Project 3 - FYS-STK4155 - Looking for Grid Cells & Place Cells in Recurrent Neural Networks

Markus Borud Pettersen

December 16, 2020

Abstract

Here goes abstract.

Introduction

In 2014, the Nobel prize was awarded to May-Britt and Edvard Moser alongside John O'Keefe, for their role in discovering so-called grid cells, a specialized type of neuron, which is thought to be essential to our ability to navigate our environment [1]. What makes grid cells particularly interesting is the fact that they are only active, or *fire* at certain locations in space, and that these *firing fields* map out a hexagonal grid in space. Furthermore, different grid cells can have slightly offset and/or scaled grids, in such a way that an ensemble of grid cells completely cover or tile the space in which the animal moves.

It is perhaps equally interesting that similar responses were recently found in the hidden layer activations of recurrent neural networks (RNNs) trained on a simple navigation task [2]. In fact, the navigation task consisted of performing so-called path integration; the network received information about its velocity (speed and heading), and was tasked with predicting its position. Since neural networks are directly inspired by biological systems, it seems reasonable that the artificial grid cells found in RNNs are related to grid cells in actual animals, and that one could study the emergent structures in navigating RNNs to learn more about how our brain performs navigation.

What makes these results difficult to interpret, is the fact that [2] used the somewhat complicated long short term memory (LSTM) architecture. However, [3] trained a regular vanilla RNN to do the same task and also found grid-cell-like responses. They, on the other hand, trained with so-called *place cell*-like labels. Place cells are another specialized neuron which only fire at a single region of space. A possible problem with the approach in [3] was that the place cells were randomly and uniformly distributed across the environment with firing fields specified by the authors, and not an emergent feature of the network. Inspired by both of these findings, I want to explore the possibility of training a simple recurrent neural network to perform path integration, while making as few assumptions about the network's place cells as possible. More specifically, I want to see if I can train the network to navigate, under the assumption that it develops place-cell like outputs, and then convert said representation to Cartesian coordinates, and use these coordinates to perform supervised learning.

As the RNN considered in this project seems to behave similarly to its biological counterpart, I also want to consider making the model slightly more realistic, by switching from a velocity input

signal, to speed and head direction input *cells*. In the brain, such cells are only responsive for certain input values. In order to emulate this, I will apply so-called radial basis functions (RBFs) to the inputs of the network.

To begin the project, a brief overview of place cells, grid cells and their relationship to each other is given in the context of navigation. Following this, I attempt to outline a possible procedure for decoding the predicted position of a navigating RNN, under the assumption that it has developed a place-cell like representation of space. Finally, I give a short introduction radial basis functions, and how a decent set of such functions can be found.

Background

Grid Cells, Place Cells, & Navigation

As mentioned, grid cells are thought to be a critical part of how animals perform navigation. But, there are a host of other cells involved in this process, including so-called place cells, border cells, speed cells and head direction cells [2]. As the name suggests, a place cell is a certain type of neuron which only fires at a certain place in space. Unlike grid cells, however, a place cell is not periodic, and typically only fires at a single location. As such, the firing rate of a place cell can be modelled using a two-dimensional Gaussian

$$f_{pc}(\mathbf{r}) = Ae^{-(\mathbf{r}-\boldsymbol{\mu})^2/(2\sigma^2)}, \tag{1}$$

where A is some amplitude, $\mathbf{r}$ is the position of the navigator, $\boldsymbol{\mu}$ is the center of the place cell, i.e. the location at which it is maximally active, while σ determines the width of the place cell.

Fig. 1 shows an illustrated place cell modelled according to (1), with $A = 1$, and $\boldsymbol{\mu} = (0.8, 0.4)$, and $\sigma = 0.05$, for input coordinate vectors $\mathbf{r} = (x, y)$, with $x, y \in [0,\ 1]$. The figure also clearly shows how a place cell gets its name; it is almost completely inactive outside of a small region around $\boldsymbol{\mu}$. Intuitively, a place cell coincides with our sense of location: most people do not think of their position in terms of an absolute coordinate, but rather in terms of being "here" or "there".

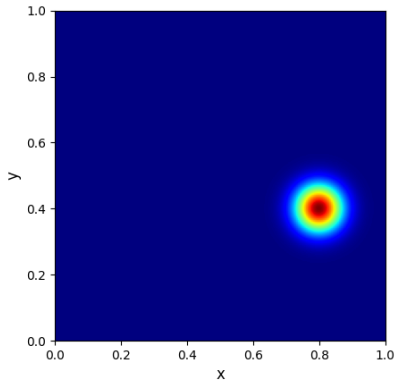

Figure 1: Two-dimensional Gaussian function, centered at $x = 0.8, y = 0.4$, illustrating the response of an idealized place cell in a square chamber.

In the same vein, Fig. 2 shows an illustration of a grid cells in the same region as the place cell. One can notice that the firing rate is maximal at points lying on a hexagonal grid

Interestingly, a grid cell can be modelled as a sum of three plane waves, as long as their wave vectors have a phase offset of $\pi/3$. For the example in Fig.2, the firing rate was calculated as

$$f_{gc}(\mathbf{r}) = \sum_{i=1}^{3} \cos\left(25\mathbf{k}_i \cdot \mathbf{r}\right),$$

where the wave vector $\mathbf{k}$ was taken to be a two-dimensional unit vector on the unit circle, with each vector $\mathbf{k}$ rotated 60 degrees relative to the next, i.e. $\mathbf{k}_1 = (\cos\pi/3, \sin\pi/3)$, $\mathbf{k}_2 = (\cos 2\pi/3, \sin 2\pi/3)$, and so on.

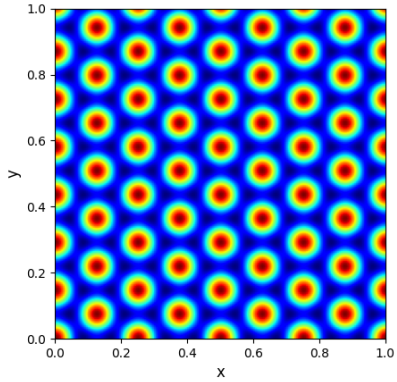

Figure 2: Interference pattern for a sum of three plane waves with wave vectors offset by 60 degrees relative to each other, illustrating the firing pattern of a grid cell in a square chamber.

The next central question is of course how grid and place cells allow us to actually navigate: As mentioned, place cells allow for a direct representation of space, as the act of a place cell firing, indicates that the animal is in the location corresponding to the center of that place cell. Grid cells, on the other hand are not as readily interpreted. As stated, the firing fields, i.e. locations where a grid cell is active, spans across the environment in a periodic fashion (as seen in the example of Fig 2). Furthermore, different grid cells may have slightly different firing fields, as a given firing field may be rotated, shifted or scaled relative to another. All this means that a set of grid cells completely cover the environment in which the agent navigates.

As for actually mapping space, it is believed that by receiving input from speed and head direction cells, grid cells perform path integrate (i.e. they receive information about their current speed and heading, and from this, update their representation of position). Intriguingly, this process is not dependent on sensory inputs such as vision, which one might be able to appreciate: If left in a dark room, one could keep track of ones position relative to the starting point, by keeping track of your speed and heading at all times.

From this very brief introduction, we can gather that grid cells completely map space, and perform path integration. As to how they are key to our able to determine our position, it turns out that grid cells also code for place cells. Using a simple model, we can model this relationship

as

$$f_{pc,i} = \sigma\left(b_i + \sum_j W_{ij} f_{gc,j}\right),$$

where $f_{pc,i}$ denotes the firing rate (or activation) of place cell number i, while b_i is the firing threshold (or bias) of said place cell, $f_{gc,j}$ the firing rate of grid cell number j, while W_{ij} is the weight relating grid cell j to place cell i. σ is some nonlinear activation function.

Recurrent Neural Networks

In order to path integrate, one needs to keep track of one's own position, and possibly speed and heading over time. In order to capture this time-dependence, a recurrent neural network may be used. At its heart, a recurrent neural network is nothing more than a simple difference equation, but as with is deceptively simple relative, the standard feed forward neural network (FFNN), RNNs are capable of solving highly non-trivial tasks, such as speech recognition CITATION.

Considering the most basic, or vanilla RNN, its governing equation can be written as

$$\begin{aligned} \mathbf{u}_t &= W_R \mathbf{h}_{t-1} + W_I \mathbf{v}_t + \mathbf{b} \\ \mathbf{h}_t &= \sigma(\mathbf{u}_t), \end{aligned} \tag{2}$$

where W_R is a matrix containing the *recurrent* weights, W_I a matrix of input weights, while $\mathbf{b}$ is a vector of biases. $\mathbf{u}_t$ is the hidden state activities at timestep t, while $\mathbf{h_t}$ is the hidden state activations at the same step. Here, $\mathbf{v}_t$ is a vector of inputs at at time t. What makes RNNs slightly different from the related FFNNs, is the fact that the hidden state at time t depends on the state value at the *previous* timestep, h_{t-1}. This, coupled with the fact that the RNN receives a new input at each step, is what enables it to solve time-series problems.

BACKPROPAGATION THROUGH TIME, ref proj. 2

As RNNs can be viewed as slightly modified, very deep feed forward neural networks, and are trained in the same way using stochastic gradient descent, it should be no surprise that RNNs tend to suffer from two major problems often associated with deep neural networks: exploding, and vanishing gradients. As the name suggests, exploding gradients occur when gradients become very large, and possibly overflow, causing training the gradient descent to diverge. The opposite problem occurs when gradients become very small, and vanish. MORE, a sentence or two on why it occurs, see e.g. `https://d2l.ai/chapter_recurrent-neural-networks/bptt.html`.

One possible way of addressing the problem of vanishing gradients in vanilla RNNs, is to use so-called identity RNN (IRNN) initialization, as proposed by [4]. The authors show empirically that by initializing the recurrent weight matrix to the identity, the biases to zero, and using the rectified linear unit (ReLU) as the recurrent activation function, a simple RNN can learn very long-term dependencies, and even outperform more complicated architectures such as LSTMs for certain tasks [4]. Note that the authors initialized the input weight matrix according to a normal distribution $\mathcal{N}(0, 0.001)$.

Explain RNN equation, possibly backpropagation through time. Discuss initializations; orthogonal vs. IRNN; vanishing/exploding gradients.

Radial Basis Functions

A radial basis function is, put simply, some function where the output somehow depends on the *distance* to some point, or center, in its domain. In keeping with the scope of this text, only real-valued functions, alongside the Euclidean distance is considered. Before asking which point distance should be measured relative to, it is worth considering that we have already seen an example of a radial basis function, in the form of the two-dimensional Gaussian used in (1), where the center of the radial basis function was the actual center of the place cell.

The reason this is interesting, is that radial basis functions not only allow for a natural way of encoding space, but also other variables, such as speed and head direction. To investigate whether adding more biologically plausible features to the navigating RNN improves performance, it would therefore be of interest to use radial basis functions to mimic speed and head direction cells, i.e. specialized neurons which are only responsive to particular input ranges. To model this in a simple manner, one can imagine that instead of having a single input neuron coding for speed, let's say, one has many neurons, each reactive to a certain range of speed values.

To see how this might be implemented, consider the illustration in Fig. 3, where a simple, linearly increasing speed signal $v(t) = 5t$, has been feed into four different simulated speed cells. For this illustration, each speed cell is taken to be an unnormalized Gaussian with a standard deviation of 0.1. The four different "cells" have centers at, or are most responsive to, a velocity of 0, 0.5, 2.5, and 4, respectively.

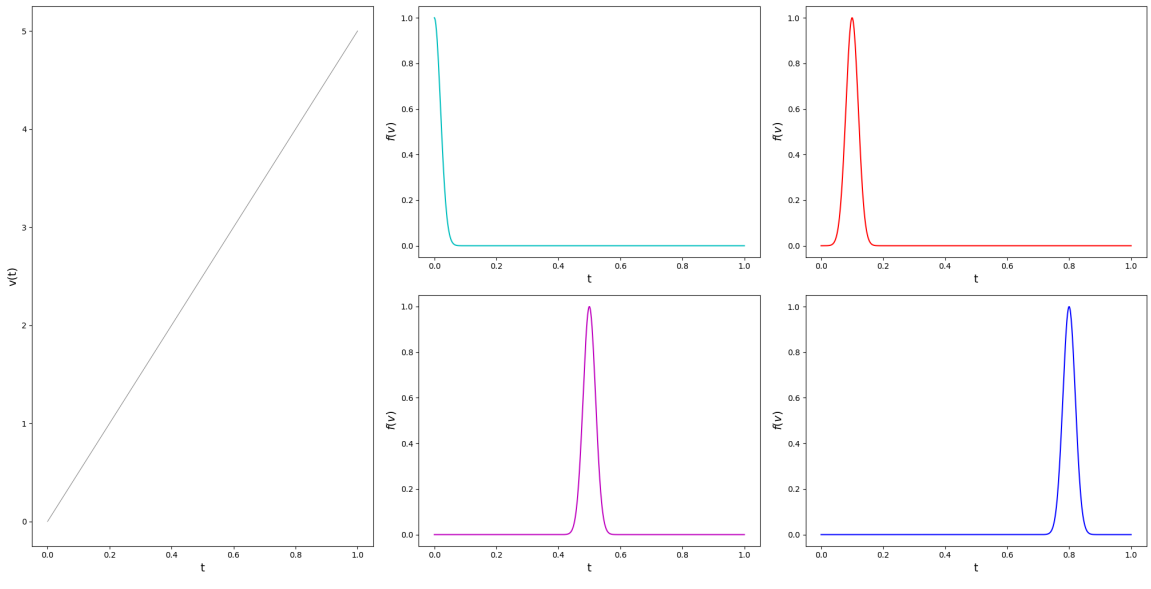

Figure 3: Response of Gaussian speed cells for a linear input signal $v(t) = 5t$. Each cell has a standard deviation of 0.1, and the cell centers are at 0, 0.5, 2.5, and 4, respectively.

As one can easily judge from Fig. 3, the simulated speed cells only react to certain input values, and are almost completely inactive for values far from their center. Based on the set of four speed cells in the figure, one can also tell that there is a clear loss of information in the representation of $v(t)$, as there are some swaths of the signal's range which are not covered any speed cell. As such,

one would either need more speed cells, or possibly wider radial basis function to have a useful basis set.

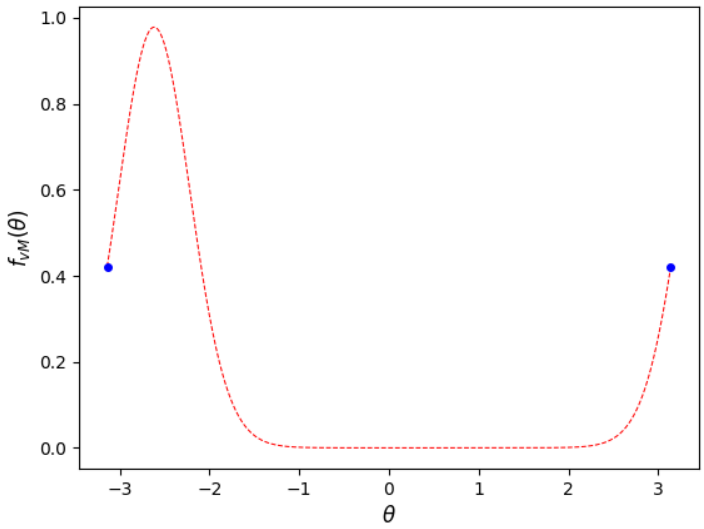

Figure 4: Response of a simulated head direction cell, for input head direction angles in the interval $[-\pi, \pi]$. The shown response is in actuality the probability density of a Von Mises distribution on the same interval.

Since speed is readily converted into a radial basis function representation, one may wonder if the same can be done for the slightly more exotic head direction. Since head direction is measured in terms of radians (angles), it is 2π-periodic, which in essence means that 2π is the same head direction as 0. This is however no obstacle, as there are probability distributions which also have this property. The simplest of which is probably the Von Mises distribution, which can be thought of as a normal distribution, but on a (one-dimensional) ring. An example of a Von Mises distribution is shown in Fig. 4, where the correspondence to a normal distribution is hopefully obvious, as is the periodicity (indicated by blue markers). For the distribution in the figure, the centre of the distribution is at $\theta = -5/6\pi$, and the width parameter, often denoted κ was taken to be 2π, which is roughly equivalent to a Gaussian with $\sigma = 1/\sqrt{2\pi}$, due to the fact that $\sigma^2 \approx 1/\kappa$.

In the same way as with the speed "cells", one can therefore mimic the behaviour of head direction cells by creating an ensemble of differently centered Von Mises distributions (corresponding to the angle to which the cell responds maximally), and pass the head direction signal through said ensemble before being fed to the RNN.

As a small sidenote, a radial basis function representation of a continuous variable such as speed does come at the cost of being very high-dimensional; we require many nodes to encode a single quantity, and there is also some redundancy, in the sense that cells may overlap, and therefore encode the same information. However, this might also be a useful feature in biological systems as it may help to guard against noise, as multiple cells hold a "piece" of the information being processed.

Decoding Place Cell Centers

As described in the introduction, the goal of this project is to train an RNN to perform path integration, and investigate whether both grid- and place cells can emerge as a solution to the navigation problem. To do this, the approach will be to *assume* that the output of the network is place cell-like, and convert/decode the output of the network in terms of Cartesian coordinates, and have this be the final output of the network. While this might sound straight-forward; one could be tempted to use the argmax function to simply extract the location of greatest activation and denote this as a place cell center, for instance. Once place cell centers are found in this manner, one can simply decode the position as the position of the most active cell at any given time. However, argmax is not differentiable, and so we need to develop a differentiable alternative, in order to be able to train models using gradient methods.

To do this, consider the fact that one can think of a place cell's activation as being proportional to the probability of finding yourself at the cell's center: If a place cell fires strongly, it is very likely that you are close to the point in space where the place cell has its center. Going one step further, and assuming that an ensemble of N such cells constitute a probability distribution at any given time (i.e. their activations sum to one at each timestep), one may hope that the expected value

$$\mathbb{E}_{PC}[\mathbf{r}(t)] = \sum_{j=1}^{N} p_j(t)\mathbf{r}_{pc}^j, \tag{3}$$

is a good estimate of the position $\mathbf{r}$ at time t. Here, $\mathbb{E}_{PC}$ is the expected value over all place cells (at time t), and $\mathbf{r}_{pc}^j$ is the center of place cell j. Note that the place cell centers are fixed in time. It seems unreasonable to think that the outputs of our network (which is only assumed to be place cell-*like*) sum to one without any modification. Therefore, inspired by the way the softmax function is determined, one can calculate the normalized outputs or probabilities, as

$$p_i = \frac{f_{pc,t}^i}{\sum_{k=1}^{N} f_{pc,t}^k}, \tag{4}$$

where $f_{pc,t}^i$ denotes the firing rate of place cell i at time t.

Fig. 5 shows the above scheme applied to motion along a circle of radius 0.5, for three simulated place cell with fixed centers, at angles $\theta = 0, \pi/4$ and $\pi/2$, respectively. Also shown is the activation of each place cell, taken to be non-normalized Gaussians, each with standard devation 0.1, alongside the corresponding output normalized according to (4). Alongside the place cell activations is a motion along the circle (solid line), the place cell centers (large dots), and the decoded position along the circle (small blue dots).

From the place cell activations it is evident that as one moves close to a given center, the normalized activation becomes almost unity (and all other activations consequently close to zero), which "picks out" the center of that place cell as the decoded position through (3). One can also see that for the regions between cells, there is some degree of interpolation of position; the decoded position is mostly a weighted sum of the two closes place cells, and so on. Finally, one can also see that the scheme fails far from a place cell: the decoded position far from the three place cells fall on a line between pc_0 and pc_1, but this is because the normalized activation once again goes to unity for the closest place cell, and almost zero for the remaining cells. This goes to show that this scheme depends on having place cells that cover the entire environment in order to get an accurate position estimate.

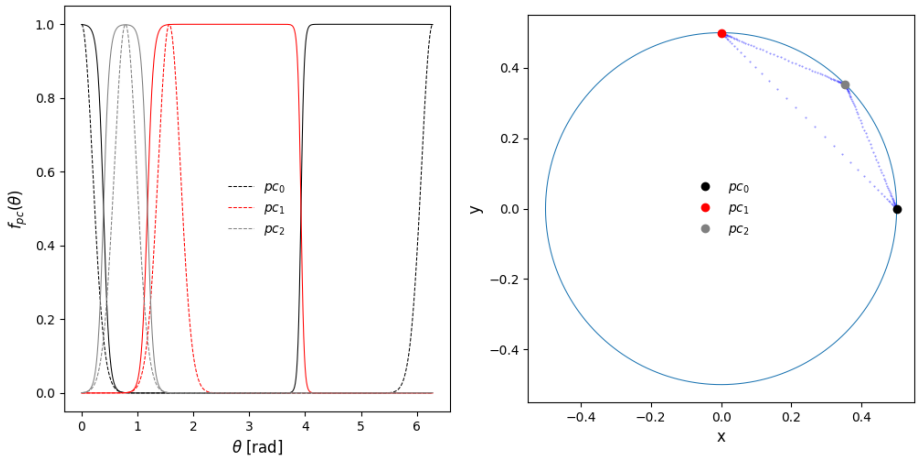

Figure 5: .

While the example in Fig. 5 hopefully illustrates the decoding of position from place cells, a method for actually finding the place cell centers is needed. To find a such a scheme, we once again assume that our network is trained, and is producing some place-cell-like output. Finding the centers of these place cells is actually a similar problem to the position decoding, but this time we want to study the entire time-series response of a given place cell, and from this response deduce the place cell center. Following the same logic as before, but instead taking the expectation value over time for a single place cell, we arrive at the estimate

$$\mathbb{E}_t[\mathbf{r}_{pc}^k] = \sum_{t=1}^{T} P_k(t)\mathbf{r}(t), \tag{5}$$

where $\mathbb{E}_t[\mathbf{r}_{pc}^k]$ is the expected value over time, of the center of place cell k, and $\mathbf{r}(t)$ is the target position at timestep t, while T is the number of timesteps, and we interpret $P_k(t)$ to be the probability of the place cell center being at position $\mathbf{r}(t)$. The same normalization procedure as before is used, but this time, as we want to take the expected value over time, we normalize the place cell activation in time:

$$P_k(t) = \frac{f_{pc}^k(t)}{\sum_{t=1}^{T} f_{pc}^k(t)}.$$

This approach can then be applied to each place cell separately, yielding an estimate for all place cell centers.

At this point, this might seem like a hopeless approach; in order to determine the place cell centers, one needs the target position, i.e. the thing the network is trying to predict. However, the goal of this network is not strictly to path integrate; it is to form grid cell/place cell firing patterns in space while doing so. MORE; GOOD SENTENCE OR TWO TYING IT ALL TOGETHER.

Methods

Simulated Paths

In order to investigate whether RNNs performing path integration develop grid- or place cell like hidden unit activations, a dataset was simulated. The dataset consisted of semi-random, semi-smooth paths in a square, 1×1 chamber. The paths themselves were created using python and the numpy library. For a given path, the starting coordinate was sampled randomly from a uniform distribution in the chamber, and both initial speed and head direction was sampled from uniform distributions.

At each timestep, the head direction was updated by adding random turn angle, sampled from a normal distribution $\mathcal{N}(0, 0.25)$, i.e.

$$\phi \to \phi + \varepsilon,$$

where ϕ is the head direction and ε the added turn angle. Also at each timestep, the distances to all walls were computed. If the distance of any coordinate component was smaller than 0.05, or the agent outside the arena, an additional turn angle was added to the head direction, so as to keep the agent inside the arena, and avoid the walls. This was done by determining the difference in angle between the closest wall normal and head direction, and turning counterclockwise if the difference was positive, and clockwise otherwise. This way, if the head direction was $\pi/3$ and the corresponding top wall normal angle was $\pi/2$, then the agent turned clockwise, eventually facing away from the wall.

In addition, the speed of the agent was updated at each step, given by an update rule similar to that of the head direction:

$$s \to s + \chi,$$

where χ was a random change in speed, sampled from a normal distribution $\mathcal{N}(0, 1)$. To avoid the speed becoming negative or too large, s was clipped between 0 and 1. For the head direction, the modulus was taken with 2π to ensure that the head direction was in the interval $[0, 2\pi]$. Finally, the position of the agent at a given time step t was computed as

$$\mathbf{r}_t = s_t(\cos\phi_t, \sin\phi_t).$$

To emulate a real-world scarcity of data, as well as save storage space, small datasets were created with sequence lengths of 100, 1000, and 10000 steps, respectively. Each dataset featured 10000 paths, or samples. In order to train recurrent neural nets on both Cartesian coordinates as well as head direction/speed inputs, datasets were created with both sets of inputs.

Recurrent Neural Networks

In order to investigate path integration in different RNN architectures, three different RNN models were created: one baseline model with uniform initialization, one IRNN model, and a custom RNN which supporded radial basis function inputs, an RBFRNN.

All models featured the same basic architecture, composed of two units; one RNN layer, and one output layer. In all cases, models were implemented using Tensorflow 2.1 and Python3.

For the baseline model, the recurrent weight matrix was initialized according to a uniform distribution, more specifically a Glorot uniform initialization scheme (see for example [5]), and the same initialization was used for the RNN bias. For this model, the recurrent activation function was tanh, and there were 100 recurrent nodes.

The output layer was initialized using the same Glorot uniform scheme, but the biases were initialized to zero. The output layer featured 50 output units (i.e. assumed place cells). To enforce a non-negative output similar to that of place cells, the output layer activation function was ReLU.

To allow for training on very long paths, the IRNN architecture described in the background section was used for the second RNN model. As described earlier, the recurrent weight matrix was initialized according to the identity matrix, the biases were set to zero, and the input weights were initialized according to a normal distribution $\mathcal{N}(0, 0.001)$. The recurrent activation function was accordingly ReLU, and the recurrent layer featured 100 hidden units. In addition, an optional L2 weight regularization was added to the recurrent weight matrix. The IRNN output layer was identical to that of the baseline model, both in terms of weight initialization and activation function. As with the baseline model, the number of output nodes, i.e. assumed place cells, was 50.

The RBFRNN was based off of the IRNN design, in order for it to be able to cope with possibly long paths. However, to allow for radial basis function inputs, a custom RNN cell, or RNN equation, was implemented. The custom RNN cell followed the same structure as (2), but featured two sets of inputs, one from an ensemble of simulated speed cells, and the other from a set of simulated head direction cells. The speed cell ensemble consisted of 20 nodes, the responses of which were calculated as a non-normalized Gaussians in a manner similar to that outlined in the background section. For simplicity, the speed cell centers were taken to be linearly spaced in the interval $[0, 1]$, corresponding to the possible values of the dataset paths. To ensure that the full interval of possible speed values were encoded by the speed cells, the width, or standard deviation of each Gaussian was taken to be $1/20$, i.e. the range of possible speeds divided by the number of cells.

The responses of the head direction cells were approximated using the Von Mises distribution described in the background section. As with the speed cells, the centers of the head direction cells were linearly spaced along the range of possible head directions as they appeared in the dataset, i.e. in the range $[0, 2\pi]$. The width parameter κ of the distribution of each cell was taken to be 2π, and a total of 50 head direction cells was used.

The activations of the speed and head direction cells were each fed into the RNN hidden state by a respective input weight matrix. Each weight matrix was separately initialized according to the standard IRNN input matrix initialization, i.e. according to a normal distribution $\mathcal{N}(0, 0.001)$. The recurrent weight matrix was initialized to identity, and the bias to zero, and the recurrent layer consisted of 100 nodes, as with the other models. The output layer of the RBFRNN was identical to those of the IRNN and baseline models.

For the IRNN and baseline models, the input to the RNN layer was the speed of the agent, alongside direction vector of the agent. In other words, the input at time t was

$$\mathbf{x}_t = (s_t, \cos\phi_t, \sin\phi_t),$$

which was done to avoid the discontinuity of the head direction at $\phi = 0 = 2\pi$, while the separate speed input was added in the hopes that it might ease training. For the RBFRNN, the speed and head direction were supplied to the head direction and speed cell ensembles for decoding, so that the input vector to the RNN layer at time t was

$$\mathbf{x}_t = (s_t, \phi_t),$$

for a given path.

In addition to head direction and speed inputs, the network itself was supplied with all target positions, so that the place cell/position decoding scheme detailed in the background section could be implemented. Each RNN performed the same decoding process, using the following steps:

1. Compute RNN hidden state activations from head direction/speed input data
2. Compute output activations using RNN activations
3. Determine place cell centers from output using eq. (5) and position labels
4. Decode output to Cartesian coordinates using place cell centers and eq. (3)
5. Yield predicted Cartesian coordinates

For all models, the initial state of the recurrent layer was trainable, meaning that each node could learn a unique starting state, but this state was the same for all samples in a mini-batch.

Training & Experiments

To compare the performance of all networks on the navigation task, all networks were trained on datasets corresponding to paths of length 100 timesteps. All networks were trained using gradient descent and the Adam optimizer. Likewise, the cost function was taken to be the mean squared error in all cases, for all models. As mentioned, each dataset consisted of 10000 samples or paths, with each path in a given dataset being the same length. In all trials, 80 % of the dataset was used for training, and the remaining 20 % reserved for testing. For training of RNNs, the batch size was 50 for all models, and after some initial testing[1], the learning rate was taken to be $5 \cdot 10^{-5}$. For these initial runs, no regularization was applied to any of the models. All models were trained for 100 epochs, and the dataset was shuffled between each epoch.

The relatively short 100-timestep paths did not allow for complete exploration of the chamber, and the place cell decoding scheme, assumes that place cells emerge as the navigator moves around the chamber. Therefore, longer, models were also trained on a dataset containing 1000-step paths, using the same training parameters described in the previous paragraph. Note that lower learning rates were tested for the longer paths, but this did only seemed to slow down learning.

Finally, to study the effects of regularization, a grid search was performed for the IRNN model, for values of the recurrent weight matrix L2 weight penalty factor. In this search, each model was trained for 50 epochs on the 1000-timestep paths, and L2 penalty factors were taken to be geometrically spaced values in the interval $[10^{-6}, 10]$.

[1] Stochastic Student Descent

Results

Simulated Paths

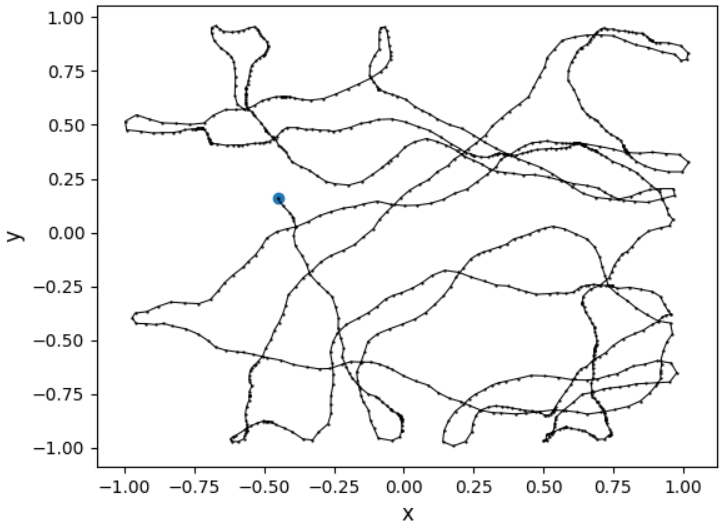

Figure 6: Caption

0.1 Recurrent Neural Networks

Table 1: Test set mean absolute error for each RNN model after 100 epochs of training, for paths of length $T = 100$ and $T = 1000$ timesteps.

Steps	100	1000
Baseline	6	87837
IRNN	7	78
RBFRNN	545	778

Discussion

Conclusion

References

[1] *The Nobel Prize in Physiology or Medicine 2014*. en-US. 2020. URL: https://www.nobelprize.org/prizes/medicine/2014/press-release/ (visited on 12/16/2020).

[2] Andrea Banino et al. "Vector-based navigation using grid-like representations in artificial agents". en. In: *Nature* 557.7705 (May 2018), pp. 429–433. ISSN: 1476-4687. DOI: 10.1038/s41586-018-0102-6. URL: https://www.nature.com/articles/s41586-018-0102-6 (visited on 11/13/2020).

[3] Ben Sorscher et al. "A unified theory for the origin of grid cells throughthe lens of pattern formation". In: *Advances in Neural Information Processing Systems* (2019).

[4] Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. "A Simple Way to Initialize Recurrent Networks of Rectified Linear Units". In: *arXiv:1504.00941 [cs]* (Apr. 2015). arXiv: 1504.00941. URL: http://arxiv.org/abs/1504.00941 (visited on 12/14/2020).

[5] Markus Borud Pettersen. "Project 2- FYS-STK4155". Project Report. Oslo: University of Oslo, 2020.