Using Machine Learning to model collective and adversarial behaviours in active matter systems

Matthew Godfrey

Contents

1 Introduction

Active matter systems can be used to understand complex biological systems such as bird flocking and bacterial behaviour. In such situations, these particles expend energy in order to drive themselves. One such model is a system of active Brownian particles. These particles are given individual directions which can be affected by external forces and forces from other neighbouring particles. These forces can effect both their translational and rotational components. In a simple system, the forces between particles causing these changes can be described as particle-particle repulsion and noise. They can also include a force governed by the environment or neighbouring particles, for example to induce the alignment of particles.

In typical systems, the directions of particles are all randomly distributed, with only a noise term effecting the change in direction over time. Flocking occurs when particles are all aligned together such that they all move towards the same direction. Clustering occurs when the orientations of the particles are misaligned such that clusters form and there is little movement of particles.

A neural network can be used for many different applications of varying fields. For example, the well known MNIST database is used to predict what digit is represented by a hand-drawn version of such digit. The method behind this is rather simple. The image of the hand-drawn digit is represented as a 28x28 pixel image with each pixel being some value between 0 and 255. Each of these images has an associated label, which is some digit between 0 and 9 to represent the hand-drawn image. The images are normalised, flattened, and then fed into the neural network, and for each image the network guesses what digit the image represents. Over time, or epochs, the network will become very accurate in being able to predict what digit the image is representing.

In the same vein as the MNIST database, a database that represents active Brownian motion could in theory be used to train a neural network to flock or cluster. Instead of using an actual image of a digit, chosen parameters of our Brownian system can be used to create an 'image', and instead of a single digit as the label, our label can be some parameter that effects the motion of our particles. Our Brownian motion system can be condensed down to be quite similar to a simple neural network problem.

2 Method

2.1 Brownian Motion

Our initial aim is to produce a simulation of active Brownian particles and investigate how such particles interact with each other, implementing a method to observe them flocking and clustering. In our model, these Brownian particles will have random angular and translational motion due to noise, but will interact initially only via repulsive pairwise forces. This interaction force can be obtained from the Leonard-Jones potential, leading to

$$F_{i,ex} = \frac{24(2 - r^6)}{r^{13}}\Theta(2^{\frac{1}{6}} - r), \tag{1}$$

where r is the distance between particles, and Θ is a Heaviside step function. Θ is used as we are only interested in the repulsive aspect of the Leonard-Jones potential. We are essentially ignoring the attractive aspect of the potential with this.

Our first task is to model our environment by creating our initial conditions. We introduce the number of particles we will have, n, and the dimensions of our 'box' that houses them, an L x L grid. Here, L is some positive number. The particles initial positions are created as random points on this grid with an x and y coordinate, but ensuring that all points are unique as to not have overlapping particles. Each particle is also given a random associated direction, an angle between 0 and 2π.

Next is to model how our particles interact with each other, and move in general. The positions and orientations of the particles, **r** and θ respectively, are updated according to the equations of motion

$$\dot{\mathbf{r}}_i = \mu \mathbf{F}_{i,ex} + v_p \mathbf{u}(\theta_i) + \sqrt{2D}\boldsymbol{\eta}_i, \tag{2a}$$

$$\dot{\theta}_i = \sqrt{2D_r}\xi_i, \tag{2b}$$

where η_i and ξ_i are Gaussian white noise with a zero-mean and unit variance, μ is the particles mobility, v_p is the particles speed, D and D_r are the translational and rotational diffusivities respectively, and $\mathbf{u}(\theta_i)$ is the orientation vector $(\cos\theta_i, \sin\theta_i)$. We define $D_r = \frac{\sigma}{l_p}$, where σ is the range of our repulsive force and l_p is the persistence length. From this we define $D = \frac{1}{3}D_r\sigma^2$ and $\mu = \frac{D}{D_r\sigma^2}$. For simplicity, we set $v_p = 1$, $\sigma = 1$, and $l_p = 60$.

When finding the force acting on each particle, we first need to know the distances between each particle. This is simple enough to do by subtracting the position of your chosen particle by all other particles. From this, the absolute difference can be found. But for our system we would like to implement periodic boundary conditions, so we need to take into account that some particles may be closer together than this distance we

have just calculated. To solve this issue, we make a check for the distances we have calculated. If the distance calculated along a plane, say x_i, is greater than $0.5L$, our new distance along that plane is now x_i - L. Otherwise, we leave it. From these new differences, we can calculate the absolute distance between particles according to

$$r = \sqrt{{x_i}^2 + {y_i}^2}, \tag{3}$$

where x_i and y_i are the true distances between particles along the x and y axis respectively. We then check if this new distance is less than $2^{\frac{1}{6}}$. From these, it is simply a matter of plugging the distance into Equation 1. This force can be split into the x and y components by multiplying by $\mathbf{u}(\theta_i)$, where θ_i is the angle between the particles. The sign of these components can be found trivially from the signs of the differences in coordinates, ensuring the force between particles is always repulsive.

Now that we know how to calculate the force, we use it alongside Equation 2 to update the particles positions at each time step. The first term in Equation 2a is due to the force, the second is the base velocity of our particles, and the third is random noise in the x and y directions. Currently the only change in the direction of our particles comes from Equation 2b, similar to the noise term in Equation 2a.

In order to keep our periodic boundary conditions, we need to do a check after each time step. We simply divide the positions of each particle by L. By saving the coordinates of all particles after each time step, we can build up an animation to visualise our simulation. While this isn't necessary, it does help us check everything is working properly.

In order to induce flocking and clustering we can append a term to our angular equation of motion, Equation 2b, such that

$$B_i^\theta = -g\partial_{\theta_i} \sum_i \hat{e}_{i,j} \cdot \mathbf{u}_i(\theta_i) \tag{4}$$

where g is some constant, and $\hat{e}_{i,j}$ is the unit vector from particle j to i. Depending on if the chosen g value is positive or negative, clustering or flocking is induced respectively. In the case of $g = 0$, the entire term vanishes and typical behaviour is resumed.

2.2 Active work

The active work is the total work done by active forces on all particles, akin to a record of all the motion achieved by all particles. It is a way of measuring the amount of energy the system has used in changing its state. Calculating the active work of our system allows us to see what state our system is in. The active work itself can be viewed as a dimensionless and non-physical quantity that allows us to see this. The total active work of a system can be calculated as

$$W_a(t) = \frac{v_p}{\mu} \int_0^t \sum_{i=1}^{N} \dot{\mathbf{r}}_i(\tau) \cdot \mathbf{u}(\theta_i(\tau)) d\tau, \tag{5}$$

where $\dot{\mathbf{r}}_i(\tau) \cdot \mathbf{u}(\theta_i(\tau))$ is known as the propulsive speed and is a projection of a given particles velocity along its orientation at time τ.

In our equation of motion, the translational motion is varying due to random noise and a velocity term. Our system also has discrete time steps, they have just been made very small to make the process look continuous. Therefore, we do not need to integrate our equation of motion like for a normal function. We can instead sum all of the propulsive speeds from all previous time steps up to the current step and multiply this sum by $\frac{v_p}{\mu}$. It is more useful however to convert this quantity to the normalised active work per particle,

$$w = \frac{W_a \mu}{v_p^2 N t}. \tag{6}$$

Plotting w against time, we find varying results depending on if the system is in a typical, flocking, or clustering state. Figure 1 illustrates these differences. In general, flocking approximately has $w = 1$, for normal dynamics of large system sizes approximately $w = 0.5$, and for clustering $w = 0$ approximately.

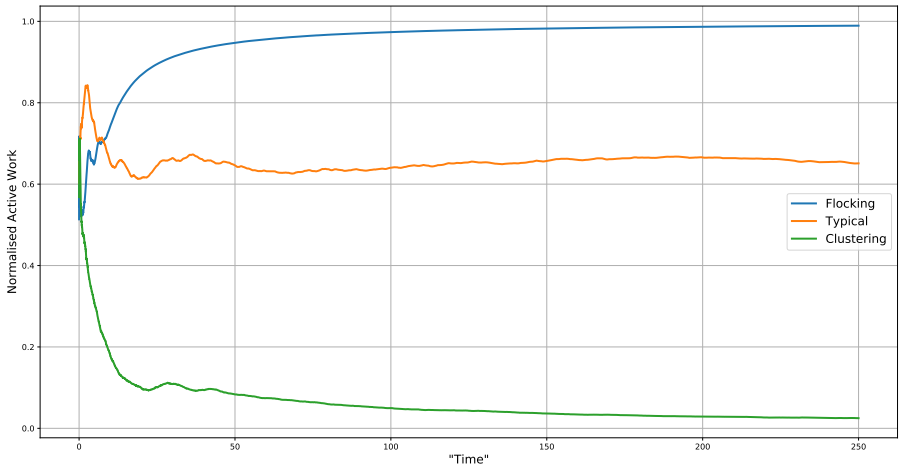


Figure 1: Variation of normalised active work for flocking, clustering, and typical systems of density = 0.4

A 'buffer' period should also be included before flocking/clustering is made to occur. Particles should be made to interact with each other normally for some period of time before Equation 4 is implemented into the system. The initial coordinates of all particles are created to be spaced out some unit distance away from each other, such that the

system is initially not realistic. If the active work is recorded when these particles are initialised, the value of w calculated would be unrealistic. This buffer period solves this issue. The effect that a buffer period has to the normalised active work is illustrated in Figure 2.

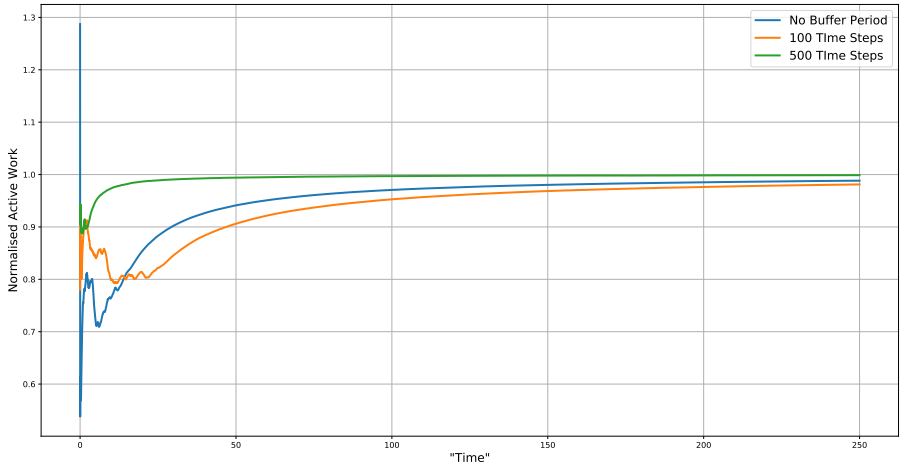


Figure 2: Variation of normalised active work of flocking state of density 0.4 due to size of buffer period

2.3 Using a Neural Network

In order to train our neural network, we need to acquire data which the network will be able to relate to our wanted output. Our training 'image' is what will be fed into the neural network. The x, y, θ, and r values relative to a chosen particle make up this 'image'. This creates an array of dimensions $[m, n, 4]$, where m is the number of images total.

From raw data acquisition, the values of x and y will initially range between values 0 and L, and the values of θ will range between 0 and 2π. We update these coordinates to now be relative to some chosen particle, and cycle through which particles reference frame we use until all particles in the system have been accounted for. Therefore in a system with n particles, we do this n times. Something to note with this is that it will result in much of the data being redundant, as the coordinates of a chosen particle to itself will always be $[0, 0, 0, 0]$. We can therefore remove all this redundant data. The distances between all particles and the chosen particle can be found trivially via Pythagoras and then appended to our data.

The corresponding output to each data 'image', the 'label', is the change in angle of the chosen particle. Initially the labels are simply the values of θ of the target particle one time step after the 'image'. When implementing the neural network into our model, we would simply update the value of θ at each time step with the prediction from our neural

network. This would mean the neural network is learning the angle in the reference frame of the system, but we want it to learn the angle in the reference frame of the particle of interest. We want this since the force is a local one that should be relative to the particles current direction, not its orientation in the box. For this reason, we instead label the training images with $\delta\theta$, a small change in θ that will shift the current value of θ from one time step to the next.

As well as creating data to train the neural network, we also need data to test this neural network. Having test data helps to stop over fitting as it measures how well the system works when implemented to data the network has never seen before.

When acquiring the data, we are only interested in the time when the system is flocking or clustering. In order to do this, we can use the normalised active work. From the active work, we can see when the system enters equilibrium and is in fact either flocking or clustering. By collecting the normalised active work for many runs, a good estimate of when equilibrium occurs for specific systems can be found. For example for a system of density 0.4, $n = 10$, and $dt = 0.001$, the system is seen to flock typically around 'time' $= 30$. We are looking for a point at which the normalised active work has stabilised around a single value. The time at which to start acquiring data will vary for each system, so using the normalised active work is helpful in this regard. Figure 3 illustrates determining the best point to pick equilibrium in a system.

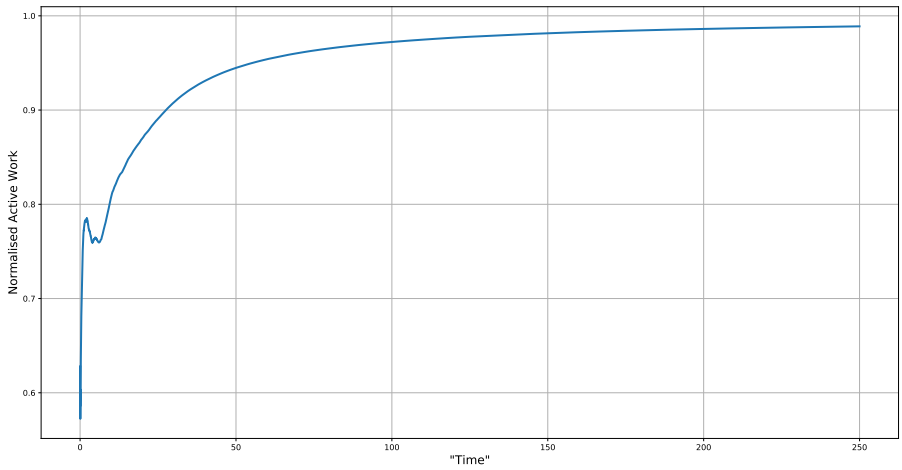


Figure 3: Average normalised active work over 10 iterations for system of density = 0.4 and dt = 0.005. System has stable normalised active work at "time" $\approx$ 150.

However, only collecting data in which the system is flocking/clustering can make training the neural network more difficult. If all the data the neural network is trained on is when the system is already in equilibrium, then it may not know what to output when particles deviate even slightly from this equilibrium. For this reason, including some data when the system is *starting* to enter equilibrium helps train the network what to do when particles deviate slightly.

We normalise all acquired data such that all values lie between -1 and 1. The training and testing images are then reshaped from the initial $[m, n, 4]$ array to a $[m, 4n]$ array. Our neural network is trying to output a value close to the actual 'label' associated with the inputted 'image', so what needs to be minimised is the difference between the actual value and value the neural network produces. For this reason, the mean squared error/mean absolute error is a good metric to use for the loss function.

To help further massage the data and improve the accuracy of our neural network, we remove unnecessary data. All instances of data in which the distance between particles is greater than $2^{\frac{1}{6}}$ is set to 0. The change in θ is due only to random noise and our flocking/clustering term from Equation 4. Therefore, the only data important to the neural network is when the flocking/clustering term is non-zero, which occurs when the distance is less than or equal to $2^{\frac{1}{6}}$.

The training labels for our data can also be massaged to improve our network. Initially our training labels are values between π to -π, meaning all label values could be unique. Having such a large range of labels can make it more difficult for the network to predict a label. Sorting these labels into bins of equal separation between the maximum and minimum values of the labels helps with this issue. Approximating each label to the value of the bin it is placed creates less uncertainty in labels values. This allows us control the number of values a label may take by varying the number of bins the system uses.

3 Conclusion/ Results

Once our neural network is trained and producing accurate results, it is rather simple to implement into our Brownian motion program. Instead of using Equations 2b and 4, the neural network will directly output the change in θ for each particle. As well as our animation, the normalised active work will help us determine if the neural network is producing accurate results.

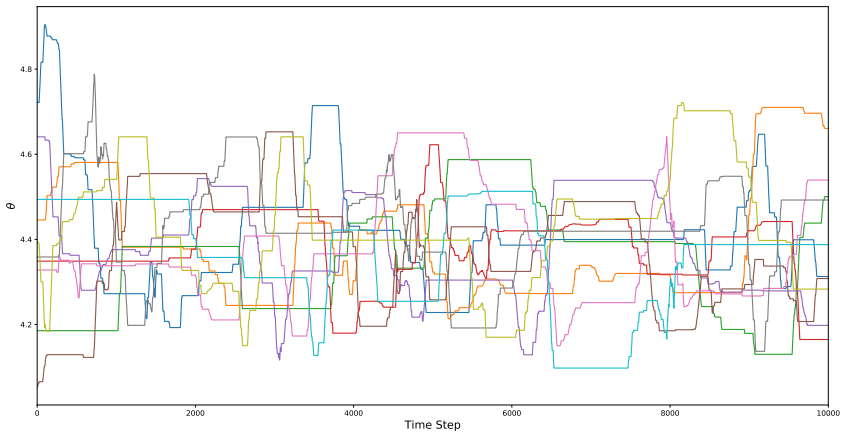
When training the neural network on data from a flocking system, the network performed well on both training and testing images supplied. Using the mean absolute error loss function, the smallest loss value achieved for testing data was 0.002 when our training labels range from 0 to 1 with 100 bins.

When implementing our neural network into our Brownian motion program, it is as simple as replacing the change in angle for all particle with an output from our neural network. When implementing our neural network into a system that has just begun flocking, the neural network kept the system flocking. In order to further analyse the differences between our neural networks outputs and our normal Brownian motion simulator, we can compare a few aspects of both.

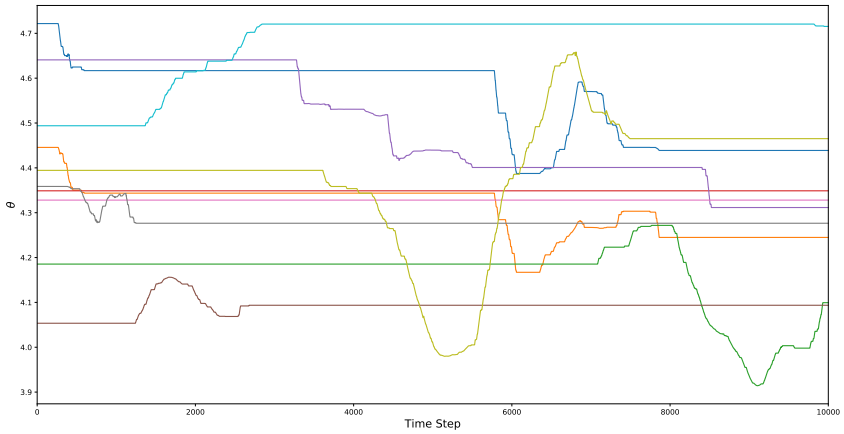
Figure 4 illustrates the change in angle of all particles over time using both our Brownian motion simulator and our neural network. While both approaches do keep the overall mean angle of the system constant, it is obvious that the neural network is not a great match to our Brownian simulator. In our Brownian simulator all particles adjust their orientations often, leading to constantly changing values of θ while maintaining a constant average direction. With our neural network however, most particles keep a constant value of θ, interacting to adjust their orientation only occasionally. While both systems do maintain a flocking scenario, they do for different reasons.

Comparing the normalised active work shown of the two approaches, shown in Figure 5, also bring up some issues. While both systems start off at similar active work levels and are very close to the value 1, the normalised active work of the neural network system deviates over time. This deviation implies that the system is not flocking correctly.

(INCLUDE THAT I COULDN'T GET ANY ACCURATE RESULTS FOR CLUSTERING?)



(a) Brownian Simulation



(b) Neural Network

Figure 4: Comparison of change in θ of particles between Brownian simulation and neural network output, in a flocking system of density = 0.4 and dt = 0.005. Both systems have the same initial coordinates

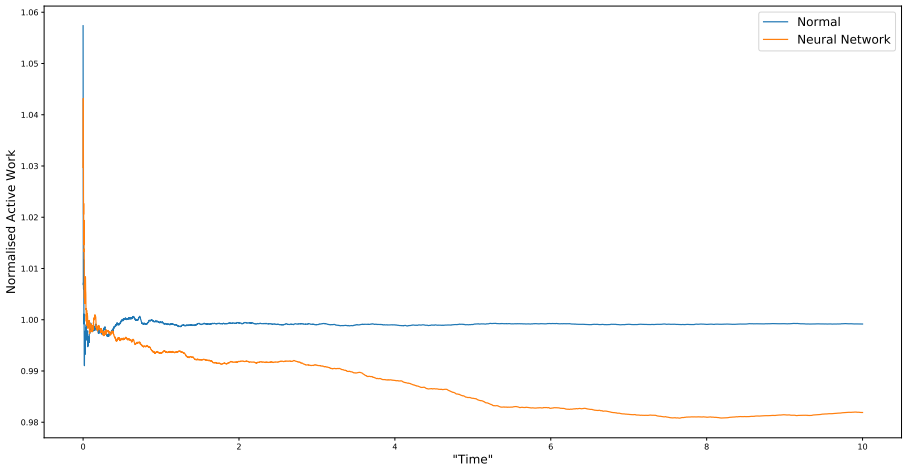


Figure 5: Comparison of normalised active work between Brownian simulation and neural network for system of density = 0.4 and dt = 0.005.