Exploring Emergent Properties of Complex Systems using Machine Learning

Finlay Sanders

August 2023

1 Introduction

Many natural phenomena display properties or behaviours more than the mere aggregation of their parts. Humans, for instance, are capable of language, cognition and intricate social behaviours, none of which are properties of individual cells. Similarly, each cell's functionality arises from the interactions between molecules, even though none possess the cell's capabilities independently. This pattern, where macroscopic properties arise from interactions between microscopic components, termed 'emergence', is a hallmark of complex systems. Emergence creates layers of abstraction within a system, where each behaves according to its own physical laws.

Formal theories of emergence have already been introduced using information theory, such as in [5]. The contribution of this paper is a novel method of identifying emergence using machine learning. By approximating the dynamics of a complex system at different spatiotemporal scales, I confirm numerically that these layers of abstraction exist, and that the dynamics of each can be learned by a data-driven approach.

I evaluate this method using the Classical XY model, a lattice model of statistical mechanics relevant to phenomena such as the melting of crystals, magnetism and superconductivity, as an example. At the microscopic scale, the model consists of a collection of spins on a lattice that can point in any direction in the plane, which operate according to the dynamics of equation 4. At the macroscopic scale, the model is characterised by emergent structures termed 'vortices' and 'anti-vortices', which describe topological flaws where groups of spins make a 2π rotation either clockwise or anticlockwise, that follow Coulomb dynamics.

To this end, I propose a dual pathway approach to predicting the trajectories of spins and vortices using graph neural networks. First, I trained a model to predict spin dynamics, from which the vortices could be extracted. Second, I trained a model that bypasses spins, instead directly predicting vortex movements. By drawing parallels to commutativity diagrams, I demonstrate that both pathways converge to accurate vortex predictions, even over extended rollouts.

2 Background

2.1 XY Model Simulation

A vast amount of data is needed to train any machine-learning model. To this end, many instances of the XY model must be simulated over many time steps to provide data with which the dynamics of spins and vortices can be learned. I chose to simulate the XY model with periodic boundary conditions (PBCs), meaning that spins on a given boundary are adjacent to spins at the opposite boundary. In essence, PBCs let a small system approximate an infinite one, as all spins have an equal amount of neighbours. Discussed below are two methods of simulating the XY model, the latter of which was ultimately chosen.

Metropolis-Hastings Algorithm

The Metropolis-Hastings algorithm, a Markov Chain Monte Carlo method, is well suited for simulating the XY model. The term 'Markov Chain' indicates that the state of the XY model at time $t+1$ is solely influenced by its state at time t. 'Monte Carlo' means that spins change according to a specified probability distribution. To this end, we assume the spins interact according to the energy given by the Hamiltonian (an equation for the sum of the system's energy), as discussed in [4]:

$$H = -J \sum_{j(i)}^{4} \cos\left(\theta_i - \theta_j\right) \tag{1}$$

where J is the coupling constant and $j(i)$ denotes summation over each of the 4 nearest neighbours, j, of lattice site i. Intuitively, the interaction energy of a pair of spins i, j is minimised when $\theta_i = \theta_j$. The algorithm for updating a single spin proceeds as follows:

1. Pick a spin θ_i on the lattice
2. Change the angle of the selected spin by some small amount $\Delta\theta$ to produce a new angle θ_i'
3. Calculate the energy change in the system due to this proposed change using the Hamiltonian:

$$\Delta E = E_{new} - E_{old} = J \sum_{j(i)}^{4} [\cos(\theta_i - \theta_j) - \cos(\theta_i' - \theta_j)] \tag{2}$$

4. Choose whether to accept the proposed change. If $\Delta E < 0$ we accept the change because it lowers the energy. If $\Delta E > 0$ we accept the change with probability:

$$P = e^{-\frac{\Delta E}{T}} \tag{3}$$

By iterating over all spins on the lattice at time t, we reach a subsequent state of the system at time $t+1$. Although reasonable, this method isn't preferred due to the inconsistency of its spin updates over extended time periods.

Numerical Integration

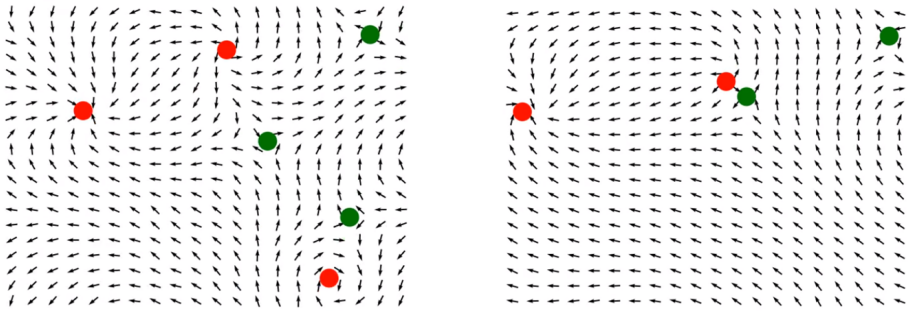
Numerical integration, as detailed in [2], produces predictable and therefore learnable, rollouts of the XY model by eliminating all randomness in spin updates. By altering the Hamiltonian above and introducing a kinetic energy term, they derived the following equations of motion:

$$\ddot{\theta}_i(t) = -\sum_{j(i)}^{4} \sin\left[\theta_i(t) - \theta_j(t)\right] \tag{4}$$

Intuitively, the right-hand side of this equation gives us the angular acceleration of each spin. To transition the state of an XY model from time t to $t+1$, we first update each spin's angular velocity by adding the computed angular accelerations. Then, we adjust each spin's angle by adding their velocities. I found that slightly damping the added velocities led to more stable vortex movements and removed the possibility of vortices spontaneously appearing.

Final Intuitions

Figure 1 is an example of how the XY model evolves over time. Vortices and anti-vortices are marked with red and green dots respectively.



XY Model at time t XY model at time $t+10$

Figure 1: An example of the XY model's spins at times t and $t+10$.

The most fundamental pattern of the system is that vortices of the same type repel and those of opposite type attract, although this is not always obvious. The first source of confusion is that the two vortices in the bottom right have disappeared, or 'annihilated', in the $t+10$ configuration. This happens when vortices inevitably meet, cancel out and allow the spins they previously distorted to become uniform. The next difficulty is that the vortices on the far left

and right boundaries appear to have moved away from each other, despite my previous claim that they should meet and annihilate. This is caused by the previously discussed periodic boundary conditions, which allow the vortices to attract across boundaries.

2.2 Graph Neural Networks

Neural Networks (NNs), a pillar of machine learning, are proven to be capable of representing any continuous function given an appropriate architecture [1]. In a trained network, input data is transformed according to internal parameters to produce useful output such as a prediction or classification. However, traditional feed-forward neural networks don't inherently capture structured relationships between input data points, like the local interactions that are crucial to the XY model.

Graph Neural Networks (GNNs) are designed to work on data structured as graphs. These networks not only consider data points but also the relationships between them. Equipped with in-built assumptions that prioritize relationships (often termed as 'relational inductive biases') [3], GNNs are particularly adept at handling systems where interactions between individual components are paramount, making them especially suitable for approximating complex systems like the XY model.

Neural Networks

Neural networks are often visualised as interconnected layers of neurons. The type of traditional neural network I used to create graph neural networks is called a multi-layer perceptron, where all nodes in layer l are connected to those in layer $l+1$. The first and last layers are the input and output layers and any intermediate layers are called hidden layers.

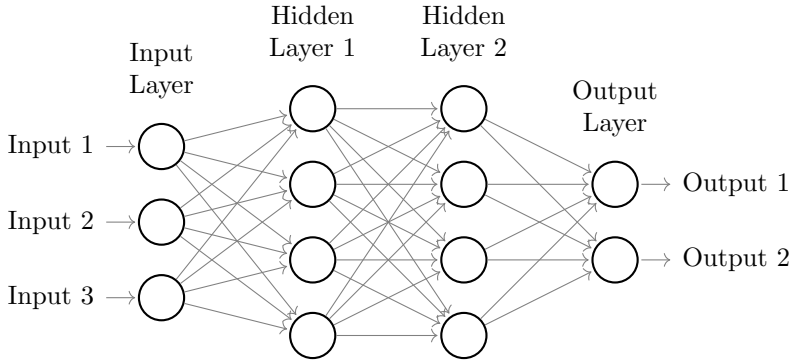


Figure 2: A multi-layer perceptron which takes 3 inputs, has 2 hidden layers with 4 neurons each and produces 2 outputs.

Each neuron has a set of parameters, specifically a weight for each of its inputs and a single bias, which transform its inputs from the previous layer into an output. We then apply a non-linear activation function which enables it to approximate non-linear functions. The operation can be summarised as:

$$Output = Activation(Inputs \times weights + bias) \tag{5}$$

The output of each neuron in layer l becomes one of the inputs to each neuron in layer $l+1$. Repeating this until we get a result from the output layer is called forward propagation.

To improve our network, we first need to quantify the difference between its outputs and the expected outputs using a loss function. For instance, the mean-squared error E, of neuron i's output, y_i, is:

$$E_{y_i} = \frac{(Expected\ Output - Network\ Output)^2}{2} \tag{6}$$

Now we traverse backward through the network, using the chain rule to quantify each parameter's contribution to the combined error, E_T, with which they will be updated. For each neuron, if z is the input to its activation function and h is its activated output, then each weight, w's, contribution is given by:

$$\frac{\partial E_T}{\partial w} = \frac{\partial E_T}{\partial h} \times \frac{\partial h}{\partial z} \times \frac{\partial z}{\partial w} \tag{7}$$

Once we have all the necessary derivatives, we update each neuron's weights according to a learning rate, α. The learning rate determines the step size in the opposite direction of the gradient. Too large a value may cause the parameters to over-correct, missing their optimal values and too small a value might make the training process excessively slow. For example, we can update each weight, w, with this equation:

$$w = w - \alpha \times \frac{\partial E_T}{\partial w} \tag{8}$$

A similar process is repeated for computing the gradients for and updating biases.

After many iterations of this 'training', the network will converge on ideal parameters for its task. In practice, we often 'mini-batch' inputs, or pass a group of them into the network simultaneously, to improve training stability and computational efficiency.

Graphs

A graph is a data structure defined by a set of nodes and edges. Nodes can be seen as individual entities, while edges depict interactions or relationships between these entities.

In figure 3, nodes x_1, x_2 and x_3 have associated features v_1, v_2 and v_3, which define numerical or categorical properties. The edges also have features,

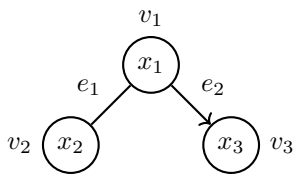


Figure 3: A graph with 3 nodes (x_1, x_2, and x_3) connected by an undirected edge e_1 and a directed edge e_2.

e_1 and e_2, which might encompass the nature or strength of the relationship between the nodes they connect. The arrowed or 'directed' edge between v_1 and v_2 denotes a one-way influence. The edge between v_2 and v_1 is 'undirected', denoting mutual influence. Undirected edges are comprised of two directed edges in opposite directions, with the same feature. Graphs can also have global features, representing holistic properties, but I have excluded them as they are not relevant to the methods in this paper.

GNN Implementation

Graph neural networks combine the concepts of traditional neural networks and graph theory. They take a graph as input and return an updated graph with the same topology, but different features, in a way that reflects the structure of the input graph. The method explained below is detailed in [3].

We begin by updating the features of each edge. For each undirected edge with feature, e_k, we use the update function ϕ^e to produce a new edge feature e'_k.

$$e'_k = \phi^e(e_k, v_{r_k}, v_{s_k}, u) \tag{9}$$

Here, v_{r_k} is the receiver node's feature, v_{s_k} is the sender node's feature and u is the global feature. By sender and receiver, we mean the node from which the directed edge originates, and the node it points to. To update an undirected edge, we update each of its constituent directed edges and then average them to produce a single feature.

Next, we update the features of each node in a three-step process. First, for each node i, we collect into a group, E'_i, its own feature, r_i, and the features of all of its incoming edges, s_k, from each node, k.

$$E'_i = \{(e'_k, r_i, s_k)\} \tag{10}$$

We pass these groups into a neural network, giving us an updated embedding for each node that considers its incoming edges. Next, for each node i, we aggregate all of the embeddings of the nodes to which it is connected to produce a new embedding, $\bar{e}'_i$, with the function $\rho^{e\to v}$.

$$\bar{e}'_i = \rho^{e \to v}(E'_i) \tag{11}$$

The aggregation function often computes the sum or mean of the features it takes as input. Finally, we update each node i's feature, v_i, with the update function ϕ^v using the embedding created in the previous equation.

$$v'_i = \phi^v(\bar{e}_i, v_i, u) \tag{12}$$

Now we can reconstruct our graph. Each edge, k, now has the feature e'_k, and each node i, now has the feature v'_i.

To create mini-batches of graphs we combine the constituent graphs into a larger, disconnected graph. This lets us use our original update and aggregation methods, no matter the mini-batch size.

3 Method

I approached the problem of predicting the trajectories of vortices using graph neural networks from 2 directions. First, I trained a GNN, the 'spin model', that predicts the state of spins, S, at time $t+1$ from those at time t. Extracting vortices from these spins gives us the first set of predicted vortex trajectories. Second, I trained a GNN, the 'vortex model', that predicts the state of vortices directly, V, at time t from those at time t, giving us another set of trajectories. The approach is summarised in figure 4:

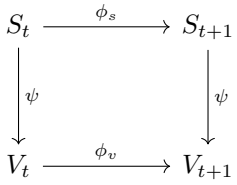


Figure 4: A commutative diagram where ψ is the function that extracts vortex positions from spins, and ϕ_s and ϕ_v are update functions approximated by the spin and node models respectively.

If both models' predictions are accurate, all paths from S_t should yield the same result, V_{t+1}, and should be similar to the movements of vortices in the actual XY model simulation.

I implemented both GNNs using the Pytorch Geometric (PyG) library for ease of use and efficient computation, and all of the accompanying code in Python.

3.1 Vortex Extraction

To find the positions of vortices in the XY model, we consider spins in square groups of four, termed 'plaquettes' as shown in figure 5.

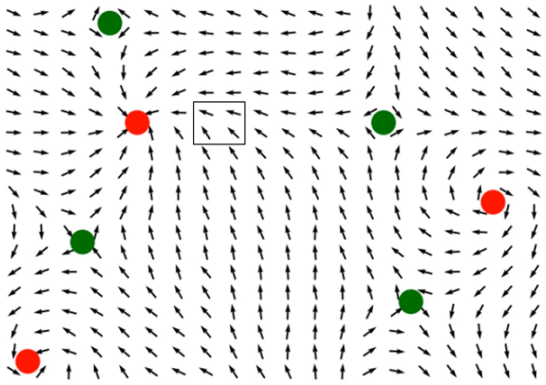


Figure 5: An instance of the XY model, where a box outlines a plaquette.

We define the circulation around a plaquette as the sum of the phase differences along its perimeter. The circulation around a plaquette is $+2\pi$ if there is a vortex and -2π if there is an anti-vortex.

3.2 Spin Model

The spin model takes as input the spins of the XY model at time t and outputs a prediction of their state at time $t + 1$.

Spin Graph Representation

In the XY model, spins only interact with their four nearest neighbours, so can be represented by a graph like in figure 6.

In this graph, each spin is a node connected by undirected edges to its neighbours. Crucially, there are edges connecting nodes that are neighbours across boundaries, such as θ_5 and θ_9, even though they are not shown. Each spin, i, with angle, θ_i, and angular velocity, ω_i, has a feature, v_i, where:

$$v_i = (\sin(\theta_i), \cos(\theta_i), \omega_i) \tag{13}$$

Including both $\sin(\theta_i)$ and $\cos(\theta_i)$ accounts for the periodicity of angles. I found that no global or edge features were necessary for the model to make accurate predictions, so have discluded them.

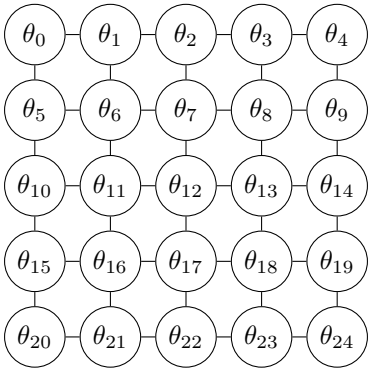


Figure 6: A graph representation of the spins of an instance of the XY model of size 5x5. It is important to note that the XY model simulated for data generation was size 20x20, and that its graph representation follows the same pattern of connectivity.

Data Preparation

Data for the dataset was obtained by simulating many instances of the XY model, each initialized randomly, using numerical integration. For each instance of the XY model, I created a trajectory, or sequence of graphs like the one above, that captures the state of the XY model's spins at consecutive time steps. If S_t is the graph representing the state of the spins at time t, then each trajectory, T, over n time steps can be expressed as:

$$T = \{S_0, S_1, ..., S_{n-1}\} \tag{14}$$

Given m such trajectories, our dataset becomes:

$$Dataset = \{T_0, T_1, ..., T_{m-1}\} \tag{15}$$

Model Architecture

The spin model is a GNN, constructed using a single instance of the Metalayer class provided by PyG. The absence of edge and global features simplifies this model to a single node update. The forward pass of the spin model consists of the following steps:

1. Aggregate Node Features: For each node, we compute the sum of its neighbors' features.

2. Update Node Features: We pass each node's features and its neighbourhood feature into a multi-layer perceptron, to obtain new node features.

Model Training

I used an iterative refinement approach to train the spin model. Specifically, I compared small rollouts of length k produced by the spin model to real sequences of spin graphs from the XY model. To obtain a real sequence of spin graphs I took a subset, T_{real}, of a spin trajectory from the dataset of length k.

$$T_{real} = \{S_t, S_{t+1}, ..., S_{t+k-1}\} \tag{16}$$

Then, we produce a predicted spin sequence, $T_{predicted}$, by applying the spin model $k-1$ times to S_t.

$$T_{predicted} = \{S_t, S_{t+1}, ..., S_{t+k-1}\} \tag{17}$$

By summing the mean-squared error of each pair of predicted and real spins at each timestep, we obtain a total loss which we can propagate back through the model. This was done in mini-batches of size 5, using the Adam Optimiser with a learning rate of 0.001, and a k of 8.

3.3 Vortex Model

The vortex model takes as input the positions of vortices of the XY model at time t and outputs a prediction of their state at time $t+1$.

Vortex Graph Representation

In the absence of clear relationships between vortices, I assumed that they all exert some influence on each other. To this end, I represented them as nodes in a fully connected graph. Figure 7 shows two vortices, V_1 and V_2, and two anti-vortices A_1 and A_2.

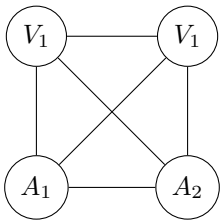


Figure 7: A fully connected graph representing the vortices of an instance of the XY model.

Each vortex's position (x, y) in an instance of the XY model of dimensions L by L, is first encoded to position (x', y') as follows:

$$x' = \frac{2\pi x}{L} \qquad y' = \frac{2\pi y}{L} \tag{18}$$

This effectively converts the position into an angle, enabling the use of trigonometric functions to account for periodic boundary conditions. Then, a feature, v_i, is constructed from the position (x', y') and the vortex's type n (1 or -1 for vortices and anti-vortices respectively).

$$v_i = (\sin(x'), \cos(x'), \sin(y'), \cos(y'), n) \tag{19}$$

Data Preparation

The dataset for the vortex model shares many similarities with that of the spin model. Using numerical integration, I simulated instances of the XY model over various time steps. These simulations produced many vortex trajectories, T, represented as sequences of graphs, V.

$$T = \{V_0, V_1, ..., V_{n-1}\} \tag{20}$$

Given m such trajectories, our dataset becomes:

$$Dataset = \{T_0, T_1, ..., T_{m-1}\} \tag{21}$$

However, this process posed some unique challenges.

The first challenge was tracking vortices through time. Extracting vortices from the XY model at a time t yields lists of both vortex and anti-vortex positions. However, these lists are unordered. To track each vortex over time, we need to pair positions at time t with those at time $t + 1$. This is an 'assignment problem', and requires us to pair positions in a way that minimises the sum of pairwise distances. I overcame this using the SciPy library's linear sum assignment function.

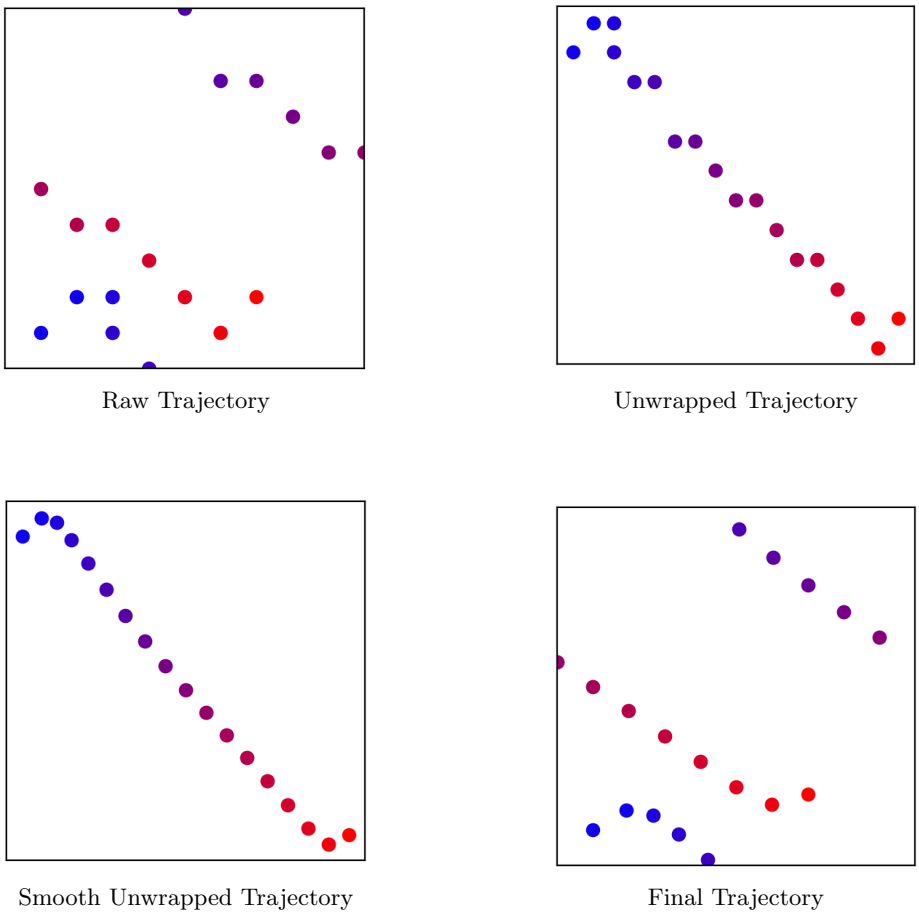
The next nuance that needed to be handled was the annihilation of vortices. When vortices annihilate, the list of positions at time t outnumbers that at time $t + 1$, leading to unpaired vortices. To manage this, I first paired any remaining vortices with their corresponding unpaired anti-vortices using linear sum assignment. Then, I found the midpoint of each pair and appended this to the trajectory of both vortices in the pair. Without this extra step, the GNN would have no data that captures vortex dynamics when very close together.

The final difficulty was tackling the discrete nature of vortex extraction, which led to erratic vortex trajectories. I chose to fit a bezier curve to each vortex's set of points, and then sample it to obtain a smoother set of points. However, fitting a curve to vortex trajectories that include periodic boundary conditions requires some extra steps. I 'unwrapped' each trajectory, equivalent to expanding the area in which vortices can move, to ensure they didn't traverse any boundaries.

I then fit a bezier curve to the unwrapped trajectory using the equation:

$$B(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i P_i \tag{22}$$

where $B(t)$ is the position on the Bezier curve for parameter t in the range $[0, 1]$, n is the degree of the Bezier curve (equal to the number of time steps in the trajectory we are smoothing) and P_i is the i-th control point (the positions of the vortex at each time step). The result of this is a smooth unwrapped trajectory, which can be re-wrapped to produce a smooth trajectory that considers periodic boundary conditions. The example below shows all these steps using an example trajectory of a single vortex, where the colour gradient reflects the vortex's position over time.



Raw Trajectory | Unwrapped Trajectory

Smooth Unwrapped Trajectory | Final Trajectory

Figure 8: A visualisation of the steps taken in vortex trajectory smoothing. The top left shows the actual trajectory of the vortex. The top right shows the 'unwrapped' trajectory. Next is the bottom left, where the unwrapped trajectory has been smoothed. Finally, the bottom right shows the smoothed trajectory of the vortex taking into account periodic boundary conditions.

Repeating this process for all trajectories creates a more suitable dataset

with which the vortex model can be trained.

Model Architecture

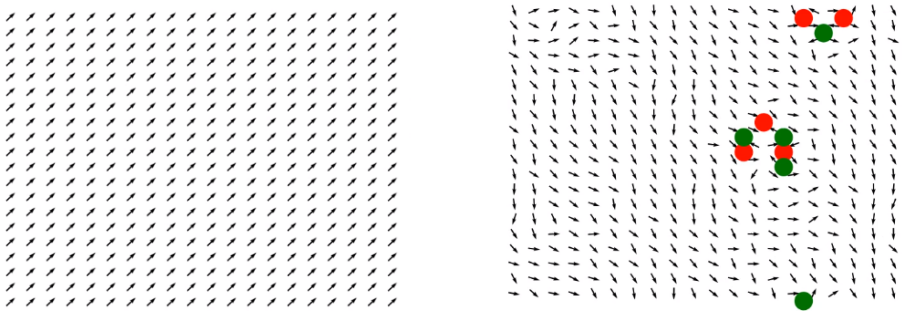
The vortex model is identical to the spin model, except in the aggregation step I chose to average the features of each node's neighbours.

Model Training

Training an accurate vortex model proved more simple the the spin model, requiring only paired graphs, V_t, V_{t+1}, representing vortex states at consecutive time steps. This more straightforward approach is equivalent to the iterative refinement approach with k restricted to 2. The loss was calculated as the mean-squared error between the model's prediction and the real state of V_{t+1}. This was done in mini-batches of size 5, using the Adam Optimiser at a learning rate of 0.001.

4 Results

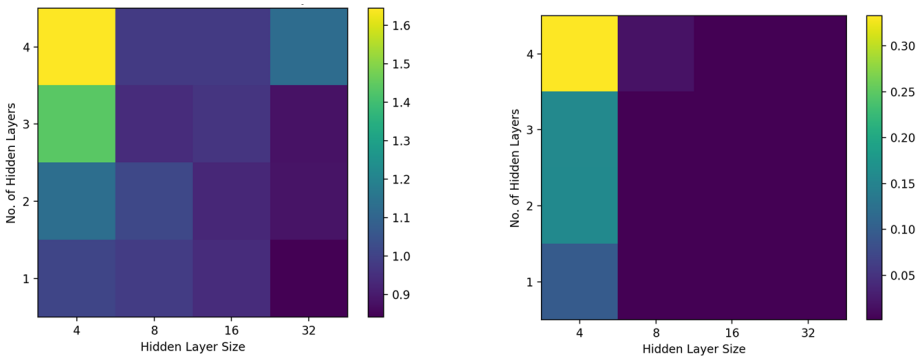
Both models were able to produce physically plausible vortex trajectories over long rollouts, even with small network sizes. However, the spin model required an iterative refinement technique to have a similar performance to the vortex model. Without this assistance, its predictions quickly devolved into chaos. One example of this is shown below:



Real Spins Predicted Spins

Figure 9: An example showing the instability of predicted spins (left) without the iterative refinement approach.

It is therefore important to consider that all of the following comparisons compare the spin model with an iterative refinement approach to the vortex model with its more simple approach. First, I conducted a brief hyperparameter search to identify the optimal network sizes, shown in 10, where darker colours denote a lower minimum loss over 50 epochs.



Spin Model Vortex Model

Figure 10: Grid searches for the spin and vortex models with smaller datasets to identify optimal network sizes.

I decided to use two hidden layers of size 16 for the training of both models with larger datasets to produce the following results.

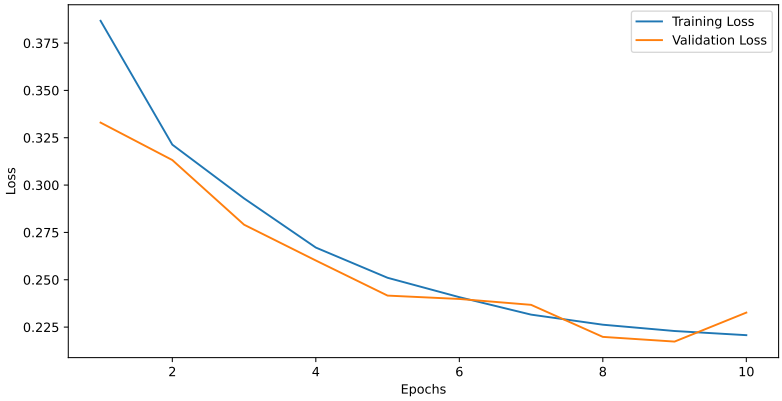


Figure 11: Spin Model Training Curve over 10 epochs.

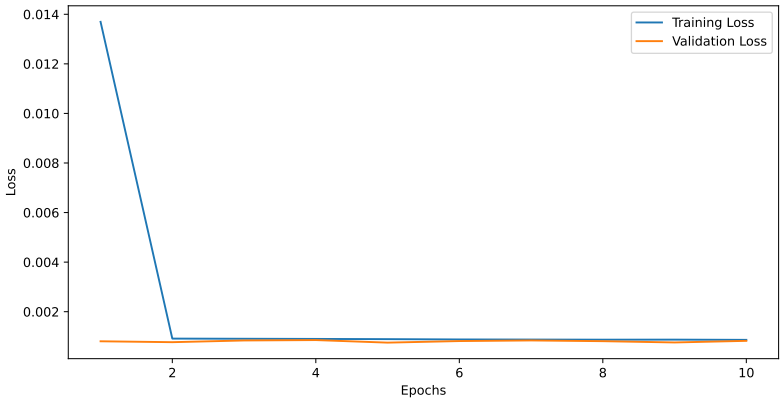
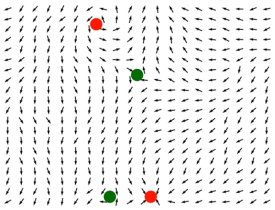


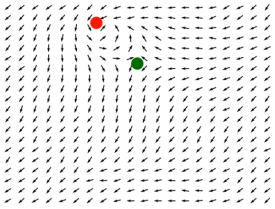
Figure 12: Vortex Model Training Curve over 10 epochs.

Model Predictions

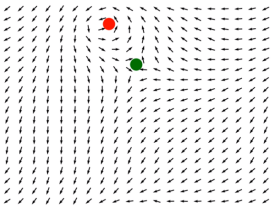
Both models performed best when predicting the dynamics of more stable instances of the XY model, such as in figure 13, where spins are most uniform. In each example below, the initial configuration is given to each model, and their predictions are compared to the real XY model.



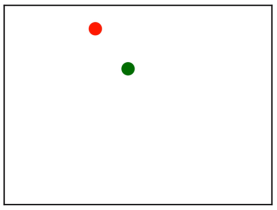
Initial Configuration



Real XY Model at $t = 5$



Spin Model at $t = 5$



Vortex Model at $t = 5$

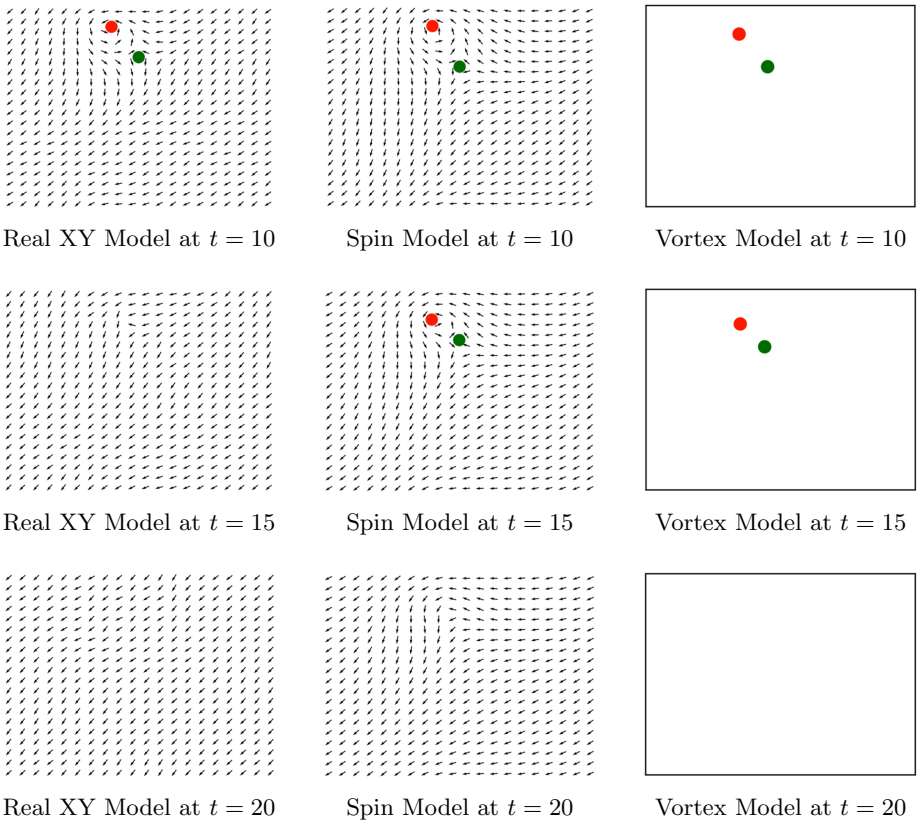
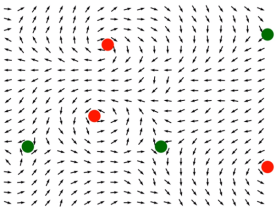
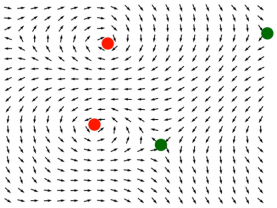


Figure 13: An example of the predictions of each model compared to the real XY model.

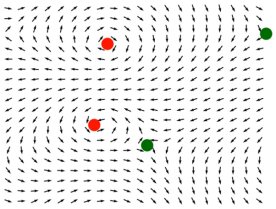
When predicting the evolution of more chaotic instances of the XY model, or those that require longer time periods to reach equilibrium, the predictions become less accurate. The vortex model especially is sensitive to the number of vortices it has to predict, likely because the majority of its training data included 2 or 4 vortices. Even though no longer identical to the real model, the predictions of both the spin and vortex model still display physically plausible dynamics, with vortices attracting and annihilating, as shown in figure 14.



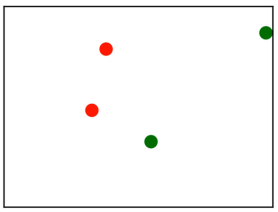
Initial Configuration



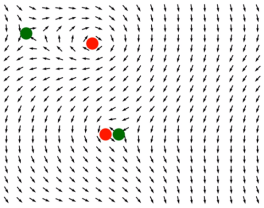
Real XY Model at $t = 10$



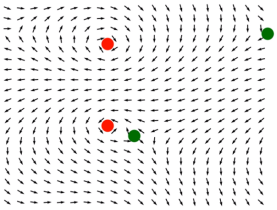
Spin Model at $t = 10$



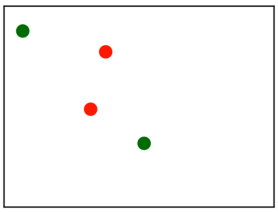
Vortex Model at $t = 10$



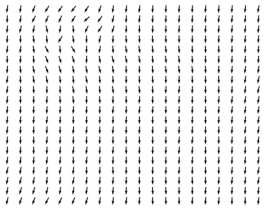
Real XY Model at $t = 20$



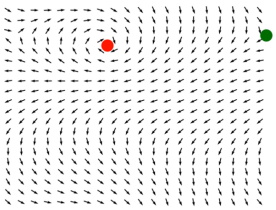
Spin Model at $t = 20$



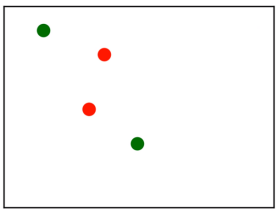
Vortex Model at $t = 20$



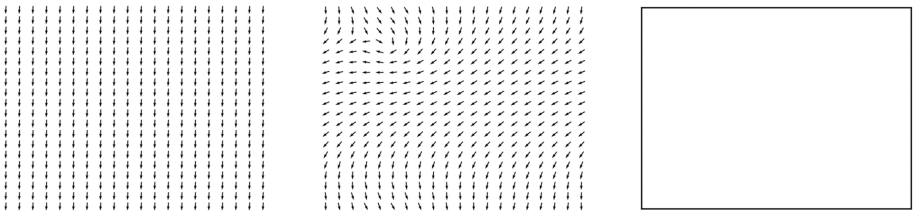
Real XY Model at $t = 30$



Spin Model at $t = 30$



Vortex Model at $t = 30$



Real XY Model at $t = 70$ Spin Model at $t = 70$ Vortex Model at $t = 70$

Figure 14: Another example of the predictions of each model compared to the real XY model. Here, both models produce very accurate vortex trajectories until around $t = 15$, after which they begin to deviate. Despite this, the trajectories remain physically plausible and reach uniformity by $t = 70$.

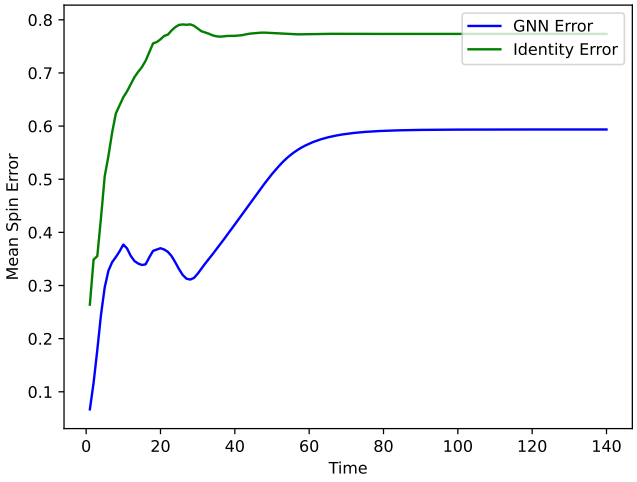


Figure 15: A plot showing the mean spin error in figure 13 between the real XY model, and the predictions of the spin model and an identity model.

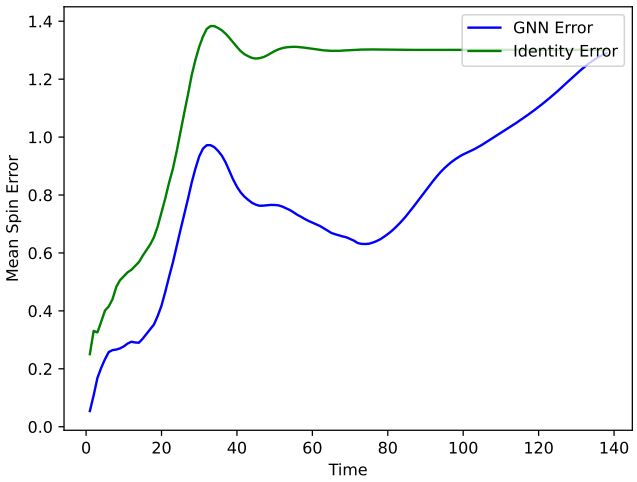


Figure 16: A plot showing the mean spin error in figure 14 between the real XY model, and the predictions of the spin model and an identity model.

5 Conclusion

Even though the error of both models is non-zero, they both predict physically plausible vortex trajectories, even over long rollouts. This confirms numerically that the spins and vortices of the classical XY model exist as different layers of abstraction, and that the dynamics both can be approximated in a data-driven way. The ability of GNNs to approximate the dynamics of the XY model, even with small architectures, shows their promise as tools for studying emergence in more intricate systems. Future research could use similar techniques to investigate brain data to see if neurons follow different dynamics from thoughts, offering new insights into the emergence of consciousness.

References

[1] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. *Multilayer feedforward networks are universal approximators*. Neural Networks, 2(5):359–366, 1989. https://cognitivemedium.com/magic_paper/assets/Hornik.pdf

[2] Xavier Leoncini, Alberto D. Verga, and Stefano Ruffo. *Hamiltonian dynamics and the phase transition of the* XY *model*. Phys. Rev. E, 57(6):6377–6389, 1998. https://link.aps.org/doi/10.1103/PhysRevE.57.6377

[3] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst et al. *Relational inductive biases, deep learning, and graph networks*. arXiv preprint arXiv:1806.01261, 2018. https://arxiv.org/pdf/1806.01261.pdf

[4] H. Jensen. *Complexity Science: The Study of Emergence*. Cambridge: Cambridge University Press, 2022, p. 144. https://doi.org/10.1017/9781108873710.

[5] Rosas FE, Mediano PAM, Jensen HJ, Seth AK, Barrett AB, Carhart-Harris RL, et al. *Reconciling emergences: An information-theoretic approach to identify causal emergence in multivariate data*. PLoS Comput Biol, 2020, 16(12): e1008289. https://doi.org/10.1371/journal.pcbi.1008289.