analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, March 2015.
- [13] Sylvie Boldo and Guillaume Melquiond. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 243–252, 2011.
- [14] Elena Botoeva, Panagiotis Kouvaros, Jan Kronqvist, Alessio Lomuscio, and Ruth Misener. Efficient Verification of ReLU-Based Neural Networks via Dependency Analysis. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):3291–3299, Apr. 2020.
- [15] Achim D. Brucker and Amy Stell. Verifying Feedforward Neural Networks for Classification in Isabelle/HOL. In *Formal Methods: 25th International Symposium, FM 2023, Lübeck, Germany, March 6–10, 2023, Proceedings*, page 427–444, Berlin, Heidelberg, 2023. Springer-Verlag.

- [16] Achim D. Brucker and Amy Stell. Formalizing neural networks. *Archive of Formal Proofs*, November 2025. https://isa-afp.org/entries/Neural_Networks.html, Formal proof development.
- [17] Rudy R Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and Pawan K Mudigonda. A Unified View of Piecewise Linear Neural Network Verification. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.
- [18] Kerem Yunus Camsari, Rafatul Faria, Brian M. Sutton, and Supriyo Datta. Stochastic p -Bits for Invertible Logic. *Physical Review X*, 7(3):031014, 2017.
- [19] Matteo Cipollina, Michail Karatarakis, and Freek Wiedijk. Formalized Hopfield Networks and Boltzmann Machines. arXiv:2512.07766 [cs.LG], <https://arxiv.org/abs/2512.07766>, December 2025. Preliminary version of the present paper.
- [20] James P. Coughlin and Robert H. Baran. *Neural Computation in Hopfield Networks and Boltzmann Machines*. Rowman & Littlefield, 1995.
- [21] Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-CoRN, the Constructive Coq Repository at Nijmegen. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *Mathematical Knowledge Management*, pages 88–103, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [22] Remi Desmartin, Omri Isac, Grant Passmore, Ekaterina Komendantskaya, Kathrin Stark, and Guy Katz. A Certified Proof Checker for Deep Neural Network Verification in Imandra. In Yannick Forster and Chantal Keller, editors, *16th International Conference on Interactive Theorem Proving (ITP 2025)*, volume 352 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:21, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [23] Manuel Eberl. The Laws of Large Numbers. *Archive of Formal Proofs*, 2021. https://isa-afp.org/entries/Laws_of_Large_Numbers.html.
- [24] Rüdiger Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verification and Analysis*, pages 269–286, Cham, 2017. Springer International Publishing.
- [25] Stuart Geman and Donald Geman. Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):721–741, 1984.
- [26] Robert Joseph George, Jennifer Cruden, Will Adkisson, Xiangru Zhong, Huan Zhang, and Anima Anandkumar. TorchLean: Formalizing Neural Networks in Lean. arXiv:2602.22631, 2026. <https://github.com/lean-dojo/TorchLean>.
- [27] Fred Glover, Gary Kochenberger, Rick Hennig, and Yu Du. Quantum bridge analytics i: a tutorial on formulating and using QUBO models. *Annals of Operations Research*, 314:141–183, 2022.
- [28] Xinjie Guo, Farnood Merrikh-Bayat, Ligang Gao, Brian D. Hoskins, Fabien Alibart, Bernabe Linares-Barranco, Luke Theogarajan, Christof Teuscher, and Dmitri B. Strukov. Modeling and Experimental Demonstration of a Hopfield Network Analog-to-Digital Converter with Hybrid CMOS/Memristor Circuits. *Frontiers in Neuroscience*, 9, 2015.
- [29] John Harrison. The HOL Light Theory of Euclidean Space. *Journal of Automated Reasoning*, 50, 02 2013.
- [30] W. K. Hastings. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika*, 57(1):97–109, 04 1970.
- [31] Donald O. Hebb. *The organization of behavior: A neuropsychological theory*. Wiley, New York, June 1949.
- [32] Johannes Hölzl. Markov Processes in Isabelle/HOL. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017*, page 100–111, New York, NY, USA, 2017. Association for Computing Machinery.
- [33] Johannes Hölzl, Andreas Lochbihler, and Dmitriy Traytel. A Zoo of Probabilistic Systems. *Archive of Formal Proofs*, 2015. https://isa-afp.org/entries/Probabilistic_System_Zoo.html.

- [34] John J Hopfield. Neural Networks and Physical Systems with Emergent Collective Computational Abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.
- [35] Ernst Ising. Contribution to the Theory of Ferromagnetism. *Z. Phys.*, 31:253–258, 1925.
- [36] Olav Kallenberg. *Foundations of Modern Probability*. Probability and its Applications (New York). Springer-Verlag, New York, second edition, 2002.
- [37] Rudolf Kruse, Sanaz Mostaghim, Christian Borgelt, Christian Braune, and Matthias Steinbrecher. *Computational Intelligence - A Methodological Introduction, 3rd edition*. Springer Verlag, March 2022.
- [38] Solomon Kullback and Richard A. Leibler. On Information and Sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, 1951.
- [39] Can Li, Jim Ignowski, Xia Sheng, Rob Wessel, Bill Jaffe, Jacqui Ingemi, Cat Graves, and John Paul Strachan. CMOS-integrated nanoscale memristive crossbars for CNN and optimization acceleration. In *2020 IEEE International Memory Workshop (IMW)*, pages 1–4, 2020.
- [40] Chenyi Li, Ziyu Wang, Wanyi He, Yuxuan Wu, Shengyang Xu, and Zaiwen Wen. Formalization of Complexity Analysis of the First-order Algorithms for Convex Optimization. *Journal of Automated Reasoning*, 2025. arXiv:2403.11437; library at <https://github.com/optsuite/optlib>.
- [41] Wang Lin, Zhengfeng Yang, Xin Chen, Qingye Zhao, Xiangkun Li, Zhiming Liu, and Jifeng He. Robustness Verification of Classification Deep Neural Networks via Linear Programming. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11410–11419, 2019.
- [42] Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher Strong, Clark Barrett, and Mykel J. Kochenderfer. Algorithms for Verifying Deep Neural Networks. *Foundations and Trends in Optimization*, 4(3-4):244–404, 2021.
- [43] Liya Liu, Vincent Aravantinos, Osman Hasan, and Sofiène Tahar. On the Formal Analysis of HMM Using Theorem Proving. In Stephan Merz and Jun Pang, editors, *Formal Methods and Software Engineering*, pages 316–331, Cham, 2014. Springer International Publishing.
- [44] Eric Hanqing Lu. A Formalization of Elements of Special Relativity in Coq. Senior thesis, Harvard University, 2017. <http://nrs.harvard.edu/urn-3:HUL.InstRepos:38811518>.
- [45] Andrew Lucas. Ising formulations of many NP problems. *Frontiers in Physics*, 2, 2014.
- [46] Alex Meiburg. Quantum Information in Lean. <https://github.com/Timeroot/Lean-QuantumInfo>, 2024. GitHub repository.
- [47] Alex Meiburg. ComputableReal: Computable implementation of real numbers in Lean 4. <https://github.com/Timeroot/ComputableReal>, 2025.
- [48] Alex Meiburg, Leonardo A. Lessa, and Rodolfo R. Soldati. A Formalization of the Generalized Quantum Stein’s Lemma in Lean, 2025.
- [49] Sean Meyn, Richard L. Tweedie, and Peter W. Glynn. *Markov Chains and Stochastic Stability*. Cambridge Mathematical Library. Cambridge University Press, 2 edition, 2009.
- [50] Charlie Murphy, Patrick Gray, and Gordon Stewart. Verified Perceptron Convergence Theorem. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 43–50, 2017.
- [51] Shaila Niazi, Shuvro Chowdhury, Navid Anjum Aadit, Masoud Mohseni, Yao Qin, and Kerem Y. Camsari. Training deep Boltzmann networks with sparse Ising machines. *Nature Electronics*, 7:610–619, 2024.
- [52] Michael Norrish and Konrad Slind. *The HOL System DESCRIPTION*. The HOL4 Contributors, 3rd edition, 2022. <https://hol-theorem-prover.org/documentation.html>.
- [53] Hiroyuki Okazaki, Yuichi Futa, and Yasunari Shidama. Formal Definition of Probability on Finite and Discrete Sample Space for Proving Security of Cryptographic Systems Using Mizar. *Artificial Intelligence Research*, 2, 2013.
- [54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan,

- Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*, pages 8024–8035. Curran Associates, Inc., 2019.
- [55] Hubert Ramsauer, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Thomas Adler, Lukas Gruber, Markus Holzleitner, Milena Pavlović, Geir Kjetil Sandve, Victor Greiff, David Kreil, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. Hopfield Networks is All You Need, 2021.
- [56] Christian P. Robert and George Casella. *Monte Carlo Statistical Methods*. Springer Texts in Statistics. Springer, 1999.
- [57] Richard Schmoetten, Jake E. Palmer, and Jacques D. Fleuriot. Towards Formalising Schutz’ Axioms for Minkowski Spacetime in Isabelle/HOL. *Journal of Automated Reasoning*, 66:715–749, 2022.
- [58] Daniel Selsam, Percy Liang, and David L. Dill. Developing Bug-Free Machine Learning Systems with Formal Mathematics. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70, ICML’17*, page 3047–3056. JMLR.org, 2017.
- [59] E. Seneta. *Non-negative Matrices and Markov Chains*. Springer, New York, 2006.
- [60] Mike Stannett and István Németi. Using Isabelle/HOL to Verify First-Order Relativity Theory. *Journal of Automated Reasoning*, 52(4):361–378, 2014.
- [61] Joseph Tassarotti and Robert Harper. A Separation Logic for Concurrent Randomized Programs. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.
- [62] Joseph Tassarotti, Koundinya Vajjha, Anindya Banerjee, and Jean-Baptiste Tristan. A Formal Proof of PAC Learnability for Decision Stumps. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2021*, page 5–17, New York, NY, USA, 2021. Association for Computing Machinery.
- [63] The Royal Swedish Academy of Sciences. Press release: The Nobel Prize in Physics 2024. <https://www.nobelprize.org/prizes/physics/2024/press-release/>, 2024. Accessed 2025-01-05.
- [64] René Thiemann. Stochastic Matrices and the Perron-Frobenius Theorem. *Archive of Formal Proofs*, November 2017. http://isa-afp.org/entries/Stochastic_Matrices.html, Formal proof development.
- [65] René Thiemann and Akihisa Yamada. Formalizing Jordan normal forms in Isabelle/HOL. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2016*, page 88–99, New York, NY, USA, 2016. Association for Computing Machinery.
- [66] Joseph Tooby-Smith. Formalization of physics index notation in Lean 4, 2024.
- [67] Joseph Tooby-Smith. Digitalizing Wick’s theorem. arXiv:2505.07939 [hep-th], 2025.
- [68] Joseph Tooby-Smith. HepLean: Digitalising high energy physics. *Computer Physics Communications*, 308:109457, 2025.
- [69] Joseph Tooby-Smith. Formalizing the stability of the two Higgs doublet model potential into Lean: identifying an error in the literature, 2026.
- [70] Hoang-Dung Tran, Stanley Bak, Weiming Xiang, and Taylor T. Johnson. Verification of Deep Convolutional Neural Networks Using Imagestars. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 18–42, Cham, 2020. Springer International Publishing.
- [71] Jean-Baptiste Tristan, Joseph Tassarotti, Koundinya Vajjha, Michael L. Wick, and Anindya Banerjee. Verification of ML Systems via Reparameterization, 2020.
- [72] Koundinya Vajjha, Avraham Shinnar, Barry Trager, Vasily Pestun, and Nathan Fulton. CertRL: formalizing convergence proofs for value and policy iteration in Coq. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2021*, page 18–31, New York, NY, USA, 2021. Association for Computing Machinery.

- [73] Koundinya Vajjha, Barry Trager, Avraham Shinnar, and Vasily Pestun. Formalization of a Stochastic Approximation Theorem. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [74] Kexing Ying and Rémy Degenne. A Formalization of Doob’s Martingale Convergence Theorems in Mathlib. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, page 334–347, New York, NY, USA, 2023. Association for Computing Machinery.