Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Medicine and Health Sciences
Kavli Institute for Systems Neuroscience

Boon Linn, Choo

Spatial representation of a virtual environment in a deep neural network

Master's thesis in Master of Science in Neuroscience (MSNEUR)

Supervisor: Prof. Raphael Kaplan

June 2020

NTNU
Norwegian University of Science and Technology

Boon Linn, Choo

Spatial representation of a virtual environment in a deep neural network

Master's thesis in Master of Science in Neuroscience (MSNEUR)

Principal Investigator: Prof. Raphael Kaplan

Subject Supervisor: Mr. Markus Frey

June 2020

Norwegian University of Science and Technology

Faculty of Medicine and Health Sciences

Kavli Institute for Systems Neuroscience

NTNU – Trondheim
Norwegian University of
Science and Technology

(This page was intentionally left blank)

Abstract

In recent decades, scientists have made great advances in characterizing the neurobiological foundations of spatial cognition. Notably, neurobiological findings have not been limited to active exploration of the physical world. Neuroscientists have more recently started to use virtual reality (VR) approaches to develop more detailed accounts of spatial coding in the brain, but the specific neural computations guiding spatial navigation remain unclear. Attempting to uncover these neural computations, we built a deep convolutional neural network (CNN) to investigate how a simulated agent learns a virtual environment. We found that the CNN learned the layout of the virtual environment mainly via unit responses that resembled previously discovered spatially modulated hippocampal neural signals, as well as novel 'corner' units. These results give a hint of the importance and efficiency of spatially modulated cells in evaluating physical environments in both artificial navigating agents and exploring organisms.

Sammendrag

De siste tiårene har forskere gjort store framskritt i å karakterisere det nevrobiologiske fundamentet for spatial kognisjon. Særskilt har nevrobiologiske funn ikke vært begrenset til aktiv utforskning av den fysiske omverden. Nevroforskere har i det siste begynt å ta i bruk virtuell virkelighet (VR) for å redegjøre mer detaljert om spatial koding i hjernen, men det er fremdeles uklart hvilke spesifikke nevrale beregninger som veileder spatial navigasjon. I et forsøk på å avdekke slike nevrale beregninger bygde vi et konvolusjonsbasert (convolutional) nevralt nettverk (CNN) for å undersøke hvordan en simulert agent lærer virtuelle omgivelser. Vi fant at vårt CNN lærte de virtuelle omgivelsene gjennom enhetsbaserte responser som lignet på tidligere oppdagede stedsmodulerte nevrale signaler i hippocampus, samt nye 'hjørneenheter'. Disse resultatene kan være hint om viktigheten og effektiviteten av stedsmodulerte celler i å evaluere fysiske omgivelser både i simulerte navigerende agenter, og i utforskende organismer.

Acknowledgements

Extra appreciation to every person who has been a part of my thesis, supporting me both in close distance and remotely regardless of holidays. Without them, I would not have been able to complete my thesis. Special thanks dedicated to my supervisor, Mr. Markus Frey for all the detailed explanations, valuable suggestions and patience guidance in assisting my thesis, especially when I have started all in this project out from scratch and zero background in deep learning and Python. Also, extra gratitude to my principal investigator, Dr. Raphael Samuel Matthew Kaplan who has been really supportive and motivating, he whom encouraged me through all the ups and downs, gave me informative feedbacks and provided me with confidence when I felt lost while conducting the project. Next, I am also very thankful to Dr. Tobias Navarro Schröder, whom has shared me the snapshots and labels information to get the project started in the first place, whilst always readily to share his experience as enlightens. Thus, I would like to dedicate this thesis outcomes to all of them, honouring their efforts by declaring this as OUR project. Additionally, also thanks to my colleague, Mr. Stian Framvik and housemate, Mr. Svein Fredrik Froshaugen Nyvoll to have helped me in translating the abstract into Sammendrag. Meanwhile, also thanks to my colleague, Ms. Natalie St. John for helping me to do the final proofreading. Lastly, of course also thanks to my family and best friends who always show me their beliefs and trusts in me despite we are thousands miles away from each other, especially my parents, whom have even been selflessly investing and aiding part of my expenses to allow my further study became possible.

(This page was intentionally left blank)

Table of Contents

List of Figures

List of Tables

List of Abbreviations

- Acc – accuracy
- AdaGrad - adaptive gradient algorithm
- Adam - adaptive moment estimation
- AR - augmented reality
- ARe – autoregressive
- BN - batch normalization
- BOLD - blood-oxygen-level-dependent
- CNN(s) - convolutional neural network(s)
- CPU - central processing unit
- DARPA - Defense Advanced Research Projects Agency
- EC – entorhinal cortex
- EKF - extended Kalman filter
- EM - electron microscope
- EMA - exponential moving averages
- fMRI - *functional* Magnetic Resonance Imaging
- GAN - generative adversarial network
- GPU(s) - graphics processing unit(s)
- GQN - generative query network
- HDD - hard disk drive
- LEC – lateral entorhinal cortex
- LN - layer normalization
- Lr – learning rate
- LSTM - long short-term memory
- Mae - mean_absolute_error
- MEC – medial entorhinal cortex
- MR - mixed reality
- Mse - mean_squared_error
- OS – operating system
- RAM - random access memory
- RGB – red green blue
- RMSprop - root mean square propagation
- RNN(s) - recurrent neural network(s)
- SATA - serial advanced technology attachment
- SGD - stochastic gradient descent
- SIMD - single instruction multiple data
- SLAM - simultaneous localization and mapping
- VR - virtual reality

Chapter 1 : Introduction

1.1 Project's Rationale and Approach

We often have the chance to travel outside familiar environments like for international travel or visiting a new shopping mall. Despite the inherent challenges posed by environmental novelty, we are typically still able to conduct intuitive exploration, where we can find our way and generalize locations within a newly perceived environment quite easily. We can even travel to targets precisely without familiar landmarks in a considerable distance (1) based on internal judgement. This judgement is known as path integration, our innate ability to use idiothetic (self-motion) cues such as angular movements and distances travelled to compute the vector between personal location and trajectory positions (2, 3). In contrast, mobile robots normally require heavy pre-training experience that requires high similarity with encounters during testing.

More than 20 decades have passed since the first invention of steam-powered railway trains, yet we are still incapable of realizing the fantasy of safety-assured-driverless-vehicles. Autonomous robotic navigation is mostly considered a failure when compared to animals' self-locating ability. Taking the Google Street View as an example of an artificial navigating agent, it is unable to calculate distance travelled without GPS tracking or the ability to label a building and street automatically without help from Google engineers. To this end, researchers are using deep learning to help resolve this issue. Research teams are developing either simulated or physically-existing navigating robots that can at least navigate autonomously in a new environment after being trained in different dynamic environments (4, 5). These robots mainly function on the basis of large training sets of labelled images (6) to learn image identification, image classification, object detection, scene understanding (semantic segmentation), and specific object recognition (7). In spite of these initial successes, most studies have failed to relate navigating computations in robots with electrophysiological findings in freely moving animals.

To reduce the gap between studies in animals and robots, we first compare the remarkable spatial cognition capabilities of animals to artificial agents. Next, we provide insights into the spatial recognition and navigation components of artificial agents, and use these new findings to uncover novel spatial computations. We then provide evidence how artificial agents could be trained to learn on virtual environments in an efficient manner almost similar to animals. We hypothesize that training a deep convolutional neural network (CNN) to learn locations in a complex virtual environment will afford biologically plausible spatial learning in a simulated agent.

1.2 Background: Neurobiology of Spatial Cognition

It was first suggested by Charles Darwin that instincts have led us to find directions in our daily lives (8). The concept of an internal mental map used for navigation was then suggested by a geographer, Gulliver (1908) (9) in an attempt of trying to make a map that is comfortably readable by everyone (10). One of the earliest spatial navigation studies was introduced by Watson and Lashley (1915), where they investigated birds' long distance homing and short distance nest-locating abilities (9). They however, failed to explain the underlying spatial cognition related to these behaviours (9). One of the most influential theories on spatial cognition was of the cognitive map proposed by Edward C. Tolman in 1948, which was devised following his experiment involving rats exploring a maze (11). This theory suggested that behaviour is guided by mental events beyond stimulus-response learning (12). For example, the concept of stimulus-response learning does not sufficiently explain the use of novel shortcuts during navigation (13), as taking a novel shortcut requires the ability to build up an internal map of an environment. Consequently, Tolman suggested that these cognitive map-like representations are necessary to help acquire, store, and recall memories (12). Uncovering specific brain regions related to memory, Brenda Milner studied an epilepsy patient, H.M., with surgical resections in the hippocampus and neighboring medial temporal lobe structures that failed to form new memories and remember everyday experiences that occurred after their surgery (13-15). Supporting a potential role for the hippocampus in forming cognitive maps, O'Keefe and Dostrovsky found that there were selective hippocampal "place" cells that increased in firing rate when rats explored a particular location within an environment (16). Inspired by these findings, O'Keefe and Nadel (1978), hypothesized that these hippocampal cognitive maps of physical environments could facilitate mnemonic function more generally (11).

Since then, the hippocampus and surrounding brain areas received more attention from neuroscientists, which led to the discovery of additional types of spatial cells. A summary of these spatial and directional cells are listed in the table below (*Table 1.1 Summary of Spatial Cells and Features*).

Spatial Cells Name	Features
Hippocampal Place Cells	• Type: Pyramidal Cells, Granule Cells (17) • Predominant Brain Area(s): Hippocampus (CA1, CA3, Dentate Gyrus) (17, 18) • Environment-specific representations. Place cells fire at specific locations in a given environment, and remap in another environment. This suggests that the cognitive map is dynamic, can be continuously updated and remapped to represent one's location in a changing environment, and may reflect stored experience in the hippocampal network (17) • May be co-modulated by grid cells and head-direction cells (17)
Head-Direction Cells	• Type: (putative, excitatory) Pyramidal Cells (19-21), Martinotti Cells (22)

	• Predominant Brain Area(s): Anterior Dorsal Thalamus (predict future head direction), Dorsal Presubiculum (postsubiculum), Retrosplenial Cortex, MEC (23), lateral mammillary nuclei and lateral dorsal thalamus, striatum • Activate when an animal is facing a particular direction regardless of spatial location. Angular distance between the cells' preferred directions is conserved across environments (23) • Mature prior to place cells and grid cells, and do not depend on grid cells on directional selectivity, suggesting an upstream expression of spatial maps (23)
Grid Cells	• Type: Pyramidal Cells, and Stellate Cells in Layer II of MEC (17) • Predominant Brain Area(s): Medial Entorhinal Cortex (MEC) (17), may also present in neocortex (24) • Grid cells possess periodic hexagonally spaced firing fields that form a regular triangular grid across the environment. They provide a measure of distance and boundaries for the entorhinal–hippocampal spatial map, and affected by path integration (direction [angular] and speed [linear] information required to transform the representation during (self-)movement by yet-to-be-determined-specialized-cells) (17). Grid fields that appear in the first exploration of a new environment persist despite subsequent changes in landmarks, with a tendency to rotate along with external reference points (25), hinting of a continuous attractor network mechanism (17). Grid cells cluster into modules of cells with similar grid scale, grid orientation and grid asymmetry but different grid phase (25) that remains constant across environment (17), hinting of oscillatory interference models (24) • May be modulated by head-direction cells and dentate gyrus (17), and drive the formation of place cells (26)
Conjunctive (Grid) Cells	• Type: Pyramidal cells in Layer III of MEC (27) • Predominant Brain Area(s): Layer III of Medial Entorhinal Cortex (MEC) (27) • Very similar to grid cells and also fire in the pattern of a regular triangular pattern. Modulated by head-direction such that the individual cell will only fire if the head is pointed towards the cell's preferred direction (27)
Border/Boundary (Vector) Cells	• Type: Stellate Cells (99%), Pyramidal Cells (1%) (28) • Predominant Brain Area(s): (All layers of) Medial Entorhinal Cortex (MEC), Parasubiculum (29, 30), subiculum (31) • Sparsely exist with just less than 10% of the local cell population. Border cells encode one's position in relation to the borders of the egocentric environment,

	and fire when one is near the edge of the local environment (29, 32), that is, they fire at specific distances from specifically oriented environmental boundaries (33), providing connecting information of egocentric (self-to-object) and allocentric (object-to-object) boundary (34). • Potentially work with head-direction cells and grid cells to plan trajectories and anchoring grid fields and place fields to a geometric reference frame (29), whereas border and grid cells associations are suggested to minimize the accumulated grid cells' error (35)
Object Cells	• Type: Principal cells of the entorhinal cortex (36) (Pyramidal, Stellate) • Predominant Brain Area(s): Lateral Entorhinal Cortex (LEC) and associated areas (33) • Object cells possess object selectivity (processing what instead of where the object is (37)), that is, respond only when the animal is at the vicinity of discrete objects (36) and not when it perceives it from a distance (33) • A sub-type of object cells are so-called object-trace cells, discovered by Tsao et al. (2013) that respond corresponding to where an object had been on a preceding trial (36, 38)
Landmark-Vector Cells	• Were initially thought to be place cells by Muller and Kubie in 1987 (39), but were later called landmark-vector cells by Deshmukh and Knierim in 2013 (31). These cells match the evidence of alternative class of place cells proposed by Rivard et al. in 2004, who named them barrier cells (40) and closely resemble the vector representation model proposed by McNaughton et al. in 1995 (31) • Type: Pyramidal (31) • Predominant Brain Area(s): Hippocampus (CA1) (31, 33) • Predicted to be bound to two or more landmarks and would fire at all locations, which have matching distance-bearing relationships to landmarks. Landmark-vector cells are functionally equivalent to boundary (vector) cells except that instead of boundaries, they signal proximity to a barrier (fixed external landmarks) (31). They differentiate between subsets of objects (33), fire at a unique spatial location (31), and encoding landmark identity and saliency (41) • Distinct from object-vector cells, which are sharply tuned from the first trial, landmark-vector cells emerge slowly and appear to depend on experience (33) • "May arise from object representations and object-based spatial representations of the LEC in conjunction

	with path integration-based inputs (such as representations of head direction and distance) from MEC" (31)
Speed Cells	• Type: Pyramidal (42) • Predominant Brain Area(s): Medial Entorhinal Cortex (MEC) (43) • Speed cells possess linear positive response to running speed, hence providing a dynamic representation of self-location that contribute to the update of grid-cell activity when one is moving across space (43)
Object-Vector Cells	• Type: (Not specified) • Predominant Brain Area(s): Medial Entorhinal Cortex (MEC) (33) • Object-Vector Cells are allocentric vectoral representations in the MEC which support position mapping of objects in distance (peri-personal space). They "discharge at specific distances and directions from salient objects, independently of the identity, size or location of the object or the orientation of one's body axis. Discrete high-contrast objects induced object-vector fields regardless of whether they were internal to the environment or attached to external bounding walls." (33) • Object-vector cells are theorized to intermingle with grid cells and head-direction cells that encode position in a distal framework (33)
There are also Misplace/Mismatch Cells (O'Keefe, 1976), Spatial View Cells (Rolls, 1999; Rolls and Xiang, 2006), Splitter Cells (Wood et al., 2000), Barrier cells (Solstad et al., 2008), Time/Sequence Cells (Pastalkova et al., 2008; McDonald et al., 2011), Band Cells (Julija Krupic et al ,2012), and (Memory-)Trace Cells (Tsao et al.,2013 (Briefed above); Bicanski and Burgess, 2018; Salman et al., 2019), Goal-Direction Cells (Sarel, 2017), Reward Cells (Gauthier and Tank, 2018), "Social Place Cells" (Danjo et al., 2018, Omer et al., 2018) and Irregular (Spatial) Cells (Meister and Buffalo, 2018) which are not discussed in this table.	

Table 1.1 Summary of Spatial Cells and Features

For a rough illustration of different spatially modulated cells' firing patterns, Behrens et al. (2018) have compiled a figure that shows the anatomical location of the hippocampus and entorhinal cortex, along with a variety of other spatially modulated cells in these regions (*Figure 1.1 Examples of Spatially Modulated Neurons in the Hippocampal Formation*).

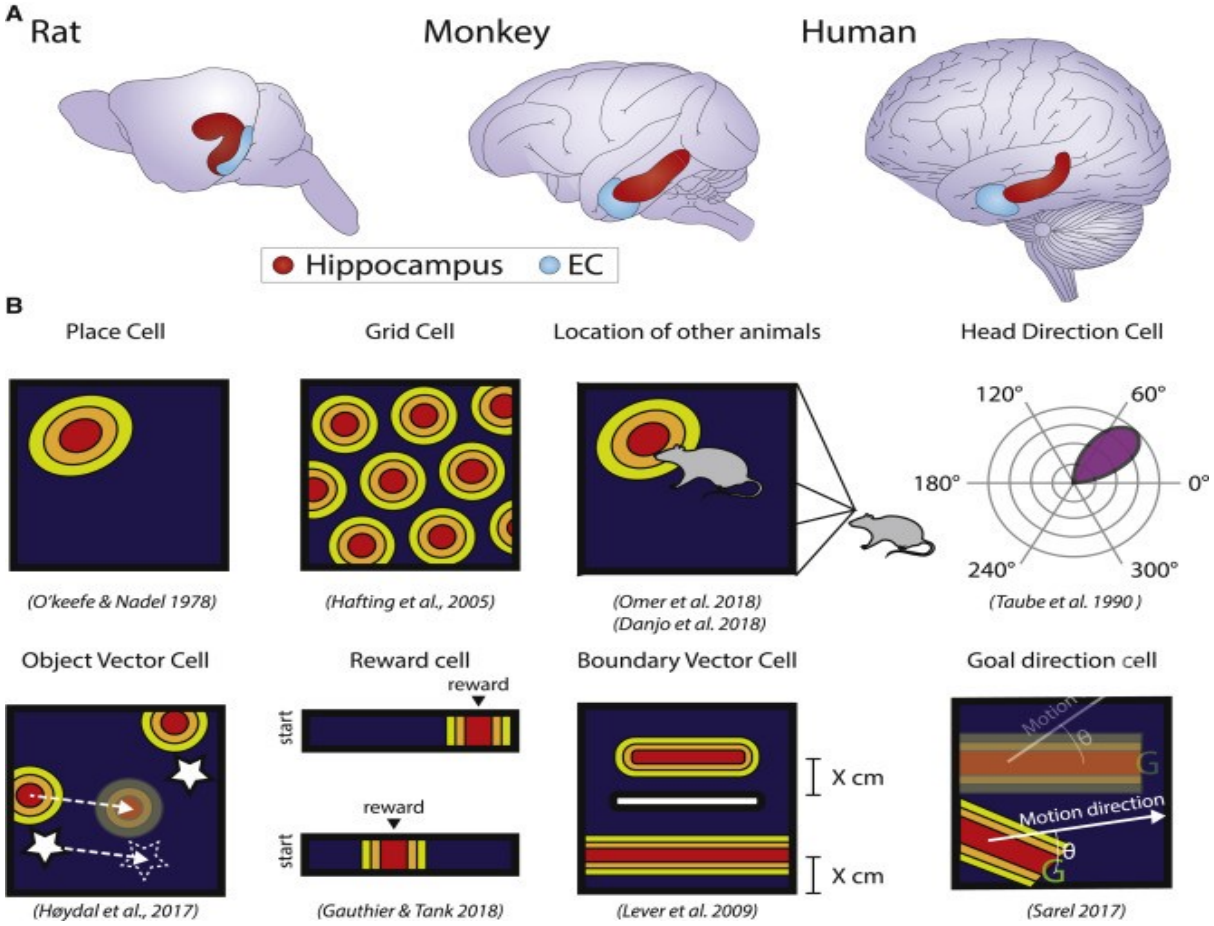


Figure 1.1 Examples of Spatially Modulated Neurons in the Hippocampal Formation [44]

As a brief overview, (A) showed the homologous network organization of the mammalian animals: rat, monkey and human (44). Meanwhile, in (B), (i) place cell is mainly associated with a single location and induces activity restrictively, (ii) grid cell which sourced in MEC is activated when the animal is situated in one of the multiple locations that located in a triangular grid, playing a role in decoding vector and distances between the locations, (iii) "social place cell" spikes in response to the location of another animal, (iv) head-direction cell activity is correlated to the animal's facing direction, (v) object vector cell decodes direction and distance information to an object (egocentric mode: self-to-object), (vi) reward cell is activated at the vicinity of reward itself, (vii) boundary vector cell gets activated by deriving the representation of boundary from a distance, (viii) goal direction cell encodes movement direction in relation to the goal (44).

1.2.1 Utilization of Virtual Environments on Spatial Cognition Studies and the Viability of Donderstown

Exploration (45), navigating and imagined navigation (46, 47) studies using non-invasive functional MRI and VR environments have found grid cell-like representations in humans.

Researchers have postulated that grid cell-like coding mechanisms in the entorhinal cortex (EC) enable us to traverse space in situations such as imagined navigation, which would lead to mental exploration and recalling of previous experience (46). Horner et al. (2016) observed blood-oxygen-level-dependent (BOLD) fMRI signal changes in EC during both movement and imagination phases in a virtual reality (VR) environment, but not during the stationary phase in the study (46). These findings were further supported by Bellmund et al. (2016) who used a newly created VR city, Donderstown that was carefully built to comply with their own set of imagined navigation tasks (47). Both of the studies found

that EC may contribute to mental simulation more generally than expected as spatial processing in the absence of visual input is also sensitive to the six-fold rotational symmetry properties of grid cells' signals (47). Meanwhile, in another study that performed behavioral spatio-temporal learning and memory tasks using Donderstown by Deuker et al. (2016), it was suggested that humans form a combined event map of memory in the hippocampus (48).

In sum, the results from these two studies using Donderstown have proven the reliability of our chosen virtual environment to study spatial cognition in simulated agents. In the next part, we will be discussing the background of deep learning and virtual environments, as well as the main algorithm behind navigating agents.

1.3 Machine Learning, Neural Networks and Virtual Environments

In an effort to simulate neural computations of space and memory using neural network models, there were two major papers that studied path integration in simulated agents with recurrent neural networks (RNNs): first by Cueva et al. (2018) who managed to find representations of band cells, border cells, irregular cells and grid cells (49), and a second by Banino et al. (2018), who used a long short-term memory (LSTM) architecture, to find grid representations (50). The models from both teams have enabled the artificial agents to conduct shortcut behaviours in a 2D virtual environment (49, 50). But where did the concept of artificial neural networks first arise?

Observations of how human thinking have contributed to the invention of automation and early robotics. Subsequently, there came the term artificial intelligence, which was phrased in 1956, and can be defined as machines that exhibit some human intelligence (51). Meanwhile, neural networks constitute a central category of artificial intelligence research. The earliest neural network was modelled by Warren McCulloch and Walter Pitts, who used electrical circuits known as logic gate circuits (52) that have thresholds and weights, but no layers (53). Subsequently, one of the most influential factors was contributed by the neuroscientist, Donald Hebb, who reinforced the concept of Hebbian learning in neural networks, where "cells that fire together, wire together" (54). This concept has then further been expanded into Hopfield networks, which are commonly known as attractor networks or auto-associative networks (17, 55) that allow generalization, familiarity recognition, categorization, and error correction in a time separated manner (56), for pattern separation and pattern completion of different representations (17).

Referring back to the development history of neural networks, the Perceptron was introduced by Frank Rosenblatt in 1957 (53), and is capable of performing basic logical operations (AND, OR, and NOT) to classify linear tasks (52). The Perceptron possesses only an input layer, a hidden layer and an output layer. This limitation was identified by Minsky and Papert in 1959, and only began being resolved around the 1980s (53), when feed-forward neural networks were introduced with multiple layers (52). Tsodyks and Sejnowski later developed neural networks that came with feedback and feedforward coupling (57) to find out the values of the hidden layer and enable weighted learning (52). The mentioned term - feedback, is commonly known as back-propagation, an algorithm that allows weights updating between layers (52).

While there has been intense corporate competition to create better hardware to support neural networks, the smart systems today are still largely linked to machine learning that requires big data and engineers (51). Additionally, despite early successes, the concept of deep learning did not emerge until Hinton's multi-layer neural networks with Restricted Boltzmann Machine and Deep Autoencoder were proposed (58). These followed Bengio's breakthrough with metamodelling and LeCun's findings using convolutional neural networks (58). Thereafter, more and more variations were surfacing either in enhancement of existing models or new kinds of neural networks (58). Alongside the improvement in technology and the invention of more and more dedicated graphics processing units (GPUs), we are currently stepping into a rapidly developing era of deep learning. Relating deep learning to everyday life, many large corporations have started to invest on deep learning, such as Google Brain, Deep Mind, Facebook AI (58), Alibaba, Microsoft Research AI and IBM Watsons. Consequently, almost all digital things today are highly affected by them, such as Google Translate, Alibaba Cloud Image Search and even the lung image diagnosis and forecast of disease spreading for pandemics.

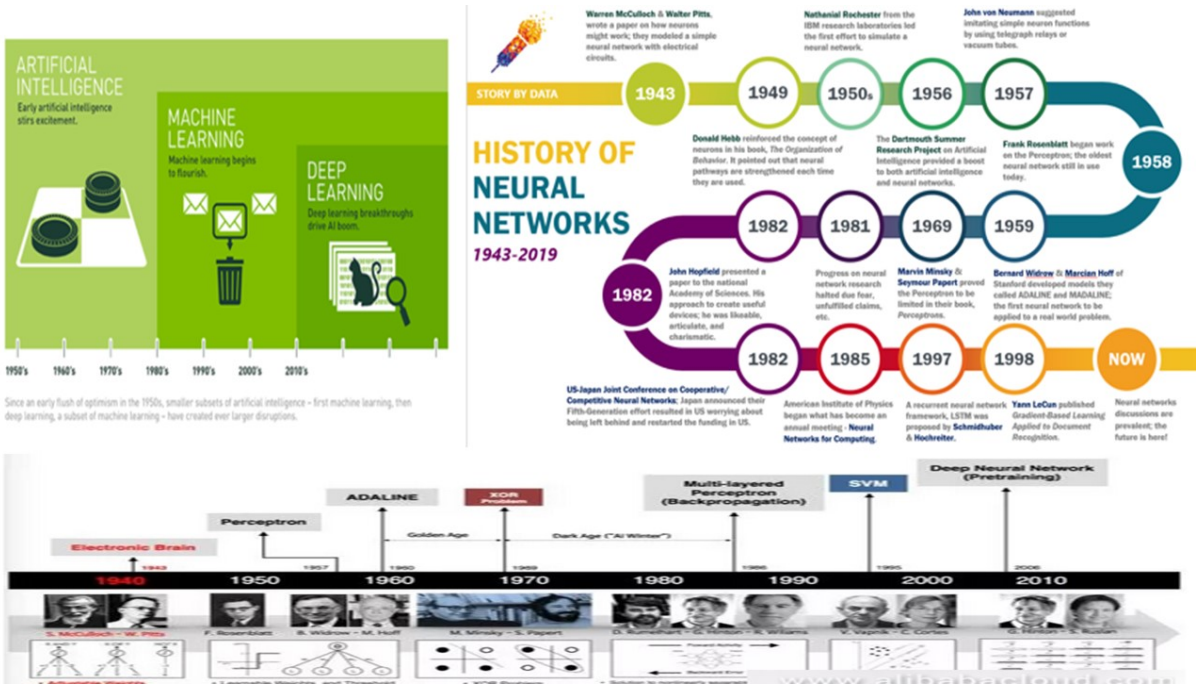


Figure 1.2 Brief Summary of the Development Timeline in Neural Networks [58] [59] [52]

A very brief overview of technologies (top left) and evolution of neural networks (top right and bottom)

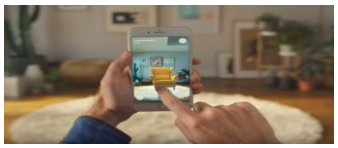
Virtual Reality (VR)	Augmented Reality (AR)	Mixed Reality (MR)
An **immersing simulated experience with computer generated environment** that can be similar or completely different from the real world, comes in 3Ds, which can be explored and interacted with, such as manipulating objects and performing actions (59, 60). For example, Google StreetView, Oculus and Donderstown. 	**Projection of virtual objects into physical space** whether viewing through lenses or projectors (61). For example, Ikea Place, Snapchat Lenses & Geofilter, Google Glass and Microsoft HoloLens.   	Mixed reality encompasses both augmented reality and virtual reality that allows a digitally interactive experience. Commonly associated **holograms with mixed reality headset** -- a device that enables users to see, grab and interact with holographic content just like physical objects and environments (62). For example, SixthSense by Pranav Mistry.   

Table 1.2 Differences between VR, AR and MR

Differences between the 3 main simulated environments available today. Our Donderstown is a VR environment.

However, one may start to wonder, how do robots in real and virtual worlds deal with navigation problems? To become as autonomous as possible, they have to be able to perform real time robotic mapping via odometry such as visual localization and motion estimation (63). This method is well known as simultaneous localization and mapping (SLAM) (63). SLAM was first proposed by Smith et al. in 1988 using extended Kalman filters (EKF), which was later implemented and expanded by others (64) (see *Figure 1.3 Example of SLAM Methods* for instances).

Figure 1.3 Example of SLAM Methods [65]

To ensure this SLAM approach can enable a machine, especially a mobile robot, to successfully localize, the model has to be capable of performing: robot position prediction (based on odometry and landmark), observation (feature extraction), measurement prediction (global-to-local frame transform, i.e., coordinate transformation between the world frame and the sensor frame), matching (data association and correlation that will be used as a prediction of map), and estimation (filter update based on learned weights) (66-68). Beyond linear navigation, the model has to be able to perform loop closure too, i.e. correct cycle detection, handling of uncertainty and integration of new landmarks in dynamic environments (66, 67). With such success, the model will be applicable not only to metric and topological maps, but also cognitive maps (65).

Artificial intelligence outputs have also inspired neuroscientists working in computational neuroscience to work further in humanizing a robot into learning spatial environments like humans and animals. An interesting model known as RatSLAM (*Figure 1.4 Overview of RatSLAM Versions*) was created by Milford and Wyeth, who were inspired by SLAM and empirical research on the rodent hippocampal cognitive map (69, 70). The very first model of RatSLAM (*Figure 1.4(a)*) was built based on competitive and continuous attractor networks that can take visual inputs (a.k.a. local view cells or landmark cues) to support its pose hypotheses and update the estimation (self-motion cues) based on the shifting of activity packet to allow further path trajectory via the wheels (70). The pose is represented by the place cells (position) and head direction cells (orientation) separately in two networks (70). The model however encountered difficulties in 2D environments due to the inability to hold multiple pose hypotheses (71). This matched the boundary problem proposed by McNaughton et al. (2006) for connections around the edge of the neurons layer (72). To overcome this issue, the same group then combined place and head direction cells into one single network, and named these cells, pose cells (70), which had very similar properties to conjunctive grid cells (70) (*Figure 1.4(b)*). This remodelled version encountered collision (1 pose cell representing >1 place) and discontinuity (>1 pose cells representing the same location), leading to goal recall breakdown (73). To overcome this issue, a third version of RatSLAM (*Figure 1.4(c)*) was generated. Activity in the pose cells is updated by self-motion cues, forming tessellating patterns similar to grid cells and calibrated by local views for data association to support path integration (74). Additionally, this newer generation had an additional feature known as a spatial experience map that formed based on information from pose cells, local view cells, and self-motion cues (74). An experience is a matched activity pattern of pose and local view cells (74). Any difference in the pairings would lead to the formation of new experience that is linked to previous experience (node) via the distance obtained from self-motion cues, and corrected via loop closure (74) involving the techniques graph relaxation (average between visual cues and self-motion cues to get the best overall map), map pruning (divide experience map into a grid and ensure one grid represented only by one place to maintain a reasonable number of places in map) and path planning (transitions- distance, speed, temporal between goal places) (69). This model provided the robot with a continuous and reusable map for navigation (69, 75). With this success, these researchers have proceeded to create an open source SLAM system known as OpenRatSLAM (76), which inspired other labs to develop their own related models, such as adapting to a humanoid robot (77), Gist+RatSLAM (78), visuo-tactile SLAM (ViTa-SLAM) (79) and active neural SLAM (80).

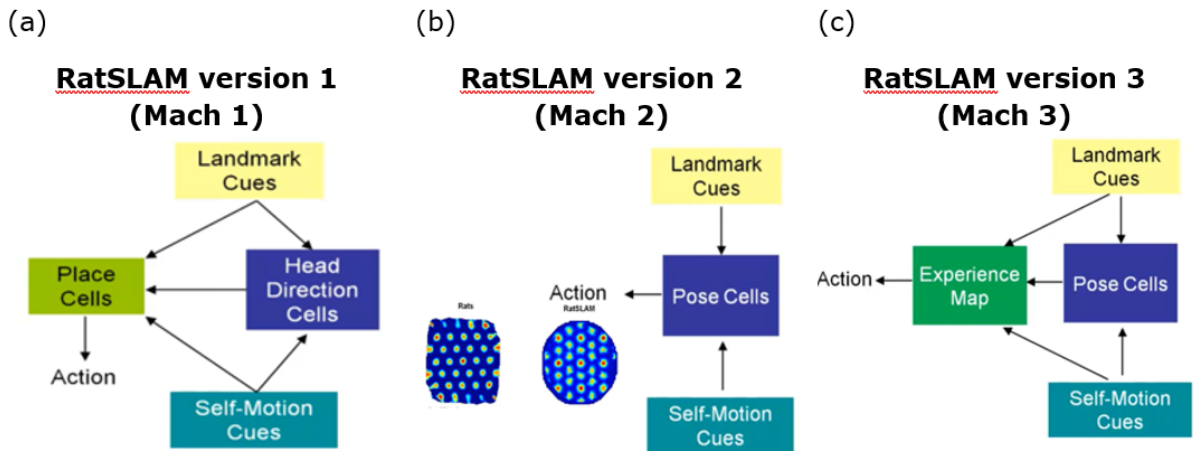


Figure 1.4 Overview of RatSLAM Versions [69]

1.4 Research Questions

Taken all of this previous research together, we wondered: (i) Is it possible to train a deep neural network to learn the representation of a 3D virtual reality environment (Donderstown) from 2D snapshots, to predict the position where a given snapshot was taken? (ii) Which type of deep neural network is most suitable? And how does it achieve its learning goal?

In reference to our research questions, we specifically built and trained a simple deep convolutional neural network (CNN) to learn part of Donderstown and tested its ability to generalize the entire environment. We were interested in determining whether computations guiding how humans learn a virtual environment can be captured using deep learning. Additionally, this model may also act as a preliminary indicator of how close we can give a simulated agent human-like spatially modulated neural coding and how this can be extended in the future.

As a side note, due to time constraints and limited resources, the model was built to learn about Donderstown via snapshots instead of self-navigation. In particular, we tested whether our model would demonstrate hippocampal-like spatial cells and homing vectors (individual coordinate prediction), and potentially predict the whole assembly of Donderstown as an event spatial map solely based on its learning of the static spatial representation environment.

(This page was intentionally left blank)

Chapter 2 : Methods

2.1 Overview

This project mainly utilizes the freeware and open-source toolboxes available online such as Python, Anaconda (Jupyter Notebook) and Keras. No living subjects are included in any part of this project. As the main data supplemented for the model testing were snapshot images from Donderstown, we decided to build a deep convolutional neural network (CNN) to process these images. To judge the novelty of our model, we have tried multiple ways of data feeding and adjustments of steps in the model. More related details will be explained in upcoming sections.

2.2 Our Selected Virtual Environment

Our datasets were created from snapshots of Donderstown. Donderstown was built using the Unreal Development Kit (81) (please refer to *Figure 2.1(a) Donderstown (top)* for illustration of the virtual town). It is a large scale, urban VR city with a complex layout of unnamed and curved streets, and irregularly outlined squares and parks inspired by the old-style German town, and is surrounded by a mountain range (47). As previously mentioned, the very first study using Donderstown VR city by Bellmund et al. (2016) showed that head direction and grid cell representations are exhibited even in imagination tasks without an actual movement (47). In another paper by Deuker et al. (2016), they have suggested that memory is related to spatio-temporal network mechanism of episodic memory (48). This indirectly suggested that spatially modulated hippocampal neurons could be engaged during memory recall tasks involving Donderstown.

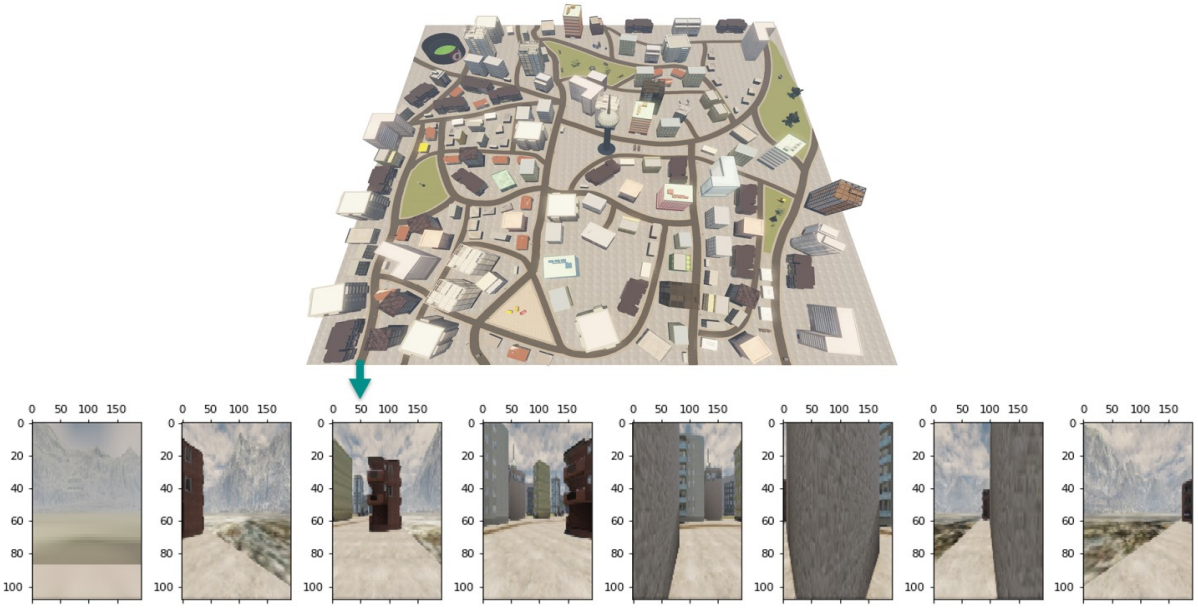


Figure 2.1 (a) Donderstown (top) and (b) Random Checking of Positional Angles (bottom)

2.3 Model Architecture

The most commonly used neural networks consist of:

- Recurrent neural networks (RNN) that are most frequently used in natural language processing and speech recognition due to its feedback connections feature (82);
- Autoregressive (ARe) models that use values from previous time series to predict future values (83); and the rapidly developing
- Convolutional neural networks (CNN) that perform object recognition via a sliding window method (84) to do classification, localization and detection that can also be used for object verification and style transfer (85).

On the other hand, there are also newer and interesting models to neuroscientists: e.g. the generative query network (GQN) that was developed by Eslami et al. (2018). In particular, the GQN can see a scene from one viewpoint and predict the same scene in another viewpoint (86). While there are many variations of these neural networks, we decided to build our model on top of the CNN basic structure (please refer to *Figure 2.2 A Typical CNN Architecture* for a brief concept of CNN general connectivity map).

2.3.1 Convolutional Neural Networks (CNNs)

We have chosen convolutional neural networks (CNNs) to be the basis of how we build the structure of our model as they fit what we needed in this project – image processing. More details about CNNs will be outlined precisely in what follows.

CNNs are generally feedforward networks and have been actively applied for visual task processing (87). Inspired by the neurobiology of visual cortex, CNNs were first proposed by Kunihiko in 1980 (88). However, the real backbone of CNNs was started by LeCun et al. who developed a CNN model named LeNet-5 (88). This LeNet-5 which is capable of obtaining image representations and visual patterns from raw pixels has multiple layers, and is capable of performing backpropagation (88). With the performance limitations detected by researchers, and along with the improvement of digital technology in hardware, datasets and algorithms (87), CNNs further evolved with many different models such as AlexNet, ZFNet, GoogLeNet, VGGNet-16, ResNet, Inception models, ResNeXt, SENet, MobileNet V1/V2, DenseNet, Xception models, NASNet/PNASNet/ENASNet, and EfficientNet (84). Despite their differences, their basic components of processing are very much alike (88).

Technically, the basic architecture of CNNs is composed of an input layer, at least one or more hidden layers which would include a combination of convolution, pooling and possibly normalization layers before the fully connected layers, classification layer and an output layer (84, 89), which can be visualised in the figure below (*Figure 2.2 A Typical CNN Architecture*). In other words, the hidden layers are made up of nodes/neurons, whereby each of them receives some inputs and has a learnable weight and bias which are capable of converting raw image pixels into class scores (90) based on user-defined parameters. Convolution layers are arranged in 3-D (height, width, depth/color channels) (89, 90), and

learn the feature representations of the input (87) by computing the dot products between inputs and filter values (84) before an optional non-linearity (87) activation is used (84).

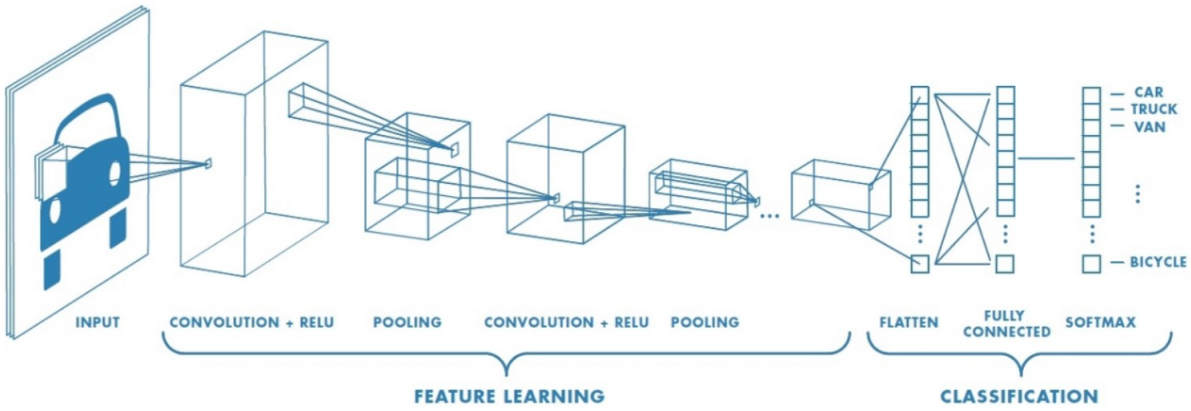


Figure 2.2 A Typical CNN Architecture [91]

Unlike in regular neural nets where each neuron in each hidden layer is fully connected to the previous layer and is functioning independently in the same layer, CNN nodes are only connected to a small region from previous layers, which helps to reduce overfitting and to generate a single vector class score arranged in the depth dimension for the image at the end of the model training (90). Next, pooling layers are used to down-sample the spatial dimensions (i.e. width and height) (90) which helps in speeding up calculations and potentially mitigate overfitting issues (84). Subsequently, fully connected (FC) layers will compute the class scores (90). FCs are fully connected to the previous layer and calculate the activation with matrix multiplication and a bias offset (84). Lastly, the loss/classification layer is used to monitor the training process by determining deviations between the true and expected labels (84) prior to obtaining the output layer.

2.3.2 Our Model Architecture

While CNNs form the blueprint for our model, the fine tuning process was the most time-consuming part in this project. Our test coverage included different hardware allocation (e.g. different GPUs, RAM size and hard disk storage type), data distribution of training and testing generators, values for numbers of grid_division, sampling size, layers of convolution (and the parameters inside such as filters, kernel size, activation, padding, strides; see appendix - *Simple Explanation of Parameters*), dense layers (and the parameters such as number of units, activation, L1 and L2 kernel/activity/bias regularizer), dropout, outputs (XY coordinates with and without angles), optimizers, learning rates, losses, metrics, steps and epochs, as well as different normalizations.

Discussing the most essential component in the model development, we have tried multiple different approaches while building our model architecture. To summarize, we generally concluded that the following elements help to train our model successfully:

(i) We started with 64 filters and systematically increased up to 256 filters for the convolutional layers in order to benefit from efficient data training and lower computation time. A lower number of filters would affect the model training.

(ii) We used a kernel size of (7,7) across all convolution layers, which produced better models than smaller kernel size. This is potentially because we did not resize our input images, but kept them relatively big at the original pixel dimensions (108,192,3).

(iii) Adding more convolution layers made the model learn better (we capped the amount of layers at 13, as additional convolution layers did not provide significant improvement to the training).

(iv) Activation in the convolution layers is a very important factor that determines whether the model learns. Relu for all convolution layers without combination with non-relu functions is the best option for our current case based on inspection over the outputs (see appendix-*Simple Explanation of Activation Function* to understand what Relu is).

(v) Utilization of dropout and stride between the convolutional layers did not affect our model training as much, but helped to save some computing time, down-sampled the inputs, reduced overfitting and perhaps helped to make the model more robust.

(vi) Using an initial learning rate (lr) of 0.0001 (1e-04) for our Adam optimizer (check out section *2.4.1.2 Optimizer* to learn about Adam) without the utilization of any learning rate scheduler contributes to a good model. However, the model's generalization ability gets improved slightly with lr=0.00001 (1e-05) – our ideal lr.

Due to time constraints from carrying out further parameter tuning, we decided to structure our model (*Figure 2.3 Model Architecture*) based on the statements we summarized above.

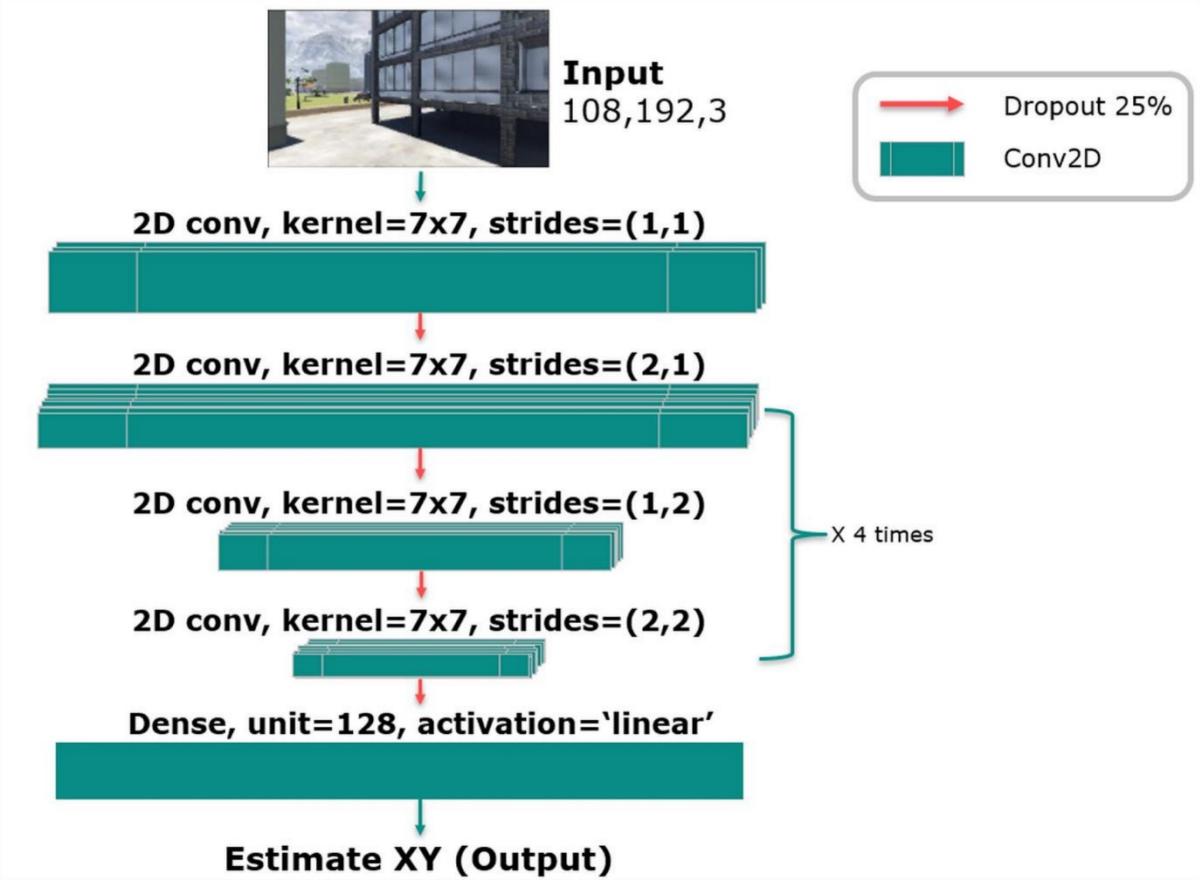


Figure 2.3 Model Architecture

Our model consists a total of 13 relu convolutional layers that were padded to avoid features down-sampling and significant features loss. Other than the 1st convolutional layer, all other convolutional layers get 25% dropout to prevent overfitting.

2.4 Model Training

The Donderstown snapshots data for this project consisted of 652,720 non-uniform, first-person perspective (egocentric) images at ground level from 10 different runs. These 652,720 images were derived from a total of 81,590 positions, which were taken at 8 different angles that differ by 45° (0°/360°, 45°, 90°, 135°, 180°, 225°, 270°, 315°). We performed simple validation of data points by plotting the coordinates into a topographic map for each run and across runs (overlaying all the 10 runs) for visual inspection (*Figure 2.4 Plotting of Data Points for Validation*). To further verify the usability of the snapshots, we have also decided to do visual inspection on all 8 angles of a given coordinate for every 100th coordinate of each run (*Figure 2.1(b) Random Checking of Positional Angles (bottom)*).

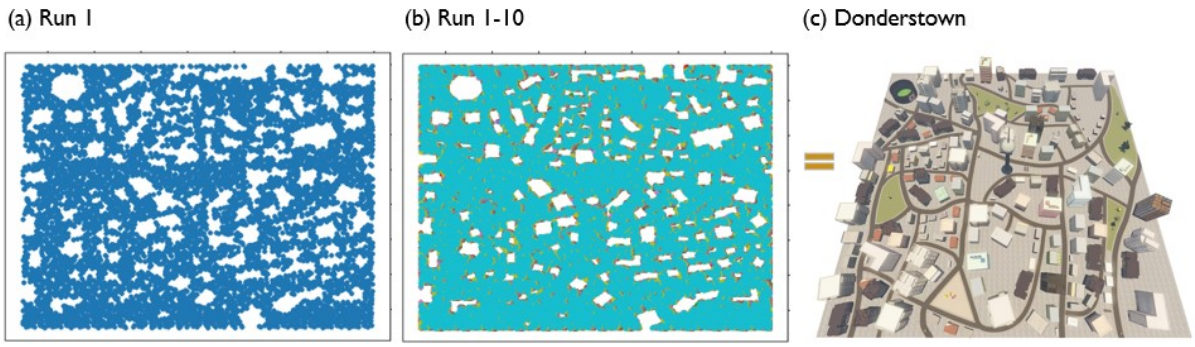


Figure 2.4 Plotting of Data Points for Validation

We plotted the positions onto a topographic map based on received labels of data to see if these plots matched a bird's eye view of Donderstown (c). (a) is the plot from single Run 1 and (b) is the overlay of all 10 runs.

To initiate image preprocessing, we calculated the group mean and group standard deviation prior to 2-steps normalization, i.e. local centering and data standardization. Normalizing the data in 2-steps meant: (i) we obtained post-mean-processing values close to 0, which generally helped speed up the learning and led to faster convergence, and (ii) we rescaled the data into a pre-defined range to ease the application for further algorithms (92). For both mean and standard deviation values, we performed the calculations across the width and height of RGB channels separately, which resulted in individual mean pixels across width and height for each of the 3 channel arrays (RGB).

Thereafter, data of each run was extracted by first redistributing 50% of all the data into training data (**X**, i.e. oblique viewing angles), while the rest were defined as testing data (**✚**, i.e. cardinal viewing angles)) (*Figure 2.5(a) Angles Redistribution*). Meanwhile, the maximal and minimal of both X and Y was respectively obtained. Based on the maximal X and Y obtained, we divided the space from 0 to maximal XY values evenly based on the chosen parameter (in our case, we fixed the value as 25, please refer to *Figure 2.5(b) Sample Illustration of 25 Evenly Divided Grids at Size of Topographic Map of the Run* for clearer showcase). These new grids then represented the new XY coordinates. Original XY coordinates that were closest to these new XY coordinates were selected. The reason why sampling locations of the snapshots were predetermined to uniformly cover Donderstown was to prevent the model from being biased towards one specific location(/position/coordinate) in the environment. Next, the related snapshots that further matched both the coordinates and desired angles criteria were extracted, with the X and Y values being downscaled to values between 0 to 1 using the formula (i.X - minX)/(maxX – minX) and (i.Y - minY)/(maxY – minY). Along with the group mean and standard deviation values, these data are stored in a table (pandas dataframe), which will later be used by the data generator to yield data for the model training.

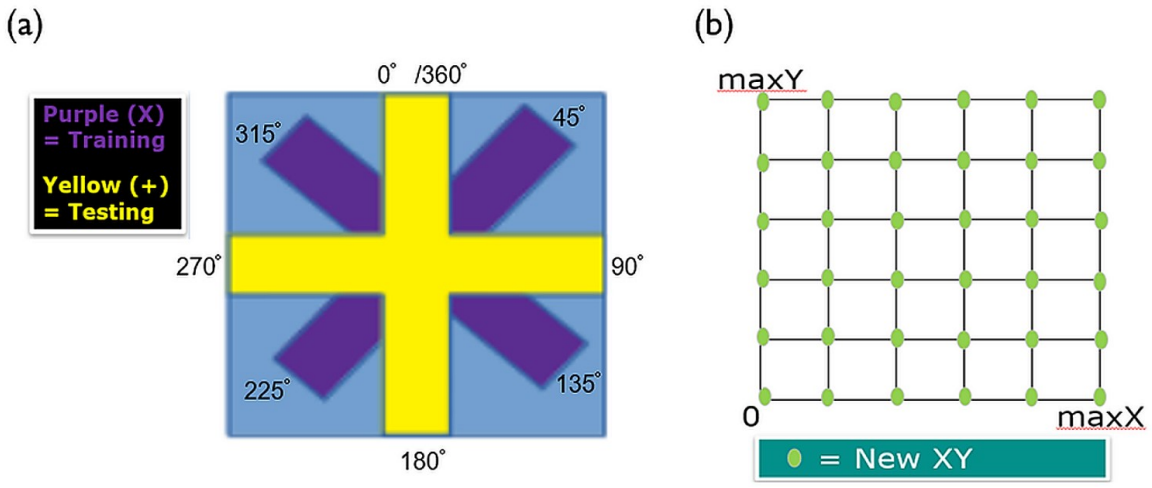


Figure 2.5 (a) Angles Redistribution (b) Sample Illustration of 25 Evenly Divided Grids at Size of Topographic Map of the Run

To ensure our self-customised method of data generation and normalization worked correctly, an additional step was built to randomly choose the data from the table array at a rate based on the parse-in value specified for the variable 'sampling_size', in a non-repetitive manner for every batch. In our case, 8 was the sampling size used for random visual checking of the data. The program-selected data were then normalized by first conducting a 2-steps normalization with (i) (img - groupMean) / (groupStd) for local centring and data standardization, and (ii) converting the image values into the range of between 0 to 1 by adapting the new image values into the formula: image = (image - np.min(image))/(np.max(image) - np.min(image)). Both the original image and end normalized product are then plotted side-by-side with the histogram and saved into 8 different figures to show the differences for visual inspection (*Figure 2.6 Example Figure of Original vs Normalized Image*).

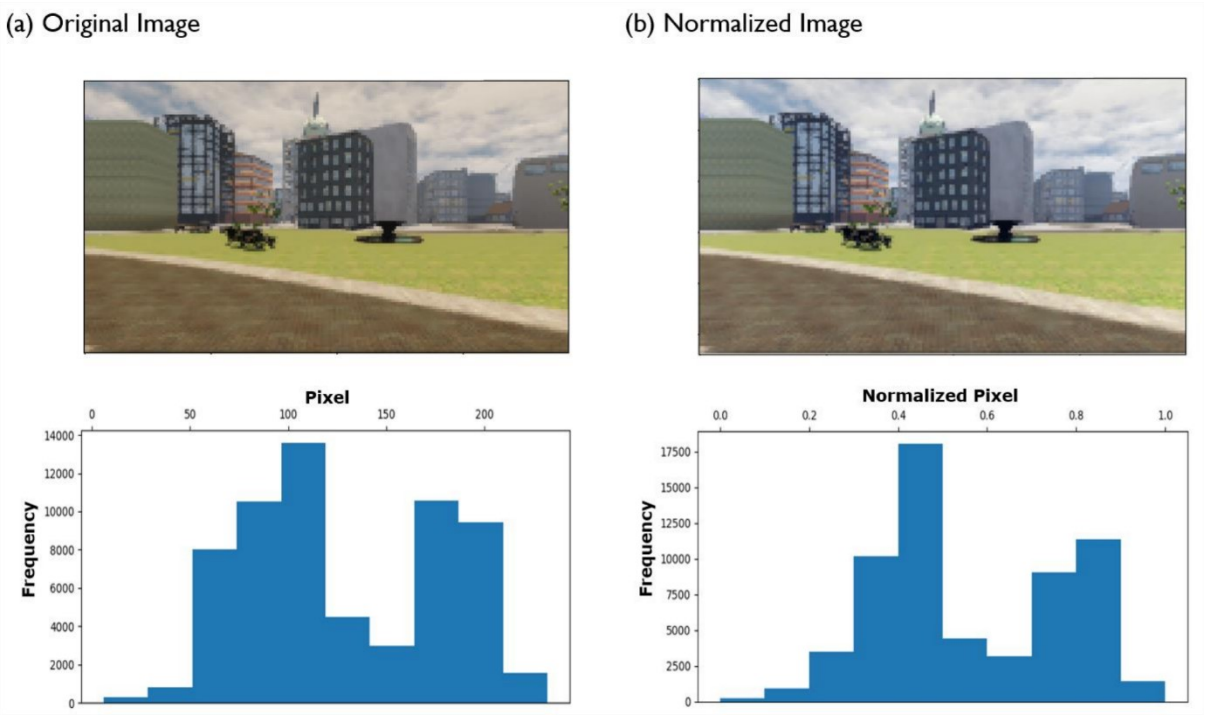


Figure 2.6 Example Figure of Original vs Normalized Image

The data generator function then repeatedly paired n number of images from the table array based on user-specified batch size in a non-repetitive manner, and underwent the similar 2-steps normalization processes mentioned above, which would yield an array of images and xy values (i.e. location/position labels) for the model. The model will take the batch's images from the training generator as input in the feed forward phase and learn the corresponding xy values, whereby the learning is validated by using the batch's dataset from the testing generator, i.e. using the images as input, and predict the xy values before comparing the deviation from the correct xy values, and performing backpropagation to update the weights (93) after each epoch. In general, we can define backpropagation as a weight update algorithm that takes actual and desired output into account (94).

2.4.1 Data Generator, Optimization and Hyperparameter Search

2.4.1.1 Data Generator

Per previously mentioned, in order to feed the snapshots into our model, we decided to use a Python yield function to build a memory efficient data generator, which keeps the data on disk until the model requests a new sample (95). Each input and output sample consisted of a snapshot of Donderstown with the corresponding X,Y position, which we combined into a batch of size 8.

2.4.1.2 Optimizer

During model training the weights were adjusted to minimize the mean absolute error between the real X,Y position and predicted X,Y position. Optimization was done using a variant of Stochastic Gradient Descent (SGD), namely Adam. First introduced by Kingma and Ba in 2015, Adam (adaptive moment estimation) was claimed to blend the benefits of both the modified SGD: AdaGrad (adaptive gradient algorithm) and RMSprop (Root Mean Square Propagation) (96). The authors suggested that this method is capable of working well with non-stationary objectives and sparse gradients, suitable for large datasets and parameters, while magnitudes of parameter updates are unaffected by rescaling of gradients. Additionally, it requires little tuning as Adam uses a moving average of the parameters, which helps in removing noise in the (batched) gradient updates and assures convergence to a robust step size.

Stochastic gradient descent (SGD) is a first-order iterative optimization algorithm that estimates the error gradient of the model's current state by looking for global minimum of one example at a time (step), and calculates the gradient (which is later being used to calculate the weights). These are repeated for all the examples throughout the training dataset in a single epoch (97) before performing backpropagation to update model's weights (98). The Adam optimizer on the other hand is a modified SGD (99), which classifies as a second-order optimization and calculates the exponential moving average for the gradients $\mathfrak{m}$ (estimated mean) and the squared gradients υ (estimated variance). Both $\mathfrak{m}$ and υ are then bias corrected to prevent them decaying to their initial value, which is typically initialized as 0. The weight is then updated with the calculation as follows:

$$W_{t+1} = W_t - \frac{\alpha}{\sqrt{\hat{\upsilon}_t} + \varepsilon} \bullet \hat{\mathfrak{m}}_t$$

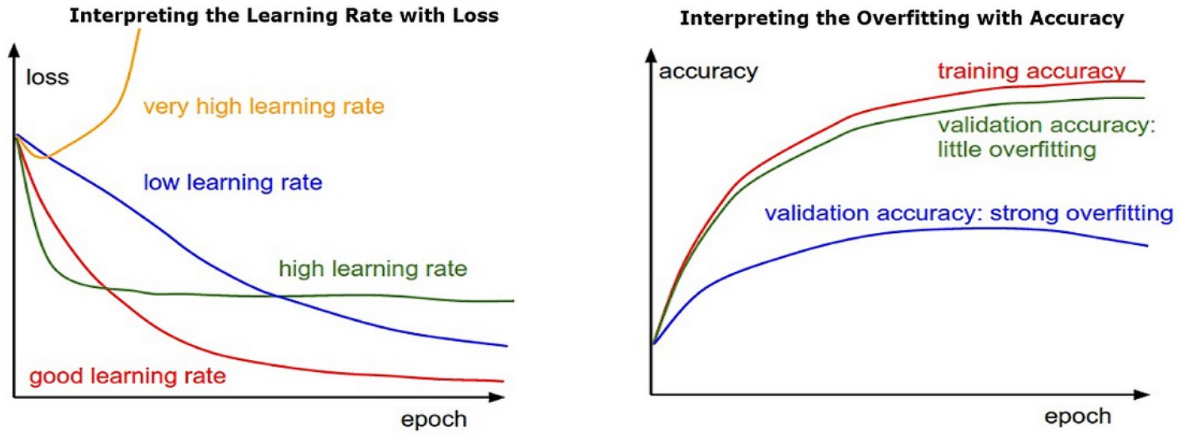
with W_{t+1} denoting new weights, W_t denoting old weight, α denoting learning rate, ε denoting a small value avoiding the division by zero, $\hat{\mathfrak{m}}_t$ denoting bias-corrected exponential moving averages, and $\sqrt{\hat{\upsilon}_t}$ denoting square root of the bias-corrected variance.

Since the introduction on Adam, the authors have also proved that Adam outperformed AdaGrad, RMSProp, SGDNesterov and AdaDelta on MNIST. Additionally, Adam has also been recommended as the default algorithm (92) by the Stanford Vision and Learning Lab. Taking all of these strengths into account, we chose Adam as the optimizer for our model.

2.4.1.3 Hyperparameter Search: Learning Rate

The only argument that we have pre-defined in our optimizer was the initial learning rate (lr). The lr determines the speed of convergence towards the optimal weights as the specified value defines the weights' adjustment in regard to the loss gradient descent (100). It is especially important to decide the right lr to be used in the model in order to enable the model to converge and save training time and resources (100). A very high lr (e.g. 0.1) may miss the optimal solutions (global optima) (100), leading to an unstable training process that mainly picks up on sub-optimal solutions (local optima) and causes pre-mature convergence (98, 101, 102), and possibly lead to protracted oscillating between certain weights (98). A lr that is too small (e.g. 0.00000001) would slow down the time to reach convergence (100). This slowing could also lead to a tendency to mainly converge into the closest local minimum (103).

While it is a time-consuming trial-and-error process to select the best lr for the model (100), it is always good to perform sanity checks before intensive model tuning such as: (i) tracking the loss function over time, and (ii) accuracy (92). Referring to the figure (*Figure 2.7 Tracking Quantities for Learning Rates: (a) Loss and (b) Accuracy*) adapted from the Stanford Vision and Learning Lab below, we need to achieve a loss function that decreases steeply but not overly drastic in reasonable time (converging) that continues with a stable decreasing trend (moving towards the optimal solution), and that resembles the red smooth decreasing plot in *Figure 2.7 (a)*. We also have to ensure that the accuracy between the training and validating datasets does not deviate too much from each other to avoid overfitting, just like the red and green plots in *Figure 2.7 (b)*. Theoretically, a model that is able to generalize fully will not be showing any overfitting or underfitting (104), but this is unlikely to happen in practice (105). Overfitting and underfitting are terms to label the deficiency of the model's performance (104). Overfitting often refers to a model that performs well on the training data, but not on the testing data. During overfitting, the model learns both the signal and noise in the training data to an extent, where there is a negative impact on the model performance when learning new data (106). Conversely, underfitting refers to a model that could not generalize both the training and testing data (105). Similarly, "a model that is underfit will have high training and high testing error while an overfit model will have extremely low training error but a high testing error" (107).



(a) A cartoon depicting the effects of different learning rates. With low learning rates the improvements will be linear. With high learning rates they will start to look more exponential. Higher learning rates will decay the loss faster, but they get stuck at worse values of loss (green line). This is because there is too much "energy" in the optimization and the parameters are bouncing around chaotically, unable to settle in a nice spot in the optimization landscape.

(b) The gap between the training and validation accuracy indicates the amount of overfitting. Two possible cases are shown in the diagram on the left. The blue validation error curve shows very small validation accuracy compared to the training accuracy, indicating strong overfitting (note, it's possible for the validation accuracy to even start to go down after some point). When you see this in practice you probably want to increase regularization (stronger L2 weight penalty, more dropout, etc.) or collect more data. The other possible case is when the validation accuracy tracks the training accuracy fairly well. This case indicates that your model capacity is not high enough: make the model larger by increasing the number of parameters.

Figure 2.7 Tracking Quantities for Learning Rates: (a) Loss and (b) Accuracy[92]

2.4.1.4 Loss Functions

In the interim, the loss function calculates how well the model accomplishes its intended task under the general assumption that the lower the value the better the performance in the dataset. In our model, we have used mean absolute error (mae) for loss, and mean squared error (mse) and accuracy (acc) for metrics. Mae is the average of the absolute differences between actual and predicted values in the test (i.e. the average prediction error) regardless of the direction of the deviation (always positive differences) (108). Mae is less sensitive towards outliers and useful for multimodal distributions, where the median tends to be the optimal prediction (108). Mse on the other hand is similar to mae, except it squares the resultant mae value (108). As mse squares the errors, the values amplify significantly, making it sensitive towards the outliers and typically results in a normal distribution around the mean (108). Lastly, the acc metric computes the mean accuracy across predictions (109). As for acc in our model, we did not specify the exact type of accuracy metric to be used, nor set any extra parameters, rather, we let the backend decide what to use. Potential accuracy metrics include binary accuracy for problems with two classes, categorical accuracy for multi-classes, and sparse categorial accuracy for sparse targets (109). In our experience, sparse categorical accuracy happened seemingly due to our sparse target's input (i.e. Our inputs were not specifically categorised or classified [e.g. 0 is cat, 1 is dog], but were being numerically vectorized using the feature weights to get our xy position labels. The actual label values were then being used to compare with the predicted label values to compute the accuracy).

(This page was intentionally left blank)

Chapter 3 : Results

3.1 Overview and Method Recap

This project started out by training a convolutional neural network (CNN) to find out if it is able to learn a representation of Donderstown, and if it can generalize to new unseen locations. To do this, it implies that the model has to be capable of storing a simple representation of Donderstown in its weights. After some initial testing when first building the model, the model was trained on four cross-diagonal angles and tested on another four out of the eight provided angles (of different snapshots) for a single position. This would eliminate the possibility of trivial associative match if we train all images, and it will be good to check if the model is robust enough to learn and estimate.

3.2 Training a Neural Network to Estimate Locations in Virtual Reality

In this part, we will show plots to demonstrate that our model successfully learns a spatial representation of Donderstown. We used the model mentioned in the previous chapter, with the convolution layers using relu as activation and fully connected dense layer using linear as the activation. Also, note that the model has only divided the dataset into two, i.e. training and testing. Thus, testing dataset is equivalent to validation of our model.

3.2.1 Training and Testing Plots

To validate the performance of our chosen model's architecture, we first checked over the training loss and validation loss values of the 100th epoch post-training, and realized that the model showed small overfitting values in general (recap: the loss was calculated using mae). Next, we inspected the results, i.e. the losses (both training and validation losses) and judged that lr=1e-06 (*Figure 3.2(e)*) is not ideal as an initial lr. On the other hand, both lr=1e-04 (*Figure 3.2(a)*) and lr=1e-05 (*Figure 3.1(a)*) indicated good lr for both the training and testing datasets. The lr=1e-05 (*Figure 3.1(a)*) showed the most consistent trend over both the training and testing datasets. The most significant update happened around the 30th epoch and continued to improve consistently in our model that trained with 1000 steps and 100 epochs. We further validated the losses with the mse (*Figure 3.1(b)*) trend and judged that the losses plotted were correct. Thereafter, we checked the accuracy, and again, lr=1e-06 (*Figure 3.2(g)*) proved itself as a bad hyper-parameter with highly inconsistent trends for both the training and testing datasets, while lr=1e-05 (*Figure 3.1(c)*) demonstrated a smaller overfitting as compared to lr=1e-04 (*Figure 3.2(c)*). A good model will keep the overfitting as minimal as possible, since a perfect model with no overfitting nor underfitting is practically non-achievable. By comparing the estimation ability, lr=1e-05 (*Figure 3.1(e)*) showed better generalization (with more evenly distributed prediction per indicated by the red dots, and lower error per indicated by the

density of the black line in the maps), as compared to lr=1e-04 (*Figure 3.2(d)*). In contrary, lr=1e-06 (*Figure 3.2(h)*) did not achieve a proper learnable weight for estimation and thus could not estimate the XY location correctly. In sum, we decided that lr=1e-05 is a better fit for our model.

We used the validation set for all subsequent evaluation figures. We plotted the XY Euclidean error with a cartesian to polar plot (*Figure 3.1(d)*) in order to have a clearer overview of the deviation between the actual and predicted value. Based on the figure (*Figure 3.1(d)-left*), we observed that our model works well with very low error as shown by the dark purple dots which clustered mainly around 0. On the other hand, the light purple dots also appeared to be spreading around all angles without clustering to specific angles. To further illustrate if the model has an angle bias tendency, we plotted a histogram (*Figure 3.1(d)-right*). The result showed that the angles were mostly random across all angles.

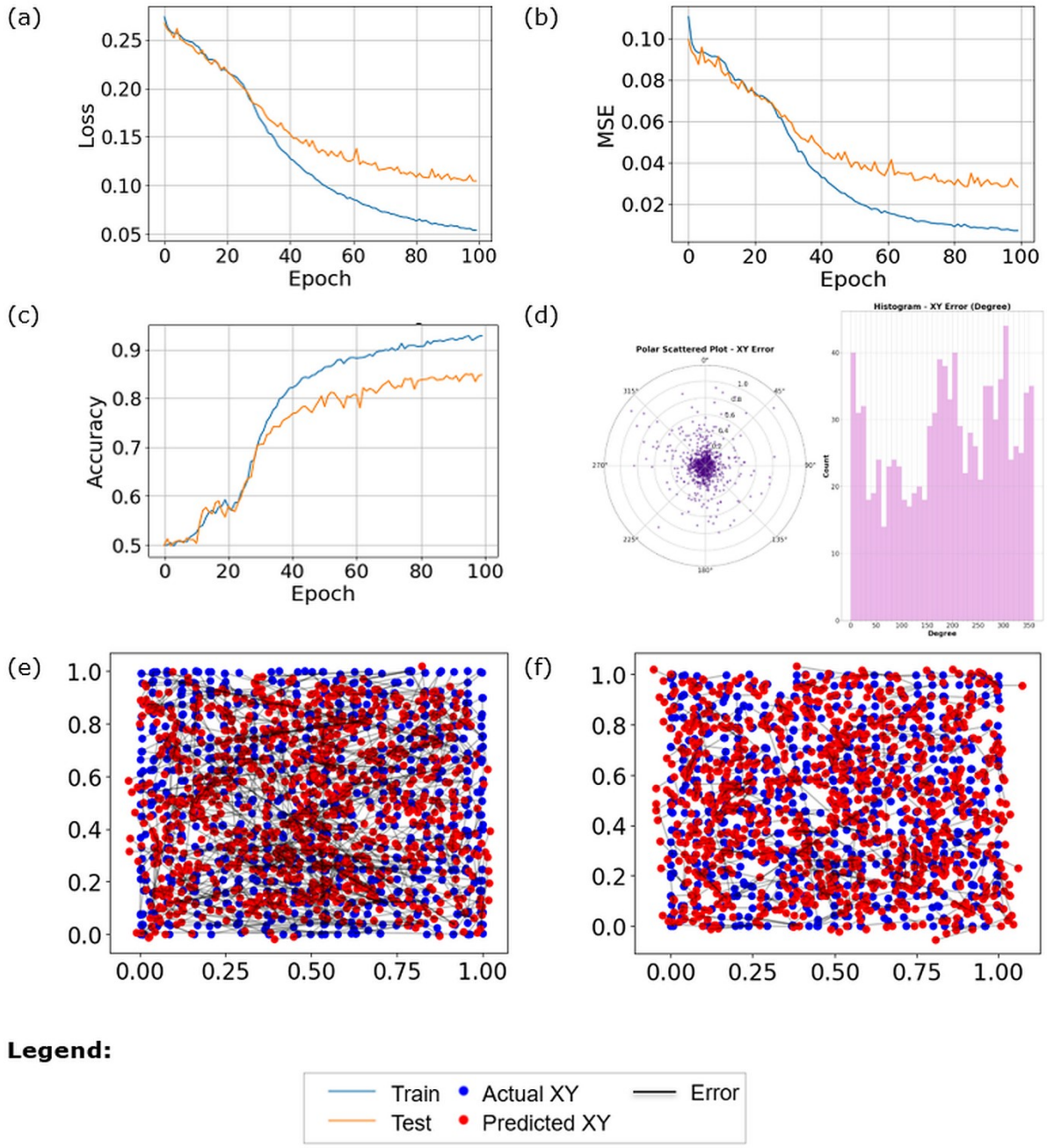


Figure 3.1 Chosen Model's Performance (1e-05)

(a) Model's (1e-05) Loss Plot. (b) Model's (1e-05) MSE Plot. (c) Model's (1e-05) Accuracy Plot. (d) Model's (1e-05) Euclidean Error Plot (left) and Degree Error Plot (right). (e)Model's (1e-05) Generalization on Testing Dataset. (f) Model's (1e-05) Generalization on Training Dataset.

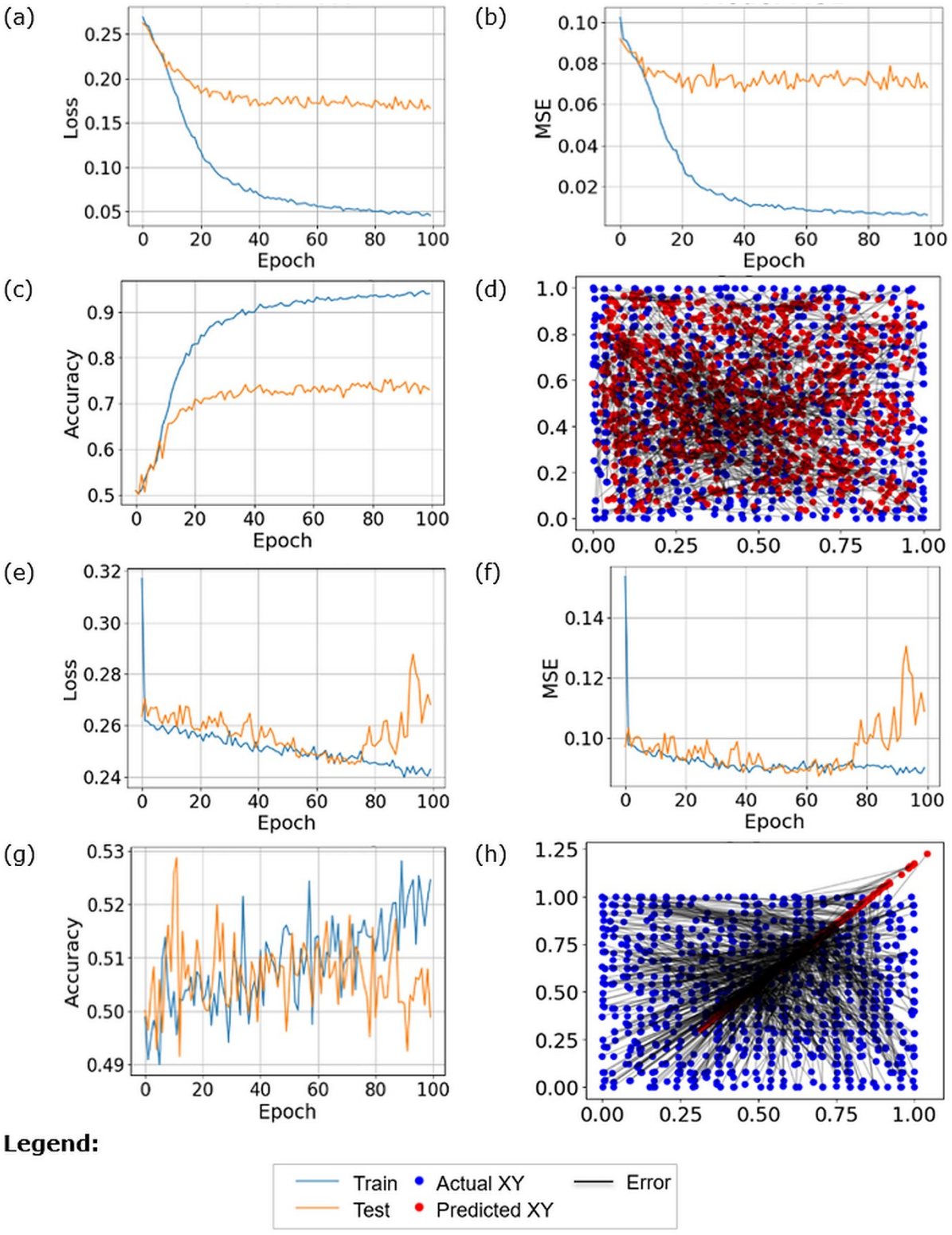


Figure 3.2 Model's Performance (1e-04 and 1e-06)

(a) Model's (1e-04) Loss Plot. (b) Model's (1e-04) MSE Plot. (c) Model's (1e-04) Accuracy Plot. (d) Model's (1e-04) Generalization on Testing Dataset. (e) Model's (1e-06) Loss Plot. (f) Model's (1e-06) MSE Plot. (g) Model's (1e-06) Accuracy Plot. (h) Model's (1e-06) Generalization on Testing Dataset.

3.2.2 Chance Level: Mismatch Data Feeding's Testing Plot and Model Performance Without Training

To further decide if our model learns correctly, using the same chosen parameters, we shuffled the data by randomising the snapshots so that input and output were incorrectly matched. This allowed us to see if the model would still manage to learn any representation and gave us an estimate of the chance level of the model. Referring to *Figure 3.4 Chance Level Trial*, as the input data were incorrect, the model seemed to be incapable to converge correctly (*Figure 3.4(a)*) and consequently was unable to predict all the outputs well into both the training and testing phases (*Figure 3.4(b)*). This suggests that our model was able to learn very fine grained representations and thus the unmatched data made it almost impossible to decode a spatial memory map correctly. The polar plot further showed that the model ran into high errors (*Figure 3.4(c)*), while the averaged activation map displayed that activation was inconsistent (not shown in figure).

We then wanted to know if the model could do a prediction without training (i.e. without running the .fit_generator command). Based on the figure (*Figure 3.3 Model Prediction without Training*) below, the model predicted every output coordinate as (0,0), as there were no trained weights and bias values to calculate the potential output correctly. This implementation consequently led to big error and a tendency to cluster between 0° to 90° (matching pattern between the predicted output representation map (*Figure 3.3(a)*) and the polar plot of Euclidean errors (*Figure 3.3(b)*)).

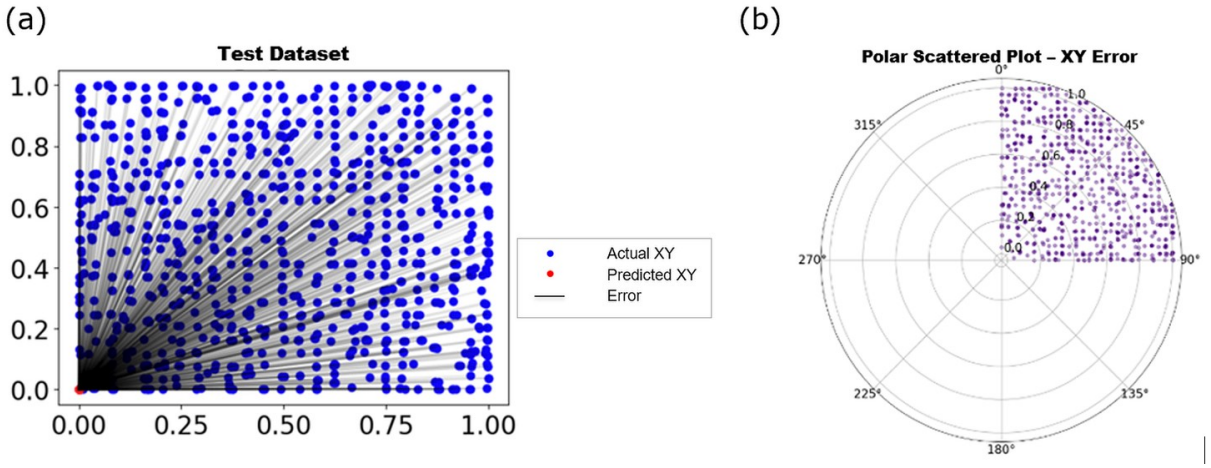


Figure 3.3 Model Prediction without Training

(a) Generalization on Testing Dataset. (b) Euclidean Error Plot.

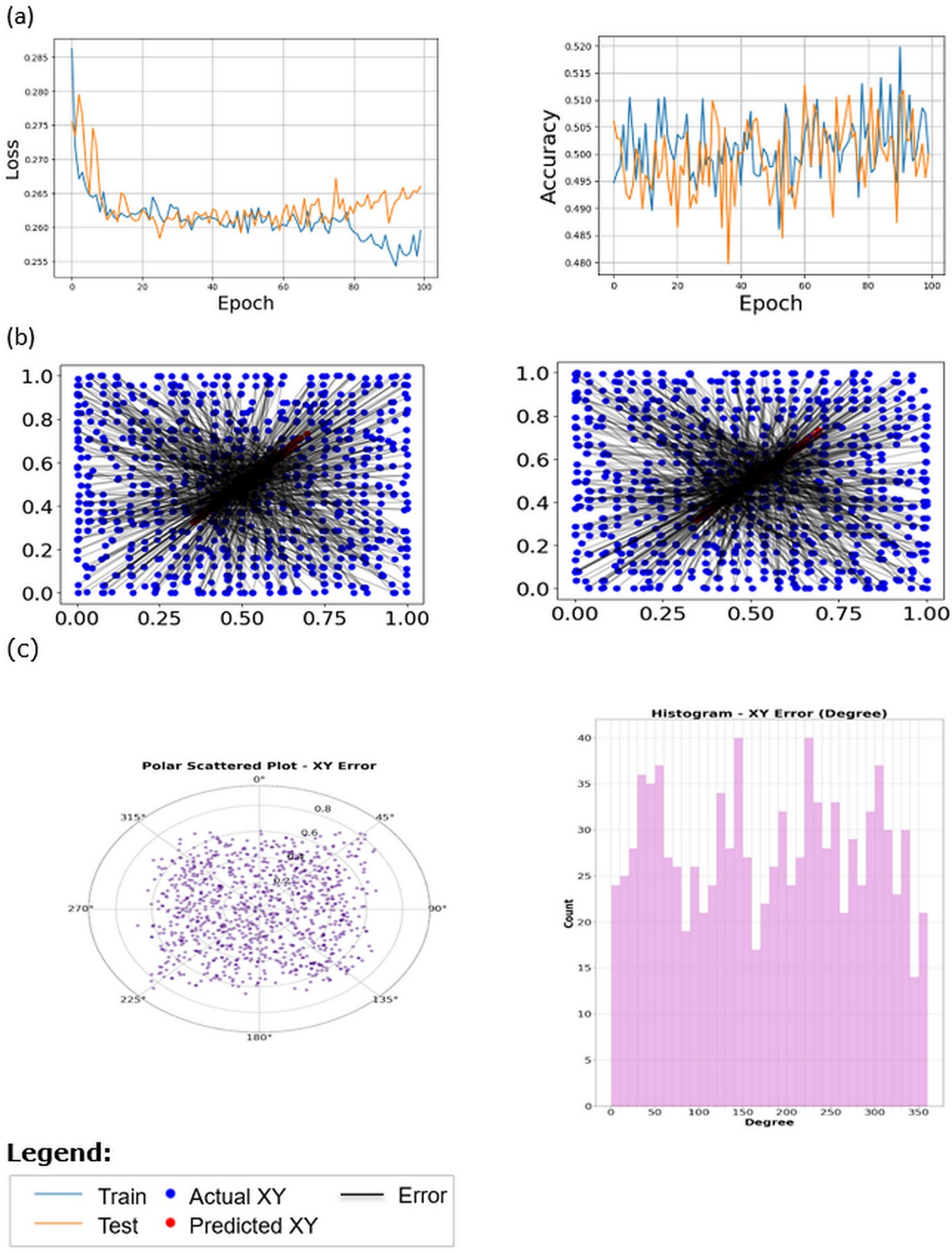


Figure 3.4 Chance Level Trial

(a) Loss and Accuracy Plots. (b) Generalization on Testing Dataset (left) and Training Dataset (right). (c) Euclidean Error Plot (left), Degree Error Plot (right).

3.2.3 Different Proportion of Data Splitting

While our presented model was using a data split of 50%-50% for the training and testing datasets based on the angles, we also tried to validate the model by using 20%-80%, 50%-50% and 80%-20% for training and testing datasets respectively. To do this, we shuffled data rows, combined the data of all 10 runs, which matched the pre-criteria (i.e. closest coordinates) into a single pandas data frame, and split the percentage accordingly before the generator does its task. Eventually, we found out that our model was effective (*Figure 3.5 Different Data Splitting Comparison*). First, provided that all training was performed using lr=1e-05, all of them showed good lr per modelled in the loss plots, with train-test pairs of 50%-50% and 20%-80% showed possible overfitting as the validation loss (*Figure 3.5(b)*) and validation mse values (*Figure 3.5(d)*) were higher than the training loss (*Figure 3.5(a)*) and training mse values (*Figure 3.5(c)*) respectively. Accuracy plots *(Figure 3.5(e) and (f)*) on the other hand proved the claim and showed that a dataset with a lower percentage (i.e. smaller pool of total data) would have better accuracy in general regardless if it is a training or testing dataset. From the plots analysis, we realised that: (i) the 80% training set has an accuracy almost similar to its 20% test set (i.e. almost no overfitting/underfitting), (ii) the 20% training set has higher accuracy than its 80% testing set (i.e. with a higher overfitting scale than its 50%-50% counterpart), while (iii) the 50% training set has higher accuracy than its 50% test set (i.e. some small overfitting). These phenomena are correct because a lower training percentage (e.g. 20%) indicates less samples available and higher repetition during training, hence lower training errors (thus smaller training loss) and more overfitting (i.e. perform well on the training data but not over the testing data). This however leads to poorer generalization to the testing set. On the contrary, as the 80% training set has more examples for the model to learn, it is thus able to estimate the testing set more easily. Having 50% on the side reflects almost having a sufficient amount of examples, but probably does not cover all features present in the testing set, which may lead to some small overfitting. In sum, the model needs sufficient training experience to generalize.

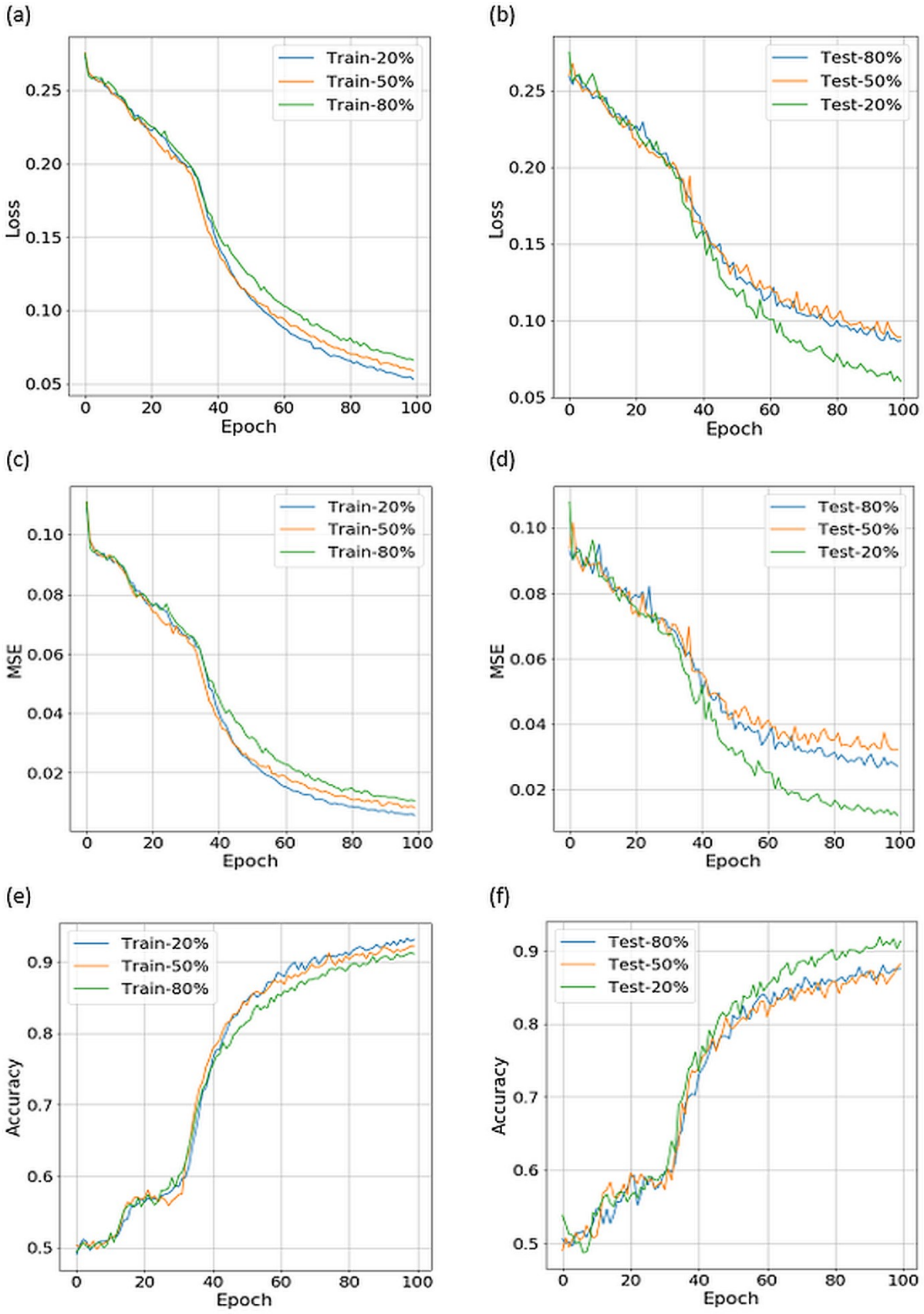


Figure 3.5 Different Data Splitting Comparison

(a) Loss Plot. (b) Validation Loss Plot. (c) MSE Plot. (d) Validation MSE Plot. (e) Accuracy Plot. (f) Validation Accuracy Plot.

3.3 Model Development and Sensitivity to Hyperparameters

3.3.1 Model Performance Under Different Epochs

To investigate how training epochs may affect the model performance such as its ability to generalize, we tested our model with 10 to 100 epochs at an interval difference of 10 epochs. We realized the model provided good estimates starting around 50 epochs (*Figure 3.6 Effects of Training Epochs on Model Prediction*), while further training slowly aids to decrease the loss values and increase the accuracy as shown in Figure 3.1(a) & (c) (and thus improved generalization ability).

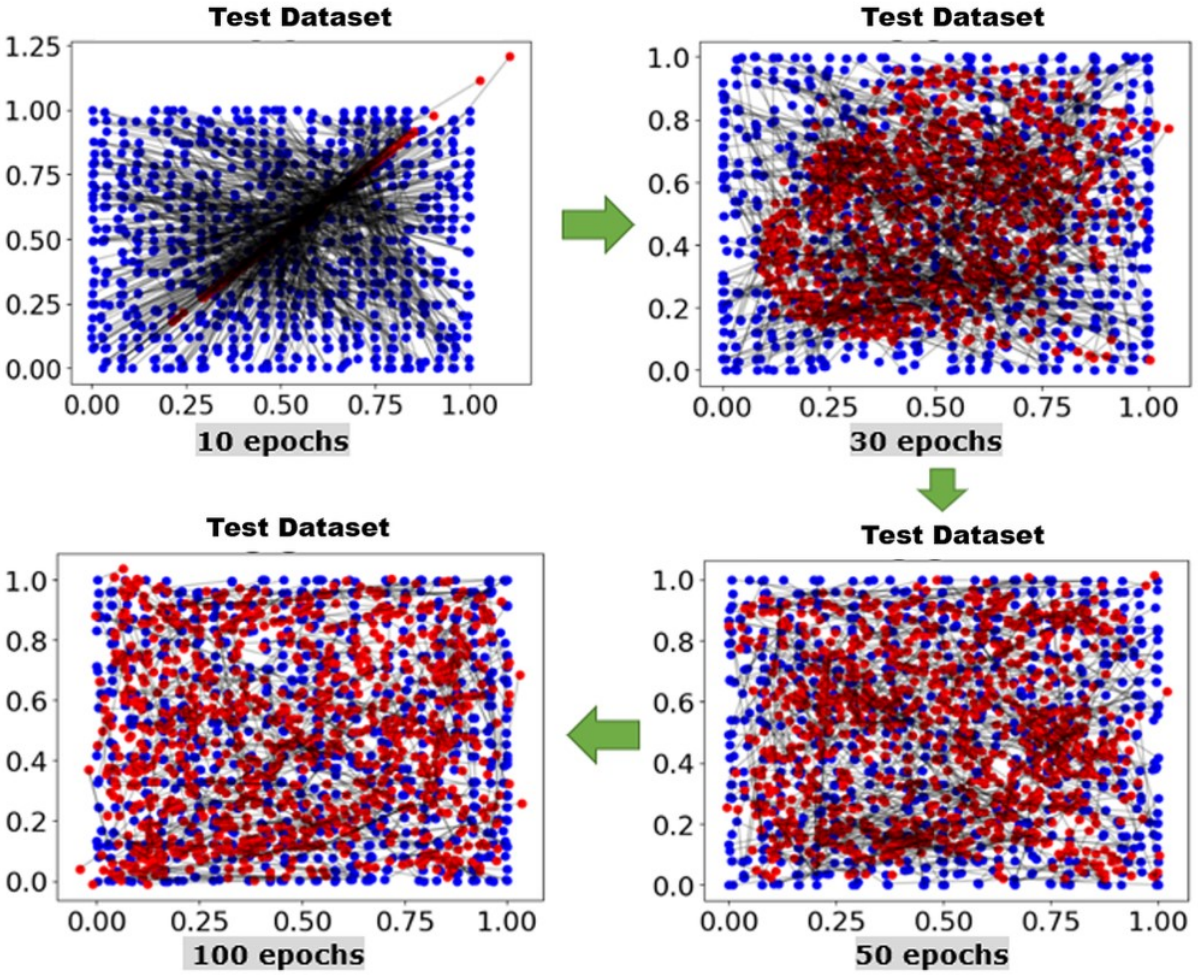
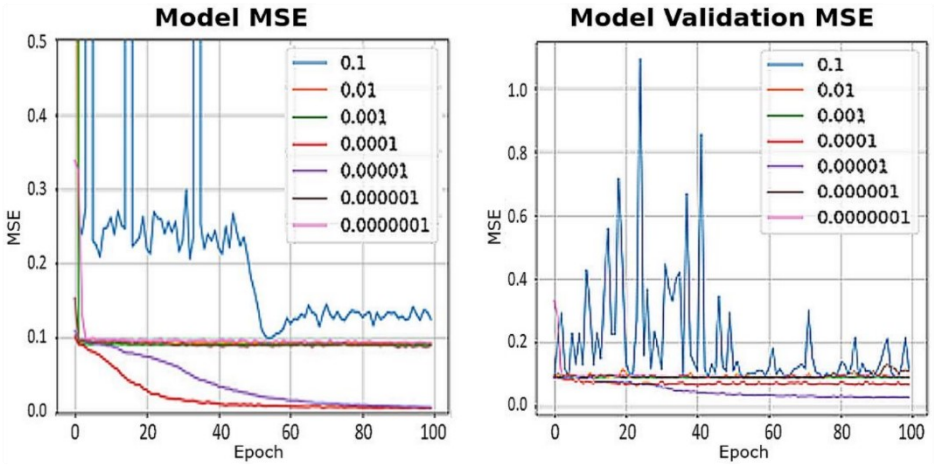


Figure 3.6 Effects of Training Epochs on Model Prediction
The generalization of the model starts to be more effective around the 50th epoch.

3.3.2 Different Learning Rate (lr) on Model Performance

As a selective expansion to section 3.2.1 above, and to see the effects of initial lr on the model training, we tested 7 different lrs: 0.1 (1e-01), 0.01 (1e-02), 0.001 (1e-03), 0.0001 (1e-04), 0.00001 (1e-05), 0.000001 (1e-06) and 0.0000001 (1e-07). We chose 0.00001 (1e-05) as the lr for our model that obtained the best score in general (see *Fig. 3.7 Effects of Learning Rate on Model Prediction* for illustration). From the loss plotting which was akin to the mse plotting, we observed that lr=0.0001 (1e-04) had the smallest training loss, but not the validation loss. In fact, lr=1e-05 has MSE trends that resemble each other more in both the training and testing (validation) (*Figure 3.7(a)*) datasets, with lower difference in values between them, making 1e-05 the ideal lr choice for the model. Lastly, we looked at the accuracy plots (*Figure 3.7(b)*), and lr=1e-05 is deemed the best model in terms of high efficiency and smallest overfitting.

(a)



(b)

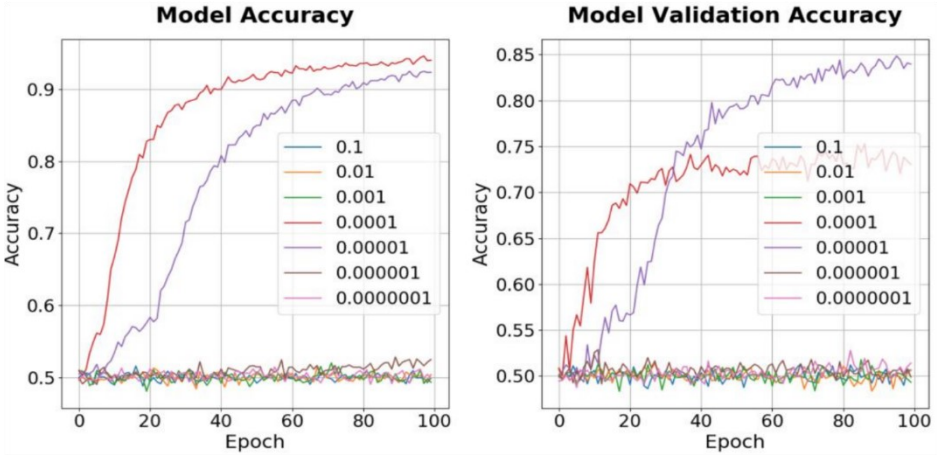


Figure 3.7 Effects of Learning Rate on Model Prediction
(a) MSE Training and Validation Plots. (b) Accuracy Training and Validation Plots.

3.4 Hidden Layer Activations Represent Spatially Informative Cells

3.4.1 Relationship between Dense Units (Nodes) in Last Hidden Layer to Our Model Performance

Knowing that our model could learn the spatial layout of Donderstown, we were interested in what activation pattern in each of the units could be attributed to the precise estimation of the input's XY location. Consequently, we investigated the last hidden layer by retrieving the activation of each node that stretched over all the positions before shrinking them into a squared bin of 2D histogram, and then further smoothed them using a Gaussian filter. This subsequently resulted in an 'activations map' plot. We found that nodes of the model learned Donderstown locations mainly via activations around the borders and corners. Based on inspection, all sides of borders and corners were covered among the 128 units of nodes applied (refer to *Figure 3.8 Border-like and Corner-like Units*). This is especially noticeable when we tried to average the activations of all the nodes to generate a single averaged activation map (*Figure 3.9 Averaged Activations*).

We initially deduced that the model is able to judge the spatial environment by simply using a combination of two gradients, i.e. it decodes an output (XY position) by using a single axis gradient in one node and combine with another node to compute the estimation. Interestingly, we realized that the model may actually be capable of learning even more fine-grained representation than expected after we plotted the mean activations across the x- and y-axes into line plots, as both the axes showed different level/trend of activities. Meanwhile, we also assumed that corner-like units in the model could be a conjunction of boundary vector representations (resembled by the border-like units).

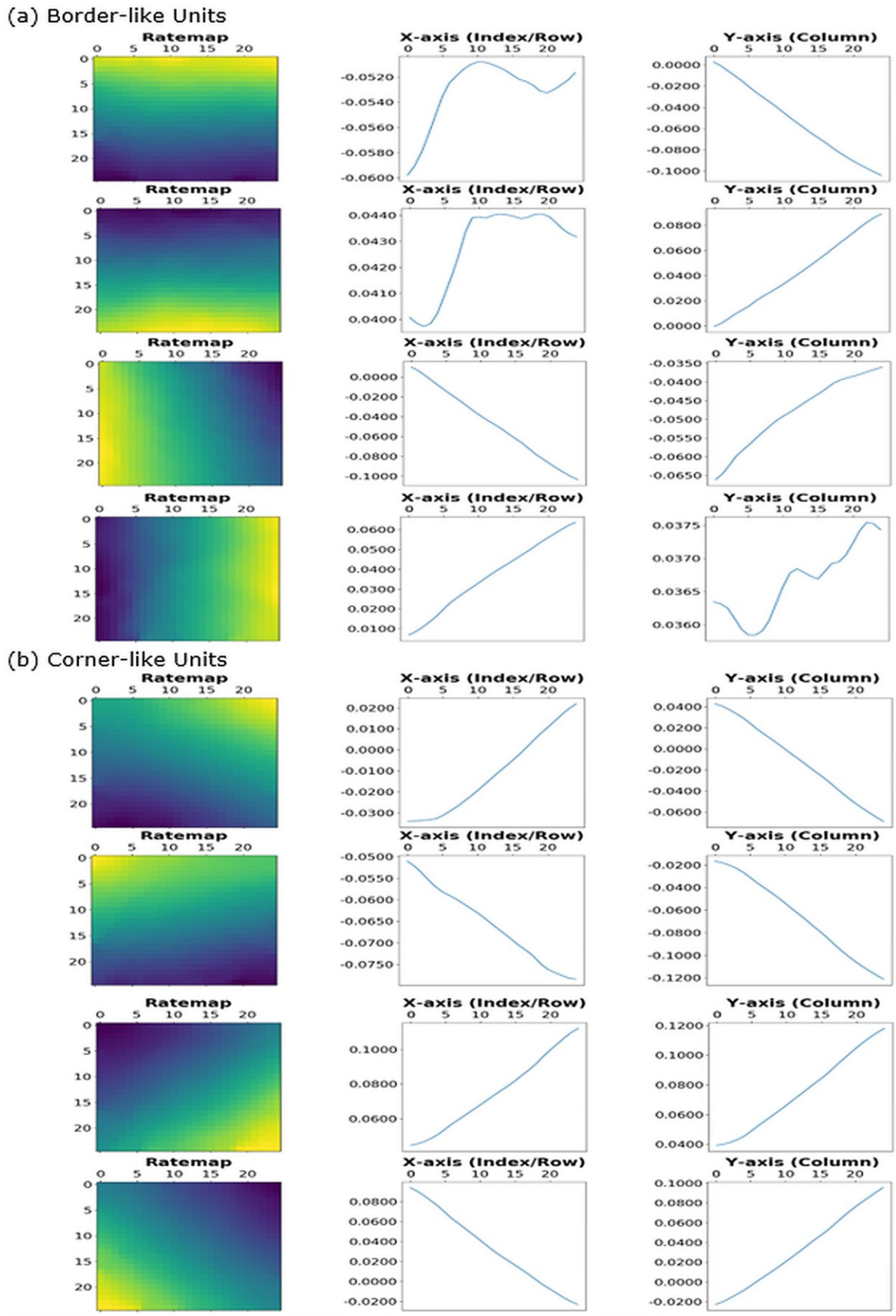


Figure 3.8 Border-like and Corner-like Units

(a) Border-like Units. (b) Corner-like Units (possible conjunction of boundary vector cell / hinting possible transformation of corner units into border units).

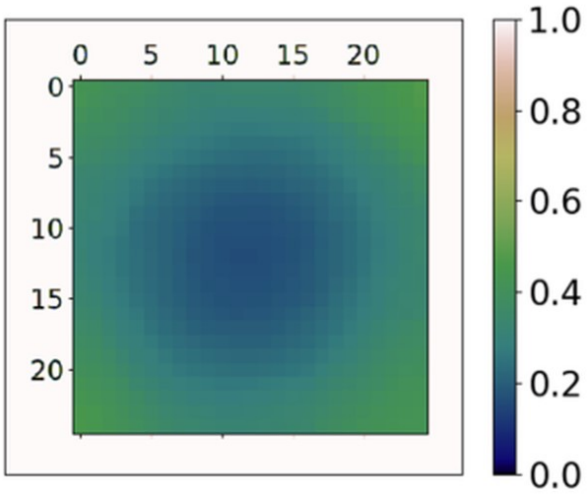


Figure 3.9 Averaged Activations

Note: Averaged activation maps in this report are subjects to be perfected over the shading due to a slight lacking in technical knowledge.

3.4.2 Effect of Number of Dense Units (Nodes) in Last Hidden Layer on Model Performance

The performance of the model is not very much affected by the dense units, unless it goes lower than 2. From the activations (*Figure 3.10 Activation over Different Dense Units*) below, it appeared that the model still mainly learned and estimated the positions via corners and borders. The results were also having low error rates and without specific angles clustering when we checked the Euclidean error in a cartesian to polar plot and histogram of degrees (not shown in figure). It was also able to predict the output well (not shown in figure). However, once the lower limit was exceeded, the model appeared to be no longer capable of generalizing, nor predict the output accurately with high errors (*Figure 3.10(a)-middle*) and had a tendency to cluster around north and south (*Figure 3.10(a)-right*) as shown by the single dense unit. We thus deduced that the model needs at least 2 nodes to work out an output. In contrary, despite having different numbers of units, model performance seemed to have the same trends over loss and accuracy (not shown in figure).

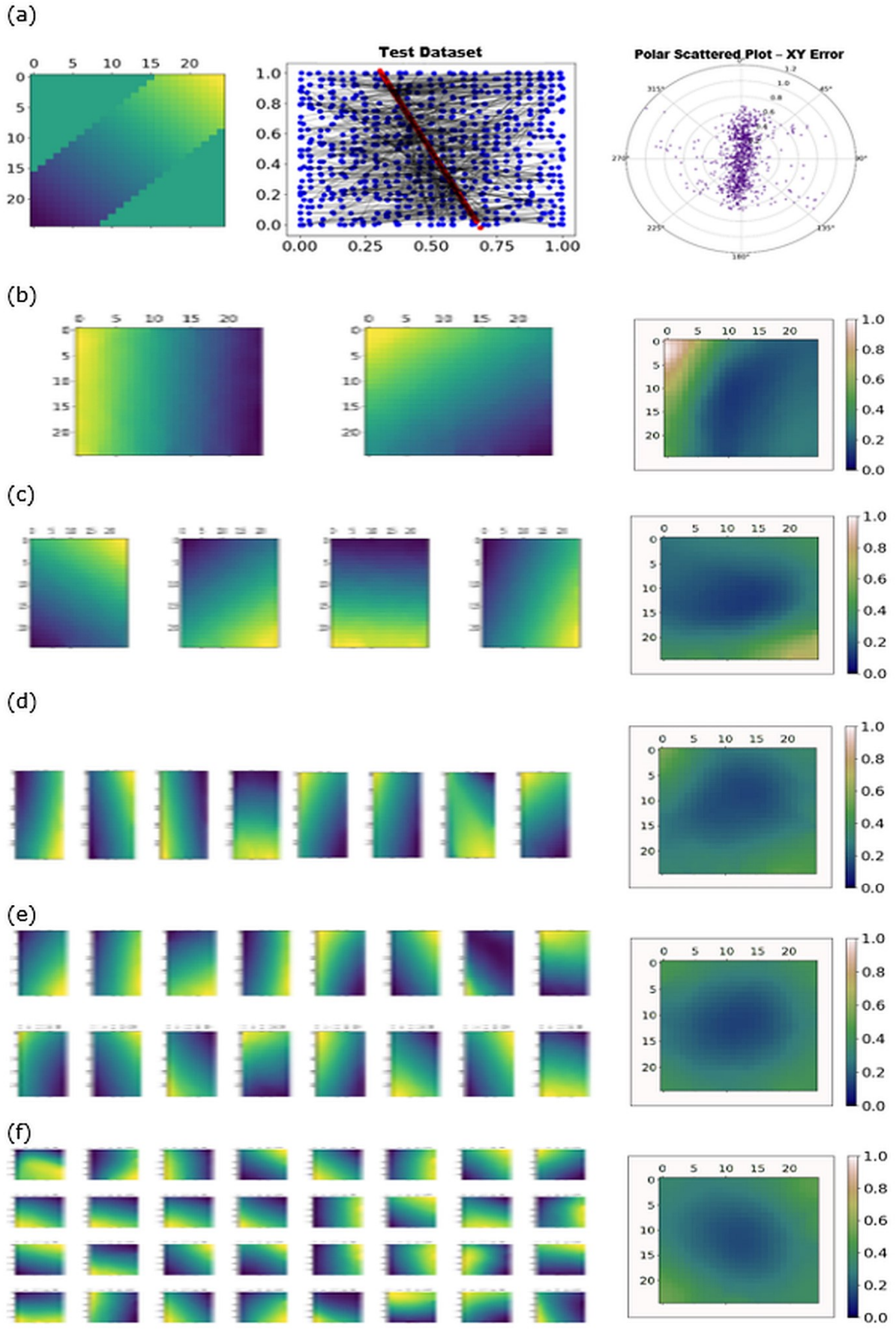


Figure 3.10 Activation over Different Dense Units

(a) 1 Dense Unit. (b) 2 Dense Units. (c) 4 Dense Units. (d) 8 Dense Units. (e) 16 Dense Units. (f) 32 Dense Units.

Chapter 4 : Discussion

4.1 Overview

We wanted to investigate whether a deep neural network can learn the spatial layout of a virtual environment, Donderstown. We consequently decided to build our neural network based on the architecture of a convolutional neural network (CNN) with our inputs being 2D snapshots of Donderstown. Subsequently, we found out that our deep neural network was able to learn the spatial layout of Donderstown mainly via corner and border-like units (refer back to *Figure 3.8 Border-like and Corner-like Units*).

4.1.1 Implications

Our deep neural network was able to learn the virtual environment by analyzing the features of the inputs and storing them as different weights in the nodes, in combination with learning labels from the training dataset. The neural network eventually learned to compute the output for the testing dataset. We found that our deep neural network needs at least two nodes to judge locations in the virtual environment. We deduced that such efficiency is due to the absence of an energy constraint in the neural network's nodes, in contrast with biological neurons. It is therefore unclear if exposure to just half of the training data would be enough for a human to judge and predict a full topographical map.

Referring back to *Table 1.1 Summary of Spatial Cells and Features*, there are four main aspects about empirically observed border cells. First, they potentially work with head-direction and grid cells to plan trajectories, thereby anchoring grid fields and place fields to a geometric reference frame (29). Second, border and grid cells associations are suggested to minimize the accumulated grid cells' error (35). Third, border cells sparsely exist with just less than 10% of the local cell population (29). Lastly, border cells encode one's position in relation to the borders of the egocentric environment, and fire when one is near the edge of the local environment (29, 32), where they fire at specific distances from specifically oriented environmental boundaries (33). This firing provides connecting information between egocentric and allocentric boundaries (34), i.e. the vectors between self-to-object and object-to-object. Along with our findings, we also wonder if the model implicates border cells as one of the most important and efficient spatial cells, which play a key role in modulating and integrating the essential information to other spatial and directional cells. Moreover, it was argued that geometrical information of distant objects may be assessed using the nearer small object and further away larger object (110). This may explain why edges such as borders become an important aspect to judge the surrounding spatial environment. For example, if one failed to judge an edge, one may fail to differentiate between different objects as well.

Our findings also included the involvement of corner-like units for all varieties of the successful models we discussed in the previous section. While it is unclear if corner-like units co-exist with border cells in biological settings, such as inside the hippocampal-entorhinal circuit, we suspect that they can play an equally important role as the border

units. For example, adjoining junctions showed higher activation in all of the averaged activation maps in our model training, this could suggest earlier detection of the corner before transforming to the adjacent edge via a mechanism similar to the continuous attractor networks. This indirectly evokes the prediction that corner detection might be a feature preceding edge detection. In addition, the corner units might have partially selective and long-lasting activation that further activated the adjoining border following the perception of a vector relating the self to the external environment. Furthermore, these corner units could be a type of highly efficient, sparsely distributed spatial cell that is yet to be detected in animals due to inadequate technology or knowledge.

Lastly, we deduced that accurate judgement of borders may be one of the keys to success in autonomous navigating agents, such as to prevent slipping and to match objects. For instance, it could help create robots that can climb the staircase independently with minimal to no accident, or predict an image location based on the extracted features that matched other learnt images in our model.

4.1.2 Advantages and Disadvantages of Convolutional Neural Networks (CNNs) Relative to Biological Systems

CNNs are recognized as one of the best models in the neural representation of visual images (111), especially when helping extract complex features from electron microscope (EM) image analysis (112). Relevant details about CNNs have previously been introduced in section 2.3.1. With the utilization of neural networks, one can know empirically difficult to control information such as the connectivity and activation of each node/layers, while also allowing one to perturb the units more easily and precisely (113).

There are still challenges for simulated neural networks to fully reflect biological ones due to higher levels of complexity (113). One to one correspondence between artificial and biological neural networks may not be technically achievable for a long time. Likewise, there is always a trade-off between complexity and interpretability in modelling (114), which can be inappropriate when using a CNN to fully describe a biological system. For instance, CNNs have no different units to process black and white colours differently during visual perception (115), besides possessing different mechanistic levels. Constraints and demands in energy of both artificial and biological networks are unique (e.g. artificial neurons can perform backpropagation, while action potentials in biological neurons are propagated unidirectionally along the axons) (113). Subsequently, biological networks are highly efficient in power and memory, but CNNs are limited by the hardware capacity, resource availability, and computational expense for large-scale analysis (113). Nonetheless, while we are yet to understand everything about the brain, we do not really understand the machinery computation processes either (113) but are likely just observing the resulting output.

4.1.3 Disadvantages of Adam Optimizer

An optimizer works to minimize the loss function using a set of algorithms that specify some particular ways in ruling the update of weight parameters (116). Due to time limitations, this project has mainly been using Adam as its optimizer. Despite positive results from Adam as mentioned in section 2.4.1.2, Adam has some drawbacks too. These included that: it was unable to converge to an optimal solution in some areas, and may have bad generalizing ability when the learning became saturated (117). Thus, in the future, those who use this ready-built model can try with other optimizers to validate the claims about Adam.

4.1.4 Others

It has been suggested that the mean and standard deviation values for local centering and data standardization (normalization) shall only be taken from the training dataset for application in the model training (92) to avoid information leakage from the testing dataset to the training dataset. However, we have tried that before, and deduced that it would be better to take the mean and standard deviation of the whole dataset due to data complexity of our dataset. To be specific, the colours intensities from different angles may be diverse largely among our dataset. We train and test alternate angles, instead of simple splitting of the data that have more or less the same nature. Therefore, it is justified to compute the values from the whole dataset to have unbiased and more representative values, so that different features can be scaled onto the same range and efficiently learnt by the model (*Figure 4.1 Comparison of Image Pre-Processing*).

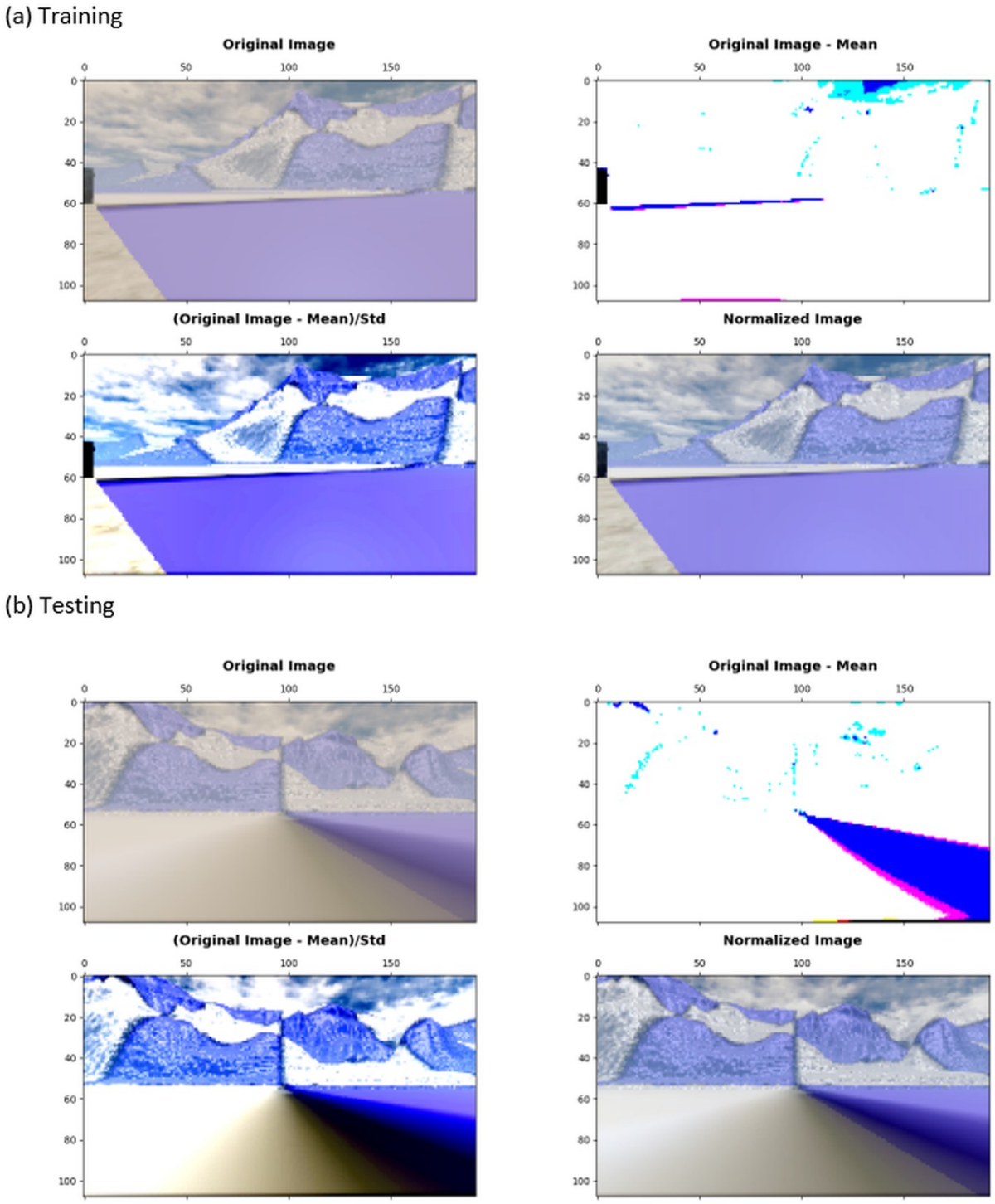


Figure 4.1 Comparison of Image Pre-Processing

Using a similar position from training(a) and testing (b) datasets that possessed different angles, it showed that our normalization method allowed good standardization of the images

4.2 Future Directions

There are many aspects which can be expanded in this project. First, we could manipulate the input selection where we train only 180° North/West and test on 180° South/East. We also could similarly distribute the map into grids or diagonals, but only train on an alternate portion and see how the representation changes. Second, we may also freeze some of the spatial inputs by deleting ('lesioning') them directly in the inputs or altering the weights of nodes. That can help us see if the model's performance becomes degraded by having both spatial and non-spatial (empty activation maps) units. Thirdly, we can also do data augmentation including translation and transformations such as shift, zoom, rotation, flip, random crop, colour shift, noise addition contrast change to the training dataset. Subsequently, we can use these features as an extra training dataset to examine if it helps the model to learn invariant features with better depth. Next, we can include angle as an output of the model, where we can evaluate if the model can handle the data to make higher complexity estimations. Lastly, we may slightly alter the architecture of the current model to train and decode on angle differences between separate snapshots. Beyond position and directional orientation, we might be able to optimize the model in estimating more realistic representations by replacing outputs XY with place cell activations and angles with head-direction activation, which might lead the model to discover more biologically plausible solutions.

Additionally, since we can refer to the representation as a prediction-based experience memory map, we could expect to see place cells if we modify the algorithms used in the model. Lastly, considering that we managed to discover spatial units in a deep neural network that learns the representation of a virtual environment, it is unclear if we may find the phenomena of spatial cells' activations similar to path integration when one is playing a non-landmark representation mobile game such as Flow Free (i.e. connecting dots game, *Figure 4.2 Flow Free*). To simulate this situation, perhaps one could modify the model to complete the dot patterns autonomously and have an insight into the model's maze-challenge-solving to determine if spatial units could efficiently facilitate this process.

Figure 4.2 Flow Free

4.3 Conclusions

We conclude that deep neural networks, specifically CNNs, can learn the representation of a virtual environment. Similar to biological organisms, our CNN model was able to learn the spatial representation of the environment using spatially modulated neuron-like activations, i.e. border units and novel 'corner' units. In particular, our model emphasized the importance of borders and corners when judging a spatial environment, which hints at the underlying importance of these spatial representations for both navigating artificial agents and biological systems.

References

1. Gould JL. Animal Navigation: A Map for All Seasons. Current Biology. 2014;24(4):153-5.
2. Collett TS, Graham P. Animal Navigation: Path Integration, Visual Landmarks and Cognitive Maps. Current Biology. 2004;14(12):475-7.
3. McNaughton BL, Battaglia FP, Jensen O, Moser EI, Moser M-B. Path integration and the neural basis of the 'cognitive map'. Nature Reviews Neuroscience. 2006;7:663-78.
4. Lei X, Zhang Z, Dong P. Dynamic Path Planning of Unknown Environment Based on Deep Reinforcement Learning. Journal of Robotics. 2018.
5. Saha O, Dasgupta R, Woosley B. Real-Time Robot Path Planning from Simple to Complex Obstacle Patterns Via Transfer Learning of Options. Autonomous Robots. 2019;43(5).
6. Jia Z, Saxena A, Chen T, editors. Robotic Object Detection: Learning to Improve the Classifiers Using Sparse Graphs for Path Planning. Proceedings of the 22nd International Joint Conference on Artificial Intelligence; 2011.
7. Fujiyoshi H, Hirakawa T, Yamashita T. Deep learning-based image recognition for autonomous driving. IATSS Research. 2019;43(4):244-52.
8. Darwin C. Origin of Certain Instincts. Nature. 1873;7:417-8.
9. O'Keefe J, Nadel L. The Hippocampus as a Cognitive Map: Oxford University Press; 1978.
10. Graham E. What Is a Mental Map? The Royal Geographical Society (with the Institute of British Geographers). 1976;8(4):259-62.
11. Eichenbaum H. The Hippocampus as a Cognitive Map ... of Social Space. Neuron. 2015;87(1):9-11.
12. Tolman EC. Cognitive maps in rats and men. Psychological review. 1948;55(4):189-208.
13. Eichenbaum H, Dudchenko P, Wood E, Shapiro M, Tanila H. The Hippocampus, Memory, and Place Cells: Is It Spatial Memory or a Memory Space? Neuron. 1999;23(2):209-26.
14. Moser MB, Rowland D, Moser E. Place Cells, Grid Cells, and Memory. Cold Spring Harbor Perspectives in Biology. 2015.
15. W.B S, B M. Loss of recent memory after bilateral hippocampal lesions. Journal of Neurology, Neurosurgery & Psychiatry. 1957;20:11-21.
16. O'Keefe J, Dostrovsky JO. The hippocampus as a spatial map. Preliminary evidence from unit activity in the freely-moving rat. Brain Research. 1971;34(1):171-5.
17. Moser E, Kropff E, Moser MB. Place Cells, Grid Cells, and the Brain's Spatial Representation System. Annual Reviews of Neuroscience. 2008:69-89.
18. Park E, Dvorak D, Fenton AA. Ensemble Place Codes in Hippocampus: CA1, CA3, and Dentate Gyrus Place Cells Have Multiple Place Fields in Large Environments. PLoS One. 2011;6(7).
19. Preston-Ferrer P, Coletta S, Frey M, Burgalossi A. Anatomical organization of presubicular head-direction circuits. eLife. 2016;5.
20. Peyrache A, Schieferstein N, Buzsáki G. Transformation of the head-direction signal into a spatial code. Nature Comminications. 2017;8.
21. Tukker JJ, Tang Q, Burgalossi A, Brecht M. Head-Directional Tuning and Theta Modulation of Anatomically Identified Neurons in the Presubiculum. The Journal of Neuroscience : the Official Journal of the Society for Neuroscience. 2015;35(46):15391-5.
22. Simonnet J, Nassar M, Stella F, Cohen I, Mathon B, Boccara CN, et al. Activity dependent feedback inhibition may maintain head direction signals in mouse presubiculum. Nature Comminications. 2017;8.
23. Whitlock JR, Derdikman D. Head direction maps remain stable despite grid map fragmentation. Frontiers in Neural Circuits. 2012.
24. Hawkins J, Lewis M, Klukas M, Purdy S, Ahmad S. A Framework for Intelligence and Cortical Function Based on Grid Cells in the Neocortex. Frontiers in Neural Circuits. 2019.

25. Moser EI, Roudi Y, Witter MP, Kentros C, Bonhoeffer T, Moser M-B. Grid cells and cortical representation. Nature Reviews Neuroscience. 2014;15:466-81.
26. Rowland DC, Roudi Y, Moser M-B, Moser E. Ten Years of Grid Cells. Annual Review of Neuroscience. 2016;39(1).
27. Kubie JL, Fenton AA. Linear Look-Ahead in Conjunctive Cells: An Entorhinal Mechanism for Vector-Based Navigation. Frontiers in Neural Circuits. 2012;6(20).
28. Tang Q, Burgalossi A, Ebbesen CL, Ray S, Naumann R, Schmidt H, et al. Pyramidal and Stellate Cell Specificity of Grid and Border Representations in Layer 2 of Medial Entorhinal Cortex. Neuron. 2014;84(6):1191-7.
29. Solstad T, Boccara CN, Kropff E, Moser M-B, Moser EI. Representation of Geometric Borders in the Entorhinal Cortex. Science. 2008;322(5909):1865-8.
30. Zhang S, Schönfeld F, Wiskott L, Manahan-Vaughan D. Spatial representations of place cells in darkness are supported by path integration and border information. Frontiers in Behavioral Neuroscience. 2014.
31. Deshmukh SS, Knierim JJ. Influence of local objects on hippocampal representations: landmark vectors and memory. Hippocampus. 2013;23(4).
32. Moser M-B, Moser EI. Crystals of the brain. EMBO Molecular Medicine. 2011;3:69-71.
33. Høydal ØA, Skytøen ER, Andersson SO, Moser M-B, Moser EI. Object-vector coding in the medial entorhinal cortex. Nature. 2019;568:400-4.
34. Bicanski A, Burgess N. A neural-level model of spatial memory and imagery. eLife. 2018;7.
35. Santos-Pata D, Zucca R, Low SC, Verschure PFMJ. Size Matters: How Scaling Affects the Interaction between Grid and Border Cells. Frontiers in Computational Neuroscience. 2017.
36. Tsao A, Moser M-B, Moser EI. Traces of Experience in the Lateral Entorhinal Cortex. Current Biology. 2013.
37. Deshmukh SS, Knierim JJ. Representation of non-spatial and spatial information in the lateral entorhinal cortex. Frontiers in Behavioral Neuroscience. 2011.
38. Weible AP, Rowland DC, Monaghan CK, Wolfgang NT, Kentros CG. Neural Correlates of Long-Term Object Memory in the Mouse Anterior Cingulate Cortex. Journal of Neuroscience. 2012;32(16):5598-608.
39. Muller RU, Kubie JL. The Effects of Changes in the Environment on the Spatial Firing of Hippocampal Complex-Spike Cells The Journal of Neuroscience,. 1987;7(7):1951-88
40. B R, Y L, PP L-S, B P, RU M. Representation of objects in space by two classes of hippocampal pyramidal cells. The Hournal of General Physiology. 2004;124(1):9-25.
41. Geiller T, Fattahi M, Choi J-S, Royer S. Place cells are more strongly tied to landmarks in deep than in superficial CA1. Nature Communications. 2017;8.
42. Czurkó A, Hirase H, Csicsvari J, Buzsáki G. Sustained activation of hippocampal pyramidal cells by 'space clamping' in a running wheel. European Journal of Neuroscience. 1999:344-52.
43. Kropff E, Carmichael JE, Moser M-B, Moser EI. Speed cells in the medial entorhinal cortex. Nature. 2015;523:419-24.
44. Behrens TEJ, Muller TH, Whittington JCR, Mark S, Baram AB, Stachenfeld KL, et al. What Is a Cognitive Map? Organizing Knowledge for Flexible Behavior. Neuron. 2018;100(2):490-509.
45. Doeller CF, Barry C, Burgess N. Evidence for grid cells in a human memory network. Nature. 2010;463:657-63.
46. Horner A, Bisby J, Zotow E, Bush D, Burgess N. Grid-like Processing of Imagined Navigation. Current Biology. 2016:842-7.
47. Bellmund JL, Deuker L, Schröder TN, Doeller CF. Grid-cell representations in mental simulation. eLife. 2016.
48. Deuker L, Bellmund JL, Schröder TN, Doeller CF. An event map of memory space in the hippocampus. eLife. 2016.
49. Cueva CJ, Wei X-X. Emergence of grid-like representations by training recurrent neural networks to perform spatial localization. ICLR 20182018.

50. Banino A, Barry C, Uria B, Blundell C, Lillicrap T, Mirowski P, et al. Vector-based navigation using grid-like representations in artificial agents. Nature. 2018;557:429–33.
51. Copeland M. What's the Difference Between Artificial Intelligence, Machine Learning and Deep Learning? : Nvidia; 2016 [Available from: https://blogs.nvidia.com/blog/2016/07/29/whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/.
52. Fei DS. All You Need to Know About Neural Networks – Part 1: Alibaba Cloud; 2018 [Available from: https://www.alibabacloud.com/blog/all-you-need-to-know-about-neural-networks-%E2%80%93-part-1_593835?spm=a2c65.11461447.0.0.755b72d0bvDhZI.
53. Hardesty L. Explained: Neural networks

Ballyhooed artificial-intelligence technique known as "deep learning" revives 70-year-old idea: MIT News Office; 2017 [Available from: http://news.mit.edu/2017/explained-neural-networks-deep-learning-0414.
54. Strachnyi K. Brief History of Neural Networks 2019 [Available from: https://medium.com/analytics-vidhya/brief-history-of-neural-networks-44c2bf72eec.
55. Wills TJ, Lever C, Cacucci F, Burgess N, O'Keefe J. Attractor Dynamics in the Hippocampal Representation of the Local Environment. Science. 2005;308(5723):873-6.
56. Hopfield JJ. Neural networks and physical systems with emergent collective computational abilities. Proc NatL Acad Sci. 1982:2554-8.
57. Yan M, Michael KYW. The dynamis of two-layer continuous attractor neural network model with moving stimulus. 2015 International Symposium on Nonlinear Theory and its Applications; Kowloon, Hong Kong2015. p. 393-6.
58. Fei DS. All You Need to Know About Neural Networks – Part 2: Alibaba Cloud; 2018 [Available from: https://www.alibabacloud.com/blog/all-you-need-to-know-about-neural-networks-part-2_593836?spm=a2c65.11461447.0.0.27b35f45XL3d4H.
59. History Of Virtual Reality: Virtual Reality Society; [Available from: https://www.vrs.org.uk/virtual-reality/history.html.
60. Poetker B. The Very Real History of Virtual Reality (+A Look Ahead): G2 Learning Hub; 2019 [Available from: https://learn.g2.com/history-of-virtual-reality.
61. Poetker B. A Brief History of Augmented Reality (+Future Trends & Impact) 2019 [Available from: https://learn.g2.com/history-of-augmented-reality.
62. Poetker B. What Is Mixed Reality? (+How It Differs From Augmented Reality) 2019 [Available from: https://learn.g2.com/mixed-reality.
63. Martin S. What Is Simultaneous Localization and Mapping? SLAM is a commonly used method to help robots map areas and find their way: Nvidia; 2019 [Available from: https://blogs.nvidia.com/blog/2019/07/25/what-is-simultaneous-localization-and-mapping-nvidia-jetson-isaac-sdk/.
64. Castellanos JeA, Neira Je, ́os JDT, editors. Limits to the Consistency of EKF-based SLAM. 5th IFAC/EURON Symposium on Intelligent Autonomous Vehicles; 2004; Lisboa, Portugal.
65. Mahrami M, Islam MN, Karimi R. Simultaneous Localization and Mapping: Issues and Approaches. International Journal of Computer Science and Telecommunications. 2013;4(7).
66. Siegwart R, Nourbakhsh IR. Introduction to Autonomous Mobile Robots: The MIT Press; 2004.
67. Luft L, Burgard W. SLAM - Simultaneous Localization and Mapping Albert-Ludwigs-Universität Freiburg2019 [Lecture Notes from http://ais.informatik.uni-freiburg.de/teaching/ss19/robotics/]. Available from: http://ais.informatik.uni-freiburg.de/teaching/ss19/robotics/slides/13-slam.pdf.
68. Thrun S, Burgard W, Fox D. Probabilistic Robotics1999.
69. RatSLAM: Using Models of Rodent Hippocampus for Robot Navigation [Internet]. Queensland University of Technology. 2012. Available from: https://www.youtube.com/watch?v=t2w6kYzTbr8.

70. Milford MJ, Wyeth GF, editors. RatSLAM: A Hippocampal Model for Simultaneous Localization and Mapping. Proceedings of the 2004 IEEE: International Conference on Robotics and Automallon 2004; New Orleans, LA: IEEE.
71. Milford MJ, Wyeth GF, editors. Hippocampal Models for Simultaneous Localisation and Mapping on an Autonomous Robot Proceedings of the 2003 Australasian Conference on Robotics and Automation; 2003; Australia: Australian Robotics and Automation Association.
72. McNaughton BL, Battaglia FP, Jensen O, Moser EI, Moser M-B. Path integration and the neural basis of the 'cognitive map'. Nature Reviews Neuroscience. 2006;7:663-78.
73. Milford M, Wyeth G, Prasser D. RatSLAM on the Edge: Revealing a Coherent Representation from an Overloaded Rat Brain. 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems; Beijing, China: IEEE; 2006.
74. Milford MJ, Wyeth GF. Mapping a Suburb With a Single Camera Using a Biologically Inspired SLAM System. IEEE Transactions on Robotics. 2008;24(5):1038-53.
75. Milford MJ, Wyeth GF. Persistent Navigation and Mapping using a Biologically Inspired SLAM System. The International Journal of Robotics Research. 2010;29(9):1131–53.
76. Ball D, Heath S, Wiles J, Wyeth G, Corke P, Milford MJ. OpenRatSLAM: an open source brain-based SLAM system. Auton Robot. 2013;34:149-76.
77. Müller S, Weber C, Wermter S, editors. RatSLAM on Humanoids - A Bio-Inspired SLAM Model Adapted to a Humanoid Robot. Proceedings of the 24th International Conference on Artificial Neural Networks (ICANN 2014); 2014.
78. Kazmi SMAM, Mertsching B, editors. Gist+RatSLAM: An Incremental Bio-inspired PlaceRecognition Front-End for RatSLAM. BICT'15: Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS); 2016: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).
79. Struckmeier O, Tiwari K, Salman M, Pearson MJ, Kyrki V. ViTa-SLAM: A Bio-inspired Visuo-Tactile SLAM for Navigation while Interacting with Aliased Environments. IEEE Cyborg and Bionic Systems (CBS)2019.
80. Chaplot DS, Gandhi D, Gupta S, Gupta A, Salakhutdinov R. Learning To Explore Using Active Neural SLAM. ICLR-20202020.
81. Bellmund JLS, Deuker L, Doeller CF. Donderstown 2018 [Available from: https://osf.io/78uph/.
82. Amidi A, Amidi S. Recurrent Neural Networks cheatsheet: Stanford; 2018 [Available from: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks.
83. PSU. 14.1 - Autoregressive Models: Pennsylvania State University (Department of Statistics); [Available from: https://online.stat.psu.edu/stat501/lesson/14/14.1.
84. Dhillon A, Verma GK. Convolutional neural network: a review of models, methodologies and applications to object detection. Progress in Artificial Intelligence. 2019.
85. Amidi A, Amidi S. Convolutional Neural Networks cheatsheet: Stanford; 2018 [Available from: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks.
86. Eslami SMA, Rezende DJ, Besse F, Viola F, Morcos AS, Garnelo M, et al. Neural scene representation and rendering. Science. 2018;360:1204-10.
87. Rawat W, Wang Z. Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review. The MIT Press Journals. 2017;29(9):2352-449.
88. Gu J, Wang Z, Kuen J, Ma L, Shahroudy A, Shuai B, et al. Recent Advances in Convolutional Neural Networks. Elsevier. 2018;77:354-77.
89. Convolutional Neural Network (CNN): Nvidia Developer; [Available from: https://developer.nvidia.com/discover/convolutional-neural-network.
90. Li F-F, Krishna R, Xu D. CS231n: Convolutional Neural Networks for Visual Recognition (Spring 2020) 2020 [Available from: https://cs231n.github.io/convolutional-networks/.

91. Wang R. Convolutional Neural Networks (CNNs): Harvey Mudd College; 2019 [Available from: http://fourier.eng.hmc.edu/e176/lectures/ch10/node8.html.
92. Li F-F, Krishna R, Xu D. CS231n Convolutional Neural Networks for Visual Recognition 2020 [Available from: https://cs231n.github.io/neural-networks-2/.
93. Zhou V. Training a Convolutional Neural Network from scratch: A simple walkthrough of deriving backpropagation for CNNs and implementing it from scratch in Python 2019 [Available from: https://towardsdatascience.com/training-a-convolutional-neural-network-from-scratch-2235c2a25754.
94. Amidi A, Amidi S. Deep Learning Tips and Tricks cheatsheet: Stanford; 2018 [Available from: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks#.
95. Agarwal A. When to use yield instead of return in Python? : GeeksforGeeks; [Available from: https://www.geeksforgeeks.org/use-yield-keyword-instead-return-keyword-python/.
96. Kingma DP, Ba JL. ADAM: A Method For Stochastic Optimization. ICLR 20152015.
97. Patrikar S. Batch, Mini Batch & Stochastic Gradient Descent 2019 [Available from: https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a.
98. Brownlee J. Understand the Impact of Learning Rate on Neural Network Performance 2020 [Available from: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/.
99. Brownlee J. Gentle Introduction to the Adam Optimization Algorithm for Deep Learning 2019 [Available from: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/.
100. Rakhecha A. Understanding Learning Rate 2019 [Available from: https://towardsdatascience.com/https-medium-com-dashingaditya-rakhecha-understanding-learning-rate-dd5da26bb6de.
101. Brownlee J. How to Configure the Learning Rate When Training Deep Learning Neural Networks 2019 [Available from: https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/.
102. Brownlee J. Why Initialize a Neural Network with Random Weights? 2018 [Available from: https://machinelearningmastery.com/why-initialize-a-neural-network-with-random-weights/.
103. Bushaev V. Improving the way we work with learning rate 2017 [Available from: https://techburst.io/improving-the-way-we-work-with-learning-rate-5e99554f163b.
104. Al-Masri A. What Are Overfitting and Underfitting in Machine Learning? 2019 [Available from: https://towardsdatascience.com/what-are-overfitting-and-underfitting-in-machine-learning-a96b30864690.
105. Brownlee J. Overfitting and Underfitting With Machine Learning Algorithms. 2019.
106. Gonfalonieri A. Dealing with the Lack of Data in Machine Learning 2019 [Available from: https://medium.com/predict/dealing-with-the-lack-of-data-in-machine-learning-725f2abd2b92.
107. Koehrsen W. Overfitting vs. Underfitting: A Complete Example 2018 [Available from: https://towardsdatascience.com/overfitting-vs-underfitting-a-complete-example-d05dd7e19765.
108. Regression loss metrics: Peltarion; [Available from: https://peltarion.com/knowledge-center/documentation/evaluation-view/regression-loss-metrics.
109. Mwiti D. Keras Metrics: Everything You Need To Know: neptune.ai; 2020 [Available from: https://neptune.ai/blog/keras-metrics.
110. Purves D, Lotto RB, Nundy S. Why We See What We Do: A probabilistic strategy based on past experience explains the remarkable difference between what we see and physical reality. American Scientist. 2002;90(3):236-43.
111. Kuzovkin I, Vicente R, Petton M, Lachaux J-P, Baciu M, Kahane P, et al. Activations of deep convolutional neural networks are aligned with gamma band activity of human visual cortex. Communications Biology. 2018.

112. Zhang X, Tan G, Chen M. A Reliable Distributed Convolutional Neural Network for Biology Image Segmentation. 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing2015.
113. Barrett DG, Morcos AS, Macke JH. Analyzing biological and artificial neural networks: challenges with opportunities for synergy? Current Opinion in Neurobiology. 2019;55:55-64.
114. Lindsay GW. Convolutional Neural Networks as a Model of the Visual System: Past, Present, and Future. Journal of Cognitive Neuroscience. 2020:1-15.
115. Silva DB, Cruz PP, Gutierrez AM. Are the long–short term memory and convolution neural networks really based on biological systems? ICT Express. 2018;4(2):100-6.
116. Khandelwal R. Overview of different Optimizers for neural networks 2019 [Available from: https://medium.com/datadriveninvestor/overview-of-different-optimizers-for-neural-networks-e0ed119440c3.
117. Bushaev V. Adam — latest trends in deep learning optimization 2018 [Available from: https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c.

Appendices

This section will give additional information from the previous sections and provide further elaboration to novice computational neuroscientists like me for mutual learning.

Our CNNs Model Codes

This is a sample of our final model script. As additional angles output is still under testing, the script is not yet to be revealed to the public.

```
def create_cnn_functional_model(num_layers):
    input_layer = Input(shape=(108,192,3))

    ## Downsampling here (Flexible to alter)
    X = Conv2D(64, kernel_size=(7,7), activation='relu', strides = (1,1), padding='same')(input_layer)
    for l in range(0, num_layers):
        X = Conv2D(64*(l+1), kernel_size=(7,7), activation='relu', strides = (2,1), padding='same')(X)
        X = Dropout(0.25)(X)
        X = Conv2D(64*(l+1), kernel_size=(7,7), activation='relu', strides = (1,2), padding='same')(X)
        X = Dropout(0.25)(X)
        X = Conv2D(64*(l+1), kernel_size=(7,7), activation='relu', strides = (2,2), padding='same')(X)
        X = Dropout(0.25)(X)

    X = Flatten()(X)
    X = Dense(128, activation="linear")(X)

    output = Dense(2, activation="linear")(X)
    model = Model(inputs=input_layer, outputs=output)

    dt = time.strftime("%Y-%m-%d_%H.%M.%S")

    # summarize layers
    from contextlib import redirect_stdout
    file = dir14 + '/' + dt +'.txt'
    with open(file, 'w') as f:
        with redirect_stdout(f):
            model.summary()
    print(model.summary())


    # plot graph
    plot_model(model, to_file=(dir11 + 'convolutional_neural_network_' + dt + '.png'))

    model.compile(optimizer=Adam(lr=0.00001),loss='mse', metrics=['mse','acc'])


    return model
```

How the Inner Activations are Calculated

Adapted from our full codes, the following code shows exactly how we retrieve the inner activation of the dense units:

```
layer_outputs = model.layers[27].output
activation_model = Model(inputs=model.input,outputs=layer_outputs)
hidden_act = activation_model.predict(real_img)

def generate_ratemap_heatmap_from_activations(neuron_activation, positions, num_bins=25, smoothing=1, max_pos=None)
    if max_pos is None:
        max_pos = np.max(positions, axis=0)
    square_bins = [num_bins,num_bins]
    # 1.) Calculate occupancy and activty
    heatmap_occupancy, xedges, yedges = np.histogram2d(positions[:, 1], positions[:, 0], bins=square_bins,
                                                       normed=False)
    heatmap_activity, xedges, yedges = np.histogram2d(positions[:, 1], positions[:, 0], bins=(xedges, yedges),
                                                      normed=False, weights=neuron_activation)

    # 2.) Calculate rate map as AVERAGE
    ratemap = heatmap_activity / heatmap_occupancy

    # 2.1) Smooth ratemap TAKE GOOD CARE OF NANS
    V = ratemap.copy()
    V[ratemap!=ratemap] = 0
    VV = gaussian_filter(V, sigma=smoothing)

    W = 0 * ratemap.copy() + 1
    W[ratemap!=ratemap] = 0
    WW = gaussian_filter(W, sigma=smoothing)

    ratemap_wo_nan = VV / WW
    ratemap_wo_nan[ratemap_wo_nan!=ratemap_wo_nan] = 0

    ratemap_with_nan = ratemap_wo_nan.copy()
    ratemap_with_nan[ratemap!=ratemap] = np.NaN

    # 3.) Return with edges
    return (heatmap_activity, ratemap, ratemap_with_nan, ratemap_wo_nan, xedges, yedges)
```

```
def rateheatMap_ratelinear_overlay(mode,realOrPred,ratemap,heatmap,rateLinear):
    mpl.rcParams.update({'font.size': 20})

    if realOrPred == 'real':
        chosenPosition = real_xy
    elif realOrPred == 'pred':
        chosenPosition = pred_xy

    lenXY = list(range(0, chosenPosition.shape[0], 1)) # inputs: pred_xy or real_xy
    lenImage = list(range(0, hidden_act.shape[1], 1)) # outputs: dense layer (Neurons number)

    real_xy_X = []
    real_xy_Y = []

    for i in lenXY:
        x = chosenPosition[i,0]
        y = chosenPosition[i,1]
        real_xy_X.append(x)
        real_xy_Y.append(y)
    maxX = max(real_xy_X)
    maxY = max(real_xy_Y)

    positions = chosenPosition

    nt = time.strftime("%d-%m-%Y_%H%M")
    dt = time.strftime("%Y-%m-%d_%H.%M.%S")

    if ratemap == 'yes':
        folder_rate = dir13 + '/' + test_runNum + '-' + mode + '_' + realOrPred + '_xy_' + nt
        if not os.path.exists(folder_rate):
            os.makedirs(folder_rate)

    if heatmap == 'yes':
        folder_heat = dir15 + '/' + test_runNum + '-' + mode + '_' + realOrPred + '_xy_' + nt
        if not os.path.exists(folder_heat):
            os.makedirs(folder_heat)

    if rateLinear == 'yes':
        folder_rl = dir18 + '/' + test_runNum + '-' + mode + '_' + realOrPred + '_xy_' + nt
        if not os.path.exists(folder_rl):


    for a in lenImage:
        neuron_activation = hidden_act[:,a]
        (hm,rm, rmwn, rmwon, xedge, yedge) = generate_ratemap_heatmap_from_activations(neuron_activation,
                                                                                       positions, num_bins=25,
                                                                                       smoothing=1,
                                                                                       max_pos=(maxX,maxY))

        if ratemap == 'yes':
            plt.matshow(rmwon)
            plt.savefig(folder_rate + '/' + 'ratemap' + str(a) + '_' + dt + '.png')

        if heatmap == 'yes':
            plt.matshow(hm)
            plt.savefig(folder_heat + '/' + 'heatmap' + str(a) + '_' + dt + '.png')
```

b

```
        if rateLinear == 'yes':
            img_rm = rmwon

            mean0 = np.mean(img_rm, axis=(0))
            shape0 = mean0.shape

            mean1 = np.mean(img_rm, axis=(1))
            shape1 = mean1.shape

            allImage = []
            allImage.extend((img_rm,mean0,mean1))

            fig, axes = plt.subplots(1,3, figsize=(20, 8))
            axes = axes.flatten()
            fig.tight_layout(pad=6.0)

            plt.rcParams['xtick.bottom'] = plt.rcParams['xtick.labelbottom'] = True
            plt.rcParams['xtick.top'] = plt.rcParams['xtick.labeltop'] = False

            for idx, ax in enumerate(axes):

                for label in (ax.get_xticklabels() + ax.get_yticklabels()):
                    label.set_fontsize(20)

                if idx == 0:
                    dataname = ('Ratemap')
                    img = allImage[idx]
                    ax.set_title(dataname, pad=10, fontsize=25, fontweight='bold')
                    ax.tick_params(axis="x", bottom=True, top=False, labelbottom=False, labeltop=False,
                                   which='major', labelsize=5)
                    ax.set_xticks([0,0,5,10,15,20])
                    ax.set_xticklabels(['0','0','5','10','15','20'],fontsize=20)
                    ax.xaxis.set_visible(True)
                    ax.xaxis.set_ticks_position('bottom')
                    ax.xaxis.tick_bottom()
                    ax.matshow(img, aspect='auto')
                elif idx == 1:
                    dataname = ('X-axis (Index/Row)')
                    img = allImage[idx]
                    ax.set_title(dataname, pad=10, fontsize=25, fontweight='bold')
                    ax.yaxis.set_major_formatter(FormatStrFormatter('%.4f'))
                    ax.xaxis.set_ticks_position('top')
                    ax.set_xticks([0,0,5,10,15,20])
                    ax.set_xticklabels(['0','0','5','10','15','20'],fontsize=20)
                    ax.plot(img)
                elif idx == 2:
                    dataname = ('\n\n\n\n\n\n\n\n Y-axis (Column)')
                    img = allImage[idx]
                    ax.set_title(dataname, pad=10, fontsize=25, fontweight='bold')
                    ax.yaxis.set_major_formatter(FormatStrFormatter('%.4f'))
                    ax.xaxis.set_ticks_position('top')
                    ax.set_xticks([0,0,5,10,15,20])
                    ax.set_xticklabels(['0','0','5','10','15','20'],fontsize=20)
                    ax.plot(img)

            fig.savefig(folder_rl + '/' + 'ratelinear' + str(a) + '_' + dt + '.png')


rateheatMap_ratelinear_overlay(mode='training',realOrPred='real',ratemap='yes',heatmap='yes',rateLinear='yes')
```

c

Simple Explanation of Parameters

This part briefly explain the general concept of parameters in between layers for understanding.

Item	Functionality
Convolution Layers	
Input	Input refers to the input data we feed into the model which is the Donderstown snapshot (1)
Conv2D	2D convolution layer. A mathematical process that combines 2 signals to form a 3rd signal as a feature map (filter) (1)
Filters	A concatenation of multiple kernels (2). Kernel is the weight data. Each filter extracts one feature
kernel_size	Kernel formed from the activation weights (weighted sum of all pixel values) of each neuron, the kernel_size here is in 2-dimensions
activation	An additional step over the layer to do non-linear transformation to the neuron's inputs (3). For example, we tried out with relu, elu, tanh, swish, mish, softmax. Please refer to the next section (*Simple Explanation of Activation Function*) for coverage of mechanism used in each activation.
strides	The way how the convolution works (1), such as steps proceeding (number of pixels shift) from left to right and top to bottom over the input matrix. Bigger step (>1,1) is a way of down-sampling
padding	Added outer boundaries to the input layer to avoid the loss of features post-convolution (1). The 'same' used in our model is make enough padding (empty pixels) boundaries to standardize the shape between the output and input
Dropout	Dropout is a regularization technique to reduce overfitting and improve generalization (4)
Fully-Connected Layer (Flatten)	
Flatten	The layer that connect all nodes from previous layer (1)
Fully-Connected Layer (Classification)	
Dense	Matrix vector multiplication used to change the dimensions of vector
activation	Similar to what mentioned above, this is an additional mathematical step to do data transformation. For example, we tried out with relu, linear and sigmoid. Please refer to the next section (*Simple Explanation of Activation Function*) for coverage of mechanism used in each activation.
kernel_regularizer	Input weight regularization, **W**x+b: Adds the regularising term to the training loss to penalize on actual corresponding kernel weights of the layer.
activity_regularizer	Activation Regularization, **y** = f(Wx + b): Adds the regularising term to the training loss over the output vector. Used to impose constraints on the model and reduce overfitting

bias_regularizer	Bias weight regularization, Wx+**b**: Adds the regularising term to the training loss to penalize on corresponding weights via bias vector of the weights
Dropout	(Similar to above)
Difference between L1 and L2 regularizers	**L1 regularization**: Lasso Regression (Least Absolute Shrinkage and Selection Operator), adds "absolute value of magnitude" of coefficient as penalty term to the loss function. Shrinks the less important feature's coefficient to zero (feature removal) (5) to reduce the model's complexity (6). Large lambda leads to model underfitting (5) **L2 regularization**: Ridge Regression, adds "squared magnitude" of coefficient as penalty term to the loss function. Works to reduce overfitting. However, if lambda is incredibly large then it will add too much weight and lead to model underfitting (5)
Fully-Connected Layer (Output)	
Dense	(Similar to above)
activation	(Similar to above). For example, linear is required for our case. Please refer to the next section (*Simple Explanation of Activation Function*) for coverage of mechanism used in the activation.

Simple Explanation of Activation Function

This section will briefly summarized the computation and general remarks of our tested activation functions.

Activation	Algorithm	Remarks
relu	f(x)=max(0,x)	Results become zero for negative input values (inactivated neuron) (3)
elu	f(x) = x, x>=0 = a(e^x-1), x<0	Values of x greater than 0 is 1, values of x<0, the derivative would be a.e^x (3)
linear	f(x)=ax	Linear regression (3)
tanh	tanh(x)=2sigmoid(2x)-1	Symmetric around the origin with the range of values between -1 to 1 (3)
swish	f(x) = x*sigmoid(x) f(x) = x/(1-e^-x)	Values range from negative infinity to infinity (3)
mish	f(x) = x.tanh(softplus(x))	Work better than both ReLU and Swish (7)
softmax	z = np.exp(x) z_ = z/z.sum()	Sum of all classes probability ended up equal to 1 (3)

sigmoid	f(x) = 1/(1+e^-x)	Sigmoid transforms the values between the range 0 and 1 (3)

Functions and Parameters to Train the Model

.fit_generator

We used .fit_generator to train our model. In contrast with .fit, .fit_generator is used in our model because .fit is more suitable to the model that can fit the whole training set into the RAM (8). Our data for the model is however too large for that. This is why we chose to utilize .fit_generator which makes use of the generator function to yield a dataset based on a specified batch size under timely manner for training, and then later performs backpropagation to update the model's weights (8). To recap, we learnt that the model weights are updated by the end of every epoch after the model learnt the whole training set (9) fed until the current moment. While the function trains the data batch-by-batch, the generator is running with CPU in parallel to the model training with GPU for time efficiency (10).

steps_per_epoch

While the generator is expected to loop over its data indefinitely, steps_per_epoch refers to total number of steps (batches of samples) to yield from the generator before an epoch finishes (10). While it was recommended to set the steps_per_epoch as total number of training dataset divided by the batch size (8), so that the model learns each sample at most once per epoch (11), we thought 1000 is a sensible value to get a quicker overview of the model.

callbacks

Callbacks are functions that are applied at every stage of the training such as the view on internal states and statistics. For instance, we used CSVLogger to save our model training result by the end of every epoch.

use_multiprocessing

Lastly, use_multiprocessing=True refers to process-based threading (10) that would separate the memory space and is able to take advantage of multiple CPUs and cores if present. In other words, it helps to process and load the generator dataset in parallel. Although multi-CPUs chips are still uncommon, modern Intel CPUs are now equipped with both multiple cores and hyper-threading technology (12). Hyper-threading (logical processors) makes a physical CPU core appears as two logical CPU cores to the operating system to speed up program execution (12). Conversely, multiple cores refers to multiple

processing units in a single physical CPU socket unit, which is capable of running different processes in parallel with less latency and quicker communication (12).

.get_layer and .predict

For post-model training, we also used .get_layer function and .predict to manipulate with our model output. Get_layer helps to retrieve a layer based on either its name (unique) or index while .predict generates output predictions for the input samples that were computed in batches (10).

Model Extended Tuning: Other Hyper-parameters Configurations

Based on previous testing, we found that the model was not affected by an additional dropout layer after the last hidden (dense) layer. In fact, there were no significant differences in the model performance and activation patterns of the activation map when compared to the chosen model that comes without dropout after the last hidden (dense) layer. We also found that the utilization of L2 regularizers in the last hidden (dense) layer can help drop the loss rate more quickly and smoothly, with only very slight overfitting, and get smoother looking activation maps representing the border-like and corner-like units. Similarly, there was no necessity to include dropout after the layer when applying L2 regularizers, as the model is capable of learning the XY position output correctly with or without it, and gave no differences in the training time as well.

Both L1 and L2 regularizers showed activations beyond border and corner-alike units in the activation map when non-linear activations such as relu were being used. However, we also observed that the use of L1 regularizers tends to affect the model ability to learn in a way that the model became almost impossible to predict the XY position. On the other hand, despite L2 regularizers improving the model, we also found that the high value of kernel_regularizer and/or activity_regularizer affected the model's ability to estimate the output, while bias_regularizer was not a sensitive factor in the model. In spite of these findings, we did not choose to use this extended model due to time constraint, and because there were a lot more tuning combinations that would be needed to try out for comparison and confirmation. To this end, our model still holds some level of inconclusive uncertainty regarding the utilization of L1 and L2 regularizers.

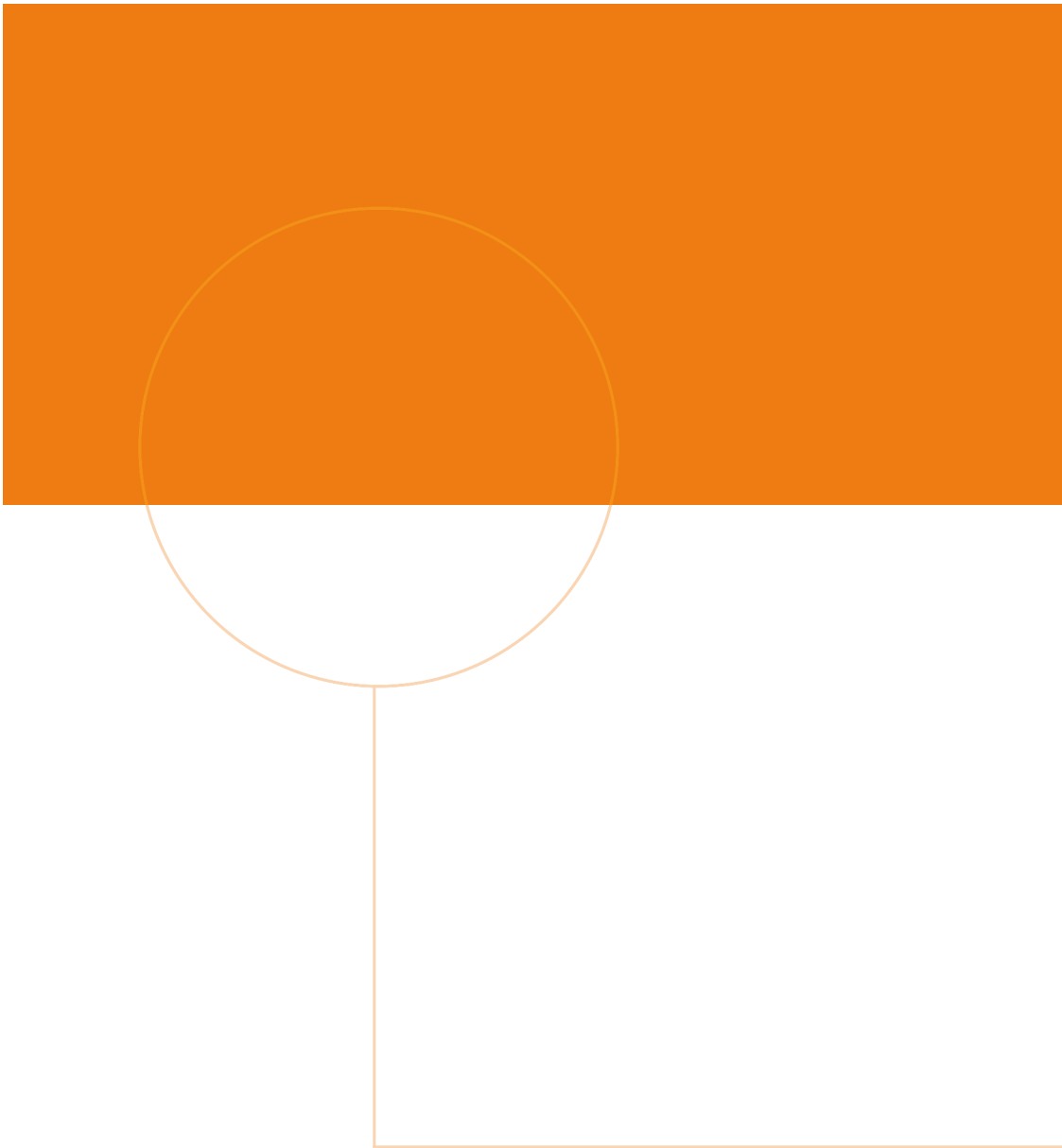
Hardware Requirements and Related Software

The original hardware in the workstation included 16 GB Random Access Memory (RAM) and Quadro K1200 graphics processing unit (GPU) with only 512 cores, 4GB GDDR5 memory and 80GB/s bandwidth. These are not enough to cater our needs for data processing and model training. Hence, we upgraded the RAM from 16 GB into 80 GB and purchased an additional GPU, NVIDIA GeForce GTX 1660 that comes with 1408 cores, 6GB GDDR5 memory and 192.1GB/s bandwidth. RAM is a primary storage which capable of quickly read and write data with the helps from Central Processing Unit (CPU, commonly known as processors) to transfer the data from and to the secondary storage, which is the hard disk (13). Meanwhile, GPU is mainly responsible to handle intensive graphics rendering tasks, taking over the heavy duty of rapid mathematical calculations from the CPU. Taking advantage of GPU's capability in breaking complex problems into simple tasks and work them out in parallel, we used it to train our model via utilization of CUDA, while RAM is mainly used for data pre-processing such as the calculation of mean and standard deviation for normalization. To further save the GPU resources for the model, we set the Quadro K1200 as default GPU for handling ordinary graphical tasks such as display projection, while GeForce GTX 1660 is only used for model running. System error is kept at minimal by installing most of the needed packages in a dedicated anaconda environment. The final version of master programs and drivers utilized in this project are summarized in hardware and software specifications table below.

Hardware and OS Specifications	
Workstation Model	Dell Precision Tower 5810
Processors	Intel® Xeon(R) CPU E5-1620 v3 @ 3.50GHz × 8
RAM	4 X 4GB + 4 X 16 GB
GPU	GeForce GTX 1660* , Quadro K1200
OS	Ubuntu 18.04.4 LTS (64-bit)
Driver: Nvidia	
Nvidia’s Driver	440.64
Application/Toolboxes: Jupyter	
jupyter-notebook	6.0.3
Application/Toolboxes: Keras	
keras	2.2.4
keras-applications	1.0.8
keras-base	2.2.4
keras-gpu	2.2.4
keras -preprocessing	1.1.0
Application/Toolboxes: CUDA & CuDNN	
cudatoolkit	10.0.130
cudnn	7.6.5
Application/Toolboxes: Tensorflow	
tensorflow	1.15.0
tensorflow-base	1.15.0
tensorflow-estimator	1.15.1
tensorflow-gpu	1.15.0
Application/Toolboxes: Conda	
conda	4.8.2
python	3.7.4.final.0

References

1. Katyal R. Convolutional Neural Networks: Why, what and How! 2019 [Available from: https://blog.usejournal.com/convolutional-neural-networks-why-what-and-how-f8f6dbebb2f9.
2. Ganesh P. Types of Convolution Kernels : Simplified 2019 [Available from: https://towardsdatascience.com/types-of-convolution-kernels-simplified-f040cb307c37.
3. Gupta DS. Fundamentals of Deep Learning – Activation Functions and When to Use Them? 2020 [Available from: https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/.
4. Brownlee J. Dropout Regularization in Deep Learning Models With Keras 2019 [Available from: https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/.
5. Nagpal A. L1 and L2 Regularization Methods 2017 [Available from: https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c.
6. Karim R. Intuitions on L1 and L2 Regularisation 2018 [Available from: https://towardsdatascience.com/intuitions-on-l1-and-l2-regularisation-235f2db4c261.
7. Misra D. Mish: A Self Regularized Non-Monotonic Neural Activation Function. arXiv. 2019.
8. Rosebrock A. How to use Keras fit and fit_generator (a hands-on tutorial) 2018 [Available from: https://www.pyimagesearch.com/2018/12/24/how-to-use-keras-fit-and-fit_generator-a-hands-on-tutorial/.
9. Amidi A, Amidi S. Deep Learning Tips and Tricks cheatsheet: Stanford; 2018 [Available from: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks#.
10. The Sequential model API: Keras Documentation; [Available from: https://keras.io/models/sequential/.
11. Amidi A, Amidi S. A detailed example of how to use data generators with Keras: Standford; 2018 [Available from: https://stanford.edu/~shervine/blog/keras-how-to-generate-data-on-the-fly#data-generator.
12. Hoffman C. CPU Basics: Multiple CPUs, Cores, and Hyper-Threading Explained 2018 [Available from: https://www.howtogeek.com/194756/cpu-basics-multiple-cpus-cores-and-hyper-threading-explained/.
13. Ferguson S, Hebels R. Computer systems and technology. In: Eyre G, Harris J, editors. Computers for Librarians (Third Edition): An Introduction to the Electronic Library. 3 ed. Charles Sturt University, New South Wales: National Library of Australia; 2003. p. 173.



NTNU

Norwegian University of
Science and Technology