**Machine learning pipeline to identify areas on a grid map**Author: Daniel Hixson  
Project supervisor: Steven Silva Mendoza  
Moderator: Carolina Fuentes Toro  
  
A Final Year Project Submitted for the Degree of Bachelor Of Science  
  
Department of Computer Science Cardiff University  
  
2024  
  
  
  
  
  
Table of Contents

[**Table of Contents 2**](#_u2k7o9ybgexd)

[**Acknowledgements 3**](#_vb9wu5ms36wy)

[**1. Introduction 4**](#_m6lwxpj4ep9x)

[**2. Background 5**](#_msczjgp1jell)

[2.1 Importance of Social Navigation 5](#_rylug984gx2p)

[2.2 Deep Learning for Social Density Prediction 6](#_mnyjikoa3g1u)

[2.2.1 Introduction to machine learning 6](#_f9psnzaw7l0n)

[2.2.2 Deep learning 7](#_75e9x51k85t8)

[2.2.2.1 CNN 8](#_48shhonruaw4)

[2.2.2.2 FCN 10](#_kk11conh9nz2)

[2.3 Robot Operating System 10](#_qdjfhpl6nxt)

[2.4 Project Justification 11](#_2zq2xej1w114)

[**3. Specification and design 11**](#_jugqhb3jled)

[3.1 Requirements 11](#_8957ka9jqmy3)

[3.2 Pipeline structure 13](#_oy2doqrdq06c)

[3.3 ROS Robot simulation 15](#_pwr2uy9kkcbq)

[3.2.1 Pepper robot 18](#_4mrvrt6j8t6z)

[3.2.2 Environment 18](#_s5ak9ola00kx)

[3.2.3 Social agents 21](#_mcw3xwl1ca8v)

[3.2.4 Navigation 23](#_5cfu3pedy59s)

[3.2.5 ROS Training program 25](#_ygjbic9513cm)

[3.3 Data collection 26](#_v0vhv6kurqg9)

[3.4 Data Preprocessing and Augmentation Techniques 28](#_t23mqi4b0jm8)

[3.4.1 Split data: Training and Ground truth 28](#_vtnbb98p10jp)

[3.4.2 Data augmentation techniques 28](#_mcbi1pbdo8et)

[3.4.2.1 Aggregation 29](#_ww6o6tcrd7up)

[3.4.2.2 Cropping 29](#_i9wrm7enaoj8)

[3.4.2.3 Padding 30](#_6puam4sdo4v5)

[3.4.2.4 Normalisation 30](#_i42zjney6zqz)

[3.4.2.5 Resizing 30](#_4n3d1my3d8si)

[3.4.2.6 Classification 30](#_tt5mbwp1xvz8)

[3.5 Model selection 31](#_fp3frg3pw5lu)

[3.5.1 FCN 31](#_h3v9fvoshkfd)

[3.5.2 CNN 32](#_pjtoo2scbgyc)

[3.6 Network Architecture Design 33](#_hhzreyo8qox)

[3.6.1 FCN 33](#_rro3hxtobrpu)

[3.6.2 CNN 34](#_rt0tduvrnyj3)

[3.7 Model training 34](#_ofya4kmgxrt0)

[3.7.1 Loss function 34](#_td603u63so)

[3.7.2 Optimiser 36](#_aj16dnw221yh)

[3.8 Model evaluation 36](#_h7nuqgv87lax)

[Model deployment 38](#_z3wwcqnkd38g)

[**4. Implementation 39**](#_tvm1awfgiwpj)

[4.1 Development Environments 39](#_zeu4b845eko)

[4.2 Code Implementation Details 39](#_rarwvrtv4jee)

[● 4.2.1 Training Route Simulation Code 39](#_yyq03dlclx2q)

[● 4.2.2 Data Collection Code 43](#_nappecyzfi6b)

[● 4.2.3 Data Splitting Code 46](#_vlqxm61xmsk3)

[● 4.2.4 Data Preprocessing Code 46](#_auu5kmrlsuis)

[● 4.2.5 Model Training Code 47](#_kfs6bxxrlup3)

[● 4.2.6 Evaluation Code 50](#_7xuqtr7mkbge)

[● 4.2.7 Model Deployment Code 50](#_6jod4ira1f6)

[4.3 Documentation 50](#_ym2osemapb7h)

[**5. Results and Evaluation 51**](#_p8js7f2gcrfl)

[**5.1 Network Performance Evaluation 51**](#_p8js7f2gcrfl)

[5.1.1 Unweighted FCN 52](#_i5tbg03o4sgf)

[5.1.2 Unweighted CNN 53](#_ohuxoqc16c7x)

[5.1.3 Class distribution 54](#_8w51adc0ynzf)

[5.1.4 Weighted FCN 54](#_4ngbvchg54wu)

[5.1.5 Weighted CNN 55](#_rmxgpcwivbiz)

[5.2 Network feasibility test 56](#_keeq6rdcexfy)

[**5.3 Discussion of Limitations and Future Work 56**](#_p8js7f2gcrfl)

[**6. Conclusion 57**](#_p8js7f2gcrfl)

[**7. Reflection on learning 57**](#_p8js7f2gcrfl)

[**8. References 59**](#_g7i8bjcq1x4r)

# 

# **Acknowledgements**

Firstly, I’d like to thank my supervisor Steven Silva Mendoza for his invaluable support and guidance throughout all aspects of the project from start to end. His expertise within the field of social navigation along with his kindness created a supportive environment that allowed me to confidently develop my solution for this project despite any shortcomings that arised.

Secondly I would like to express my gratitude towards my family for their support and encouragement throughout this challenging time of my life. Their belief in me and my abilities have been the shining light in my times of doubt.

Reflecting on this project, I understand that setbacks and challenges are part of life. It’s more important to maintain a positive outlook and focus on the progress made, than to live in the past. It will always be better in the end.

# **1. Introduction**

Recently it has become more common for robots to exist in areas where human activity can disrupt their operation. This is especially the case if the robot needs to navigate around that area, posing problems for both robot navigation and safety. Robots need to adapt their navigation to ensure it is safe for humans and vice versa. This can be done by having systems to assist the robot in identifying clear, safe and efficient paths by using data of its surroundings and obstacles like people.

Identifying which areas are most likely to have people is important data for the robot when navigating. Grid maps are data structures that represent a 2-dimensional area as an array, where each cell in the array denotes the probability of an obstacle being in that area. These grid maps can be used to represent social density, where each cell would represent the probability of a person being in that area. Being able to estimate the social density grip map of an area would allow the robot to calculate safer routes by choosing areas of lower social density.

This project employs the use of deep learning to estimate these grid maps, specifically fully convolutional networks (FCNs) and convolutional networks (CNNs). These networks are trained to analyse the occupancy grid maps made by the robot of its surroundings and then categorise areas of low, medium and high social density. The network breaks down the grid maps of the surroundings into their relevant features and learns patterns that correspond with the presence of agents (people).

Robot Operating System (ROS) is the program we use to be able to generate data to train the networks. This software allows us to efficiently simulate routes which the robot takes through environments, including agents that socially emulate people. We can change the routes that the robot takes as well as the areas in which it navigates.

The deep learning models will be evaluated using unseen data. Metrics like accuracy and mean squared error are used for each model to identify its performance and characteristics.

All aspects of the project will be contained as a ROS package and structured as a “pipeline”. This package will also undergo a network feasibility test.

This machine-learning pipeline could be used for other robot use cases beyond social navigation. For instance, it could be adapted for robots that interact with objects in an environment, allowing them to estimate object density and plan their movements. The model could even be adapted for other applications entirely like predicting metal plate failure due to concentrated heating from analysing thermal data. This would allow the pipeline to be repurposed for an industrial setting.

Overall this project aims to demonstrate the effectiveness of deep learning networks for social density estimation, ultimately contributing to improved robot navigation in social environments. The predicted data from the network can inform the robot, allowing it to adapt its navigation strategies based on the concentration of people around it.

# **2. Background**

## 2.1 Importance of Social Navigation

Examples of social areas where robots are being deployed are warehouses, factories and homes [1]. However, the presence of people creates challenges for the robots that need to navigate safely and efficiently like motion planning and evaluation methodologies[2]. Therefore methods have been created to assist in creating these paths that take into account their surroundings and any collisions with people.

Any damage to either the robot or the humans must be minimised. It should be a priority to ensure measures are in place to prevent either from being harmed. Physical damage caused to humans by the robot and vice versa can be costly and disrupt operations. Ensuring this optimises the efficiency of all in the working environment.

As surveyed by Mavrogiannis et al. [2], dynamic environments demand dynamic solutions. Pre-planned static routes do not have the adaptability needed to ensure a safe path for the robot. Unlike controlled settings, social environments contain people who can move and behave unpredictably. Crowded areas add another layer of complexity to the path planning of the robot. The higher the number of people, the more factors that need to be monitored to calculate clear routes for the robot to take. With high traffic, what can be considered a safe area can change in real time, further restricting the options for safe travel. The dynamic nature of these social spaces creates challenges for hazard-free travel.

To address these challenges in dynamic environments, social navigation equips robots with the ability to perceive and respond to social cues like the presence, behaviour and density of people. These methods provide a smoother interaction with people, reduced risk of collisions and more efficient path planning [3].

## 2.2 Deep Learning for Social Density Prediction

### 2.2.1 Introduction to machine learning



Machine learning is “a field of study in artificial intelligence concerned with the development and study of statistical algorithms that can learn from data and generalise unseen data, and thus perform tasks without explicit instructions” as defined by Gero et al. [5]. As shown in Figure 1, Machine learning is a subset of AI. It allows computers to learn from data without needing specific instructions. The algorithms that are used operate by building models that learn the patterns and relationships with the data that it trains on. Once trained, the models can take inputs and use their knowledge of the training data to predict new data.

Adopting machine learning in robotics provides many advantages when focusing on social navigation. Models that can perceive and understand patterns concerning social activity and its surrounding environment. Data can be gathered from sensors like LiDAR, cameras and ultrasonic sensors, providing an almost sensory perception of the area around it [6]. Combining all the data available, systems can be used to emulate ideas of human behaviour like personal space [7]. Personal space can be defined as the area around an agent that the robot should try to avoid whilst navigating. Using machine learning to learn these patterns, the personal space of people around the environment can be taken into account when planning paths.

### 2.2.2 Deep learning



However, social environments can be complex, with dense areas of people and unpredictable movement. Traditional methods struggle with the many dimensions of data that should all be considered in real time to choose the correct routes through these environments.

Deep learning is a subset of methods that exist in the field of machine learning in Fig 1. It employs the use of artificial neural networks to model data. Inspired by the structure of the neurons in brains, neural networks take the form of interconnected nodes that mimic the activity of neurons in the brain [9]. Each neuron (node) contains weights and biases that it uses to pass data on to other nodes in the network. Neural networks consist of three main layers: the input layer, the hidden layer and the output layer. The input layer takes in the raw data like images through several nodes to pass through the network. The hidden layer operates between the input and output layers. These layers can contain any number of layers and neurons stacked up against one another. All the nodes in one layer are connected to the one before and after. Adding different types of layers with different numbers of nodes allows the network to be optimised for different tasks. Data passing through the hidden layers is transformed, extracting features from it each time it passes through the layer. The weights and biases in these nodes create pathways for the model to learn complex, non-linear relationships in the input data. Other traditional machine learning techniques struggle with identifying the non-linear aspects of data. The output layer is the last layer of the network, this finalises the data for the output once passed through the entire model. Output data can range from calculating single values like indexes representing credit risk or complex output data like classifying images [10][11].

To train the model, the weights and biases of the nodes across the layers need to be adjusted. Training data is passed through the model (forward pass) and the output is compared to the desired output of the model. Using a method called backpropagation, the model changes the weights between each node in each layer so subsequent outputs of the model align to the desired output. The model effectively learns from its mistakes and improves predictions iteratively.

Being able to provide the robot with the ability to handle higher dimensional data like 2-dimensional grid maps and the ability to learn and improve is a strong tool for social navigation. Using conventions like grid maps, neural network architectures exist that can help identify the areas that contain low levels of activity.

#### 2.2.2.1 CNN



Convolutional neural networks (CNNs) are an architecture that specialises in analysing visual data like images and grid maps, extracting its key features. This is done through identifying the spatial relationships in the image. The input layer takes in data as images, containing the height, width and colour channels like grayscale or RGB. In the hidden layer, convolutional layers apply filters that pan across the pixels of the image, extracting features like lines and shapes. Each of the filters learns to identify these features in different parts of the image. Pooling layers exist to downsample the output of the convolutional layers to reduce the complexity of the model whilst keeping the key features. Non-linearity layers are applied to the data after convolution to ensure that the relationships that the model learns are not linear. For example, the intensity of a pixel is not a direct determination of whether it exists as an edge feature, so we employ the use of functions like ReLu that prevent the model from learning features like this.The convolution operation is the method for feature extraction. It involves a filter (kernel) that contains its weights and biases. The kernel is a square filter (3x3, 5x5 etc.) that slides across the image, calculating the element-wise multiplication between its weights and the pixels it is panning over. The result is then moved to what is called a feature map. The feature map represents extracted features across the entire image.

CNNs core task is image classification. By extracting features across the image, the model can classify objects. For example, a model can be trained to identify if it contains a cat or a dog, by outputting a value that constitutes its confidence with the classification. For social navigation, this provides an important way to detect features from the images generated by the robot. Object detection, motion detection and scene recognition can be employed to create more factors that the robot can consider when choosing a safer path [14][15][16]. However, for social robot navigation concerning identifying safe areas, classes need to be defined within the areas of the grid maps. The robot needs to know which cells contain low social activity so it can identify the safest path.

#### 2.2.2.2 FCN



CNNs excel at image classification tasks, assigning labels to the entire image. However, what if the use case required each pixel to be classified? Fully Convolutional Networks address this limitation [17]. Unlike CNNs, FCNs do not use any fully connected layers for classification. It employs a method called semantic segmentation to label each pixel across an image. The output of the model retains the spatial resolution of the image, allowing pixel-wise labelling.

For social robot navigation, 2-dimensional data like images and grid maps can be passed through the model to identify which cells are classified as low, medium and high social activity. When path planning, the robot has the context to choose safer paths and avoid crowded hazardous areas.

## 2.3 Robot Operating System

Robot Operating System (ROS) is an open-source software primarily used for robot software development. It provides many tools and libraries to assist with robot programming.

ROS nodes are the backbone of the program. This allows a modular approach to development where adding, removing and editing components is simple. These nodes communicate with each other using messages and service calls, creating networks for the robot's operation.

ROS provides abstraction between the hardware and software, simplifying the process of porting the code across different platforms. The ROS software also has visualisation capabilities like RViz, allowing developers to simulate the environments and robots.

ROS provides simulation and development capabilities for social navigation and will be the platform used to generate data for the machine-learning pipeline.

## 2.4 Project Justification

In the paper by Silva et al. [3], a framework that allows a robot to navigate around human “agents” is created in ROS. It employs path planning and world modelling with the implementation of a social heat map. The social heat map provides a grid map representing the personal space of agents in the shape of a circle. This social grid map is taken into account by the path planner so the robot can avoid agents. In Discussion and Conclusions, it’s mentioned that exploration with the path planner and “learning-based methods” could benefit the framework. This is what the report aims to achieve. The machine learning network aims to identify areas of areas of low social activity so that the sampling-based planner can only take into account these areas when processing paths, ignoring areas that are known for high social activity. The machine learning network should be able to be used as a ROS package where the package can subscribe to messages concerning grid maps, and output the predicted social density heat map.

# **3. Specification and design**

## 3.1 Requirements

Functional requirements

* Machine learning model for social density estimation
  + The system must accurately estimate social density in a simulated environment using grid maps.
  + The machine learning model should predict areas into low, medium, and high social density classes when data is inputted.
* Pipeline architecture
  + The system must be in the architecture of a pipeline which consists of modules and a clear flow of data
* Grid map data generation
  + The system should be able to generate grid maps (obstacle grid maps and social heat maps) for model training
* Data collection and preprocessing
  + The system must be able to collect grid map data from a ROS simulation
  + Data preprocessing should be implemented to ensure data is robust and uniform when training the model
* Model training and evaluation
  + The system should include the ability to train machine learning models using data collected from the system.
  + The system should contain tools for the evaluation of the trained model's performance
* Integration with ROS
  + The entire system must be integrated as a ROS package
  + The ROS package should be capable of deploying a network for model deployment

Non-Functional requirements

* Documentation
  + Documentation must be provided detailing how to run the training
  + Include tests presenting the feasibility of the network
* Usability
  + Setup, training and deployment of the system should be straightforward

Constraints

* Software: The system exists with the ROS framework and the PyTorch machine learning libraries.

## 3.2 Pipeline structure



In software engineering, a pipeline refers to a series of processing elements arranged in a linear fashion. Data from each layer passes to the layer in front of it. This naturally creates a modular format in which the nodes can be added, removed and adjusted on without necessarily affecting the entire system. Within these nodes, data is operated on by the program to be a suitable format for the next layer. With a machine learning pipeline, all aspects to train, deploy and evaluate a machine learning network should be included.

**Training route in ROS**

This program will use ROS to simulate a robot travelling from a starting position to end position in a crowded social environment. People in the environment are simulated as agents that move around the environment. These agents are either static, moving around on their own or moving in groups. The robot has the frameworks detailed in the study by Silva et al.[3]. This program can be looped to provide a constant supply of newly generated data.

**Data collection**

As the ROS simulation runs in the background, this code will record all data needed for the networks as textfiles. The data will consist of obstacle grid maps, social heat maps and coordinates. This will be achieved through subscribers to the topics of the robots framework, where each message will be formatted and then recorded into the textfiles.

**Split data in training data and ground truth**

Since the raw data contains both the training data and the ground truth data, it will need to be split to be able to train the network. This program splits a text file into 2 separate text files, one containing data to be passed through the model during training and the ground truth to be compared with output from the forward pass.

**Data preprocessing**

Since the robot is unable to calculate the social density heat map over the entire area. This program will be needed to collapse all the social heat maps in the final social density heat map around the area. This final social heat map will be normalised to ensure consistent data representation. By cropping this final social heat map to the training data, the input and label pairs can be made for training.

**Model training**

This node will contain the deep learning neural network models defined using Pytorch. During the training, the model will use the preprocessed data to adjust the models weights and biases, improving its ability to predict low, medium and high areas of social density. The model is then output as a file after the training has been concluded.

**Evaluation**

This program will pass unseen data through the trained models to evaluate and test the models. Using statistics like accuracy and intersection over union, the models' performance can be analysed and along with generalising it’s ability to deal with real-world scenarios beyond the training data. This allows the user to reveal the limitations and areas of improvement of the model.

**Model deployment for ROS network**

Once the model is deemed suitable, it can be deployed as a ROS node program with a publisher and subscriber. The subscriber is able to receive obstacle grid map messages, where it is formatted to be input through the model. The predicted social density grid map is then output as a message to a ROS topic via a subscriber.

This pipeline architecture is simple and linear. This makes it straightforward for the user to understand where they need to start, and the sequence at which they should continue through the pipeline. Any unnecessary complexities in the design were avoided to ensure a logical and streamlined workflow. This minimises errors, enhances productivity and the reliability of the final program being deployed.

## 3.3 ROS Robot simulation

The main packages used in the ROS simulation within the project are:

* **smf\_move\_base\_planning**

This module is used for controlling the navigation of the robot. This creates the paths for which the robot follows to move from its starting position to its final destination.

* **smf\_move\_base\_mapping**

This module is used to create maps from which the robot can understand it’s environment through grid maps which are two-dimensional data structures representing the probability of obstacles in that area.

## 

The primary objective of ROS robot simulation is to simulate the robot navigating through a virtual environment, manoeuvring around obstacles like walls, furniture, and dynamic elements like social agents. The robot will be placed into a large, indoor setting called “map”, simulating a real world social space such as an office or hospital, where it will be given a destination. Using sensors like LIDAR and depth cameras, the robot will collect data in real-time and create obstacle grid maps, which are two-dimensional data structures with cells that contain the likelihood of an obstacle being present. There is a grid map for the static obstacles called the “obstacle grid map” and a grid map called the “social density grid map” for the dynamic obstacles. Safe paths from the robot to the destination are calculated with both grid maps which the robot moves along to reach the final destination. The goal is to collect the data from the robot as obstacle grid maps and the social grid maps to train a machine learning model so that areas of low and high social density can be identified in the environment.



As mentioned in the project justification, the framework of the robot simulation ROS is provided by my supervisor and is based on the framework provided in the paper by Silva et al. [3]. The main ROS package I will be using is the *pepper\_social\_nav\_tests* package along with additional packages that allow the simulation to run.

A core concept in ROS is the use of “nodes” and “topics”. Nodes are processes that perform computation. These nodes can subscribe to topics to retrieve data and publish to topics to send data. This system allows modularity, where each node can be tested, developed and reused without affecting the entire system. The framework used to simulate the environment are made of these nodes and topics.

In ROS, data between nodes are sent as “messages”. Messages are the data structures that are exchanged between these nodes, which consist of fields that can contain data, e.g int, str, array. Data is streamed through serialisation and deserialised upon arriving at the destination node.

### 3.2.1 Pepper robot

### 

The robot provided by the framework is a simulation of the pepper robot from SoftBankRobotics [18]. (See figure 9.) Its primary function is to navigate the environment in ROS by finding safe paths to the end destination. Pepper is a robot designed for interacting with people making it suitable for a simulation in a crowded social space. Its appearance is human-like and with a head, two arms, a body and wheels. It contains sensors to help understand its environment and anticipate the changes happening around it.

### 3.2.2 Environment

The environments which are presented in the ROS simulation consist of using a structure called a map. A map is a complete representation of an area. This includes things like walls, doors, furniture and other objects that would exist in a real-life area.

Gazebo is the ROS tool to load and simulate these environments. Within the project four maps are tested, these are:

**Bookstore**

****

**Office**



**Small house**

****

**Small warehouse**

****

Each map provides its own challenges through their different characteristics. This diversity allows the robot to be tested on many different scenarios which the robot could face if deployed in a real-life scenario involving crowded, indoor, social spaces.

RViz is the ROS tool to run the environment and the simulation assets such as the robot and social agents. This allows the environment, the robot and the social agents to interact.

### 3.2.3 Social agents



Since the environments being emulated are social spaces, people are simulated walking around the environment. These are called “social agents” and they’re primary goal is to mimic human behaviour within a crowded social space. When moving around, they will respond to the robot and other people by moving away from them when they are too close, mimicking discomfort. Agents will also either move alone or in groups. When in groups, they’re movements will be coupled to further simulate situations in a social space which the robot would have to anticipate whilst moving around. (See figure 11.) The concept of a “personal space” is created through an occupancy grid map called a Social heatmap. Every agent will have a radius around them deemed as personal space which the robot will avoid whilst moving around. (See figure 12.) In figure 13, the social heatmap is calculated for each cell. Where is the number of agents considered for the cell, is the state taken from the consideration of the EPSM in X.-T. Truong et al[19]. The value persistence is detailed in figure 14, where, is current time whilst is start time. A is a time decay factor. These equations help emulate human presence so that the robot is able to safely route around people and obstacles. As seen in Figure 15, all the social agents are managed by the **/pedsim\_simulator** node, specifically the **/pedsim\_simulator/simulated\_groups** and **/pedsim\_simulated\_agents** nodes. These are also sent to the **/pedsim\_vizualiser** nodes so they can be simulated in the ROS environment.





### 3.2.4 Navigation



The navigation of the robot consists of a framework. This is needed to allow the robot to understand how to move around an environment. This is done through many modules within the framework which take place in specific ROS nodes.

**Perception sensors**

Perception sensors retrieve data from the surroundings and are sent to the world modelling component.

**World modelling**

The world modelling component creates data structures called maps to create a better representation of the environment. The 3D map is made from the depth sensor and is a volumetric representation of its environment. The social heat map is generated from the presence of the social agents. This is the most important module to the project as it is responsible for generating the data needed to train the machine learning model.

The other modules assist in successfully navigating through the environment to its destination. Using the planner, a path is created which the robot will follow. Through the path following components the movement of the robot is realised.

**Environment**

The robot then moves along a calculated path. The environment immediately around the robot then changes and the cycle repeats until the final destination is reached. The environment is set up in gazebo and RViz.

In the figure above, you can see the path indicated as a green line which the robot creates from the navigation framework.

### 3.2.5 ROS Training program

In order to run the program, the map, the start and end positions of the robot need to be set. This is done through a python program in the */pepper\_social\_nav\_tests*. However, for this project I have edited that file and developed a custom training route program. The primary goal of the custom program is to increase the quality of data collected so that the machine learning models develop a more accurate and reliable representation of the social space. Additionally the code has been designed to integrate with the data collection program so that the obstacle grid map data and social grid map data can be stored.

In the original training route program, the robot is only able to route to a single destination. In the custom training code, the robot is able to route to many destinations. However these destinations are not user defined coordinates but waypoints situated on the map. This was implemented to allow the robot to explore the map more before ending the program. This ensures that the data from the grid maps represents a more whole representation of the environment and improves the quality of data. As the number of waypoints which the robot visits increases, the longer the route will be. To ensure that the robot is given enough time within the test to reach each waypoint, the user can change the maximum time for a route between waypoints.

Although exploring waypoints aids with producing a wider representation of the data, by choosing random samples this is not always the case. In situations where the waypoints are close to each other, the robot does not explore much of the area. A solution to this could be to create a system which employs a systematic selection strategy or heuristic approach to ensure that the complete route around all waypoints covers the majority of the map in order to create a more balanced representation of the environment.

The user is able to choose the number of waypoints which the robot visits. However this is a static number and the chosen number by the user may not be optimal for the area. Larger areas would need more waypoints to visit to create a reliable amount of data whilst smaller areas would not. In some cases as well if the number wasn’t static. The issue of the lack of exploration from the robot on a complete route could be dealt with through increasing the number of waypoints it visits. Adjusting the number of waypoints the robot visits based on a system that identifies properties of the environment like size and number of obstacles would benefit the training route program extensively.

## 3.3 Data collection

The data from the training route program is essential for the training of the machine learning networks and evaluating the model. The goal of the data collection program is to store a representation of the environment which the robot creates through obstacle grip maps, social heatmaps and robot coordinates so it can be used on the machine learning networks to train.

The primary data that needed to be received are the grid maps (obstacle grid map and social heatmap). For every obstacle grid map, there is a social grid map pair. It is essential that the program records both parts of the pair and contains a method to match them to each other when accessed later on by the data preprocessing. These 2D data structures should be collected through a method that is reliable and efficient to ensure that the data is not corrupted or lost when being recorded and stored.

The coordinates of the robot when the maps are generated is also used when training the machine learning networks. This will need to be recorded along with the grid map pairs. The recording of the coordinates should be reliable and correct.

**/smf\_move\_base\_mapper** is the ROS node that is responsible for the creation of the maps within the framework. The node contains two topics, **/smf\_move\_base\_mapper/octomap\_map** and **/smf\_move\_base\_mapper/social\_grid\_map**. The octomap is the 3D map created from the depth sensor. This 3D map is then used in the **/grid\_map\_vizualisation** node to create the obstacle grid maps and streamed over the **/grid\_map\_visualization/obstacles** topic. This topic hosts **OccupancyGrid** messages. The other topic from **/smf\_move\_base\_mapper**, **/smf\_move\_base\_mapper/social\_grid\_map** produces the social grid maps and hosts **GridMap** messages. Therefore the maps that are needed are streamed over the **/smf\_move\_base\_mapper/social\_grid\_map** and **/grid\_map\_visualiztion/obstacles** topics. The maps in the messages exist in a one-dimensional row-major format, therefore the height and width of the grid needs to be converted into a 2D format. All messages across these two topics contain a .seq field. This is an identifier field which can be used to match the obstacle grid map and social grid map pairs.

The coordinates of the robot are streamed to the topic **/pepper/odom\_groundtruth** and hosts **Odometry** messages. These messages contain the x and y coordinates of the robot.

In order to record these topics, the program contains ROS nodes that subscribe to these topics so that when they are published the program is able to access them. When the subscribers receive the messages, the fields are then accessed and converted into a format suitable for saving so the data within can be accessed logically. The format being:  
  
[ I, S , H , W,{ X, Y, }D… ]

Where I is an identifier for the grid map (0 for obstacle and 1 for social) , S is the header or sequence used to match pairs of maps together, H is height of the grid map, W is the width of the grid map, X is the x coordinate of the robot, Y is the Y coordinate of the robot and D is the actual grid map data. The coordinates should only exist if the program is recording training data for the model taking into account the coordinates of the robot.

There should be separate files for recording data without coordinates and for data with coordinates. The reason for there being two files is for modularity and clarity during development. The files are tailored to each use case and allow configuration to specific scenarios. It is simple for the user to identify which file needs to be run when using the machine learning pipeline.

This data should be saved to files for use for the next section of the pipeline. The files should follow an appropriate naming convention. The data should be searchable and indexable. The data should be human readable so allow for easy debugging.

File names are the same for both recordings with and without coordinates. This feature was chosen to make it easier for functions to access the files since they will both have the same naming convention. However if the user were to want to find out if a file contained coordinates or not, they can check the format of the data saved as it is human readable. Another approach could be to have different name conventions for the recording files so it can be identified without opening.

## 3.4 Data Preprocessing and Augmentation Techniques

In order for the data to be effective when training the machine learning models, the data needs to be checked and altered to ensure that the quality of data is high enough to allow proper training.

### 3.4.1 Split data: Training and Ground truth

The split data program is used to split the recording file into two text files, the obstacle grid map text file and the social grid map file. The raw text file generated from recording the training route is a mix of both the maps so they need to be separated. The code also removes odd pairs of maps within the raw text file. Using the header, grid maps that do not have a matching ID are not written onto the new text file. The separated files are saved ending with “\_socialGridMap” and “\_obstacleGridMap” respectively and start with the recording number. The obstacle grid map data will be the training data whilst the social grid map data is the ground truth data.

The reason for having two files is to have more efficient handling and processing. Data from both the obstacle grid maps and social grid maps are manipulated in different ways. Two files ensures specialised processing. This method also reduces complexity as the same data exists amongst each other.

### 3.4.2 Data augmentation techniques

Data augmentation is crucial for enhancing the robustness of the data that the machine learning models train on. This ensures that the data is better represented by manipulating the existing data. Feature extraction is the process of transforming the data into only the relevant features so it is a suitable format for the machine learning network.

#### 3.4.2.1 Aggregation

The social grid maps after the split maps program are the immediate social grid maps generated by the robot at that current time. The goal of the project is to be able to identify the areas of the environment that contain low, medium and high social activity. To do that we need to combine all the social heat maps into a final social density heatmap, where every cell represents how often social activity occurs in that area. This is done through aggregation methods. This will create a data structure that represents the activity across the entire environment which can be used to train the model.

The method most suitable in the term of aggregation and the use case is the average. A representative measure of the overall activity per map is what is useful when training the model. It is a much more balanced view of the social activity within each cell. Sum aggregation is simple and can be used to represent the total social activity across all the maps; however this is not a normalised or balanced view of the social activity. Max aggregation is only useful for the highest levels of social activity. The main goal of the project is to identify low areas of social activity thus making it unsuitable.

#### 3.4.2.2 Cropping

By cropping this final social density heat map to the size of the obstacle grid map, the pair represents a better representation of the obstacles against the overall social activity in that area. This new heat map will only cover the same area as the obstacle grid map. This optimises training as the model will only understand the relationships of the obstacles and social density in the relevant areas (areas around the robot that it has discovered).

#### 3.4.2.3 Padding

The method of padding will be used to allow the aggregation of social maps. By artificially expanding the grid maps, the social heat maps can be averaged in a simple manner. This is done through zero-padding so as to not affect the data.

#### 3.4.2.4 Normalisation

In addition, the values of interest are the range of values presented in the social density heat map and not the raw values. Normalisation can be used to stabilise the data across the final social density map. This will set all values across the map into a standard range (0-1). Also reduces sensitivity to initial data range and improves the training of the machine learning model. Min-max normalisation is the only method that makes sense within the goals of the project. By min-maxing to a set range. It is much easier to produce set classes with the range of numbers representing social density.

#### 3.4.2.5 Resizing

When training the machine learning network, all sizes of the training and ground truth data should be consistent. Resizing the grid maps to a consistent and uniform size allows the model to train better. Having uniform data also reduces the hardware overhead when training. The user is able to change the resolution of the resize to tailor the training of the network to a use case. Interpolation is used to resize as it is simple and reliable.

#### 3.4.2.6 Classification

As the aim of the machine learning network is to produce values corresponding to three distinct categories, the raw data will need to be assigned and classified to fit the concept of low, medium and high social activity. This is done through classification. By using bounds, the continuous normalised data can be converted into three distinct classes. These bounds can be adjusted by the user to fit each use case when producing a trained machine learning network. This is very important for the training of the machine learning model as accurate classification is essential for an effective trained model. The model is better able to understand the relationship between the training data and the ground truth with this method.

Three classes were chosen with discussion from my supervisor to be appropriate. Benefits of three classes include flexibility with many use cases. Data

## 3.5 Model selection

Utilising the correct machine learning model is paramount to achieve the goal of predicting social density. The model should be able to handle spatial data from the grid maps when training so that it can understand the relationship between its surroundings and the potential social activity around it. Since the training data consists of grid maps, the model should be able to handle different sizes and complexities of input data. The model must also be able to use classification methods as we are only interested in the distinct classes of social activity. Implementation of the model should be achieved within the ROS framework. The trained model should be robust and its performance should be within acceptable ranges for the use case.

An additional machine learning model is also used with the combination of the coordinates of the robot. Using this method the model should be able to better understand where the robot is within the obstacle grid map and therefore produce better predictions concerning social activity.

Fundamentally, the task of identifying areas of social density is a segmentation problem. The grid map is to be broken up into areas containing different classes, where each cell will be identified with a class (low, medium, high). Therefore a deep learning algorithm excels at this task.

Traditional machine learning methods that do not involve a neural network struggle with the complexity of spatial data like two-dimensional grids. They miss important patterns and hierarchies within spatial data. Additionally, traditional machine learning methods are not versatile with varying data and complexities within it.

### 

### 3.5.1 FCN

Fully Convolutional Networks (FCNs) are a suitable choice as they are efficient at segmentation tasks, especially pixel-wise prediction. They are able to handle varying sizes of data making them versatile for different use cases. FCNs understand spatial information by retaining the hierarchical features as the network goes through different layers. This is the main feature that makes this network effective for the project's goals. In addition, like all deep learning models, the layers can be customised by the user to fix the use cases of the machine learning network. By developing an architecture that is specifically designed for analysing the grid maps, the model will better understand the relationships of the spatial data of the grid maps. However, FCNs only contain convolutional layers meaning that the coordinates can not be used alongside the obstacle grid maps when training.

### 3.5.2 CNN

Similar to the FCNs, Convolutional Neural Networks (CNNs) contain convolutional layers however they are capable of other layers within their architecture. This flexibility allows non-spatial data like the coordinates of the robot to be trained with in conjunction with the grid map data. Using non-spatial data as well, could possibly lead to more accurate predictions, as the model will be able to create feature representations of the robot’s location within the grid map.

U-net was another consideration as it offers the same advantages as the FCN however it contains skip connections that makes it better at identifying smaller details within the spatial data. As the grid maps are not extremely detailed like medical images from N. Siddique et al. [20], there is a high probability that the model would introduce noise when predicting the social density. This would hinder the ability to deploy the trained model within the network.

## 

## 3.6 Network Architecture Design

### 3.6.1 FCN

**Encoder**

This is responsible for extracting the spatial features and reducing the spatial dimensions of the grid map. As the grid map can be treated like a grayscale image there is only one input channel. The first two pairs of Conv2d and ReLu capture the features of the grid map and extract them. The MaxPool2d layer down-samples the feature map in order to lower their spatial size whilst keeping the important features. Afterwards another layer of Conv2d, ReLu and MaxPool2d is followed by a Conv2d to further extract the features.

**Decoder**

This part of the architecture reconstructs the image back to the original dimensions with the output channels for the low, medium and high social activity classes (3 channels). ConvTranspose2D upsamples the data. More convolutional layers occur before the grid is passed through the final ConvTranspose2D layer where it is output from the machine learning network.

There is no need for an activation function as Cross Entropy Loss is used to train both models which relies on the data to be logits.

### 3.6.2 CNN



The CNN architecture allows the input of both the coordinates of the robot and the obstacle grid man in order to produce a social density heat map. This is achieved through feature extraction outside of the encoder and decoder pair. The coords enter a linear layer where the features are extracted into 32 hidden nodes. The obstacle grid map data enters a convolutional layer where the features are extracted into a 8-channel feature map. By concatenating the features maps before going into the encoder, the model is able to understand the relationships between the robot’s position and the obstacle grid map.

The encoder then takes in the 40 channel feature map of the combined features of the coordinates and obstacle grid. This encoder is the same as the FCN but with the addition of dropout layers. These layers help prevent the model overfitting by dropping a unit at random during training. This assist with generalisation

The decoder is the exact same as the FCN.

There is no need for an activation function as Cross Entropy Loss is used to train both models which relies on the data to be logits.

## 3.7 Model training

### 3.7.1 Loss function

The loss function is used to obtain the difference between the predicted result and the ground truth. Choosing a good loss function is crucial to ensure the model is training effectively and efficiently. Cross entropy loss is used to train both the FCN and the CNN models since it works well with the goal of semantic segmentation.

**Cross entropy equation:**



Where:

* C is the total number of classes
* is the ground truth label for class
* is the predicted probability for class obtained after applying the softmax function to the logits.

**Softmax function:**



Where:

* is the logit for class
* is the exponential function applied to
* The denominator is the sum of exponential of all logits and normalises the output so probabilities add up to 1.

**Weighted loss function**

If there is an imbalance of classes within the training data, the model may bias towards the classes that are more common. Therefore, the accuracy of predicting the underrepresented classes goes down. To help prevent this, the loss function can be weighted to bias towards the less frequent classes.



Where:

* is the weight for the true class of the n-th sample
* is the logit for the class for the n-th sample
* is the exponential function applied to the logit for class
* The denominator normalises the logits so probabilities add to 1

### 3.7.2 Optimiser

The loss function is able to produce the difference between the predicted output and the ground truth. In order to train the parameters within the model using the loss function, an optimiser must be used. Gradient descent is the fundamental concept employed by optimisers. This is where the parameters of the deep learning model are adjusted with the aim of reducing the value produced by the loss function. Choosing the correct optimizer is based mainly on the trade-offs of convergence speed, stability and computational overhead.

The models train using the Adaptive Moment Estimation or Adam for short [21]. This model contains an adaptive learning rate which are hyperparameters that are adjusted by the optimiser itself. It is also very versatile with the various data and deep learning network architectures.

Other optimisers could be used like RMSProp however be appropriate when the hyperparameters are adjusted accordingly. Not having to adjust these makes the program much simpler and convenient in the context of the pipeline.  
  
The machine learning models should contain other options for training which users of the pipeline can choose between for different use cases.

## 3.8 Model evaluation

Trained models should be able to be evaluated using a program along with evaluation sets to analyse its performance. The values should be useful with addressing any issues that the model has, as well as giving the user a clearer understanding of the characteristics of the trained model.

**Accuracy**



This is the measure of the overall correctness of the predicted classes. This is calculated by analysing the ratio of the correctly classified predictions with the total number of predictions  
  
**Per class accuracy**



This value is the accuracy for each class. This is useful to identify which classes the model is struggling with.

**Mean Squared Error**

### 

This is the Mean Squared Error(MSE) where N is the total number of samples, is the actual value for the -th sample, and is the predicted value for the i-th sample. This value is the average of the absolute differences between the predicted values and the ground truth values.

**Jaccard index**



This is the intersection over union or the Jaccard Index where A set of pixels classified by the model and B are the pixels of the ground truth. This measures the overlap between the segments produced by the model and the ground truth.

**Precision**

  
Precision is the measurement of the ability of the model predicting the correct classes. Where TP is the number of true positives and FP is the number of false positives.

**Recall**



This is the recall value where TP are the true positive predictions and FN are the false negatives. This reflects how well the model classifies all relevant pixels.

**Confusion matrix**  


This shows how the model classifies the values, where TP are the true positives, FP are the false positives, TN are the true negatives and FN are the false negatives. This is very important for identifying issues with classification.

### 

This average loss generated through the evaluation loop made by the loss function, where is the number of loss values and -th value is the loss value of that evaluation loop.

### Model deployment

The trained models will be deployed using a program which allows the model to be contained within a ROS node that will stream the data into the model and then out of the node.

# **4. Implementation**

## 4.1 Development Environments

The programming language used in the pipeline is Python. This is due to its extensive data science libraries for deep learning and robot development. I am also quite comfortable with coding in the language as I have had many years of experience with it. The other language that was proposed to be used for the project description was C++. I did not see the benefit in choosing this over Python as C++ is not known for its strengths with the machine and deep learning. The version of Python used throughout the project is 3.8.10.

For the deep learning framework, two options stood out. Tensorflow and Pytorch. After researching the comparisons between each other, it boiled down to a shared perception. Tensorflow is best for production, whilst Pytorch is better for research studies. This project is a study of how the deep learning model would operate in the use case so I considered Pytorch to be the best option. The version of Pytorch used is 2.2.2.

ROS will be used for the simulation of the robot as well as to generate the routes for data collection. ROS code based on the paper by Silva et al. , was provided to me for data collection. This framework contains different environments that the robot can navigate through. In the simulation, agents simulate the activity of people within the area. This code also contains options for navigation. The project uses ROS Noetic on ROS version 1.16.0.  
  
The package */pepper\_social\_nav\_tests* was provided to me with all the packages needed to test and run the ROS framework for the social robot navigation system.

The pipeline was developed on VSCODE. Providing an environment for development and debugging.

The machine-learning pipeline was developed with Git and GitHub for version control and collaboration. The repository for this ros package exists on GitHub:  
  
https://github.com/xanialh/ml\_ros\_package

## 4.2 Code Implementation Details

## 4.2.1 Training Route Simulation Code

***Training\_route.py***

****

In order to allow the program to visit multiple waypoints, *smf\_nav\_stack\_requester.py* program has been changed so that the return of the function is removed.

Training\_route.py uses ROS nodes and topics to set up the framework within **/pepper\_social\_nav\_tests** so that the simulation can be run. Emulating the robot routing through an indoor, crowded social space.

Using a YAML file, the code loads configuration data into python variables to be used throughout the program through the *load\_config()* function. This includes the map to be loaded, the number of waypoints, the starting position and the maximum time for every route between waypoints.

****

The class *ProcessListener* is responsible for monitoring the lifecycle of processes started by the training route program. The function *init\_launch* starts various processes that are essential for running the simulation environment. The object created from the *ProcessListener* class tracks the status of these processes. Mainly, if a process dies or is completed, the *process\_died* function is called and logged. This is important to have during development and testing so that failed processes can be debugged and dealt with.

The *init\_launch* function usings the config data to start the launch file in the */pepper\_social\_nav\_tests* package.



A **waypoint\_listener** node is used to receive data from the simulated environment on the position of the waypoints. This node uses the **waypoint\_topic** and receives the message in the form of **MarkerArray** where the function callback *waypoint\_callback* operates on the data so that it can be used in the program. The *waypoint\_callback* function receives the message form the listener and iterates over the data. As the only useful information is the x and y position of the waypoint, the loop iterates over the data structure and appends the x and y coordinates to the *waypoints* list defined in the program. The *waypoints* list is then set to a *rosparam* so it can be used by the program. The function then unregisters the node so that no more waypoints are collected.

Within the main loop of the training route program, the program will give the ROS framework data about the waypoints, removing each one if it is visited by the robot or the maximum route time for that waypoint has been exceeded. Once all the waypoints have been removed from the waypoints lists the program ends.

The training route program contains integration with the data recording program. Using the **record\_pub** node, the recording program sets a False message to stop all recording before starting a new recording of data. Once the main loop is run, the node streams *True* messages to subscribers so that the recording starts and continues until the program ends.

When collecting data, the training route program can be automated to restart using the script *route\_script.sh* where the user can define how many times the program is restarted. This is especially useful when recording a large amount of data so that the user does not need to restart the program manually.

I am also limited with the framework of the ROS package */pepper\_soc\_nav\_tests*. The navigation is not perfect and the robot is prone to getting stuck on obstacles. There are also situations in which the robot is unable to find a route past a social agent causing it to freeze in place. This hinders the quality of data which I am able to produce. The framework also does not contain a method to increase the speed of the simulation. Whilst collecting the data I am unable to speed up the simulation to increase the efficiency of data collection. The framework also has difficulties with long routes. The route will crash often if the routes of the robot take too long. This meant that exploring more than three waypoints across the four maps was very unstable. In addition, when using the provided script for automating the rerunning of the program, it would sometimes fail to load and halt script, therefore the script would need to be restarted manually. In order to work around this I had to keep checking on the program to ensure that the program was running and restarting it manually if need be.

## 4.2.2 Data Collection Code

***record\_maps.py and record\_maps\_coords.py***

There are two programs for recording the data, *record\_maps.py* and *record\_maps\_coords.py*. The former is used for recording data for the machine learning networks without coordinates whilst the latter is used for the networks with coordinates. Using a YAML file, the code loads configuration data into python variables to be used throughout the program.

The main ROS node of the program is “**gridMapCoordCollector**”. This node exists to bind the existing ROS nodes from the simulation to the program so that data from their topics can be collected.



The ROS\_recorder class handles recording data from the obstacle grid map (**/grid\_map\_visualisation/obstacles**), social grid map topics (**/smf\_move\_base\_mapper/social\_grid\_map**) and coordinate data if used (**/pepper/odom\_groundtruth**). This object is instantiated for every simulation using the *\_\_init\_\_* function. The record method sets up the subscribers to the grid map nodes, where the two subscribers each have their own callback function. The callback functions extracts the data from the messages they received and convert them into a list with the format mentioned in the design section:

[ I, S , H , W,{ X, Y, }D… ]

Where I is an identifier for the grid map (0 for obstacle and 1 for social) , S is the header or sequence used to match pairs of maps together, H is height of the grid map, W is the width of the grid map, X is the x coordinate of the robot, Y is the Y coordinate of the robot and D is the actual grid map data. The coordinates should only exist if the program is recording training data for the model taking into account the coordinates of the robot. Once formatted the data is written onto a text file using np.savetxt() with the naming convention:

{id}\_recording\_{datetime}.txt

Where id is the id of the recording object and date time is the date and time of the start of the recording program.



The *dataCollector* class manages the overall data recording process. When the program is run it will make a *dataCollector* object, where for every simulation, it will make a *Ros\_recorder* object, this is achieved through the *loop()* function. Whenever the simulation ends, the *Ros\_recorder* object will be stopped and a new one made as the program restarts. The *dataCollector* object will also only record a maximum number of files defined in the config file, where the program will stop as soon as this maximum is reached.

Initially I planned to use rosbags to record data from the training program however there were many issues with this approach. The rosbags were not suitable for recording when the simulation environment closes and restarts. This would desync the data being recorded to the rosbags and prevent it from being indexed properly. Also when something like this occurred, no errors were raised by the program and until the rosbag data was checked and the data would be corrupted. Rosbag also contained issues with storage limitations, when the grid maps would reach a certain size they would not be able to be recorded to the bag. This happened often enough that I decided to use text files instead.

The code does not have an option to save files for evaluation when recording. This means the user will have to set aside some files for evaluation. However this can be implemented into the program by changing the file location once enough training files have been created so that and continue recording data.

## 4.2.3 Data Splitting Code

***split\_maps.py***

The *split\_maps.py* file is used to split the raw data into separate obstacle grid map and social grid map files.

A YAML file is used to load the input and output directories of the program through the *load\_config()* function.

Within the *split()* function, the program will find matching files using the header ID. All data beginning with “0” will be written onto the obstacle grid map file and data beginning with “1” will be written onto the social grid map file. Any odd pairs will not be written on to the new files. New files will end in “\_socialGridMap.txt” and “obstacleGridMap.txt” respectively with the same filename which was loaded into the program.

This program could be omitted from the pipeline if the recording program saved data to two separate files making the pipeline less complex. However through this approach, all incomplete data is removed before being separated into the training data and ground truth data.

## 4.2.4 Data Preprocessing Code

***create\_SDHM.py and create\_SDHM\_coords.py***

The programs *create\_SDHM.py* and *create\_SDHM\_coords.py* are responsible for preprocessing and augmenting the code so that the data is more suitable for training the machine learning networks. The program loads the data from the input files and augments them and then writes them into the final text files used for training the machine learning models.

The *main()* function starts the program where it also loads configuration data from a YAML config containing the input and output file paths for the program using the *load\_config()* function.

The *load\_from\_txt()* function contains the code that deals with the preprocessing and augmentation of the data. Once the data is loaded, it is separated into the social grid map and social grid map pairs. The largest social grid map is found and saved to the “*reshape\_final\_social\_grid*” variable. All social grid maps are then zero-padded using the *pad\_array\_to\_shape()* function to the same dimensions of the *“reshape\_final\_social\_grid”* variable and then averaged and aggregated to produce the social density heat map of the entire area. The social density heat map is then passed through the *min\_max\_norm()* function where it is normalised using min-max scaling to values between zero and one. Afterwards each obstacle grid map is paired with the social density heat map that is cropped to the same dimensions. Each obstacle grid map and cropped social density heat map is then written to their new respective files.

Both the programs (with and without coordinates) deal with data preprocessing in the same way however the *create\_SDHM\_coords.py* retains the coordinate data across the entire process.

This code does not classify the data. This is done so that there is flexibility and experimentation in mind. Having the data augmented into classes here would mean that if the user wanted to change the boundaries between classes, they do not have to run the data through the data preprocessing code.

## 4.2.5 Model Training Code

The two programs, *FCN.py* and *CNN.py* contain deep learning models and tools for training.

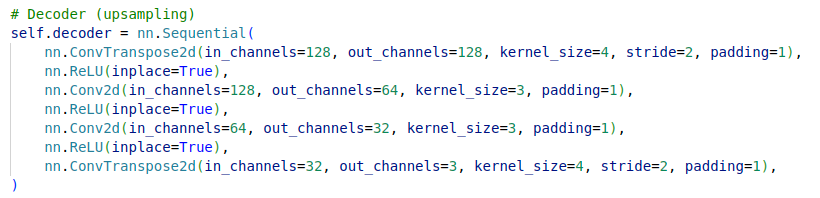
A YAML file is used to load the input and output directories of the program through the *load\_config()* function.

Both programs load data from the files produced from the *create\_SDHM.py* and *create\_SDHM\_coords.py* programs.

The class *FCN* defines the architecture in the *FCN.py* program; this consists of an encoder and a decoder.



The architecture for the encoder is the same as in the design section.



The architecture for the decoder is the same as in the design section.

The forward method of this class passes all data through the encoder and then the decoder.

The *CNN* class contains the additional layers detailed in the design section to allow non-spatial data to be trained on. The field *self.robot\_pos\_encoding* contains a linear(*nn.linear*) and a rectifier layer (*nn.ReLU*) to create a feature map of the coordinates. *Self.obstacle\_map\_encoding* contains a convolutional layer(*nn.Conv2d*) and a rectifier layer (*nn.ReLU*) to create a feature map of the obstacle grid map. Using the function *torch.cat()*, the feature maps are combined and passed through the encoder and decoder layers.

The encoder and decoder layers are the same as in *FCN.py* with the addition of dropout layers(*nn.Dropout*) to help prevent the model from overfitting.

**HM\_Dataset class**

The *HM\_Dataset* class is used to contain all the training data and ground truth data so the model can train.  
  
The fields *self.data* contains the obstacle grid map data, *self.labels* contains the social grid map data and *self.coords* contains the coordinate data.

The *add\_data* method is the method that stores the data into the object. This converts all the received data in the form of lists into tensor objects. The data is also resized into the user defined dimensions before it is stored. This is done through the *cv2.resize()* function and the interpolation method *cv2.INTER\_AREA* is used.

The *\_\_getitem\_\_* method is a defined method which the training function will use to pull data from to train the model.

***Def social\_map\_to\_labels(social\_gridMap)***

This function is used to classify the social grid map data into three classes. Each element in the array *social\_GridMap* is iterated through. If the value is less than 0.33, it is assigned 0 for low density, if it is above 0.66 it is assigned 2 for high density and all values between are assigned 1 for medium density. The new classified social grid map is then returned.

The *train()* function is the main function for training the machine learning models. After loading the configuration data, the model is instantiated. The accuracy of the model through the training is measured using the *torchmetrics.Accuracy* object.

If the configuration data contains a True weighted value, the weighted cross entropy loss object is created. This will use the class matrix from the class matrix data from the config. The weights are calculated using the presence of classes that appear from training without the weighted optimizer. If the weighted value is false then a standard cross entropy loss function is created.

After the loss function is created, an Adam optimiser object is created.

After all data is loaded, the loss functions and optimiser functions are defined, the training loop starts. This training loop uses user defined epochs and batch sizes. An epoch is one complete pass of the dataset through the model and batch sizes are the number of training examples used in one pass through the model’s parameters. These are set by the user. After the model trains through the entire loop, the model is exported as a .pt file with the name detailed in the configuration file.



Simpler models of the FCN and CNN were experimented with initially however the models struggled to capture the spatial features within the grid maps.

I was unable to implement an early stopping system to prevent overfitting due to all the resulting models underfitting when evaluating. This system uses a validation set to analyse the loss against unseen data after training through each epoch, where if the loss did not improve for a set number of epochs, the model would be output. The lack of validation testing is a limitation with the model as a whole.

## 4.2.6 Evaluation Code

*eval\_FCN.py* and *eval\_CNN.py* are the programs which produce the evaluation results for their respective trained models.

A YAML file is used to load the input and output directories of the program through the *load\_config()* function.

The code loads the files and data into the dataset and then into a data loader to be run through an evaluation loop. The metrics are objects created through the torchmetrics package and are updated as the loop runs. The program then prints out all the metrics once the loop is complete.

## 4.2.7 Model Deployment Code

The program used to deploy the code is the network.py program. This code creates a ROS node called “**mlNetwork**”. This node can subscribe to the topic that produces the obstacle grid maps. Using the function *o\_gridmap\_callback()*, the obstacle grid map is processed as an **OccupancyGrid** message. Using the processed data, the grid map is run through the loaded machine learning model. This predicted grid map is then published as an **OccupancyGrid** message.

There is no model code for the CNN models due to time constraints however it would be very easy to implement using another subscriber.

## 4.3 Documentation

The pipeline files should be used in this order in order to either create an FCN or CNN:

**FCN**  
*Training\_route.py -> record\_maps.py -> split\_maps.py -> create\_SDHM.py -> FCN.py -> eval\_FCN.py*

**CNN**  
*Training\_route.py -> record\_maps\_coords.py -> split\_maps.py -> create\_SDHM\_coords.py -> CNN.py -> eval\_CNN.py*

For the correct deployment of the pipeline, it is essential that the *pipelineConfig.yaml* file is edited and the parameters are set to ensure the program runs correctly. For each program, depending on where you run the code, it may not file the config file, this is crucial to set up within the program to ensure the parameters are loaded when running the program.

# **5. Results and Evaluation**



Above is a visual representation of the models producing social density grid maps using the evaluation program on an unweighted FCN model trained on the office map.

## 5.1 Network Performance Evaluation

Each model was trained using 100 epochs and 32 batches. This was set as a control across all tests. Each model was trained using 20 training route files. These files were recorded through the *record\_maps.py* and *record\_maps\_coords.py* respectively. Both programs were started in parallel before the training program started ensuring that the routes they recorded were the same. Then they were passed through the appropriate data preprocessing and augmentation programs of the pipeline (*split\_maps.py and create\_SDHM.py/create\_SDHM\_coords.py*).

### 

### 5.1.1 Unweighted FCN

|  | | | | | | | Per class metrics (low, medium, high) | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Model Type | Model | Map | Average Accuracy | Average Loss | Jaccard Index (IOU) | MSE | Accuracy | Precision | Recall | F1-Score |
| Unweighted | FCN | Office | 0.983 | 0.388 | 0.329 | 0.0288 | [0.997, 0.00350, 0.0] | [0.985, 0.0176, 0.0000] | [0.997, 0.0035, 0.0000] | [0.991, 0.0059, 0.0000] |
| Unweighted | FCN | Small House | 0.957 | 0.652 | 0.326 | 0.0771 | [0.992, 0.0201, 0.0054] | [0.964, 0.0718, 0.0470] | [0.992, 0.0201, 0.0054] | [0.977, 0.0315, 0.0097] |
| Unweighted | FCN | Small Warehouse | 0.921 | 0.908 | 0.319 | 0.141 | [0.980, 0.0460, 0.0044] | [0.939, 0.1009, 0.0337] | [0.980, 0.0460, 0.0044] | [0.959, 0.0632, 0.0078] |
| Unweighted | FCN | Bookstore | 0.923 | 0.811 | 0.343 | 0.131 | [0.973, 0.0749, 0.0756] | [0.948, 0.1230, 0.1873] | [0.973, 0.0749, 0.0756] | [0.960, 0.0931, 0.1077] |
|  |  | Average across maps | 0.946 | 0.68975 | 0.32925 | 0.094475 |  |  |  |  |

All the unweighted FCN models achieve high average accuracy above the 90th percentile. The average loss is high for the Small Warehouse and Bookstore map, intermediate for the small house and low for the office. The Jaccard index for each map is around 0.3 which implies a low overlap between the predicted and ground truth grid maps. MSE for all maps are low and acceptable for the use case. However if we look at the per class metrics we can see that the model only performed well within the low class metrics. All the low class metrics are extremely high, all falling above the 90th percentile. All other metrics are below the 20th percentile. This skews the averaged metrics to emulate good performance, when the fact is that the model performed poorly.

### 5.1.2 Unweighted CNN

|  | | | | | | | Per class metrics (low, medium, high) | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Model Type | Model | Map | Average Accuracy | Average Loss | Jaccard Index (IOU) | MSE | Accuracy | Precision | Recall | F1-Score |
| Unweighted | CNN | Office | 0.983 | 0.276 | 0.33 | 0.0293 | [0.997, 0.0076, 0.0017] | [0.986, 0.0439, 0.0057] | [0.997, 0.0076, 0.0017] | [0.997, 0.013, 0.0017] |
| Unweighted | CNN | Small House | 0.948 | 0.527 | 0.326 | 0.0939 | [0.983, 0.0251, 0.0177] | [0.964, 0.0499, 0.409] | [0.9832, 0.0251, 0.0177] | [0.974, 0.0334, 0.0247] |
| Unweighted | CNN | Small Warehouse | 0.917 | 0.837 | 0.319 | 0.156 | [0.976, 0.0469, 0.0061] | [0.939, 0.1019, 0.0200] | [0.976, 0.0469, 0.0061] | [0.957, 0.0642, 0.0094] |
| Unweighted | CNN | Bookstore | 0.925 | 0.736 | 0.349 | 0.129 | [0.980, 0.0625, 0.0983] | [0.944, 0.153, 0.238] | [0.980, 0.0625, 0.0983] | [0.962, 0.888, 0.139] |
|  |  | Average across maps | 0.94325 | 0.594 | 0.331 | 0.10205 |  |  |  |  |

The unweighted CNN models show very similar results. The average accuracy is very high still above the 90th percentile. The loss for each map however is less than the FCN models, this shows that the model does in fact learn better with the addition of non-spatial data within that regard. The average loss across all maps is 16% higher than the FCNs. The average Jaccard index represents poor overlap here with a similar value to the FCNs. MSE is very low which is the same as the FCN models and all values are very similar. The per class metrics show the same pattern as the FCNS. Performance is great for the low classes however all other classes perform extremely badly. With all other classes than low being under 0.3.

### 5.1.3 Class distribution

|  |  |  | Predicted |  |
| --- | --- | --- | --- | --- |
|  |  | Class 0 (low) | Class 1 (medium) | Class 0 (high) |
|  | Class 0 (low) | 8164651 | 17897 | 5185 |
| Actual | Class 1 (medium) | 91246 | 322 | 51 |
|  | Class (high) | 27271 | 65 | 0 |

The reason for the bias towards the low class is due to class distribution. From the class matrix produced by the FCN office model, you can clearly see that the low class dominates the dataset. The low class contains 98 percent of the classes within the ground truth data. The medium class exists in about 1 percent of the data and the high class exists in about 0.3 of the data. There is just not enough data from the medium and high class to properly train the models on those classes.

In order to try to counteract this imbalance of classes, a weighted loss function can be used to ensure that underrepresented classes are trained with more importance than the dominant classes.

For the weighted models, the class matrix from their unweighted models were used to calculate the weights of each class for training.

### 5.1.4 Weighted FCN

|  | | | | | | | Per class metrics (low, medium, high) | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Model Type | Model | Map | Average Accuracy | Average Loss | Jaccard Index (IOU) | MSE | Accuracy | Precision | Recall | F1-Score |
| Weighted | FCN | Office | 0.756 | 0.607 | 0.257 | 0.686 | [0.7651, 0.0983, 0.0972], | [0.9856, 0.0123, 0.0022] | [0.7651, 0.0983, 0.0972] | [0.8615, 0.0218, 0.0043] |
| Weighted | FCN | Small House | 0.743 | 1.38 | 0.266 | 0.516 | [0.763, 0.251, 0.13] | [0.970, 0.0415, 0.0159] | [0.763, 0.251, 0.130] | [0.854, 0.0712, 0.283] |
| Weighted | FCN | Small Warehouse | 0.719 | 1.5 | 0.26 | 0.811 | [0.758, 0.0776, 0.0265] | [0.9400, 0.0492, 0.0276] | [0.758, 0.0776, 0.247] | [0.840, 0.0603, 0.0496] |
| Weighted | FCN | Bookstore | 0.93 | 0.85 | 0.344 | 0.123 | [0.982, 0.0599, 0.0715] | [0.947, 0.145, 0.207] | [0..982, 0.0599, 0.0715] | [0.840, 0.0603, 0.0496] |
|  |  | Average across maps | 0.787 | 1.08425 | 0.28175 | 0.534 |  |  |  |  |

The average accuracy across all maps is lower than the unweighted models which is a reduction of 16%. This is due to the averages being less skewed by the low class. The bookstore map seems to have a higher average than the rest of the maps. The average loss for all maps is much higher than the unweighted models which is almost doubled from the unweighted models. Both the small\_house and small\_warehouse seem to struggle with training especially as both values are above 1. The Jaccard index is also lower than the unweighted models with a reduction of 16%, showing that there is less overlap when the model training is weighted. The MSE for each model is much higher than the untrained models with an increase of 500%. For the per class metrics, the other underrepresented classes see an improvement throughout. The accuracy for the small\_house model for the medium class is 0.25 which is an increase of 0.23 from the unweighted model. However for the rest of the maps, the metrics are still not reaching an acceptable threshold for performance of identifying these classes. No values even reach 0.3 which would be considered an “ok” value.

### 

### 5.1.5 Weighted CNN

|  | | | | | | | Per class metrics (low, medium, high) | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Model Type | Model | Map | Average Accuracy | Average Loss | Jaccard Index (IOU) | MSE | Accuracy | Precision | Recall | F1-Score |
| Weighted | CNN | Office | 0.863 | 0.52 | 0.3 | 0.306 | [0.873, 0.131, 0.116] | [0.988, 0.0196, 0.0069] | [0.873, 0.131, 0.116] | [0.93, 0.0341, 0.116] |
| Weighted | CNN | Small House | 0.851 | 0.803 | 0.3 | 0.334 | [0.880, 0.108, 0.0993] | [0.970, 0.0425, 0.0182] | [0.880, 0.108, 0.0993] | [0.921, 0.610, 0.0307] |
| Weighted | CNN | Small Warehouse | 0.815 | 1.25 | 0.301 | 0.475 | [0.8602, 0.0981, 0.213] | [0.947, 0.0782, 0.0444] | [0.860, 0.0981, 0.213] | [0.901, 0.0870, 0.0735] |
| Weighted | CNN | Bookstore | 0.855 | 0.851 | 0.324 | 0.269 | [0.902, 0.148, 0.16] | [0.947, 0.0931, 0.0751] | [0.902, 0.148, 0.60] | [0.930, 0.114, 0.102] |
|  |  | Average across maps | 0.846 | 0.856 | 0.30625 | 0.346 |  |  |  |  |

Similar to the FCNs, the weighted CNNs see an average accuracy decrease of about 10%. The average loss across maps increased by about 0.3 showing the model is also struggling with learning with the weights. The average Jaccard index is lower than the unweighted models however it is only slightly by about 0.03. The MSE triples in value compared to the unweighted models showing more error with the additional weights. With the per class metrics, the model sees a much slighter increase with the addition of weights compared to the difference between the FCNs.

## 5.2 Network feasibility test

Using a previous version of the machine learning pipeline, a network feasibility test was run. This proved that the models were able to be deployed within a network. As the development of the project continued, a test using the latest version of the package was not done due to time constraints. However with the code provided it would very likely work as a network.

## 5.3 Discussion of Limitations and Future Work

Overall the models succeed in identifying the low areas of social activity with the grid maps but fail to do so with the other classes. The dataset is very low social activity dominated, making the other two classes underrepresented when training. Collecting data through the training program and ROS framework also contains limitations. As of right now it is very difficult to collect a very large data set within a reasonable time limit and without intervention needed if the program. Additionally, there is no validation set training implemented in the final code to verify any hyperparameters. The pipeline has to be run as separate programs instead of through a main program. No generalisation tests were done. This could give insight on the models ability to generalise across the maps that they were not specifically trained on.

Weighting did not prove to help the training program learn to detect the medium and high levels of social activity. Using oversampling to generate data with more cases of medium and social activity could prove very useful with dealing with these issues. However with this method, the model could try to predict unrealistic data as the artificially generated data has no basis within the actual environment. On the opposite end, undersampling could also be used to remove more of the low activity class so that the other classes can be represented. But in the same way, this data is artificially created and has no basis with the actual environment. As the social grid maps are normalised, the boundaries were set at every third within the range. As most of the data occurs within the low areas, perhaps the boundaries could be adjusted rather through percentiles, where the low classes occur within the 33rd percentile of the code, the high class above the 66th percentile and the medium between those. This could prove very useful in realising the actual low, medium and high areas within the grid maps. Focal loss is a loss function which focuses learning on the underrepresented classes which could be a better function than Cross-entropy loss. Cost-sensitive learning is similar to this, where costs are assigned to classes so that the model is skewed towards better learning the underrepresented classes.

Including non-spatial data proved to assist the model with learning the spatial features of the grid maps. Including other metrics like distance from objects, the robots velocity, data on the social agents and other metrics important to the environment could help the model identify the trends of social activity. Within the environments, waypoints existed as Points-of-interest (POIs). Perhaps including data about the presence of POIs could prove useful as these places are deemed as important and therefore could be busy. Through more feature engineering, the input model can become a more whole and balanced representation of the entire environment around the robot, leading to better training of the model.

With regards to the entire machine learning pipeline, there are many error checking and validation systems which could be added to the functions. This would make the system more robust. As mentioned before, the pipeline could be implemented in a way in which the entire pipeline could be automated so that it is easier to use. There is also a need for a network deployment program for CNNs which could easily be implemented.

# **6. Conclusion**

In conclusion, the minimum requirements of the project are realised but contain many glaring flaws that make the entire pipeline not very useful. The system does contain a system which allows the generation and storing of training data for the model. Also, the system contains code which is able to preprocess said data and augment it for training the model. The training data can then be used to train a model by the system, where the model can then be evaluated using metrics. The finalised model can finally be deployed within a network. However, the model is only very capable of identifying areas of low activity which is not ideal if the model is unable to reliably predict any other class. The pipeline contains flaws and can be improved to be better suited for any user planning to use the system.

Future improvements and implementation are needed to ensure the system can function properly and achieve all requirements.

# **7. Reflection on learning**

Overall this project has been very interesting and I have delved into some fascinating concepts regarding social navigation and machine learning.

In order to run ROS, I needed a linux machine. I only had limited experience with linux from a module I did in university but I never really used it to its fullest extent. Luckily, my supervisor was kind enough to lend me a laptop with linux and ROS installed. Throughout the program I have developed a keen understanding of the functioning and usefulness which the linux operating system provides.

When I began the project I had no experience with ROS. I was given tutorials on YouTube which I completed in order to have a foundation to allow myself to better understand the ROS framework that I was given for data generation. I had to learn through experimentation, research and assistance from my supervisor to understand the functions of all the parts of the system, the reasoning behind them and how they all fit together to allow the system to emulate the environment, the navigation, the robot and social agents. This was very rewarding as once I understood everything, I could find a way to produce a program which tailored the system to the project's needs.

Once the training route code and the data collection code was created I could start development on all aspects of the machine learning model. PyTorch was something I had not used or understood. I had done machine learning before but never deep learning. Researching the models which I could use along with their very complex features and characteristics has been a highlight through the entire development process. Understanding and researching the aspects of social navigation was very interesting as the uses seemed to be relevant within the contemporary world of technology. With this new knowledge I discovered the existence of Convolutional Neural Networks which seemed to check all the boxes for my project. Implementing them using the PyTorch library proved challenging but the rewards outweighed the setbacks.

In working on the project, I have developed an appreciation for good coding practices. When developing I would focus on ensuring the program was functioning ignoring aspects like code readability, maintainability and efficiency. Over time I realised that if the code was structured clearly to begin with that it would prevent headaches later on in the development process. Code that was neat was easier to debug. It was also much more efficient and easier to improve in the future. By prioritising a clean code structure and better understanding the presence of code standards, I have gained a better state of mind for developing my own projects in the future.

Researching throughout the project has made me a much more independent and informed person. Reading through articles, research papers, tutorial videos and many other formats has given me the skills to better understand interesting and complex topics, where I can critically and confidently apply information to my own use case. I better understood the potential real-world applications which went beyond the scope of my project.

While I have invested many hours into this project, I have come to understand how my time management skills need to be honed and perfected. I have come to the realisation that projects that extend over a long time span need a more consistent and balanced approach. Working on the project in short intense bursts was not viable over a long period of time. Being able to balance multiple tasks, fully understanding that tasks will always take longer than expected and prioritising effective use of time is the main focus for my future projects.

Even if my project did not reliably achieve all the objectives, the knowledge, experience and realisations that I have gained through the process of developing this project will leave me with a better foundation for any future project development I may be a part of in the future.

# **8. References**

**Bibliography**

[1]

V. Jain, “Robotics for Supply Chain and Manufacturing Industries and Future It Holds! Robotics for Supply Chain and Manufacturing Industries and Future It Holds!,” *International Journal of Engineering Research & Technology (IJERT)*, vol. 8, no. 3, Mar. 2019, doi: https://doi.org/10.13140/RG.2.2.25071.07842.

[2]

C. Mavrogiannis *et al.*, “Core Challenges of Social Robot Navigation: A Survey,” *ACM Transactions on Human-Robot Interaction*, vol. 12, no. 3, pp. 1–39, Apr. 2023, doi: https://doi.org/10.1145/3583741.

[3]

S. Silva, N. Verdezoto, D. Paillacho, S. Millan-Norman, and J. David Hernández, “Online Social Robot Navigation in Indoor, Large and Crowded Environments,” 2023. Available: https://orca.cardiff.ac.uk/id/eprint/156489/7/2023\_ICRA\_Steven\_IEEEHeader.pdf

[4]

M. I. Jordan and T. M. Mitchell, “Machine learning: Trends, perspectives, and prospects,” *Science*, vol. 349, no. 6245, pp. 255–260, Jul. 2020, doi: https://doi.org/10.1126/science.aaa8415.

[5]

J. S. Gero and F. Sudweeks, *Artificial Intelligence in Design ’96*. Springer Science & Business Media, 2012.

[6]

V. De Silva, J. Roche, and A. Kondoz, “Robust Fusion of LiDAR and Wide-Angle Camera Data for Autonomous Mobile Robots,” *Sensors*, vol. 18, no. 8, p. 2730, Aug. 2018, doi: https://doi.org/10.3390/s18082730.

[7]

E Elena Torta, RH Raymond Cuijpers, and JF James Juola, “Design of a Parametric Model of Personal Space for Robotic Social Navigation,” *International journal of social robotics*, vol. 5, no. 3, pp. 357–365, May 2013, doi: https://doi.org/10.1007/s12369-013-0188-9.

[8]

“NN SVG,” *alexlenail.me*. https://alexlenail.me/NN-SVG/index.html

[9]

Y. LeCun, Y. Bengio, and G. Hinton, “Deep Learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015, doi: https://doi.org/10.1038/nature14539.

[10]

P. Addo, D. Guegan, and B. Hassani, “Credit Risk Analysis Using Machine and Deep Learning Models,” *Risks*, vol. 6, no. 2, p. 38, Apr. 2018, doi: https://doi.org/10.3390/risks6020038.

[11]

S. Loussaief and A. Abdelkrim, “Machine Learning framework for image classification,” *Advances in Science, Technology and Engineering Systems Journal*, vol. 3, no. 1, pp. 01-10, Jan. 2018, doi: https://doi.org/10.25046/aj030101.

[12]

“Deep Convolutional Neural Networks,” *www.run.ai*. https://www.run.ai/guides/deep-learning-for-computer-vision/deep-convolutional-neural-networks

[13]

*Opengenus.org*, 2024. https://iq.opengenus.org/content/images/2023/01/2023\_01\_20\_0te\_Kleki-min.png

[14]

W. Zhiqiang and L. Jun, “A Review of Object Detection Based on Convolutional Neural Network,” Jul. 2017.

[15]

M. Siam, H. Mahgoub, M. Zahran, S. Yogamani, M. Jagersand, and A. El-Sallab, “MODNet: Motion and Appearance based Moving Object Detection Network for Autonomous Driving,” Nov. 2018.

[16]

H. Seong, J. Hyun, and E. Kim, “FOSNet: An End-to-End Trainable Deep Neural Network for Scene Recognition,” *IEEE Access*, vol. 8, pp. 82066–82077, 2020, doi: https://doi.org/10.1109/access.2020.2989863.

[17]

E. Shelhamer, J. Long, and T. Darrell, “Fully Convolutional Networks for Semantic Segmentation,” 2015. Accessed: May 08, 2024. [Online]. Available: https://arxiv.org/pdf/1605.06211v1

[18]

Softbank Robotics, “Meet Pepper: The Robot Built for People | SoftBank Robotics America,” *us.softbankrobotics.com*. https://us.softbankrobotics.com/pepper

[19]

X.-T. Truong and T. D. Ngo, “Toward Socially Aware Robot Navigation in Dynamic and Crowded Environments: A Proactive Social Motion Model,” *IEEE Transactions on Automation Science and Engineering*, vol. 14, no. 4, pp. 1743–1760, Oct. 2017, doi: https://doi.org/10.1109/tase.2017.2731371.

[20]

N. Siddique, S. Paheding, C. P. Elkin, and V. Devabhaktuni, “U-Net and Its Variants for Medical Image Segmentation: A Review of Theory and Applications,” *IEEE Access*, vol. 9, pp. 82031–82057, 2021, doi: https://doi.org/10.1109/access.2021.3086020.

[21]

D. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *Computer Science*, 2014, doi: https://doi.org/10.48550/arXiv.1412.6980.