Recreational walking: Choose your own adventure path

Master thesis

Calvin Lloren Mikkelsen & Jacob William Kragh Phillips

Aalborg University
Department of Architecture, Design, and Media Technology

The Technical faculty of IT and design
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Recreational walking: Choose your own adventure path.

Theme:
Master Thesis

Project Period:
Summer Semester 2024

Project Group:
MED10 Group 10

Participant(s):
Calvin Lloren Mikkelsen
Jacob William Kragh Phillips

Supervisor(s):
Andreas Møgelmose
Anders Skaarup Johansen

Copies: 1

Page Numbers: 93

Date of Completion:
May 29, 2024

Abstract:

Navigationsapplikationer i dag fokuserer på at optimere ruten ved at finde den hurtigste vej fra punkt A til B, men de er nødvendigvis ikke til at lave rekreative ruter. Derfor har vi udviklet en løsning der kan generere ruter ud fra en persons præfenrener. Det har vi gjort ved at lave vores eget datasæt over Aalborg centrum ud af street view billeder, hvor vi har annoteret 32 forskellige klasser fordelt over 7 overclasser. Vi har brugt overklasserne til at lave en demo med henblik over skræddersyet ruter ud fra kategorier som en person kunne vælge. Derudover, har vi også lavet en machine learning model som kan kategorisere de 7 overklasser med succes over billder i Aalborg. Test-personerne kunne godt lide konceptet bag idéen ved at prøve en tilsvarende virtuel rute, og kunne se dem selv bruge det nye steder som de ikke har været før. Denne forskning er en start på hvordan man kan lave skræddersyet ruter til personer ved hjælp af machine learning, men der mangler at blive lavet en mobilaplikatiion til at bruge idéen i praktis.

Contents

Chapter 1

Introduction

"Recreational walks" refer to leisurely walks taken for enjoyment and relaxation, where there is not necessarily an ulterior goal or destination in mind [1]. The research project "Denmark in motion" (n = 163.000) found that at least 30% of the population, once per week or more, commutes to and from work by walking or cycling as their primary transportation method, with cycling being the more popular transportation method. In their spare time, 66% use walking as their primary transportation method at least once per week in, with an additional 19% occasionally opting for walking during a month [2]. It is also less popular to use cycling in their leisure time, with 41% doing it at least once per week. This indicates that people tend to use faster transportation methods for working, but prefer to do physically demanding tasks in their spare time, such as recreational walking. The study also questioned what motivated users to use walking as their primary transportation method in a specific order, which is:

1. Because I like to walk.
2. To maintain or improve my well-being (Good shape, health, etc).
3. To feel good afterward.
4. To do something good for myself.

Recreational walking often includes walking one's dog [2, 1]. The most common factors for preventing walking as a transportation method, are distance, bad weather, and a preference for motor vehicle transportation [2]. Since this was taken for the general demographic of Denmark, distance and a preference for taking the car, might be less of importance to those living in the urban environment. The research also found that it is commonly the younger demographic (15-29 of age) that engages in recreational walking, this might correlate with that the majority of young people are living in the urban environment [3] compared to other age

groups.

Digital applications, often plays an important role when doing recreational walks, this could be supporting in planning one's route, or using navigational services to go to places. The primary objective of mainstream navigation services, such as Google Maps[4], and Apple Look Around [5] revolves around optimizing routes based on time efficiency. When a user uses these popular navigation applications, various routes are proposed, where they primarily prioritize the quickest routes from point A to B, whereas other proposed routes might be tagged as less trafficked, but may be faster under certain circumstances.

In hiking applications, such as Komoot [6] and AllTrails [7], route generations considers factors like elevation, terrain type, and other relevant elements for hiking, rather than time efficiency. However, in urban areas, the terrain primarily is asphalt and flat which makes the hiking applications propose the fastest route, likewise in the mainstream navigation applications.

While, Komoot is may not be effective at urban environments, the application was analysed to see how what kind of features they used, to tailor it towards their target audience of recreational walkers, hikers, and bikers. The application can be used on its website, or as a downloadable application on Google Play Store or Apple Store. The routes generated by Komoot are defined by characteristics such as what kind of sport the user is engaging in (hiking, biking, etc). After a route is proposed, Komoot will give the user a detailed geographic insight, such as terrain type, and elevation. It is also possible to choose submitted routes by other users, and see highlights of uploaded images, that the creator has taken.

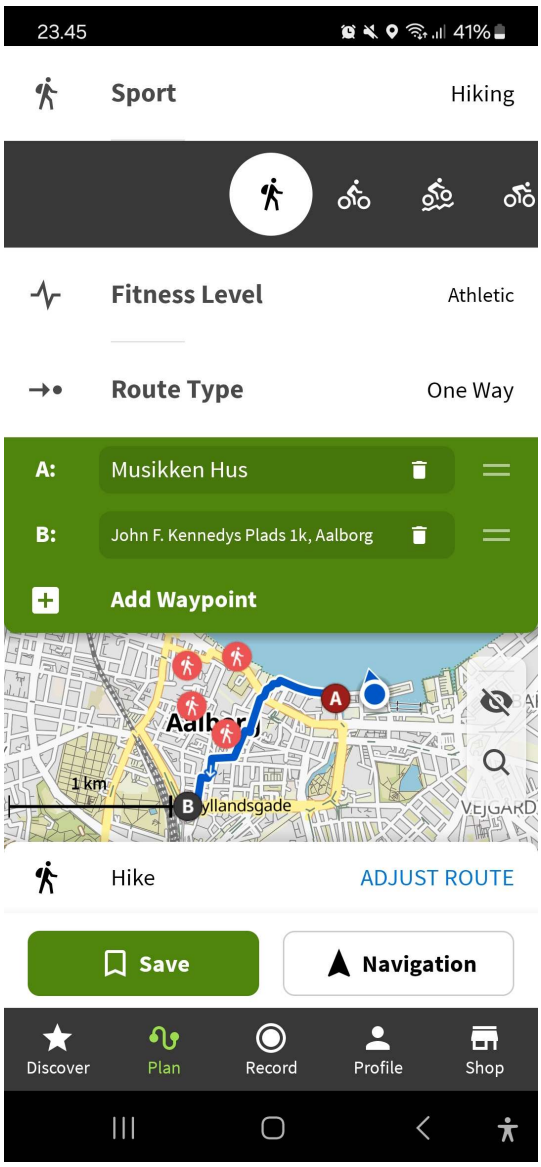


Figure 1.1: User interface of Komoot application

Figure 1.1 displays the user interface of the Komoot application. One feature is the route changes based on what different kind of sport is chosen. Another notable feature, is the option for two-way routes, which is particularly useful when incorporating multiple waypoints. This functionality allows users to plan circular routes, navigating from point A to B to C and back to A. The fitness level is simply an approximation of how fast a route is completed, as this is their defined intensity of a users' performance. These features are also found in other applications in the hiking domain, but lack customization for suited personalized routes in the urban

environment domain.

1.1 Related Works

Many researchers have investigated the impact that the environment can have, on walking preferences. For example, dog owners might favour greener environments, "*the natural environment plays a significant role in encouraging residents to walk*" [8]. This indicates that personal preferences and the purpose of walking, such as recreational walking, can be influenced by the quality of the environment and the specific features of the walking route, and that the circumstances can have an indirect effect on one's walking preferences.

Yan et al. [8] noted that "*residents are more concerned with the physical environment of community streets*" and these environments should include "*well-developed infrastructure, such as lighting, guardrails, seating, and paved ground" to encourage walking*". Yan et al. also investigated what other aspects could influence the preferences and what people had a preference for. From this Aesthetic Quality was discovered, Aesthetic Quality by their definition refers to the concept that people are more drawn to aesthetically pleasing environments. They noted that "the environmental quality of physical and aesthetic aspects has an impact on residents' choices, and the aesthetic environment quality has a stronger impact". The Aesthetic qualities that had most impact were Urban Greenways and Waterfront Paths. Urban Greenways are green elements such as trees, parks, and Waterfronts Paths are area of water bodies, such as a lake. These factors could enhance the appeal of walking routes, and therefore becoming a more popular choice. [8].

1.1.1 Pedestrian Walking Preferences

Understanding the factors that influence pedestrian walking preferences is crucial for developing effective and user-friendly navigation applications. Recent studies have delved into various aspects that affect how pedestrians choose their routes [9, 10, 11, 12]. One such study by Urmi et al. [11] explored these preferences in depth, focusing on how personalized navigation systems can cater to individual needs and priorities. They identified similar results in the aspect of user preferences for walking routes. Here they looked at personalized navigation system for pedestrians, where they investigated the importance of user-based preferences when creating routes from point A to point B.

They identified seven walking preferences namely; Safety, Tactile paving, Leisure

spots, Residential neighbourhood, low traffic, straightforwardness and step-free access. Their implementation focuses particularly on the aspect of Safety, with an emphasis on female pedestrians, who have to traverse longer distances at night-times [11].

- **Safety Priority**: Pedestrians, especially women walking alone at night, seek alternative routes that offer a sense of security and protection
- **Tactile Paving**: tactile indicators to provide feedback, aiding individuals who are blind or visually impaired with
- **Leisure Spots**: Preferred destinations among tourists and recreational walkers, including parks, shopping centres, and tourist attractions.
- **Preferences for residential neighbourhoods**: Routes passing through residential areas offering typically quieter and less crowded paths compared to industrial or commercial zones.
- **Low Traffic Volume**: Preference for routes with minimal traffic noise, and emissions for a healthier walking experience
- **Straightforwardness**: Defined by routes with fewer crossings and turns reducing the complexity of a path, which could be favoured by elderly people or joggers.
- **Inclusion of step-free access**: Routes that accommodate individuals needing barrier-free access such as wheelchair users, pushing strollers or carrying larger luggage.

Here, a modified version of the Dijkstra algorithm is employed to generate the personalized route based on the seven walking preferences. They highlight the need to include information about these walking preferences for a specific route. Such as the display of information on crime rate, and route complexity [11].

Quercia et al. proposed creating walking routes based on user-based preferences as well, with an emphasis on creating happy, quiet, and beautiful routes by using Flickr images and its metadata, which consisted of votes of such categories [9]. The research has shown to work with the proposed routes to some degree, however they encountered some shortcomings worth mentioning; There was no picture control on the Flickr images, and uploaded images may not have an accurate representation of its urban environment, and emphasized the need for software to determine the content of image "*Thus, it might be useful to use existing techniques (e.g., computer vision algorithms) to determine the extent to which a picture represents its associated urban location.*"[9] Another, important finding is the importance of personalization when generating routes, this is due to the perception of happiness

may differ from one individual to another based on personal preferences and interest like shopping, hiking or nature.

Golledge et al. explored pedestrian path selection, by asking participants to rank different criteria for route preferences, ranking from a scale from 1-10 with 1 being the top priority. The results showed that the shortest distances was the most important criterion, followed by least amount of time spent. Scenic/aesthetic aspect ranked in fourth place, suggesting that while the participant prioritized efficiency in terms of temporal aspect and distance, the aesthetic scenery is also taken into consideration [10].

Wakamiya et al. looked into how one might create pleasant walking routes from a given start and end point. Here, they focus on the routes with higher amounts of greenery and brightness by doing analysis based on the colour and object information extracted from Google Street View panoramic images along the road. They would give a score on the images based on the green pixels, and the presence of tree/parks. Based on this approach, they are able to create custom walking routes that favours more greenery routes [13].

Chapter 2

Problem Formulation

To the best of our knowledge as of May 2024, there is still a lack of comprehensive solutions that integrates multiple walking preferences which aims to create personalized walking routes in the urban environment, both in the commercial aspect and academically. Existing state-of-the-art solutions often focus on specific aspects such as safety, scenic value, or efficiency. But fail to consider the diverse range of user preferences in urban environments. Larger cities often include a diverse range of attractions, that could be utilized. The data used are often meta-data gathered from online services such as Flickr, where images were shown to not always be an accurate representation of the urban environment. Moreover, these solutions do not sufficiently leverage the potential of computer vision or machine learning for contextual understanding of what an image represents, this could be used to classify and evaluate walking preferences for route generation.

From this, we finalize the problem formulation with:

How can we leverage machine learning methods to develop a navigational digital artifact that creates personalized walking routes in urban environments, taking into account a diverse range of user preferences.

Chapter 3

Background information

3.1 Multi-label image classification

In any given image, multiple objects may be present. While these images can still be used in binary and multi-class classification, these methods can only predict one object in a given image. Multi-label classification, however, can handle multiple objects by using a combination of binary and multi-class classification. For each label it has been tasked to predict, multi-label classification performs a binary classification to predict whether a given label is present in the image. If the confidence of a predicted label exceeds an estimated probability, the multi-label classification method will predict that the object is present in the image.[14] An example of this concept can be seen in Figure 3.1.

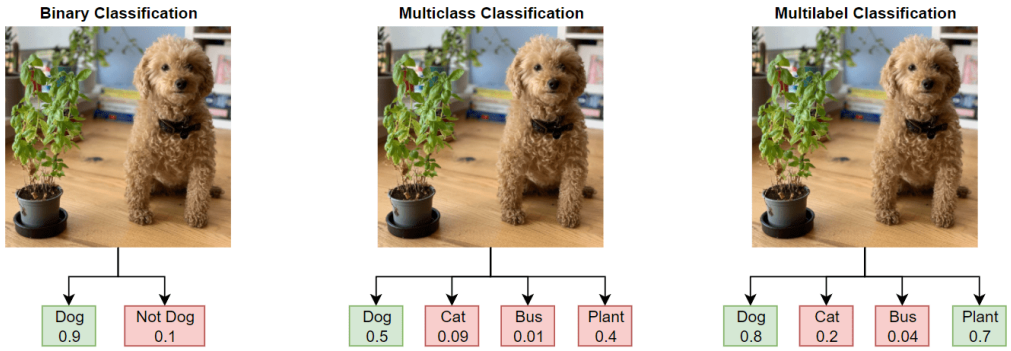


Figure 3.1: Different kinds of classification[15]

Here, it can be seen that in binary classification it predicts dog, in multi-class classification it is both detecting features of dog and plant, but the dog has the highest confidence score, so it predicts dog. In the aforementioned methods, it is also important to notice that the confidence score has a sum of one. In the multi-label classification, it can be seen that every label possible label is predicted

as a binary classification, and making it possible to predict both dog and plant, by having a confidence score of 0.8 and 0.7 respectively. For this scenario, the confidence threshold can vary between 0.2 and 0.7, however it is commonly set at 0.5 as this makes it a “true” binary classification by having equal thresholds for being detected or not.

3.2 Convolutional neural network

Convolutional neural network (CNN) is a popular deep neural network method in relation to the appliance for computer vision tasks since its first adoption of LeNet-5 in 1998 for recognizing handwritten digits[16]. Today more sophisticated CNN architectures exist like ResNet[17] which architecture has been extended upon for more advanced tasks such as object detection(Faster R-CNN[18]), instance segmentation (Mask R-CNN[19]) and panoptic segmentation [20]. Additionally, ResNet can also be used for classical computer vision problems such as binary classification, multi-class classification [17] and multi-label classification. [21]

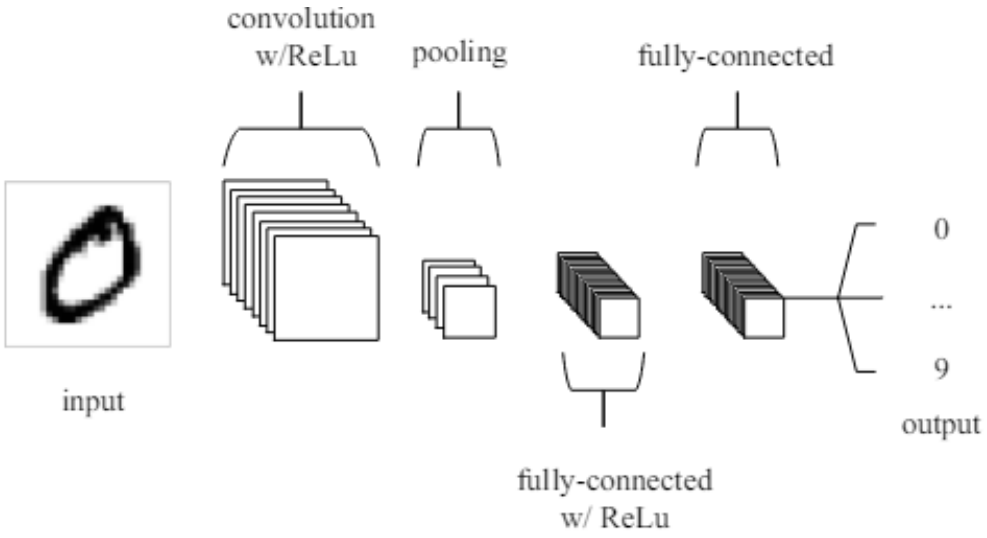


Figure 3.2: Simple five layer CNN architecture(MNIST classification) [22]

A typical CNN can be divided up into 3 different segments: an input layer, a hidden layer, and an output layer.[22] The input layer is where the "raw" data is feed through, such as an image, video, etc. The hidden layer consist of Convolutional layers followed by an activation function, pooling layers and the fully connected layer. The hidden layer consists of an arbitrary number of convolutional layers, and there may be other operatisations such as pooling layers between or after the convolutional layers. The goal of these convolutional layers is to learn features

simple features in the early layers such as simple patterns, edges and textures, to more abstract features in the last layers. These features are then fed into the fully connected layer. The fully connected layer consists of a traditional fully connected neural network, meaning all neurons in one layer, are connected to all neurons in the next layer. The fully connected neural network creates a relation between the features extracted from the Convolutional layers and the output layer. The output layer outputs a score for every class with their own probability.

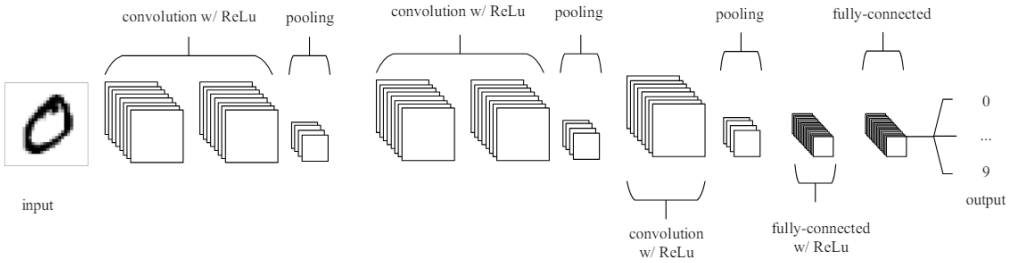


Figure 3.3: traditional form of CNN architecture (MNIST classification) [22]

In figure 3.3 it can be seen how the different layers are present in a traditional CNN. The input layer, is a sample of a handwritten digit. The hidden layer consist of 3 convolutional layers, with a pooling layer between all the convolutional layers, and a pooling layer after the final convolution layer. Afterwards, it is feed into the fully-connected layer with ReLU before being feed forwarded to the last fully connected layer. Lastly this is forwarded to the output layer which will predict a number between 0-9. Now that we have a good understanding of how CNN works, we will explore how the different methods in the hidden layer are used in-depth.

3.2.1 Convolution

Convolution is a concept used for a CNN to extract features from images by applying kernel filters in the convolution process. This process involves a sliding operation across the entire input, to create a new feature map. Kernels can vary in size of $k_h \, x \, k_w$, but it is common for the height and width to be equal and odd dimensions. [23]

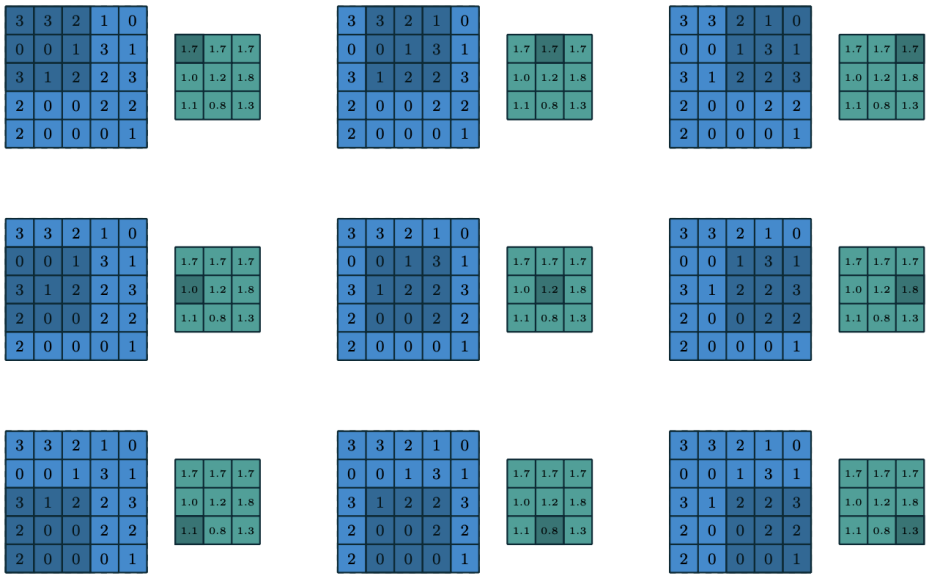


Figure 3.4: Convolution operation [23]

Figure 3.4 illustrates how the convolution operation is applied, where a kernel is applied to an input image to generate a new feature map. Multiple kernels can be used to create multiple feature maps, each referred to as an output channel. For instance, if a convolutional layer has 64 out channels, it means 64 different kernels has been applied to create 64 new feature maps. These new feature maps are then feed forwarded to subsequent convolutional layers, where additional kernel filters are applied to the previous feature maps in order to create more feature maps for detecting complexed features.

Each kernel detects different features within a given input. However, for simplicity, it can be shown how it is possible to use filters for edge detection.

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{3.1}$$

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{3.2}$$

Equation 3.1 represents a filter detecting horizontal edges, and equation 3.2 for detecting vertical edges.

3.2.2 Stride and padding

Stride is defined as the step size over a convolution filter, and padding is refereed to what kind of operation is completed for the edges of an input feature map. Stride and padding are hyperparameters for a convolution filter, and adjusted to a desired use case. Typically, these hyperparameters are set in a way that the input feature map dimensions are maintained, essentially allowing the new kernels to be applied without altering the output size of the new feature maps. This is also referred to as same padding.

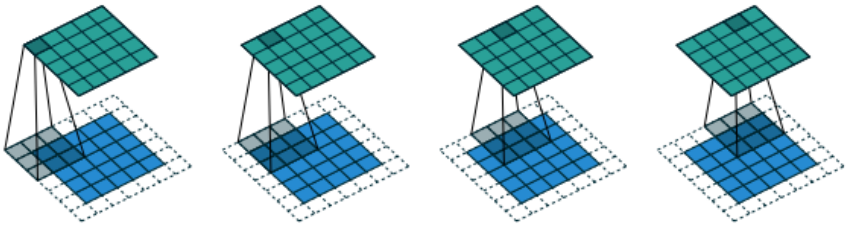


Figure 3.5: Stride and padding[23]

For instance, using a padding of 1, with a stride of 1 will ensure that the input feature map and output feature map will have the same spatial dimensions, if a kernel of 3×3 is applied. This can be seen in Figure 3.5, where the transparent squares are the padded inputs.For padding, it is most commonly known to use zero-padding.
Valid Padding is where no padding is applied to a given input feature map. This results in a reduction of the output feature map if the same scenario as previously mentioned is applied, however a pooling layer for this scenario if often favoured.

3.2.3 Pooling

Pooling layers is another commonly used method throughout CNN architectures. The purpose of a pooling layer is to reduce the size of feature maps by using a pooling window. This pooling window slides through the entire feature input, by applying a pooling function to each sub-region of the input. [23] The most commonly used functions are max-pooling and average pooling. Max-pooling takes the max value of a given sub-region, while average pooling calculates the average for the output. The sliding window is defined as x, y, but is most commonly, a square (e.g., 2x2 or 3x3). This can be combined with stride in a cleaver way, so it reduces the dimensions of the input features.

$$\text{Average pool} \begin{bmatrix} 2 & 8 & 8 & 9 \\ 6 & 3 & 5 & 4 \\ 4 & 7 & 9 & 7 \\ 1 & 4 & 8 & 8 \end{bmatrix} \rightarrow \begin{bmatrix} 5 & 6 \\ 4 & 8 \end{bmatrix} \tag{3.3}$$

In Equation 3.3 an example of an average pool can be seen, Here the subregions are colour-coded simulating a sliding window of 2x2.

$$\text{Max pool} \begin{bmatrix} 2 & 8 & 8 & 9 \\ 6 & 3 & 5 & 4 \\ 4 & 7 & 9 & 7 \\ 1 & 4 & 8 & 8 \end{bmatrix} \rightarrow \begin{bmatrix} 8 & 9 \\ 7 & 9 \end{bmatrix} \tag{3.4}$$

In Equation 3.3 an example of average pool can be seen. The subregions are coloured, simulating a sliding window of 2x2.

3.2.4 Activation functions

After specific layers, such as the convolution layers, fully connected layers, or the output layer, it is crucial to use an activation function. Activation function determines whether a neuron should be activated or not.[24] There are many activation functions, used for different use cases, but two of the most popular ones are Reflected Linear Unit (ReLU) or the sigmoid activation function. These activation functions are non-linear which add a new level of complexity to a machine learning model, by adding non-linearly to the machine learning problem, which enables the model to learn complex to learn complex patterns. [24]

$$f(x) = \max(0, x) \tag{3.5}$$

Equation 3.5 is the formula of ReLU, where if $f(x)$ is a negative value it will be equal to 0 else $f(x)$ will be equal to x. This is illustrated in the graph in figure 3.5.

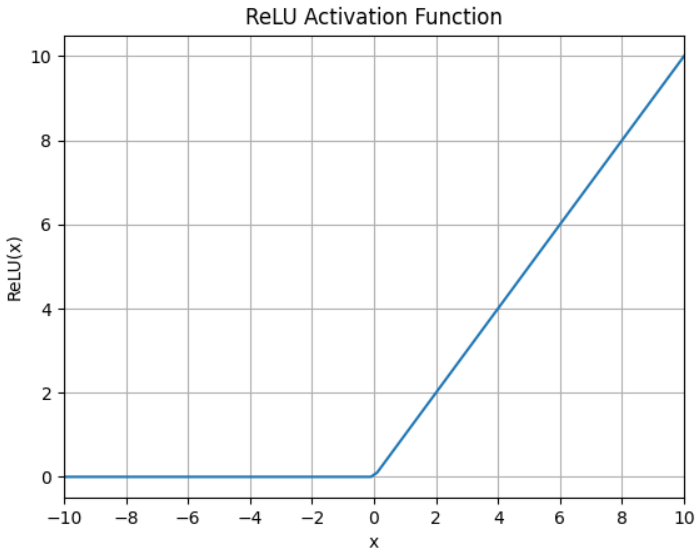


Figure 3.6: ReLU activation function

Sigmoid activation function takes an input value and outputs a value between 0 and 1.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.6}$$

Sigmoid activation function can be seen in equation 3.6. The larger the $f(x)$ is for positive numbers, the closer it will be to 1, while the lower the value is for negative numbers, the closer is it will be 0. If $f(x) = 0$, x will be equal to 0.5. Figure 3.7, shows a graph of the sigmoid activation function, where it is possible to see a correlation between an input value and the sigmoid activation function.

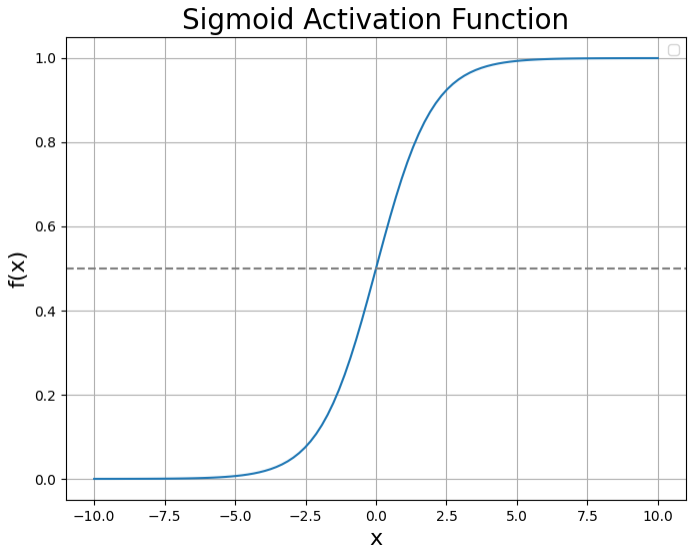


Figure 3.7: Sigmoid activation function

3.3 Training a neural network

3.3.1 Processing data

In relation to training a neural network, data has to be processed before being fed into a neural network. In the computer vision domain, it is important to have a metadata file that consist of image path and annotations, in order for a data loader to extract relevant information related to an image. A complete dataset if often split up into training validation and test set that has each their own purpose [25]. The training set is solely used for training the Neural network, where validation is used to track the process and ensure the model is learning throughout the training with generalization and to prevent overfitting, by analysing the training process and hyperparameter tuning. Lastly, then training is completed, the model is evaluated through the test set to evaluate the model. In order for the different sets to be compatible with the neural network architecture, preprocessing steps needs to be applied, such as resizing of images, or normalization. Specifically for the training set, it is also common to use augmentations techniques, horizontal flip, brightness, colour adjustment or other common computer vision techniques in order to create "new" unseen samples, as augmentations has shown to improve a models'generalization.[24, 26]

Figure 3.8: Example of augmentation[27]

These preprocessing and augmentations steps can be seamlessly incorporated using the PyTorch library and the build in data loader [28].

3.3.2 Batch and epoch

Images require tons of memory, therefore it is nearly impossible to feed the entire dataset in one iteration. To solve this, the concept of batch is used. A batch is a subset of a dataset, that is forwarded through the neural network before a model's internal parameters are updated. Batch size can be any, arbitrary size, but in general lower batch sizes seems to perform better because of training stability and generalization performance. [29] However, a larger batch size will speed up the training process, as the model's internal parameters are not updated as frequently. Epoch, is when each subset of the dataset is forwarded through the network, meaning it represent a full cycle of the entire dataset. The number of epochs is also a hyperparameter, and the ideal size depends on the problem and detailed analysis of the training process.

3.3.3 Forward propagation

Forward propagation refers to the calculation, and storage of intermediate variables from the input layer to the final output layer. This happens sequentially, meaning each layer process the input, and passes it through the next layer, and intermediate variables are stored in that order.[24] These variables are important to track for a neural network, in order to adjust them in Backpropagation.

3.3.4 Loss functions

The goal of a loss functions is to measure how close a model's predictions are to the true labels. The lower the loss is, the more accurate the predictions. For classification tasks, it is common to use Cross-Entropy Loss as the loss function [24]. Here, as the problem is multi-label classification, we can use Binary Cross Entropy Loss, as the problem can be classified as a series of binary classification problem as described in Section 3.1, and this approach has been used in similar problems for multi-label classification [30, 14]. This loss function operates under the premise that each category is a separate binary decision—essentially a 'True' or 'False' to whether a category is predicted or not in a sample. The model's predicted probability for each category's presence is compared against the actual label, with the binary cross-entropy formula[31] applied to each category independently:

$$\text{BCE Loss } = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \cdot \log\left(p_i\right) + \left(1 - y_i\right) \cdot \log\left(1 - p_i\right)\right] \tag{3.7}$$

Here, (N) represents the total number of instances, (y_i) is the actual label, and (p_i) is the model's predicted probability for the instance belonging to the respective category. Essentially, punishing the model for incorrect predictions by outputting a high loss value for incorrect predictions of a label, and a low loss value for correct predictions. The loss is computed for each category and then summed for a total loss, providing an approximation of the model's performance across all categories for an instance.

3.3.5 Backpropagation

Backpropagation is a method to calculate the gradient of the loss function in relation to the parameters of a neural network parameters, by traversing in reverse order, from output to the input layer. This is accomplished by the chain rule of calculus. The calculated gradients indicate how much the loss would increase or decrease if a particular parameter is adjusted. [32] This is crucial, in order to adjust weights and biases with the ultimate goal to decrease the overall loss.

3.3.6 Optimizers

Optimizers are used to adjust the gradient throughout the network to improve the loss during backpropagation. The most simple approach, is by using gradient descent.It systematically adjusts the model's parameters in the direction that reduces the loss function. The step taken by gradient descent is defined by a learning rate, the bigger the learning rate, the bigger the step. However, a big learning rate might overshoot the local and global minima. A too low learning rate, convergence will be slow and ineffective, furthermore it can be prone to the gradient descent to be stuck in local minima as it is unable to escape it.

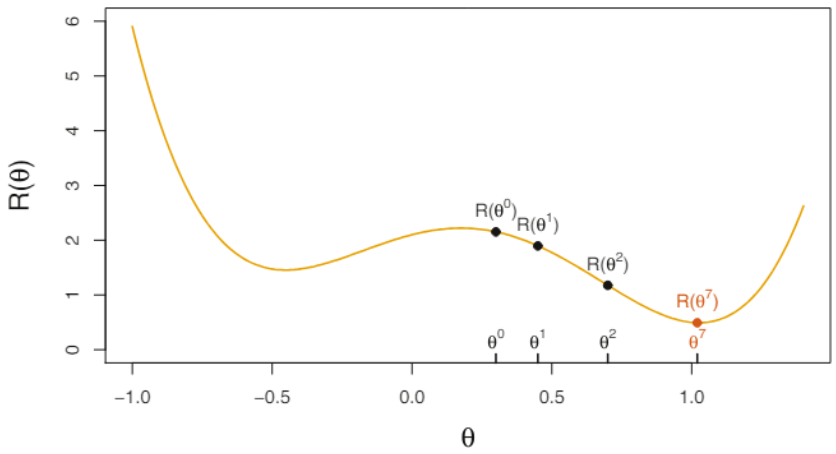


Figure 3.9: Illustration of gradient descent for one-dimensional θ. The objective function R(θ) is not convex, and with two minima. [32]

Figure 3.9 illustrates the gradient descent process, showcasing a graph with two distinct minima. The local minimum is defined as a point where the function's value is lower than that of any immediate neighbouring points, and for this graph, it occurs at θ = -0.46. The global minima are the absolute lowest point of a given function, and in this case, θ = 1.02. The primary objective of optimization algorithms, such as gradient descent, is to locate this global minimum, this is equal to the minimum possible loss. The objective function, (loss function) traverse between the graph in order to find the global minima with 7 steps. [32] This gradient descent is also known as batch gradient descent or vanilla gradient descent. However, this method is not feasible for computer vision tasks, as it require loading the entire dataset into memory.[33]
One way to come out of this problem is by using Stochastic Gradient Descent (SGD) optimization algorithm. This updates the weights and parameters after each sample, however as SGD performs updates in a high frequency, it causes high variance for the objective function, leading to an unstable convergence. Therefore, Mini-Batch Gradient Descent is used in practice, it is essentially SGD, but instead of 1

sample at the time, it will feed a n number of samples at the time, also referred to as the mini-batch which is the same concept as the batch as mentioned in Section 3.3.2. This leads to a reduction of variance in the parameter update, and a more stable convergence.[33]
Adaptive Moment Estimation (Adam) optimization, [34], incorporates the concepts of momentum, and Root Mean Square Propagation (RMSProp). Momentum is a technique, that accelerate the direction the gradient is pointing if it is consistent over a period of steps, allowing for larger steps towards minima. RMSProp adapts the learning rate for each parameter based on the gradients' magnitudes. Parameters with high gradients receive smaller updates to prevent overshooting the minima, while parameters with low gradients are allowed larger updates. This adaptive learning rate, coupled with momentum, ensures that Adam does not overshoot the minima. This results in Adam having a faster convergence compared to other optimization algorithms such as SGD, or Mini-Batch Gradient Descent, essentially requiring less training time to reach minima. However, it is not guaranteed to find sufficient minima that enable a model to be good at generalization, and it is through here analysis and hyperparameter tuning, is used to find suitable minima for generalization. While, Adam also is a more accurate through the early stages of training, it is shown that SGD can outperform Adam optimizer through longer training sessions. [35] However, Adam was chosen as the optimizer, as this would require less training to achieve noticeable results.

3.4 ResNet

ResNet is a proposed machine learning architecture to train deep neural networks, while solving the vanishing gradient decent problem of previously proposed networks. It does this by using a concept of identity mapping.[17] Identity mapping is where a proposed layer is duplicated before being fed through multiple convolution operations and is then added together with the duplicated layer, and is the core operation of a Resent block. Only the new features being are being added to the final layer of a ResNet block.

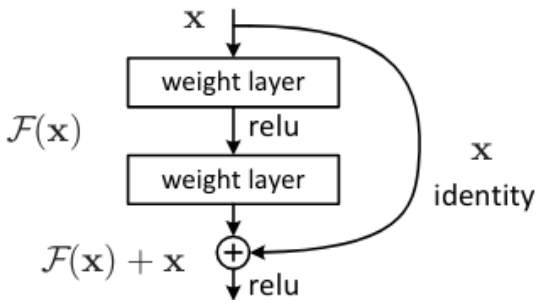


Figure 3.10: Concept of ResNet Block and identity mapping

In Figure 3.10 it can be seen how a ResNet block is constructed, where it has $F(x)$ that is fed through the convolution operation, and the identity x that is added after all the convolution operations in the end, followed by the ReLU activation function.

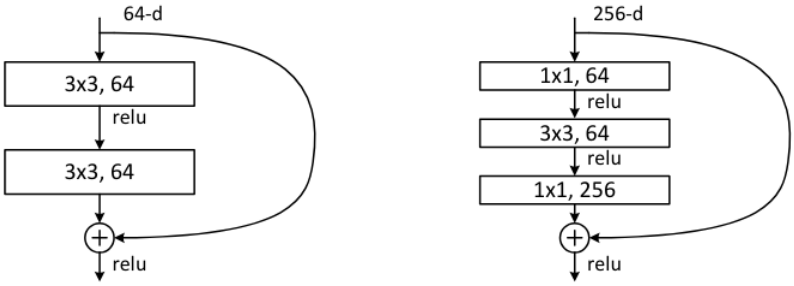


Figure 3.11: ResNet block and bottleneck block

In Figure 3.11, there are two different kinds of blocks. It can also be seen how the identity map is the same dimensions as the last convolution operation of a particular block. The ResNet block and the bottleneck block are also slightly different, where the bottleneck blocks has different kernel filters for convolution operations, and there are 3 layers in total. This bottleneck block is used for ResNets with 50 or more layers in total.[17]

3.4.1 Customizing ResNet for multi-label image classification

In order to use ResNet for multi-label image classification, it needs to be modified. In the original model proposed by He et al. after all the ResNet blocks, it is followed by a fully connected layer and output layer. [17] The output layer consist of a soft-max activation function, but this is not optimal for multi-label classification, as soft-max sums up all the outputs to 1, by normalizing the values. For a more optimized solution, the output activation function is changed to sigmoid instead as this outputs every class between 0-1, where a threshold hyperparameter is defined afterwards for deciding if a class detected or not. The sigmoid activation function

for the output layer has been an approach that works well with mutli-label classification problems. [30, 36] Furthermore, the output layer for the model is also modified so it is customized to the amount of classes for the proposed solution of the project.
As this is a multi-label classification there are also changes to the input label tensor so it follows is one-hot encoding format.

$$\text{Classes}: \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \\ \vdots \end{bmatrix} \rightarrow \text{Threshold}(\mathbf{x}) = \begin{cases} 0 & \text{if } x_i < 0.5 \\ 1 & \text{if } x_i \geqslant 0.5 \end{cases} \rightarrow \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \\ \vdots \end{bmatrix}$$

In equation 3.4.1, it can be seen how a one-hot encoding input is outputted. It is important to mention that, this is after the sigmoid output layer, so, the classes x_n is a value of the final sigmoid output. Binary Cross Entropy Loss is the loss function, as this is suitable for multi-label classification as mentioned to in Section 3.3.4.

3.5 Datasets for the urban environment

Large datasets such as COCO [25] and CityScapes[37] are commonly used for various computer vision tasks revolving around the urban environment. COCO contains 80 classes in total, but not all of these classes are relevant for appliances in the Urban environment. Meanwhile, CityScapes has 30 classes all suited for the urban environment. However, it is mainly used for traffic situations and understanding relevant to self-driving cars. For instance, the "buildings class" contains all kinds of buildings including stores, stadiums, museums, train stations, etc.
Considering the specific needs of creating personalized routes, based on preferences, it was determined that COCO nor CityScapes could solve this problem. Therefore, the decision was made to develop a custom dataset tailored specifically to urban environment analysis relevant to recreational walkers. This approach also offers greater flexibility and control to assist in solving the problem of proposing personalized routes, for example the choice of classes. The construction of the dataset can be found in Chapter 4.

3.6 Slippy map: A Web-based cartograph

Web-based cartography is the process of displaying geospatial data through interactive maps on the World Wide Web, examples of such could be Google Maps,

Microsoft Bing Maps and OpenStreetMap. These distributed maps rely on a tile-scheme system, which translates maps' positions on Earth into a two-dimensional surface into a series of tiling grids. Each tile can represent various data types including images, vectors or other geospatial data.

In Figure 3.12 we illustrate how these grid spaces work. The top section displays the coordinates Z, X and Y. Z denotes the zoom level; a larger Z value returns a more detailed map. Meanwhile, X and Y denote columns and rows. It is essential to highlight that we are using a non-zero indexing in this illustration. Some explanations may use zero-indexed, so the X and Y would start on (zoom, 0,0) [38, 39].

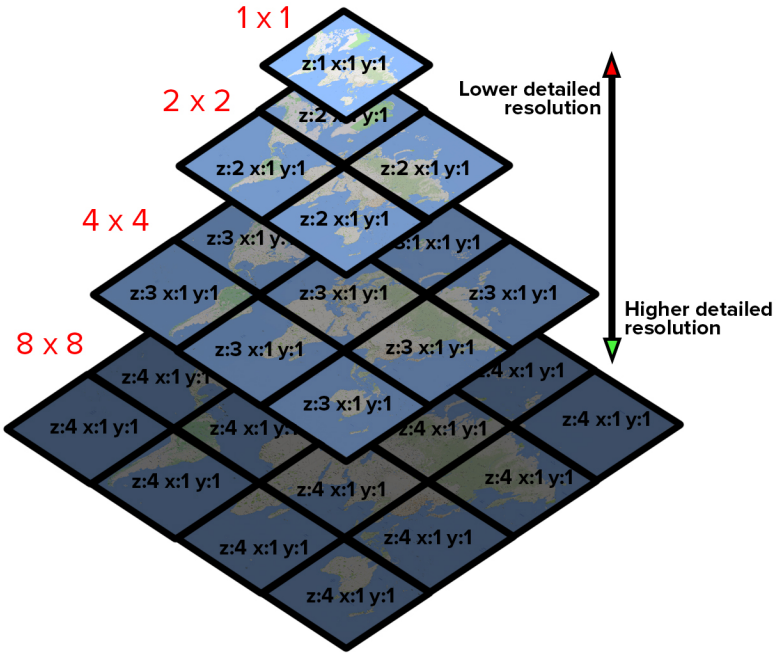


Figure 3.12: Example of a dynamic Tiling Scheme

Google Maps utilizes a tiling scheme where the zoom level of one divides the representation of the entire world into four distinct tiles (2 x 2) each measuring 256 x 256 pixels. The zoom level of two will display the entire world into 16 tiles (4 x 4) likewise measuring 256 x 256 pixels, and this pattern continues exponentially with each successive power of four, denoted as 4^n, where n^n is the zoom level [40].

3.7 Introduction to Graphs

This section is based on the book *Discrete Mathematics and Its Applications* Chapter 10 and 11 by Kenneth H. Rosen [41]

Graphs are a perfect example of how one might map the relation between entities, for example this could be the problem of finding the shortest path between two given destinations. Here, a weighted graph could be utilized, where the weighted cost could represent the distance between each vertex.

A graph can be explained as a set of vertices (also called nodes) and the edges that link these vertices (singular vertex) together. Mathematically, we can represent graph G a pair of a set V and E, denoted as $G = (V, E)$ here:

- V is a non-empty set of vertices.
- E is a set of edges, where each edge represents the link between a pair of vertices

We remark that a set of vertices of a graph G may contain infinite vertices, and thus a can contain an infinite number of edges between the vertices. The graph also contains no rules that dictate rules for connection among the vertices. Graphs can be classified into different types based on the properties of the graph, in this context we will focus on the directed, undirected, weighted and the unweighted properties.

3.7.1 Directed graph

A directed graph also known as a digraph, is a graph in which the edges have a point of direction. These directions are non-bidirectional unidirectional edges to other vertices in the graph, these directed edges are associated with an ordered pair of vertices. This can be denoted as (u, v) where these two vertices are in a sequential order, here u is the first vertex and u is the second vertex.

For a directed graph we can reuse the notation used previous G = (V, E), but where the edges E is an ordered pair of vertices represented as (u, v). Here (u, v) is said to start at U and end at V.

3.7.2 Undirected graphs

Undirected graphs compared to directional have bidirectional edges, this means that the edges do not contain directional edges, you can travel between the two connected vertices in either direction with the same edge. It is possible for a graph

to be consisting of both directed and undirected in the same graph, and this is called a mixed graph.

3.7.3 Weighted vs. Unweighted Graphs

In a weighted graph, edges has an associated value (weight or cost) with it. Here, such weights can represent length, cost. These weighted graphs are typically used with when encountering the shortest paths problems. Unweighted graphs are the opposite of weighted graphs, here there is no additional weight for an edge between the vertices.

3.7.4 Graph terminology

Here, we define some terminology related to graphs theory

- **Neighbors** can be defined as two vertices (u, v) are connected by an edge. So seen in Figure **??** vertex V1 and V2 are neighbours, as they have an edge connecting them, this can also be called adjacent vertices.
- **Degree** is the number of edges connected to a given vertex.
- **Path** is a sequence of vertices connected by edges, a path length is the number of edges in a path.
- **Cycle** is a path that starts and ends at the same vertex.
- **Connectivity** referees to connecting between two vertices, and that two vertex are connected if a path exists between them, and a graph is said to be connected if all the vertices are connected.
- **Circuit** refers to a path that begins and ends with the same vertex.

3.7.5 Tree

A tree falls under the subset of an undirected graph with no cycles, where each tree has N vertices N-1 edges, where there is a distinct path between any two vertices. The edges of a tree are also referred to as branches, and despite the name, a tree does not need to branch out from bottom to top.

3.8 Finding the shortest path

Several algorithms have been used to attack the problem with finding the shortest path between two given vertices. Here Dijkstra's algorithm has been extensively

used, or derivates of it, in creating navigational systems [42, 43, 44, 45, 13].

Wakamiya et al. employed Dijkstra's algorithm to determine the shortest path within a weighted graph. Here they changed the weighting of the edges to all adjacent vertices based colour based ranking and object based ranking. [13].

Shah et al. used another popular search algorithm namely A*, here they used both Dijkstra and A* to generate the shortest in respect for safety. A safety index was calculated based on crime rate data. Here they noted that while A* algorithm performed better in regard to finding the shortest route, Dijkstra was suited better for their current system, even though their implementation of Dijkstra suggested longer routes, the safety with Dijkstra resulted in a higher safety index. [45].

Zaoad et al. Similarly used A* for urban route safety, here A* was used together with recent crime data, to adjust the weighting on the edges between each adjacent vertices. Here they had positive results with using A* [46].

3.8.1 Dijkstra's algorithm

When it comes to finding the shortest path, Dijkstra's algorithm is frequently regarded as being one of the best. It finds the shortest path between a starting vertex and an ending vertex in a non-negative weighted graph. The algorithm maintains two sets of vertices, visited and unvisited, and tracks the total cost from the starting vertex to end vertex. For a deeper insight into the Djikstra algorithm, I refer to the original paper: *A note on two problems in connexion with graphs* [47].

1. Starts the search from the start vertex, and set the distance cost value to zero.
2. For the current vertex, check adjacent vertices, and calculate the cumulative cost to traverse to.
3. If the cost is lower than the recorded distance for the adjacent vertex, then update the distance cost.
4. Move the current vertex to the visited set.
5. Select the new unvisited vertex based on the lowest recorded distance as the new current vertex to start from.
6. Repeat this pattern until all vertices in the graph have been visited, and then backtrack the costs value to find the shortest path.

3.8.2 Path A*

A* is another popular pathfinding solution, like Dijkstra's it aims to find the shortest path between two given vertices in a weighted graph. It does this by combining both a "greedy" method like Dijkstra's and a heuristics approach for selecting vertices to traverse to. For a more in-depth explanation of the A* algorithm, I refer the readers to: "*A Formal Basis for the Heuristic Determination of Minimum Cost Paths*" [48].

At its core, the A* evaluates vertices based on a cost metric, the cost from the starting vertex to the current vertex, this can be denoted as:

$$f(n) = g(n) + h(n)$$

where n is the next vertex on the path, g(n) is the cost of the starting vertex to the current vertex n and h(n) is the heuristic estimation of the cost from the current vertex to the goal. F(n) is the total cost of g(n) + h(n)

Like Dijkstra the A* maintains two sets, here we will call them an open-set and a closed-set, the open-set are the set of vertices that needs to be evaluated, and the closed-set are the vertices that have already been evaluated. Explain the Algorithm goes :

1. **Initialize starting vertex**
    - Set the starting vertex to the open-set, and initialize the cost to zero.
2. **Evaluate adjacent vertices**
    - Select the vertex from the open-set with the lowest f-value.
    - Move the selected vertex into the closed-set, and Calculate the g-value for each adjacent vertices for the selected vertex.
3. **Update Open Set**
    - If the adjacent is not in the open set, add it to the open set.
    - If the adjacent is already in the open-set and the newly calculated g-value is lower than the current g-value, then update its current g-value and set its parent to the current vertex.
4. **Calculate Heuristic**
    - For each adjacent added to the open set, calculate the h-value using a heuristic function. The choice of heuristic function depends on the specific problem and should estimate the cost to reach the goal from the adjacent.

5. **Repeat**

    - Repeat with selecting vertices and updating the open Set until either the open Set is empty or the current vertex found is equal with the goal vertex.

```
1    while openSet is not empty
2        current := the node in openSet having the lowest fScore
3        if current = goal
4            return := reconstructed path
5
6        openSet.Remove(current)
7        for each neighbor of current
8            tentative_gScore := gScore[current] + d(current, neighbor)
9            if tentative_gScore < gScore[neighbor]
10               cameFrom[neighbor] := current
11               gScore[neighbor] := tentative_gScore
12               fScore[neighbor] := tentative_gScore + h(neighbor)
```

Listing 1: pseudocode for A*, here we focus on the most important aspect of the algorithm, where the neighbour selection is calculated

In their analysis, Rachmawati et al. compared both Dijkstra and A* algorithm. They observed that essentially both algorithm would return the same output to a smaller graph. A* would outperform Dijkstra in terms of speed [49] on larger graphs. This performance difference is attributed due to A*'s heuristic-driven approach, which guides the search towards the destination, as opposed to Dijkstra's systematic exploration in all given directions. As a consequence, Dijkstra may end up exploring larger areas before identifying the shortest path, leading to slower performance when compared against A*.

3.9 OpenStreetMap: Navigational data

OpenStreetMap (OSM) is a collaborative project that aims to create a free and open-source mapping software of the world by collecting and organizing geographic data contributed by crowdsourcing. They provide free map images and the underlying map data. These data points can be used in conjunction with things like: road data, nearby shops, public toilet, walking path, road network and more [50]. Because of this free and open-source, multiple studies have included data from it [11, 51, 13, 12].

The OSM data format is based on three basic elements [50]:

- **Nodes**: These represent a single point and are defined by its latitude, longitude, and node ID. These are typically used to define a "path" of a way, However nodes are not limited to this purpose, for example a node can be tagged with "amenity=cafe"
- **Ways**: A way is an ordered list of nodes that represents linear features such as a road, river, or wall
- **Relations**: Is a multipurpose data structure which links the relationship between other data elements (ways, nodes and/or other relations).

Each of these elements can have other associated tags, which can provide additional information about the features of the given element, such as its name, type (e.g., highway, building, amenity), and other properties.

```
1  <osm version="0.6" generator="CGImap 0.9.2 (723230 spike-08.openstreetmap.org)"
   ↪  copyright="OpenStreetMap and contributors"
   ↪  attribution="http://www.openstreetmap.org/copyright"
   ↪  license="http://opendatacommons.org/licenses/odbl/1-0/">
2  <way id="72597051" visible="true" version="14" changeset="146483869"
   ↪  timestamp="2024-01-20T15:50:37Z" user="osmviborg" uid="379467">
3  <nd ref="861638675"/>
4  <nd ref="861638757"/>
5  <nd ref="3913585026"/>
6  <nd ref="3913585033"/>
7  <nd ref="11534186031"/>
8  <nd ref="11534186024"/>
9  <nd ref="11534186018"/>
10 <nd ref="11534186009"/>
```

Listing 2: OpenStreetMap data - XML format

At Listing 2 we see an example of how an OSM data looks like, this here is based on the Utzon Park in Aalborg, Denmark. At line 2 here we are seeing a Way element, we can see when it was last updated, and it's associated ID. At Line 6 we see an example of a given node, here the numbers are representing the node ID

Chapter 4

Dataset design

4.1 Dataset

To train a machine learning model, a dataset is needed. This needs to be communicated in the form of "meta-data" that ensures the machine learning model understands the annotation logic behind of the images fed into the model. This can be an own custom meta-data text file, and/or use well-known established dataset formats: COCO [25], PASCAL VOC [52], and the yolo format. As this is a multi-label classification problem, it is easier to make a custom meta-data file, with a custom dataloader for full control.

4.1.1 Building a dataset

In creating a dataset tailored to personalized routes for recreational walks, some criteria have to be fulfilled.

- Ensure that the classes included in the dataset are easily identifiable and commonly found in an urban environment.
- The selected classes need to be relevant for personalized categories such as culture, nature, etc.
  - Classes must be useful for pedestrians' points of view.
- Limit the number of classes or combine closely related objects into 1 class.

As a route can vary in length and encounter various unique ones, it is important to analyse what purpose a route has. For example, is the intent for the route to go shopping, sightseeing architectural pieces, explore parks placed in the city, or just leisurely walking? One way to approach this problem is through the use of classes and superclasses. By categorizing route features into classes linked to superclasses,

it becomes easier to analyse and compare routes between each other and propose routes to user preferences.

List of Classes						
Culture	**Harbor**	**Nature**	**Entertainment**	**Commercial Zone**	**Residential Zone**	**City Infrastructure**
Street art	Ship/Boat	Park	Sport fields	Supermarket	Apartment building	Bus stops
Modern architecture	Seawater	Trees	Stadium	Mall	Fence/Walls/hedges	Parking Area
Historic buildings	Dock cleat	Pond/River	Playground/outdoor workout	Stores	Garden	Urban greening
Statues/Sculptures	Dock	Bush	Bar/Pub	Restaurants/Cafe	House	Transport hub
Bridge				Pedestrian street		Hospital/police stations
				Hotel		

Table 4.1: List of Categories in the City Environment

Table 4.1 is a list of the proposed list of superclasses and their inherited classes that the dataset will consist of. The chosen classes are meant to be static objects that do not change rapidly as the flow of time can have a great impact on the city, for example Christmas decorations. However, we can not blindly annotate images, and therefore it is good to have contains and requirements before an image can be annotated/tagged with these classes.

4.1.2 Culture

This superclass includes cultural objects that are commonly present in cities.

4.1.2.1 Street art

Street art is small to large-scale graffiti projects with artistic features and motives. This will exclude graffiti with only letters, as this might be perceived differently. Any images containing any form of street art will be annotated.

4.1.2.2 Modern buildings

Modern buildings are architectural structures or new buildings. They are annotated if there is a clear view of the building. A good example of a modern building is "Musikkens hus" in Aalborg centrum.

4.1.2.3 Historical buildings

Historical buildings are defined as buildings that have survived the time of modernization and historical monumental buildings such as churches, castles, palaces, etc.

4.1.2.4 Statues/sculptures

Statues and sculptures, both annotated in images provided there is a good line of sight regardless of position or size, serve as historical references to objects or persons. However, sculptures encompass distinct creations beyond the classification of statues.

4.1.2.5 Bridge

Bridges can come in many variations, here it is mainly focused on bridges that are considered huge, or play a big part in the city landscape.

4.1.3 Harbor

Harbours are frequently situated in coastal cities such as Aalborg. Therefore, this category is included to annotate typical features that distinguish a harbour's identity. However, for the sake of simplicity and consistency, only static objects, such as docked ships, will be annotated.

4.1.3.1 Ships/boats

As mentioned earlier, only static objects will be taken into consideration, therefore it is only ships and boats that are docked and close that will be annotated.

4.1.3.2 Seawater

Seawater will be annotated in images featuring a harbour if it's unequivocally evident, considering the absence of prior knowledge of the location. Therefore, there must be no doubt regarding the presence of seawater before such annotation is made.

4.1.3.3 dock cleat

If there is a dock cleat in the picture, it will be considered that the location is around a harbour, as these are uncommonly found elsewhere.

4.1.3.4 Dock/pier

If there is a dock/pier visible in the image, it will be annotated as such.

4.1.4 Nature

The superclass nature is meant to define images in a more natural setting in the urban green environment in the form of green areas and parks.

4.1.4.1 Park

If an image unquestionably depicts a park setting, it will be labelled as such. A park typically features green terrain with bushes, trees, benches, etc., along with gravel or dirt pathways.

4.1.4.2 Tree

The annotation for "tree" will be limited to instances where either one large single tree serves as the primary focus in a green environment of the image, or when

there is a group of trees present in a natural setting anywhere within the image. It is important, it is limited to this, as urban greening also can consist of trees.

4.1.4.3 Pond/river

A pond or a river will be tagged if there is a pond or a river in the image, no matter the size. However, it needs to be present in the image before annotating, where there is no doubt the annotated object is present.

4.1.4.4 Bush

Bushes will be annotated if it is natural bushes, meaning all human-made bushes acting as hedges will be excluded from the annotation of a bush.

4.1.5 Entertainment

4.1.5.1 sport fields

Sports fields encompass outdoor sports complexes, including facilities like outdoor basketball courts, football fields, etc. These will be annotated if they are present and recognizable in the image.

4.1.5.2 Stadium

Stadiums will be tagged only when they are unmistakably recognizable from a distance, ensuring there is no doubt before applying the annotation. This criterion applies to identifiable outdoor stadiums such as football stadiums, as indoor sports may be held in an unidentifiable building without prior knowledge.

4.1.5.3 Playground/outdoor workout

Playgrounds consistently have obstacles, or other kinds of infrastructure very unique. Same with outdoor workouts placed in the city. The machines or tools are identifiable. If any of these things are present in the images, they will be tagged as playground/ outdoor workout in the annotation process

4.1.5.4 Bar/Pub

If there is a clear sign of a bar/pub, it will be annotated as such. This may be signs with bear logos, or the aesthetic of the bar/pub

4.1.6 Commercial Zone

There are many kinds of shops, but they are often clustered together. That is why restaurants and cafés also are included in this, and pedestrian streets as also most likely to include these.

4.1.6.1 Supermarket

Supermarkets may look very different from country to country, however in Denmark, they are commonly inherited by a bigger corporation. Here it will be tagged if there is a clear sign of it belonging to a supermarket presently in the images. Furthermore, if there is a signature sign from the firms.

4.1.6.2 Mall

Malls often have showcases of a lot of different goods by the windows of different companies, but this might be hard to see in a close-up image. Another takeaway is the entrances are followed up by different kinds of stores surrounding it, or it has a lot of promotions at the building. If there is a clear indication that the image is focused on a mall, it will be annotated as such.

4.1.6.3 stores

All kinds and types of stores will be annotated as stores. Typically, stores have open windows and promotions by the windows. If there is a clear visibility to a store and there is no question about it, it will be annotated as such.

4.1.6.4 café/Restaurants

Cafés are closely related to restaurants, however, they are more commonly to have a more relaxed atmosphere and outdoor sitting. Therefore, if there is an outdoor sitting and there is a doubt about it being a restaurant or café, the default tag will be a café.

4.1.6.5 Pedestrian street

Pedestrian streets are often huge streets. This can be more difficult to annotate if it's empty, but if there is a clear sign that this street is for pedestrians only it will be tagged as such.

4.1.6.6 Hotel

Hotels are often recognizable by their sign or their entrance. If these characteristics are present, it will be annotated as hotel.

4.1.7 Residential zone

Residential zones are apartment buildings, fences/walls/hedges, gardens, or houses that are commonly found in both the outer parts and centrum of the urban environment.

4.1.7.1 Apartment building

An apartment building is a multi-story residential structure containing separate living units or apartments. An image of an apartment building would typically consist of multiple windows and doors suggesting individual living units. Furthermore, multiple balconies would also suggest that the building is used as a residential structure, especially if the buildings have a similar pattern design that are the individual living units. It can also be the case the ground floor is used for stores. In this case, it will only be annotated if there is a clear view of most of the living units.

4.1.7.2 Fence/wall/hedge

If there is a fence, wall, or hedge covering a house or garden it will be annotated accordingly as this category. Walls in parks will be excluded in the annotation process as this is not part of the residential zone super category.

4.1.7.3 Garden

Front gardens for houses and occasionally for apartment complexes serve as aesthetically pleasing green spaces that enhance the overall ambiance of their surroundings, which is relevant for recreational walkers. Gardens will be annotated if they are present in the images and, consist of more than just a small field of grass.

4.1.7.4 House

Houses are commonly found in residential zones in the urban environment. These will be annotated if there is a house present in the image,

4.1.8 City infrastructure

City infrastructure is defined as crucial parts that make the city tick. The categories taken into consideration have to be unique enough to be recognizable, but also rare that they are present in every single image. With this in mind the final subclasses for this category it as follows, bus stops, parking areas, urban greening, transport hub, and hospital/ police station.

4.1.8.1 Bus stops

Bus stops are important for transportation around the city and stand as an important factor for a city's functionality. These will be annotated if there is a clear view of a bus stop.

4.1.8.2 Parking area

A parking Area is defined as a designated area for parking, but to limit the number of annotations, it needs to be a large open parking place or a parking house.

4.1.8.3 Urban greening

Urban greening is important to make the city more lively. This will often come in the form of small green decorations such as trees

4.1.8.4 Transport hub

Transport hubs are considered as train stations, bus stations, etc. These are important for transport both around the city and towards other cities.

4.1.8.5 hospital/ Police station

Hospitals and police stations are often recognizable by their unique building design and signs. If this is present in the image it will be annotated, with a sign of hospital or police station it will be annotated as such.

4.2 Methodology

As we were two annotators, it is important the guidelines are followed. However, there were still many edge case scenarios where bias will be the deciding factor of whether an object is annotated or not. To counter this, we started annotating subsets of chosen tiles containing a wide range of images and annotating them both ourselves. Afterwards, we compared our results, and discussed why we annotated the chosen objects, and if there were any differences we went through them and found a middle ground. This process was done one additional time as a reassurance, before the final annotation began.

4.3 Annotation process

The annotation were done in Label studio, which is a customizable framework for annotation tasks. We created our own framework, which made it possible to label

multiple labels for each image, and ideal for multi-label classification. We also adjusted the annotations, so it was outputted as integers instead of string, while still being the class string for the annotation framework interface.

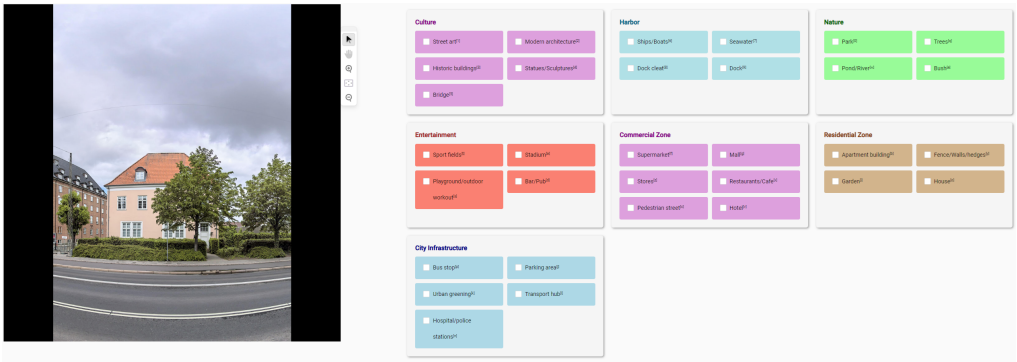


Figure 4.1: Interface of the annotation application

In Figure 4.1 is a screenshot of the annotation framework. It consists of an image that needed annotation, and boxes for each overclass containing all the classes that it inherited.

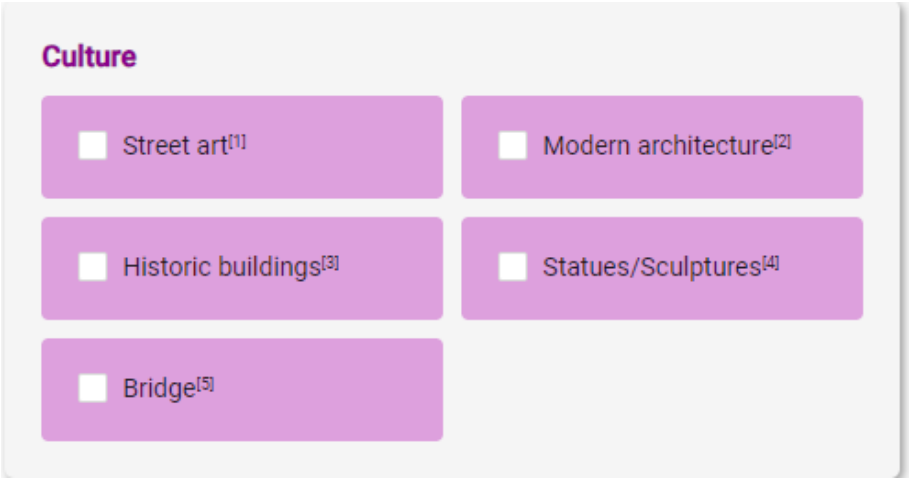


Figure 4.2: Single overclass interface

In Figure 4.2 shows how an example of the superclass box containing its inherited classes. The boxes were checkboxes, that needed to be clicked in order to be annotated. When all classes in the images were annotated, the image was submitted.

4.3.1 Preparing the dataset for Machine learning

As label studio was used for the dataset, it came with much unnecessary metadata when the annotations were exported. This was cleaned up, so we ended up with a JSON file containing, ID, annotations, and file path.

```
{
    "id": 1,
    "annotations": [
        "16",
        "23",
        "29"
    ],
    "image_data": "images/69144_40119/11006107550226572815_0.jpg"
},
```

Listing 3: JSON file annotation

The dataset has a total of 46739 images, where it was randomly split into training, validation, and testing datasets with an 80-10-10 split, resulting in they had 37391, 4673, 4675 images respectively.

```
{
    "id": 1,
    "annotations": [
        "3",
        "5",
        "6"
    ],
    "image_data": "images/69144_40119/11006107550226572815_0.jpg"
},
```

Listing 4: JSON file annotation for all overclasses

The JSON files were also adjusted, so the annotations were mapped to the superclass annotations instead. This was duo to some classes in the dataset being underrepresented.

Superclass	Training (n=37391)		Validation (n=4673)		Tesitng (n=4675)	
	Instances	Percentage	Instances	Percentage	Instances	Percentage
Culture	7391	9.53%	914	9.39%	924	9.60%
Harbor	1975	2.55%	261	2.68%	272	2.83%
Nature	6103	7.87%	731	7.51%	747	7.76%
Entertainment	691	0.89%	83	0.85%	91	0.95%
Commercial Zone	9947	12.82%	1283	13.19%	1256	13.05%
Residential Zone	29100	37.51%	3688	37.91%	3590	37.31%
City Infrastructure	22371	28.84%	2769	28.46%	2741	28.49%

Table 4.2: Distribution of overclass in training, validation, and testing dataset

In Table 4.2 the classes distributions can be seen for the different datasets. It is not perfectly distributed, and some superclasses such as Entertainment has very few instances with 0.89% in the training dataset. A way to counteract this is to train the model with adjusted weights for the classes that are underrepresented, to see if this would improve the generalization for the underrepresented classes.

4.4 Experimental design

The goal of the experimental design was to evaluate the quality of a generated route. Therefore, this experimental design will focus the aspect of the walking routes, and how they preferences the users had. We implemented a digital artifact where users were given a preconfigured route, here they had the ability to say they wanted to see on the route. Due to time constraint we opted for a digital walking experience, In this setup, the users were guided through the suggested route in a panoramic street view experience, allowing them to look around and explore, here both with and without their preferences, meaning they tested one route with their preferences and one where the algorithm would find the shortest path.

Semi-constructed interview is conducted afterwards, where we asked them :

- Did you notice any differences between Route 1 and Route 2?
- Did you prefer one route over the other?
- What did you like most about your favourite route?
- Did your preferences influence your choice of route?
- Did your choice make a difference in how you explored the route?

- Would you prefer to walk the fastest route, or would you choose the one you liked more, even if it was slower?
- Can you see yourself using an app like this for navigation when exploring a new city?

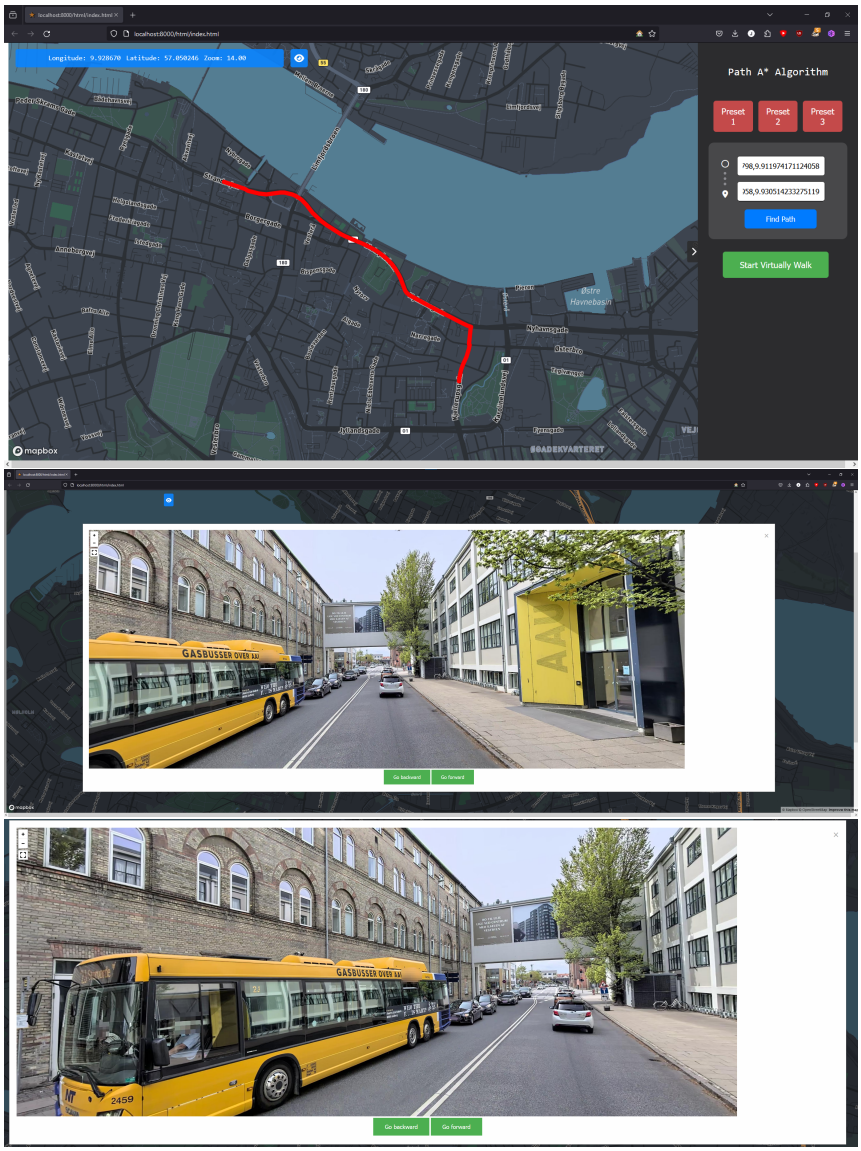


Figure 4.3: Screenshots of the prototype, here the path the participants could see, and the virutal street walking

4.5 OSM Data

We are utilizing OSM data to map out both pedestrian and vehicular routes within the city centre of Aalborg, this data will be used for creating the search through our graph, to create my custom routes based on preferences.

With OpenStreetMap's own website, it is possible to extract OSM data in an XML format [53] based on a bounding box. This data includes everything within the given area, meaning it is all unfiltered data.

```
<node id="3669452734" version="7" timestamp="2024-03-18T02:38:25Z" uid="2677992"
↪  user="mikkolukas" changeset="148796421" lat="57.0526632" lon="9.9151147">
    <tag k="button_operated" v="no"/>
    <tag k="crossing" v="traffic_signals"/>
    <tag k="crossing:markings" v="zebra"/>
    <tag k="highway" v="crossing"/>
    <tag k="tactile_paving" v="no"/>
    <tag k="traffic_signals:sound" v="no"/>
    <tag k="traffic_signals:vibration" v="no"/>
</node>
```

Listing 5: OSM — Data example

At Figure 4.4 we see a plotted image of the OSM data visualized, essentially this is all the roads, streets that can be utilized.

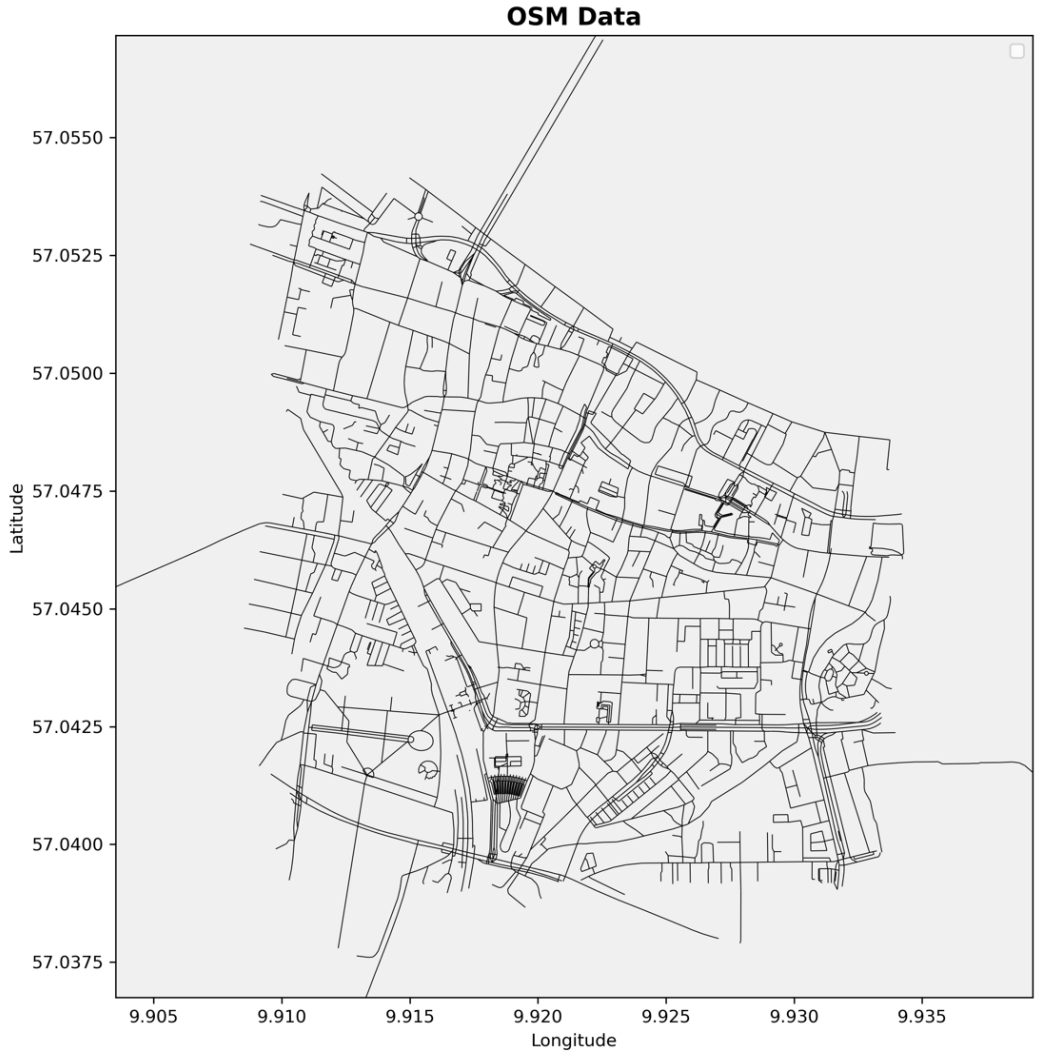


Figure 4.4: OSM Data visualized

Chapter 5

Implementation

In this section, we will describe the implementation of our solution which consists of scraping images, creating the dataset, machine learning and the logic behind the final implementation.

5.1 Implementation of the convolutional neural network

In this section, it will be described in detail how the implementation of the neural network was.

5.1.1 Loading the dataset

For loading the dataset, we had a class for handling relevant information such as annotations labels, image metadata, and transformation. The label was converted into a one-hot encoding tensor.

```python
class CustomDataset(Dataset):
    def __init__(self, json_file, root_dir, num_classes, transform=None):
        self.json_file = json_file
        self.root_dir = root_dir
        self.num_classes = num_classes
        self.transform = transform
        with open(json_file, 'r') as f:
            self.data = json.load(f)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        img_name = os.path.join(self.root_dir, self.data[idx]['image_data'])
        image = Image.open(img_name)
        annotations = [int(x) for x in self.data[idx]['annotations']]

        label_tensor = torch.zeros(self.num_classes)
        for label in annotations:
            label_tensor[label] = 1

        if self.transform:
            image = self.transform(image)
        return {'image': image, 'annotations': label_tensor}

    def class_distribution(self):
        class_counts = torch.zeros(self.num_classes)

        # Iterate through the dataset and count each class
        for item in self.data:
            annotations = [int(x) for x in item['annotations']]
            for label in annotations:
                class_counts[label] += 1
        return class_counts

    def class_weights(self):
        class_counts = self.class_distribution()
        total_samples = len(self.data)

        # Compute the weight for each class
        weights = total_samples / (self.num_classes * class_counts)
        weights[class_counts == 0] = 0

        return weights
```

Listing 6: Dataset Loader

In Listing 6 the getitem function, creates a one-hot encoder string, where it afterwards replaces the 0 with 1 if there is a label. If class_weights is used for weights for the loss function, the class distribution needs to be calculated, as this is required in the formula.

```
transform_train = v2.Compose([
    # v2.Resize((512, 512)),
    v2.ToTensor(),
    v2.ToDtype(torch.float32, scale=True),
    v2.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    v2.RandomHorizontalFlip(p=0.5),
    v2.ColorJitter(contrast=0.1, saturation=0.1),
    v2.RandomAdjustSharpness(0.1, p=0.5),
    v2.RandomEqualize(p=0.5),
])
```

Listing 7: Augmentation of images

Listing 3.8 shows the different transformation and augmentations that are applied to an image. As the images are already resized to the desired size before being loaded into the dataloader, it is not necessary to resize them. 3 out of the 4 the augmentations has a probability of 50% to trigger, where ColorJitter, is always applied to an image with a randomized value for contrast and saturation between 0.9-1.1. This transformation is only applied to the training set, whereas, for the validation and testing set, it is only the preprocessing steps that is applied. The normalization is taken from the ImageNet dataset[54] as taken from the default value[55]

```
train_dataset = CustomDataset(json_file='train.json', root_dir=r'D:/Pyphon
↪  projects/p10/', num_classes=7, transform=transform_val)

dataloader_train = DataLoader(train_dataset, batch_size=32, shuffle=True)projects/p10/',
↪  num_classes=7, transform=transform_train)
```

Listing 8: Initialize data loader

Listing 8 shows the code, where we extract relevant information, such as where is the root directory, the location of the metafile, transformation algorithm, and the total number of classes. As the model is only trained on the superclasses (see chapter 4) it is equal to 7.

5.1.2 ResNet model

As the ResNet model needs to be customized to our model, it needs to be adjusted, by adjusting some layers.

```
1  class ResNet50(torch.nn.Module):
2      def __init__(self, num_classes):
3          super(ResNet50, self).__init__()
4          self.resnet50 = models.resnet50(pretrained=False)
5          num_features = self.resnet50.fc.in_features
6          self.resnet50.fc = torch.nn.Identity()  # Remove the original fully connected layer
7          self.fc = torch.nn.Linear(num_features, num_classes)  # Add a new fully connected
           ↪  layer
8
9      def forward(self, x):
10         x = self.resnet50(x)
11         x = self.fc(x)
12         return x
```

Listing 9: ResNet50 Model implementation

In Listing 9, we remove the original fully connected layer, in order to customize to our implementation. Therefore, we changed the number of features the linear layer has being equal to the number of features of the previous layer, and change the output layer to be equal to the number of classes. It is important to mention the output is raw, however this is intentional for the chosen criterion of BCEWithLogitsLoss.

5.1.3 Training the model

In order to train the model, some hyperparameters have to be defined. From the data loader the batch sized is 32. However, the optimization algorithm, learning rate, number of epochs, and loss function are initialized.

```
1
2  # Define hyperparameters
3  learning_rate = 0.001
4  num_epochs = 25
5
6  # Define loss function and optimizer
7  weights = train_dataset.class_weights().to(device)
8  print(weights)
9  criterion = nn.BCEWithLogitsLoss(weight = weights)
10 optimizer = optim.Adam(model.parameters(), lr=learning_rate)
11
12 # Train the model
13 resume= False
14 save_dir = r"/kaggle/working/"
15 epoch = 1
16
17 if resume:
18     resume_from = f"/kaggle/working/model_epoch_{epoch}.pt"
19     train_model(model, dataloader_train, dataloader_train, criterion, optimizer, device,
       ↪ num_epochs, save_dir, resume_from)
20 else:
21     train_losses, val_losses, train_accuracies, val_accuracies = train_model(model,
       ↪ dataloader_train, dataloader_val, criterion, optimizer, device,
       ↪ num_epochs,save_dir)
22
```

Listing 10: Model hyper parameters

In listing 10, shows an overview of the hyperparameters. The number of epoch is equal to 25, with the BCEWithLogitsLoss [31], which is essentially a sigmoid activation function followed by the Binary Cross Entropy loss function as described in section 3.4.1. Furthermore, for efficiency of training time, it was decided to choose the Adam optimizer as the optimizer algorithm, with a learning rate of 0.001. Furthermore, this is optional, but weights can also be defined for the loss function which are commonly used for imbalanced datasets, in order to make classes that are underrepresented in the dataset more relevant for the calculation of loss.

```
1  def train(model, dataloader, criterion, optimizer, device):
2      model.train()
3      running_loss = 0.0
4      correct_predictions = 0
5      total_predictions = 0
6
7      # Get number of classes from the first batch
8      batch = next(iter(dataloader))
9      num_classes = batch['annotations'].shape[1]
10
11     true_positives = [0] * num_classes
12     false_positives = [0] * num_classes
13     true_negatives = [0] * num_classes
14     false_negatives = [0] * num_classes
15
16     #For each batch
17     for batch in dataloader:
18         inputs, labels = batch['image'].to(device), batch['annotations'].to(device)
19
20         optimizer.zero_grad()
21         outputs = model(inputs) #Forward propagation
22         loss = criterion(outputs, labels) #Calculate loss
23         loss.backward() #backpropagation
24         optimizer.step() #Step in the gradient descent
25
26         running_loss += loss.item()
27
28         predicted = (outputs > 0.0).float()
```

Listing 11: Train model

Listing 11 provides the first part of the train function. In this function, we initialize many values, that are used to track the training performance of each class during training. As the loss function already has a build in sigmoid activation function before being fed into the Binary Cross Entropy loss, the output needs to be the "raw" value as seen in line 27. This is done as a Sigmoid activation function, values over 0 is over 0.5, meaning a class is predicted, and for values under 0, sigmoid activation function returns less than 0.5 resulting it in not being predicted, as it can be seen in Figure 3.7 in Section 3.2.4. Therefore, the prediction also needs to be defined as true, if the raw value is positive, in order to stay true to the loss functions logic.

```
1
2           for i in range(num_classes):
3               true_positives[i] += ((predicted[:, i] == 1) & (labels[:, i] ==
                ↪  1)).sum().item()
4               false_positives[i] += ((predicted[:, i] == 1) & (labels[:, i] ==
                ↪  0)).sum().item()
5               true_negatives[i] += ((predicted[:, i] == 0) & (labels[:, i] ==
                ↪  0)).sum().item()
6               false_negatives[i] += ((predicted[:, i] == 0) & (labels[:, i] ==
                ↪  1)).sum().item()
7
8           total_predictions += labels.size(0) * num_classes
9           correct_predictions += (predicted == labels).sum().item()
10
11      train_loss = running_loss / len(dataloader)
12      train_accuracy = correct_predictions / total_predictions
13
14      precision = [0] * num_classes
15      recall = [0] * num_classes
16      f1_score = [0] * num_classes
17
18      for i in range(num_classes):
19          if true_positives[i] + false_positives[i] > 0:
20              precision[i] = true_positives[i] / (true_positives[i] + false_positives[i])
21          if true_positives[i] + false_negatives[i] > 0:
22              recall[i] = true_positives[i] / (true_positives[i] + false_negatives[i])
23          if precision[i] + recall[i] > 0:
24              f1_score[i] = 2 * (precision[i] * recall[i]) / (precision[i] + recall[i])
25
26      return train_loss, train_accuracy, true_positives, false_positives, true_negatives,
        ↪  false_negatives, precision, recall, f1_score
```

Listing 12: Calculate loss and other important parameters through training

For Listing 12, the different parameters for each class is tracked and output after each epoch. These values are stored for further analysis, in order to analyse the training. For the validation set, the process is mostly the same, however from Listing 11, in line 2, it is mode.eval() instead, and before feeding the batch into dataloader, we have torch.no_grad() in order to prevent the gradient to be calculated and updated in relation to the optimizer.

5.2 Scraping street view images

The designated area for our data acquisition will be in Aalborg Centre in the North Jutland of Denmark. Our dataset comprises street-view images from Apple Look Around, but other services like Google Street View can also be utilized for this purpose as long the images are fairly recent.

5.2.1 From Bounding box to tile coordinates

We define a bounding box based on geographical coordinates (latitude and longitude) to find the tile coordinates, respectively representing X and Y coordinates. The bounding box can be seen at Figure 5.1, where the selected area on Aalborg Centre is selected.

Figure 5.1: Bounding box area selected from OpenStreetMap

OpenStreetMap provides ready-to-deploy code, when it comes to converting longitude and latitude to tile representations which is seen at Listing 13

```python
import math
def deg2num(lat_deg, lon_deg, zoom):
    lat_rad = math.radians(lat_deg)
    n = 2.0 ** zoom
    xtile = int((lon_deg + 180.0) / 360.0 * n)
    ytile = int((1.0 - math.asinh(math.tan(lat_rad)) / math.pi) / 2.0 * n)
    return (xtile, ytile)
```

Listing 13: Lon./lat. to tile numbers [56]

With this code, we can find all possible tiles of a given bounding box seen at Listing 14. In the nested for loop, the outer loop iterates over the range of x-coordinates, while the inner loop iterates over the range of y-coordinates. Each combination of X and Y coordinates is appended to a list containing the tile coordinates.

```python
top_left_coords = deg2num(57.0516, 9.9142, 17) # = (69145, 40120)
bottom_right_coords = deg2num(57.0425, 9.9348, 17) # = (69153, 40126)

tiles = []
        for x in range(top_left_coords[0], bottom_right_coords[0] + 1):
            for y in range(top_left_coords[1], bottom_right_coords[1] + 1):
                tiles.append((x, y))
        return tiles
```

Listing 14: Finding all possible tiles in a given bounding box

5.3 OpenStreetMap Data

We utilized the library OSMnx and NetworkX for loading the OSM data. As the OSM data is in a .OSM format, we used those for processing the file.
With OSMnx we are able to load the OSM data, and project the graph. The two lines of code are using the OSMnx library to create a graph representation of OpenStreetMap (OSM) data.

```
self.graph = ox.graph_from_xml(file_path, simplify=False)

self.drivable_graph =
↪  ox.project_graph(ox.utils_graph.truncate.largest_component(self.graph,
↪  strongly=True),to_crs="EPSG:4326")
```

Listing 15: Code showing how we create a graph from osm file

Here we are converting the OSM data into a graph. "The simplify=False" will control whether, the resulting graph will retain the original vertices and edges from the OSM data. If "simplify=True" will result in pruning non-essential data from the vertices and edges. We optioned for setting the value to "False" as this would give us the ability to manually remove unwanted data, but also gives us the option if needed to manually weight vertices based on if they are pedestrian routes or road routes where cars are driving.

```
self.graph = ox.graph_from_xml(file_path, simplify=False)
```

Listing 16: Loading OSM data as graph

The *ox.utils_graph.truncate.largest_component(self.graph, strongly=True)* is responsible for identifying the largest, strongest connected edges within the graph and connecting them into a directed sub-graph to ensure that every vertex is researchable from every other vertex. This is useful in the context of navigational networks, as it ensures that a route is possible from any given point in the graph.

```
self.drivable_graph =
↪  ox.project_graph(ox.utils_graph.truncate.largest_component(self.graph,
↪  strongly=True), to_crs="EPSG:4326")
```

Listing 17: Ensure that the vertices have edges connecting them together

5.3.0.1 Manually filtering the OSM data

The filter_graph_by_nodes function is used to filter the nodes of the graph based on a predefined list of vertex IDs. Here we remove unwanted data from the graph. At Listing 35 we find what vertices to keep, "self.drivable_graph.nodes" is a collection of all the vertices from the graph, plus the adjacent vertices. This ensures that the vertices we keep, still have edges connected between them.

```
nodes_to_keep = set()
for node_id in self.node_ids_to_keep:
    if node_id in self.drivable_graph.nodes:
        nodes_to_keep.add(node_id)
        nodes_to_keep.update(self.drivable_graph.neighbors(node_id))
        neighbors = list(self.drivable_graph.neighbors(node_id))
        if neighbors:
            nodes_to_keep.add(node_id)
            nodes_to_keep.update(neighbors)
```

Listing 18: Filtering of OSM data — Vertex based filtering

We check if the filtering works, by looking at the length of vertices and edges of the graph.

```
Before
####################################
Number of nodes before filtering: 10540
Total number of neighbors before filtering: 23340
Length of the graph edges before filtering 26108
################################################
After
################################################
Number of nodes before filtering: 2504
Total number of neighbors before filtering: 4788
Length of the graph edges before filtering 5036
################################################
```

Listing 19: Filtering — Showing a change in amount of vertices and edges in the graph

5.3.1 Creating Custom weight based on Image Annotations

For the A* we want to change the weighting cost of selected vertices based on whether, they include annotations.

We give all vertices in the graph a preset of empty annotations, next we match against a list of vertices we have annotations for and updates the graph vertices to include the annotations

```
for node_id in self.drivable_graph.nodes:
    if node_id in annotations:
        self.drivable_graph.nodes[node_id].update(annotations[node_id])
    else:
        self.drivable_graph.nodes[node_id].update({
            'City infrastructure': 0,
            'Residential Zone': 0,
            'Commercial Zone': 0,
            'Entertainment Zone': 0,
            'Nature': 0,
            'Harbour': 0,
            'Culture': 0
        })
```

Listing 20: Filtering — Updating vertices with annotations

We can see if we print a given vertex from the graph, it now includes the annotations.

```
before update, node ID: 3735248896 data: {'y': 57.0434176, 'x': 9.9253107, 'lon':
↪  9.9253107, 'lat': 57.0434176}

after update, node ID: 3735248896 data: {'y': 57.0434176, 'x': 9.9253107, 'lon':
↪  9.9253107, 'lat': 57.0434176, 'City infrastructure': 1, 'Residential Zone': 1,
↪  'Commercial Zone': 0, 'Entertainment Zone': 0, 'Nature': 0, 'Harbour': 0, 'Culture':
↪  1}
```

Listing 21: Showing each vertex in the graph now includes the annotations

Now that each vertex in the graph now includes our annotations, we can now select evaluate each vertex based on whether it includes an annotation. We have currently implemented that up to three preferences can be used in a priority manner, namely pref_one,pref_two,pref_three. Here pref_one is the top priority.

```
default_values = 10

pref_one = 1.0
pref_two = 3.0
pref_three = 5.0

if fast_route == False:
    custom_weights = {
        'City infrastructure': pref_three,
        'Residential Zone': pref_one,
        'Commercial Zone': pref_two,
        'Entertainment Zone': default_values,
        'Nature': default_values,
        'Harbour': default_values,
        'Culture': default_values
            }
else:
    default_values = 1
    custom_weights = {
        'City infrastructure': default_values,
        'Residential Zone': default_values,
        'Commercial Zone': default_values,
        'Entertainment Zone': default_values,
        'Nature': default_values,
        'Harbour': default_values,
        'Culture': default_values
            }
```

Listing 22: Code showing how we are using our custom weights

Here is what a preference would look like, here we weight in favour of that the vertices with the annotations with Nature, Harbour, Culture.

```
custom_weights = {
        'City infrastructure': pref_three,
        'Residential Zone': pref_one,
        'Commercial Zone': pref_two,
        'Entertainment Zone': default_values,
        'Nature': pref_three,
        'Harbour': pref_one,
        'Culture': pref_two
}
```

Listing 23: Showing each vertex in the graph now includes the annotations

Here we find annotations for both vertices, and this is based on what key value if the value of 1.

```
annotations_node1 = {k: v for k, v in u.items() if k in custom_weights and v == 1}
annotations_node2 = {k: v for k, v in v.items() if k in custom_weights and v == 1}
```

Listing 24: Finding the annotations from both vertices

If we find no annotations, then we set the cost factor for either vertices to equal one. Otherwise, it calculates the average weight based on the annotations' weights. The final cost factor is calculated by finding the minimum value between two vertices, finally it returns the custom cost based on the final cost factor and the distance.

```
if len(annotations_node1) == 0:
    cost_factor_node1 = 1
else:
    cost_factor_node1 = sum(custom_weights.get(annotation, 1) for annotation in
    ↪  annotations_node1) / len(annotations_node1)

if len(annotations_node2) == 0:
    cost_factor_node2 = 1
else:
    cost_factor_node2 = sum(custom_weights.get(annotation, 1) for annotation in
    ↪  annotations_node2) / len(annotations_node2)

cost_factor = min(cost_factor_node1, cost_factor_node2)

return distance * cost_factor + road_type_factor
```

Listing 25: Code showing how we calculate the cost factor for each vertex and how the final cost factor is calculated

5.4 Path A* Search Algorithm

A* was chosen as the most optimal search algorithm, due to that it is a more resource friendly than Dijkstra's algorithm, thus making it more suitable to be running locally on a smartphone.

Our implementation is designed to search through a graph based on OSM data. The heuristic function used with the A* is based on the Euclidean distance.

```
def heuristic(self, a, b):
    (x1, y1) = self.graph.nodes[a]['y'], self.graph.nodes[a]['x']
    (x2, y2) = self.graph.nodes[b]['y'], self.graph.nodes[b]['x']
    return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
```

Listing 26: Python Implementation of the Euclidean distance between two given vertices

5.4.0.1 Class Definition

We define a python class named "AStarSeach", here in the parameters of the class, we take the graph, the start, and end vertex and a custom cost. We use the same for the initialization method as well, taking those parameters to make them local, accessible within the class itself.

```
class AStarSearch:
    def __init__(self, graph, start, end, custom_cost):
        self.graph = graph
        self.start = start
        self.end = end
        self.heuristic_cost = self.heuristic
        self.custom_cost = custom_cost
```

Listing 27: Class implementation of the A*

5.4.0.2 A* Algorithm

Here we define A*, we are creating the necessary dictionaries namely:

- open_set is the list to keep track of all the vertices that should be evaluated
- open_set_dict is made to check if a node is in open set or not
- came_from is to keep track of the path, which are the nodes we have traversed to.
- g_scoore is to keep track of the cheapest path from start to a given vertex
- likewise with f_score, we want to store the estimated total cost

```
def a_star_search(self):
    open_set = PriorityQueue()
    open_set.push(self.start, 0)
    came_from = {}
    g_score = {}
    came_from[self.start] = None
    g_score[self.start] = 0
```

Listing 28: The initial code for initializing the A*

The while loop ensures that the algorithm continues running as long as the open_set is not empty, since when the open_set is empty, all vertices have been evaluated. At line 5, we check the open_set to find the vertex from the set with the lowest f-score value, and select that vertex for evaluation. If the current_node is equal to the end node, we return the path. After that, it removes the current node from the open_set and the open_set_dict.

```
while open_set:
    current_node = None
    current_priority = float('inf')

    for node in open_set:
        if f_score[node] < current_priority:
            current_priority = f_score[node]
            current_node = node

    if current_node == self.end:
        return self.reconstruct_path(came_from, self.start, self.end)

    open_set.remove(current_node)
    del open_set_dict[current_node]
```

Listing 29: Code showing the while loop of the A*

Here we are getting the adjacent vertices for the current vertex

- we calculate a tentative_g_score for the start to the adjacent vertex
- If the adjacent vertex is not in the open_set then add it.
- If the adjacent vertex is already in the open_set and the new g-value is lower than the current g-value, then we update the g-value and set its parent to the current vertex.
- If the adjacent vertex is not in open_set_dict then append it to the open_set and open_set_dict

The Line at 2 at Listing 30 is where we add our custom weighting, Specifically the tentative_g_score = g_score[current_node] + **self.custom_cost(current_node, neighbor)**. Here we check if the current vertex and the adjacent vertex checking, if just one of these, includes our annotations when we will raise the cost to travel to the adjacent vertex.

```
for neighbor in self.graph.neighbors(current_node):
    tentative_g_score = g_score[current_node] + self.custom_cost(current_node, neighbor)
    if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
        came_from[neighbor] = current_node
        g_score[neighbor] = tentative_g_score
        f_score[neighbor] = tentative_g_score + self.heuristic_cost(neighbor, self.end)
        if neighbor not in open_set_dict:
            open_set.append(neighbor)
            open_set_dict[neighbor] = f_score[neighbor]
```

Listing 30: Code showing the main part of the A* here we evaluated the adjacent vertices

```
    def a_star_search(self):
        open_set = [self.start]

        open_set_dict = {self.start: 0}

        came_from = {self.start: None}

        g_score = {self.start: 0}
        f_score = {self.start: self.heuristic_cost(self.start, self.end)}

        while open_set:
            current_node = None
            current_priority = float('inf')

            # Find the node in open_set with the lowest f_score
            for node in open_set:
                if f_score[node] < current_priority:
                    current_priority = f_score[node]
                    current_node = node

            if current_node == self.end:
                return self.reconstruct_path(came_from, self.start, self.end)

            open_set.remove(current_node)
            del open_set_dict[current_node]

            for neighbor in self.graph.neighbors(current_node):
                tentative_g_score = g_score[current_node] + self.custom_cost(current_node,
                ↪  neighbor)
                if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
                    came_from[neighbor] = current_node
                    g_score[neighbor] = tentative_g_score
                    f_score[neighbor] = tentative_g_score + self.heuristic_cost(neighbor,
                    ↪  self.end)
                    if neighbor not in open_set_dict:
                        open_set.append(neighbor)
                        open_set_dict[neighbor] = f_score[neighbor]

        if self.end not in came_from:
            print("No path found to the end node.")
            return None

        path = self.reconstruct_path(came_from, self.start, self.end)
        return path
```

Listing 31: Our implementation of A*

5.5 OSM Data : Creating a search-able grid

We utilized the library OSMnx and NetworkX for loading the OSM data. As the OSM data is in a .OSM format, we used those for processing the file.
With OSMnx we are able to load the OSM data, and project the graph. The two lines of code are using the OSMnx library to create a graph representation of OpenStreetMap (OSM) data.

```
self.graph = ox.graph_from_xml(file_path, simplify=False)

self.drivable_graph =
↪  ox.project_graph(ox.utils_graph.truncate.largest_component(self.graph,
↪  strongly=True),to_crs="EPSG:4326")
```

Listing 32: Code showing how we create a graph from osm file

Here we are converting the OSM data into a graph. "The simplify=False" will control whether, the resulting graph will retain the original vertices and edges from the OSM data. If "simplify=True" will result in pruning non-essential data from the vertices and edges. We optioned for setting the value to "False" as this would give us the ability to manually remove unwanted data, but also gives us the option if needed to manually weight vertices based on if they are pedestrian routes or road routes where cars are driving.

```
self.graph = ox.graph_from_xml(file_path, simplify=False)
```

Listing 33: Loading OSM data as graph

The *ox.utils_graph.truncate.largest_component(self.graph, strongly=True)* is responsible for identifying the largest, strongest connected edges within the graph and connecting them into a directed sub-graph to ensure that every vertex is researchable from every other vertex. This is useful in the context of navigational networks, as it ensures that a route is possible from any given point in the graph.

```
self.drivable_graph =
↪  ox.project_graph(ox.utils_graph.truncate.largest_component(self.graph,
↪  strongly=True), to_crs="EPSG:4326")
```

Listing 34: Ensure that the vertices have edges connecting them together

5.5.0.1 Manually filtering the OSM data

The filter_graph_by_nodes function is used to filter the nodes of the graph based on a predefined list of vertex IDs. Here we remove unwanted data from the graph. At Listing 35 we find what vertices to keep, "self.drivable_graph.nodes" is a collection of all the vertices from the graph, plus the adjacent vertices. This ensures that the vertices we keep, still have edges connected between them.

```
nodes_to_keep = set()
for node_id in self.node_ids_to_keep:
    if node_id in self.drivable_graph.nodes:
        nodes_to_keep.add(node_id)
        nodes_to_keep.update(self.drivable_graph.neighbors(node_id))
        neighbors = list(self.drivable_graph.neighbors(node_id))
        if neighbors:
            nodes_to_keep.add(node_id)
            nodes_to_keep.update(neighbors)
```

Listing 35: Filtering of OSM data — Vertex based filtering

We check if the filtering works, by looking at the length of vertices and edges of the graph.

```
Before
####################################
Number of nodes before filtering: 10540
Total number of neighbors before filtering: 23340
Length of the graph edges before filtering 26108
##################################################
After
################################################
Number of nodes before filtering: 2504
Total number of neighbors before filtering: 4788
Length of the graph edges before filtering 5036
################################################
```

Listing 36: Filtering — Showing a change in amount of vertices and edges in the graph

5.5.1 Creating Custom weight based on Image Annotations

For the A* we want to change the weighting cost of selected vertices based on whether, they include annotations.

We give all vertices in the graph a preset of empty annotations, next we match against a list of vertices we have annotations for and updates the graph vertices to include the annotations

```
for node_id in self.drivable_graph.nodes:
    if node_id in annotations:
        self.drivable_graph.nodes[node_id].update(annotations[node_id])
    else:
        self.drivable_graph.nodes[node_id].update({
            'City infrastructure': 0,
            'Residential Zone': 0,
            'Commercial Zone': 0,
            'Entertainment Zone': 0,
            'Nature': 0,
            'Harbour': 0,
            'Culture': 0
        })
```

Listing 37: Filtering — Updating vertices with annotations

We can see if we print a given vertex from the graph, it now includes the annotations.

```
before update, node ID: 3735248896 data: {'y': 57.0434176, 'x': 9.9253107, 'lon':
↪  9.9253107, 'lat': 57.0434176}

after update, node ID: 3735248896 data: {'y': 57.0434176, 'x': 9.9253107, 'lon':
↪  9.9253107, 'lat': 57.0434176, 'City infrastructure': 1, 'Residential Zone': 1,
↪  'Commercial Zone': 0, 'Entertainment Zone': 0, 'Nature': 0, 'Harbour': 0, 'Culture':
↪  1}
```

Listing 38: Showing each vertex in the graph now includes the annotations

Now that each vertex in the graph now includes our annotations, we can now select evaluate each vertex based on whether it includes an annotation. We have currently implemented that up to three preferences can be used in a priority manner, namely pref_one,pref_two,pref_three. Here pref_one is the top priority.

```
default_values = 10

pref_one = 1.0
pref_two = 3.0
pref_three = 5.0

if fast_route == False:
    custom_weights = {
        'City infrastructure': pref_three,
        'Residential Zone': pref_one,
        'Commercial Zone': pref_two,
        'Entertainment Zone': default_values,
        'Nature': default_values,
        'Harbour': default_values,
        'Culture': default_values
            }
else:
    default_values = 1
    custom_weights = {
        'City infrastructure': default_values,
        'Residential Zone': default_values,
        'Commercial Zone': default_values,
        'Entertainment Zone': default_values,
        'Nature': default_values,
        'Harbour': default_values,
        'Culture': default_values
            }
```

Listing 39: Code showing how we are using our custom weights

Here is what a preference would look like, here we weight in favour of that the vertices with the annotations with Nature, Harbour, Culture.

```
custom_weights = {
        'City infrastructure': pref_three,
        'Residential Zone': pref_one,
        'Commercial Zone': pref_two,
        'Entertainment Zone': default_values,
        'Nature': pref_three,
        'Harbour': pref_one,
        'Culture': pref_two
}
```

Listing 40: Showing each vertex in the graph now includes the annotations

Here we find annotations for both vertices, and this is based on what key value if the value of 1.

```
annotations_node1 = {k: v for k, v in u.items() if k in custom_weights and v == 1}
annotations_node2 = {k: v for k, v in v.items() if k in custom_weights and v == 1}
```

Listing 41: Finding the annotations from both vertices

If we find no annotations, then we set the cost factor for either vertices to equal one. Otherwise, it calculates the average weight based on the annotations' weights. The final cost factor is calculated by finding the minimum value between two vertices, finally it returns the custom cost based on the final cost factor and the distance.

```
if len(annotations_node1) == 0:
    cost_factor_node1 = 1
else:
    cost_factor_node1 = sum(custom_weights.get(annotation, 1) for annotation in
    ↪  annotations_node1) / len(annotations_node1)

if len(annotations_node2) == 0:
    cost_factor_node2 = 1
else:
    cost_factor_node2 = sum(custom_weights.get(annotation, 1) for annotation in
    ↪  annotations_node2) / len(annotations_node2)

cost_factor = min(cost_factor_node1, cost_factor_node2)

return distance * cost_factor + road_type_factor
```

Listing 42: Code showing how we calculate the cost factor for each vertex and how the final cost factor is calculated

Chapter 6

Evaluation

6.1 Evaluation of the training of the model

The model was trained with adjusted weights and without weights, in order to see what model would perform best. This was done by analysing metrics such as loss accuracy during training. The training was equivalent to 25 epochs (37391 images per epoch) for both models for a fair comparison.

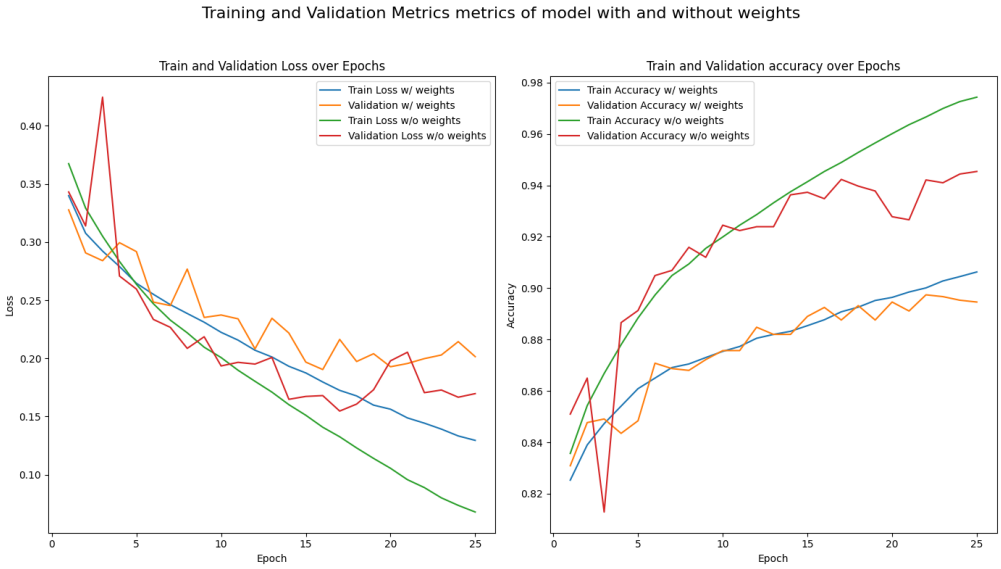


Figure 6.1: Metrics of the model over epochs

Figure 6.1 shows graphs for each model. Here it can be seen that the loss between the two models remain at a consistent pattern between them. The validation slope loss its lowest at epoch 16 and 17, where the loss rises and fluctuates. Meanwhile,

the training loss keeps decreasing over epochs. This may be an indication that the model is overfitting, and not be able to generalize well. This is further supported when looking at the accuracy curve for the graph. Here the validation accuracy is peaking between epoch 15-18 and then the slope flatten, while it keeps increasing for the training accuracy over epochs. Overall, the model without weighted classes performed better according to the lower loss and accuracy across the two graphs. Furthermore, the large fluctuations for the model, could mean that the model has a high time converging and finding a suitable minimum. Meaning, the model has a hard time generalizing to the validation set.

6.2 Evaluation of the machine learning model

The machine learning model was evaluated by using the test set and creating confusion matrixes. As this was a multi-label classification, a confusion matrix was constructed for each of the labels in the dataset.

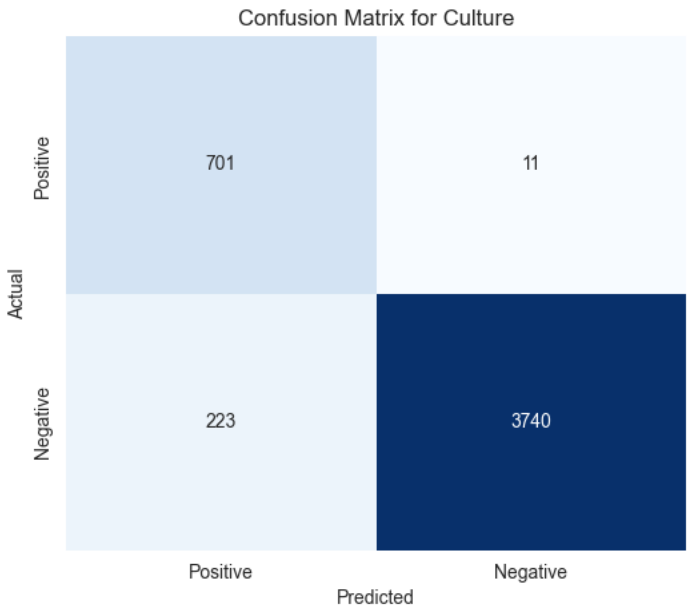


Figure 6.2: Culture confusion matrix Without weights

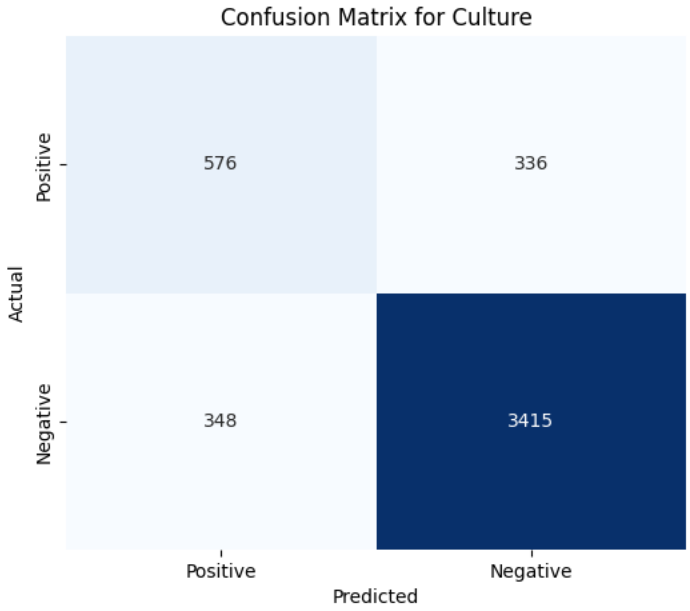


Figure 6.3: Culture confusion matrix with weights

Figure 6.2 and Figure 6.3 shows the results of the model of the class culture with and without adjusted weights. Here, culture with weights have a precision and recall of 0.98, and 0.76 vs the one with adjusted weights with 0.63 and 0.62 respectively. This indicates that even if the model without weights did not generalize better to an underrepresented class, as adjusted weights should have an advantage over.

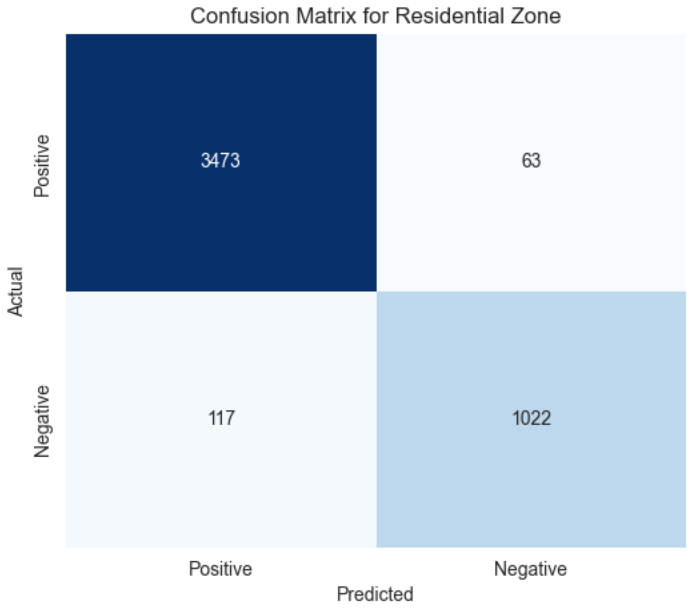


Figure 6.4: Residential zone confusion matrix without weights

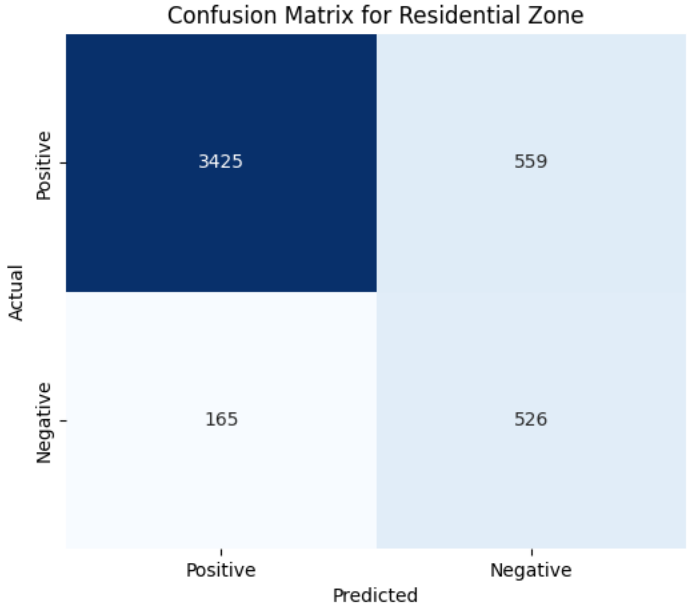


Figure 6.5: Residential zone confusion matrix with weights

& Precision & Recall & F1 Score & Precision & Recall & F1 Score

Figure **??** and Figure 6.5 shows a similar story, where the model with weights underperform compared to the one without. This is noticeable at the precision as the one without weights has a precision of 0.98 compared to 0.81.

Superclass	Without Weights			With Weights		
	Precision	Recall	F1 Score	Precision	Recall	F1 Score
Culture	0.98	0.76	0.86	0.63	0.62	0.63
Harbor	0.96	0.95	0.95	0.95	0.84	0.89
Nature	0.95	0.96	0.95	0.96	0.71	0.82
Entertainment	0.92	0.75	0.82	0.81	0.64	0.71
Commercial Zone	0.92	0.91	0.92	0.61	0.76	0.68
Residential Zone	0.98	0.97	0.97	0.86	0.95	0.90
City Infrastructure	0.98	0.97	0.97	0.86	0.90	0.88
Average	**0.96**	**0.90**	**0.92**	**0.81**	**0.77**	**0.79**

Table 6.1: Precision, Recall, and F1 Score for superclasses without and with weights

For the conclusion of, the models without weights is better compared to the model with weights is enhanced by looking at the metrics for average precision, recall and f1 score in Table 6.1. Here it has shown that, that the model without adjusted weights outperformed the model with adjusted weights, with an averaged f1 score of 0.79 and 0.92 respectively.

For final remarks, it is important to mention that the final accuracy of the test set without weights is 97.25%, which is higher than the validation accuracy at its global maxima. The training set had a total accuracy of 97.43%. This is most likely to the test set looking near identical compared to the training set. This will be discussed further upon in the discussion. 7

6.3 Evaluation of A*

We evaluated the suggested solution based, on same start and end node, but different preferences. Here we focused on several aspects,

- Total time for a given route
- Number of instances where the annotations were encountered
- A* with and without weighting preferences
- Visual difference

At Figure 6.6 preferences were not used to create the weighting. This is the fastest route suggested between the start and end point.

```
Total distance: 1.7313530000000001 km
Estimated time: 20.611345238095 24 minutes
{'City infrastructure': 94, 'Residential Zone': 94, 'Commercial Zone': 76, 'Entertainment Zone': 12, 'Nature': 23, 'Harbour': 49, 'Culture': 74}
```

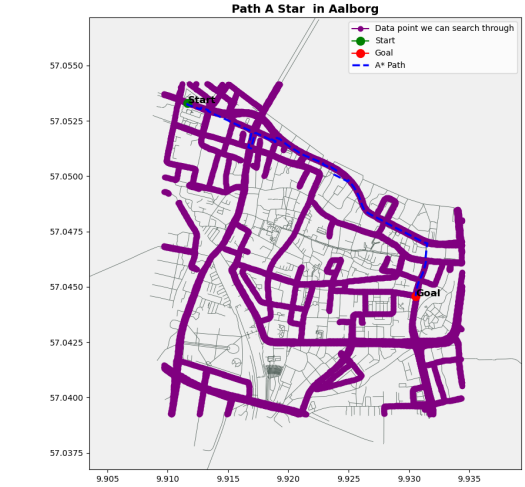
Listing 43: Preferences off, meaning fastest route

Figure 6.6: Two pictures showing the same suggested path, with no preferences set

At Figure 6.7 preferences were used to create the weighting. Here the preferences were. Harbour, Nature, Culture, with Harbour being top priority.

```
Total distance: 1.975177 km
Estimated time: 23.514011904761905 minutes
{'City infrastructure': 76, 'Residential Zone': 76, 'Commercial Zone': 62, 'Entertainment Zone': 16, 'Nature': 17, 'Harbour': 36, 'Culture': 56}
```

Listing 44: Preferences off, meaning fastest route



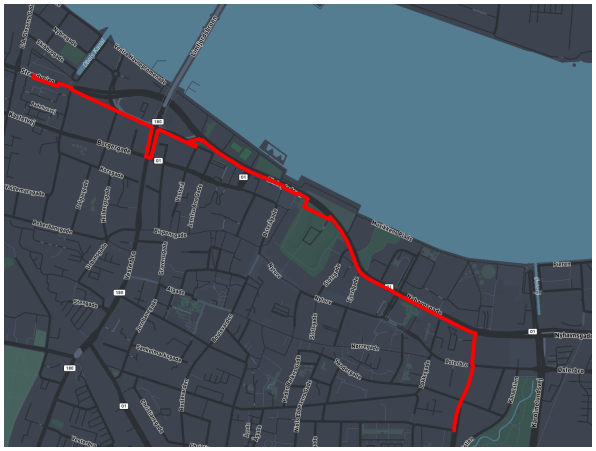


Figure 6.7: Two pictures showing the same suggested path, with no preferences set

At Figure 6.8 preferences were used to create the weighting. Here the preferences

were. Residential Zone, Commercial Zone, City infrastructure, with Residential Zone being top priority.

```
Total distance: 1.851242 km
Estimated time: 22.03859523809524 minutes
{'City infrastructure': 94, 'Residential Zone': 99, 'Commercial Zone': 88, 'Entertainment Zone': 32, 'Nature': 0, 'Harbour': 12, 'Culture': 53}
```

Listing 45: Preferences off, meaning fastest route

Figure 6.8: Two pictures showing the same suggested path, with no preferences set

6.4 Evaluation of Prototype

' We end up testing on a total of seven participants, while this is not enough by a long mile, we still somewhat gathered some useful data.

- Most participants noticed some minor differences between the two routes.
- The preferences set by the participants (e.g., wanting to see nature, culture, harbour, etc.) did seem to influence the routes to some extent, although not drastically for all participants, which resulting in participants noting that some routes had no difference.
- Participants seemed to explore the routes more actively when it was their first time, As if they were actively looking out for their preferences. On subsequent routes, they were less exploratory if they perceived the routes to be too similar.
- When asked about preferring the fastest route or a more scenic/preferred route even if slower, responses were mixed. Some preferred the fastest route, while others were willing to take a longer route, but it was depending on the current situation (e.g., being in a hurry or not, exploring a new city, etc.).
- Most participants said they could see themselves using an application that allows setting preferences for navigation and exploration, especially when visiting a new city or for recreational walks. Regarding the acceptable deviation in regard to time from the fastest route, responses ranged from 5 to 10 minutes to 15–30 minutes, with some suggesting up to 20-25% in minutes longer than the fastest route.
- It was noted that, the participants had somewhat of a hard time differentiating which of their routes was the fastest one. We switch the routes for every participant, so some would start with the fastest, while others would start with their preferences.

Chapter 7

Discussion

7.1 Model is not tested in other cities

The model was not tested in other cities than Aalborg centrum. This has led to the dataset being biased. This is further enhanced, that the images used for annotating had very little variance, as each image of panorama images was very close to each other location wise. This may also have led to the training, validation and test set looking nearly identical even after random shuffling. This could be a possible explanation why the training set had a near identical accuracy compared to the test set, and validation had the lowest.

7.2 Analysing and compare to another street view applications

While researching, we encountered that we are not allowed to test on Google Street View by reviewing their terms of service. Therefore, we could not test and evaluate the model on another street view application in Aalborg. If this was possible, it would have given us a fairer comparison, as temporal information might have been enough to get a better understanding of if the model could generalize to unseen data to some degree. Google Street View also has a wide range of images for different seasons and/or weather conditions that would make the model more robust if used to gain more data specifically in Aalborg city.

7.3 More data gathering

Another way to prevent the problem of overfitting, is to get a more diverse dataset. This could be done by gathering more data from other cities in Denmark or other countries that look similar to Danish cities. The temporal aspect is also important,

as weather, lighting, and seasons might have a bigger impact on some superclasses such as nature. Apple Look Around was also limited to certain classes such as harbour and parks, as it would only drive at car roads, which are very limited in those scenarios.

7.4 Hypertuning parameters

The model was only trained on the initial parameters, as well as a total epoch of 25 for weights and without weights. For future work, the parameters would have been analysed, and methods such as grid search would have been used, in order to find a new minima that had a better compared to the current one. Furthermore, SGD could have been used, if time was not an issue, as this may be slow compared to Adam, but has shown that it can be more accurate and better for generalizing on unseen data over longer training periods.

7.5 Normalization of images

We used the normalization mean and standard deviation from the PyTorch documentation (ImageNet's values), however this was the wrong approach as it should be an approximation of the dataset used. This may have lead to a decrease in performance while training the model.

7.6 Images for the dataset

The images of the dataset were taken by Apple look around. Apple look around lacked sufficient data in regard to having images on smaller streets in Aalborg City, this had a negative effect in regard to suggesting custom walking routes. We collected a total of 48524 images of Aalborg Centre which was enough images to get a sufficient amount in regard to the machine learning implementation.

7.7 Unnatural imbalance of the dataset

Some annotations for the dataset needs to be adjusted/combined. One way we limited this, was by only using superclasses. However, the superclass entertainment, with a total instance rate of 1& was a very underrepresented class. While the 2nd closest was harbour with over 2.5%. However, the low percentage was duo to Apple Look Around not having street view images closer to the harbour, as they only had images of car roads.

7.8 Annotating images

We spend too much time annotating images, which subsequently had a negative impact on the rest of the project, such as not having enough time to develop a mobile application.

In regard to the annotation process, annotations biases could exist. Important data was lost, this data included both of the annotators annotating the same images. With this it would be possible to debunk if annotation bias exist in the dataset.And due to that we may have a bias in when annotation the images on the dataset. This could lead to a negative impact in regard to the machine learning model. Furthermore, even if it was to the greatest of our ability to have a consistency between the two annotators, it could not be possible to analyse all images, and might have led to some inconsistencies for the annotation accuracy. Some classes such as modern architecture, or historic buildings may differ greatly from annotator to annotator, as this can be hard to define compared to a class such as seawater. As the images were also very close to each other and therefore look very much alike, habituation, might also have encored. This means that we may have been accustomed to the images to such a degree that it would be hard to detect subtle changes, which could have been impactful for the annotation process. A way to prevent this could be to shuffle the images that needed annotating, so there is no logical order.

7.9 A*

For the A* implementation we lacked sufficient data, as the route we could only evaluate were based on the imagery data we had annotations from, this means that we are missing a lot of smaller roads, pedestrian paths which could not be included. This would subsequently result in generated routes becoming longer of more convoluted than necessary, this is illustrated at Figure 7.1 where the blue path represents data points where the images are taken from.

Figure 7.1: Screenshot of the searchable paths in our graph

The reasoning behind favouring the A* compared to Dijkstra was, that originally the application was supposed to be a mobile application, but due to time constrains we did not have enough time to develop and test a mobile version.

If we had time, we would have developed and compared Dijkstra against A* solution. As A* checks only the cheapest adjacent vertices in a graph, it often misses vertices that are quite far away. Here, Dijkstra could be used instead, as Dijkstra systematically explores all adjacent vertices for a given vertex. Because of that, this can allow us to "always" force the route towards the user's preferences.

We suggest using more images in the context of that this would improve the feasibility of allowing us to search on more streets, resulting in more accurate representation in terms of the shortest path, plus seeing if Dijkstra's performance would lead to any negative effects in regard to if it was implemented on a mobile device.

7.10 Design of Experimental design

While testing the route quality on a digital approach was not the original idea, it still yields somewhat useful insight. The participant noted that visually, many of the routes were very similarly, and thus resulting in the participants "hurrying" when they did the second route. This could have had an impact in how they view the quality of the route.

We intended to do a study walk with the participants on with a given route with their preferences. Here we would have had the chance to observe them in their natural setting while walking with a navigational application. And this could also give us the ability to compare our solution to others. Not both in terms of suggested route, but also user interface, and system requirements.

Chapter 8

Conclusion

We have successfully created a navigation application that generates routes based on a user's preference in Aalborg city. This was done through the path A* algorithm to generate a heuristic route based on custom weights. By using the ground truth from the dataset, it was possible to create highly accurate and reliable routes that consider real-world conditions. This was noticeable for the participants that have used the application through questioning them in a qualitative semi structured interview. As part of the dataset creation, it was possible to create a convolutional neural network by altering ResNet-50 to a multi-label classification problem. The model showed high accuracy and other metrics, however this was duo to overfitting, and similarity between training, validation and test datasets. The concept has shown to have great potential to work around in the urban environment around the world.

Bibliography

[1] Helena Nordh et al.
"Walking as urban outdoor recreation: Public health for everyone".
In: *Journal of Outdoor Recreation and Tourism* 20 (2017), pp. 60–66.
ISSN: 2213-0780. DOI: https://doi.org/10.1016/j.jort.2017.09.005. URL:
https://www.sciencedirect.com/science/article/pii/S2213078017300555.

[2] Bjarne Ibsen, Jens Høyer-Kruse, and Karsten Elmose-Østerlund.
Danskernes bevægelsesvaner og motiver for bevægelse: Resultater fra undersøgelse af bevægelsesvaner. Dansk.
Undersøgelsen er gennemført med økonomisk støtte fra Nordea-fonden.
Center for Forskning i Idræt, Sundhed og Civilsamfund, SDU, 2021.
ISBN: 978-87-94006-29-3.

[3] Anne Kaag Andersen et al. "Indbyggere og jobs samles i byerne".
In: *Danmarks Statistik* (2018).
URL: https://www.dst.dk/da/Statistik/nyheder-analyser-publ/Analyser/visanalyse?cid=30726.

[4] *Google Maps*. Mobile application software. Accessed on March 21, 2024.
2024. URL: https://www.google.com/maps/.

[5] *Apple Look Around*. Mobile application software.
Accessed on March 21, 2024. 2024. URL: https://www.apple.com/maps/.

[6] Komoot. *Komoot | Find, plan and share your adventures — komoot.com*.
https://www.komoot.com/. [Accessed 20-03-2024]. 2010.

[7] *AllTrails*. Mobile application software. Accessed on March 21, 2024. 2010.
URL: https://www.alltrails.com/.

[8] Qian Yan, Shixian Luo, and Jiayi Jiang.
"Urban Residents' Preferred Walking Street Setting and Environmental Factors: The Case of Chengdu City". In: *Buildings* 13.5 (2023).
ISSN: 2075-5309. DOI: 10.3390/buildings13051199.
URL: https://www.mdpi.com/2075-5309/13/5/1199.

[9] Daniele Quercia, Rossano Schifanella, and Luca Maria Aiello.
"The shortest path to happiness: recommending beautiful, quiet, and happy routes in the city".
In: *Proceedings of the 25th ACM Conference on Hypertext and Social Media*.
HT '14. Santiago, Chile: Association for Computing Machinery, 2014,
pp. 116–125. ISBN: 9781450329545. DOI: 10.1145/2631775.2631799.
URL: https://doi-org.zorac.aub.aau.dk/10.1145/2631775.2631799.

[10] Reginald G. Golledge. "Path selection and route preference in human navigation: A progress report". eng.
In: *Spatial Information Theory A Theoretical Basis for GIS*.
Lecture Notes in Computer Science.
Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 207–222.
ISBN: 9783540603924.

[11] Jia Wang Urmi Shah. "A Personalised Pedestrian Navigation System".
In: (Sept. 7, 2023).
URL: https://drops.dagstuhl.de/storage/00lipics/lipics-vol277-giscience2023/LIPIcs.GIScience.2023.67/LIPIcs.GIScience.2023.67.pdf.

[12] Panote Siriaraya et al. "Beyond the Shortest Route: A Survey on Quality-Aware Route Navigation for Pedestrians".
In: *IEEE Access* 8 (2020), pp. 135569–135590.
DOI: 10.1109/ACCESS.2020.3011924.

[13] Shoko Wakamiya et al.
"Pleasant Route Suggestion based on Color and Object Rates".
In: *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*. WSDM '19.
Melbourne VIC, Australia: Association for Computing Machinery, 2019,
pp. 786–789. ISBN: 9781450359405. DOI: 10.1145/3289600.3290611.
URL: https://doi.org/10.1145/3289600.3290611.

[14] Ke Zhu and Jianxin Wu. "Residual Attention: A Simple but Effective Method for Multi-Label Recognition". In: *CoRR* abs/2108.02456 (2021).
arXiv: 2108.02456. URL: https://arxiv.org/abs/2108.02456.

[15] MathWorks. *Multilabel Image Classification Using Deep Learning*.
Accessed: May 18, 2024. 2024.
URL: https://se.mathworks.com/help/deeplearning/ug/multilabel-image-classification-using-deep-learning.html.

[16] Y. Lecun et al. "Gradient-based learning applied to document recognition".
In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.
DOI: 10.1109/5.726791.

[17] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[18] Shaoqing Ren et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. 2016. arXiv: 1506.01497 [cs.CV].

[19] Kaiming He et al. *Mask R-CNN*. 2018. arXiv: 1703.06870 [cs.CV].

[20] Alexander Kirillov et al. *Panoptic Segmentation*. 2019. arXiv: 1801.00868 [cs.CV].

[21] Ke Zhu and Jianxin Wu. "Residual Attention: A Simple but Effective Method for Multi-Label Recognition". In: *CoRR* abs/2108.02456 (2021). arXiv: 2108.02456. URL: https://arxiv.org/abs/2108.02456.

[22] Keiron O'Shea and Ryan Nash. "An Introduction to Convolutional Neural Networks". In: *CoRR* abs/1511.08458 (2015). arXiv: 1511.08458. URL: http://arxiv.org/abs/1511.08458.

[23] Vincent Dumoulin and Francesco Visin. *A guide to convolution arithmetic for deep learning*. 2018. arXiv: 1603.07285 [stat.ML].

[24] Aston Zhang et al. *Dive into Deep Learning*. https://D2L.ai. Cambridge University Press, 2023.

[25] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. 2015. arXiv: 1405.0312 [cs.CV].

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[27] Alexander Buslaev et al. "Albumentations: Fast and Flexible Image Augmentations". In: *Information* 11.2 (2020). ISSN: 2078-2489. DOI: 10.3390/info11020125. URL: https://www.mdpi.com/2078-2489/11/2/125.

[28] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *CoRR* abs/1912.01703 (2019). arXiv: 1912.01703. URL: http://arxiv.org/abs/1912.01703.

[29] Dominic Masters and Carlo Luschi.
"Revisiting Small Batch Training for Deep Neural Networks".
In: *CoRR* abs/1804.07612 (2018). arXiv: 1804.07612.
URL: http://arxiv.org/abs/1804.07612.

[30] Emanuel Ben Baruch et al.
"Asymmetric Loss For Multi-Label Classification".
In: *CoRR* abs/2009.14119 (2020). arXiv: 2009.14119.
URL: https://arxiv.org/abs/2009.14119.

[31] PyTorch. *BCELoss — PyTorch 2.3 documentation*.
https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html.
Accessed: 2024-05-20. 2024.

[32] Gareth James et al.
An Introduction to Statistical Learning with Applications in Python.
Springer Texts in Statistics. Cham: Springer, 2023. ISBN: 978-3-031-38746-3.
DOI: 10.1007/978-3-031-38747-0.
URL: https://link.springer.com/book/10.1007/978-3-031-38747-0.

[33] Sebastian Ruder.
"An overview of gradient descent optimization algorithms".
In: *CoRR* abs/1609.04747 (2016). arXiv: 1609.04747.
URL: http://arxiv.org/abs/1609.04747.

[34] Diederik P. Kingma and Jimmy Ba.
Adam: A Method for Stochastic Optimization. 2017. arXiv: 1412.6980 [cs.LG].

[35] Nitish Shirish Keskar and Richard Socher.
"Improving Generalization Performance by Switching from Adam to SGD".
In: *CoRR* abs/1712.07628 (2017). arXiv: 1712.07628.
URL: http://arxiv.org/abs/1712.07628.

[36] Shilong Liu et al.
"Query2Label: A Simple Transformer Way to Multi-Label Classification".
In: *CoRR* abs/2107.10834 (2021). arXiv: 2107.10834.
URL: https://arxiv.org/abs/2107.10834.

[37] Marius Cordts et al.
"The Cityscapes Dataset for Semantic Urban Scene Understanding".
In: *CoRR* abs/1604.01685 (2016). arXiv: 1604.01685.
URL: http://arxiv.org/abs/1604.01685.

[38] Ricardo García et al. "Web Map Tile Services for Spatial Data
Infrastructures: Management and Optimization". eng.
In: *Cartography: a tool for spatial analysis*. Ed. by Carlos Bateira.
Rijeka [Croatia]: IntechOpen, 2012, pp. 25–45. ISBN: 953-51-5005-7.
DOI: 10.5772/46129. URL: https://doi.org/10.5772/46129.

[39] John T Sample and Elias Ioup.
Tile-Based Geospatial Information Systems: Principles and Practices. ger ; eng.
1. Aufl. New York, NY: Springer Science + Business Media, 2010.
ISBN: 1441976302.

[40] Google. *Google Maps Map and Tile Coordinates*. URL: https://developers.google.com/maps/documentation/javascript/coordinates (visited on 03/13/2024).

[41] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. 5th.
McGraw-Hill Higher Education, 2002. ISBN: 0072424346.

[42] Haosheng Huang and Georg Gartner. "Collective intelligence-based route recommendation for assisting pedestrian wayfinding in the era of Web 2.0".
In: *Journal of Location Based Services* 6 (Mar. 2012), pp. 1–21.
DOI: 10.1080/17489725.2011.625302.

[43] Jan Balata et al. "Automatically generated landmark-enhanced navigation instructions for blind pedestrians". In: *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 2016, pp. 1605–1612.

[44] Shoko Wakamiya et al. "Lets not stare at smartphones while walking: memorable route recommendation by detecting effective landmarks".
In: *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. UbiComp '16.
Heidelberg, Germany: Association for Computing Machinery, 2016,
pp. 1136–1146. ISBN: 9781450344616. DOI: 10.1145/2971648.2971758.
URL: https://doi-org.zorac.aub.aau.dk/10.1145/2971648.2971758.

[45] Sumit Shah et al. "CROWDSAFE: crowd sourcing of crime incidents and safe routing on mobile devices". In: *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*.
GIS '11. Chicago, Illinois: Association for Computing Machinery, 2011,
pp. 521–524. ISBN: 9781450310314. DOI: 10.1145/2093973.2094064.
URL: https://dl.acm.org/doi/10.1145/2093973.2094064.

[46] Syeed Abrar Zaoad, Md. Mamun-Or-Rashid, and Md. Mosaddek Khan.
"CrowdSPaFE: A Crowd-Sourced Multimodal Recommendation System for Urban Route Safety". In: *IEEE Access* 11 (2023), pp. 23157–23166.
DOI: 10.1109/ACCESS.2023.3252881.

[47] E. W. Dijkstra. "A note on two problems in connexion with graphs". eng.
In: *Numerische Mathematik* 1.1 (1959), pp. 269–271. ISSN: 0029-599X.
URL: https://link.springer.com/article/10.1007/BF01386390s.

[48] E. W. Dijkstra. “A Note on Two Problems in Connexion with Graphs”. In: *Edsger Wybe Dijkstra: His Life,Work, and Legacy*. 1st ed. New York, NY, USA: Association for Computing Machinery, 2022, pp. 287–290. ISBN: 9781450397735. URL: https://doi.org/10.1145/3544585.3544600.

[49] Dian Rachmawati and Lysander Gustin. “Analysis of Dijkstra’s Algorithm and A* Algorithm in Shortest Path Problem”. In: *Journal of Physics: Conference Series* 1566 (June 2020), p. 012061. DOI: 10.1088/1742-6596/1566/1/012061.

[50] openstreetmap foundation. *OpenStreetMap*. URL: https://wiki.openstreetmap.org/wiki%20About_OpenStreetMap (visited on 05/28/2024).

[51] Adam Rousell and Alexander Zipf. “Towards a Landmark-Based Pedestrian Navigation Service Using OSM Data”. In: *ISPRS International Journal of Geo-Information* 6.3 (2017). ISSN: 2220-9964. DOI: 10.3390/ijgi6030064. URL: https://www.mdpi.com/2220-9964/6/3/64.

[52] M. Everingham et al. “The Pascal Visual Object Classes (VOC) Challenge”. In: *International Journal of Computer Vision* 88.2 (June 2010), pp. 303–338.

[53] openstreetmap foundation. *OpenStreetMap*. URL: https://www.openstreetmap.org (visited on 05/28/2024).

[54] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *CoRR* abs/1409.0575 (2014). arXiv: 1409.0575. URL: http://arxiv.org/abs/1409.0575.

[55] seberino. *Why image datasets need normalizing with means and stds specified like in transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])?* PyTorch Forum. PyTorch. 2023. URL: https://discuss.pytorch.org/t/why-image-datasets-need-normalizing-with-means-and-stds-specified-like-in-transforms-normalize-mean-0-485-0-456-0-406-std-0-229-0-224-0-225/187818.

[56] openstreetmap. *Lon./lat. to tile numbers*. Jan. 24, 2024. URL: https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames#Lon.2Flat._to_tile_numbers_2 (visited on 03/14/2024).

List of Figures

List of Listings

Chapter 9

Appendix A

Appendix B act as a digital appendix.