Universiteit Leiden

Master Computer Science

Embedding geographical locations
using triplet loss networks

Name:	R.M. van Hal
Date:	13/06/2019
Specialisation:	Advanced data analytics
1st supervisor:	Dr. W.J. Kowalczyk
2nd supervisor:	Prof.dr.ir. F.J. Verbeek

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

The success of embedding spaces created by machine learning models in natural language processing has raised the interest to use embedding spaces in other fields. In this thesis, the application of a deep learning model in the field of geoinformatics is explored. Using a convolutional neural network, tensors representing geographical locations are mapped to vectors, with the intention of keeping the location's features and their spatial properties. Using a triplet loss network, an embedding network is trained using two different loss functions. Using a dimensionality reduction algorithm the embedding spaces of both networks are visually inspected, which clearly shows that locations with similar features are mapped to similar vectors, while different locations are mapped to distant vectors. Additionally, it is shown that vectors computed by the embedding networks can preserve spatial properties of a location's features, in contrast to the location features frequency distribution where all spatial information is lost. A practical application of the embedding networks is an application which finds similar locations using the distances between their vectors in the embedding space as a measure of similarity. Lastly, a simple machine learning model is deployed to predict whether a location is residential or not based on the location's embedding vector alone. Without any fine-tuning, the model is able to do this classification correctly in 73% and 78% of the cases, which demonstrates that geographical locations can be substituted by their much simpler embedding vector representations when used as inputs for machine learning.

Acknowledgements

The thesis you are looking at would not exist without the help of other people. Therefore, I will take this moment to thank everyone who contributed or supported me in any way.

I would like to give special thanks to my supervisor Wojtek Kowalczyk from the Leiden Institute of Advanced Computer Science (LIACS) for his guidance and insights during this project. Being available often for discussions and input, he made this project possible.

I am also grateful to Fons Verbeek, also from LIACS, for being the second supervisor and his comments.

Furthermore, I would like to thank Erwin Haas and colleagues at Landscape for the original idea and support. The substantial amount of time I could spend at their company was a key factor in completing this project, during which I was surrounded by motivated experts in the Data Science world. Their knowledge and resources that I had unlimited access to were invaluable assets during this project.

Additionally, there are some great collaborative initiatives in making software that is accessible to everyone. Some of these initiatives were key in this thesis project. The map data used is copyrighted by the OpenStreetMap contributors and available from `www.openstreetmap.org`. The programming language used is Python, from the Python Software Foundation. The Keras library, developed by François Chollet and others, was used to implement the artificial neural networks.

Last but not least, I want to give a warm thanks to family and friends for their love and input, supporting me in this project both directly and indirectly.

Contents

1 Introduction

Over the past few years, many machine learning approaches have risen in the field of natural language processing. One of these approaches is the use of a machine learning model to generate word embeddings, where a word is mapped to a numerical, low-dimensional vector, while preserving the semantic relationships between words. Vectors that represent words with a similar meaning are located close to each other in the resulting embedding space and unrelated words are mapped to distant vectors. Usually, such word embedding models are trained in a semi-supervised manner, since it is hard to 'measure' the semantic similarity or relationship between words. Therefore, there are no explicit word vectors available that can be used as targets during training. Instead, the model is not only given a word, but also a context, or sentence, that the word appears in. This way, the model can learn which words are often used together and are therefore probably related. This information can be used to train a model that generates similar word vectors for words that are related. Popular word embedding models are *word2vec* [20] and *GloVe* [26], which both show state-of-the-art results on various natural language processing tasks while requiring less processing compared to previous methods.

The impressive results in the field of natural language processing lead to the interest of applying machine learning models to create embedding spaces in other fields as well. Examples are the embeddings of graphs [23, 28], images [11], medical concepts [5] or even emoji [10].

In this thesis, we explore the application of a convolutional neural network in the field of geoinformatics. Comparing geographical locations regarding their properties is difficult. Current approaches use visual comparison or quantitatively compare the features of the locations. However, visual comparison is tedious for large amounts of locations and statistical methods only cover the distribution of features in a location, not the spatial properties, or *semantics*, among them.

With the help of a deep learning model, we capture the features and semantics of geographical locations in an embedding space. In this embedding space, the vector representations of similar locations are similar, while the vectors of different locations are not. The distance between embedding vectors can be used to measure similarity of the locations that they represent. The smaller the difference, the bigger the similarity. This approach is inspired by a blog post from Sentiance [31], where the embedding vectors are used to determine the venue that a user is visiting, based on the surroundings.

The deep learning model that we deploy is a convolutional feed-forward neural network, trained using a triplet loss network. The input of the network is a three-dimensional tensor that describes a location, with two dimensions describing the spatial components and one dimension describing the features that are present. The network outputs a vector of fixed length, mapping the input tensor to a location in an embedding space.

The embedding space that we create can be used for a vast amount of applications. In this thesis, two of them are demonstrated. In the first, a distance metric is defined based on the distances between vectors in the embedding space,

which serves as a 'similarity score' between locations. We develop an application that uses this distance metric to build a ranking system that 'recommends' locations that are similar to a location of the user's choice. This application shows that distances in the embedding space are a measure of similarity in terms of the features of the locations, as well as their spatial properties. In the second application, we show that the generated embedding vectors can function as inputs for a machine learning model. With only the embedding vectors as input, a neural network can predict whether a location is residential are not. This demonstrates that the vectors can accurately preserve properties of locations.

This thesis is organized as follows. In Section 2, preliminary work is introduced. Important concepts that are used in this thesis are described. In Section 3, a general overview of the approach is given. First, the dataset that contains the features of geographical locations is introduced and we explain how such a location can be converted to a tensor that describes the location. Next, we explain how this tensor is used by an artificial neural network to produce an embedding vector, how such a network can be trained and how we can evaluate the results. Section 4 details the experimental setup, including how the locations are represented, what the embedding network's architecture is and how the network can be trained to generate representative embedding vectors. After the embedding network has been trained, the quality of the embedding vectors that it generates is assessed in Section 5. This includes the ability of the network to preserve features, as well as the ability to preserve spatial properties of those features. In Section 6, we take a look at how these embedding vectors can be used as a replacement for the locations themselves in practical applications. Two possible use cases of the created embedding space are demonstrated. First, we use the vectors to rank locations by similarity. Secondly, the vectors are used as inputs for a machine learning model. In the last section, Section 7, a conclusion of the research is provided, concluded with possible future work.

2 Related research

This section describes some preliminary work that is used in this thesis. First, the inspiration for the approach is outlined. Section 2.2 introduces briefly the concept of embedding vectors, followed by Section 2.3 which introduces artificial neural networks. A way to train these neural networks is using a triplet loss network, as explained in Section 2.4. Finally, a dimensionality reduction algorithm is introduced which is used to reduce high-dimensional vectors to lower dimensions.

2.1 Sentiance's venue mapping algorithm

In May 2018, a company called Sentiance released a blog post in which they describe their approach to implement a venue mapping algorithm [31]. Their goal is to develop an artificial neural network that learns an embedding space that captures the similarity of locations. This neural network is trained using a triplet loss network, that tries to cluster similar locations and separate distinct locations. However, while the general approach is described, the description of the neural network and its parameters is slim. The blog post provides a starting point for the approach of this thesis, where we use a neural network that is trained using a triplet loss network to describe an embedding space that captures semantics of locations. This space can be used for numerous applications. Sentiance uses their embedding space to classify venues by their use. In this thesis, two applications are demonstrated. In the first, the similarity of embedding vectors is used to find and rank similar locations based on a location that is provided by the user. Secondly, the embedding vectors are used as inputs for a machine learning model, showing that they can successfully preserve the semantics of a location.

2.2 Embedding vectors

The input of the deep learning model is a location, which maps it to an *embedding vector*. An embedding vector is a continuous vector representation of discrete variables [17]. In our case, the discrete variables are a location's features and their layout. The model maps these to a vector representation, which is the embedding vector for the location.

Using embedding vectors instead of the complex object they represent has some advantages. Embedding vectors are a lot smaller and simpler than the original objects. This *dimensionality reduction* makes them easier to use and manage. A second benefit is that the use of embedding vectors allows us to create an *informed mapping*. Instead of representing the complex objects by random vectors, we assign vectors in a smart way. Similar objects can be given similar vectors, with a small distance between them. This way, distances between the embedding vectors act as a measure for similarity.

An example is shown in Figure 2.1. It is a possible three-dimensional embedding space for word embeddings. The figure shows the mapping of six words to their

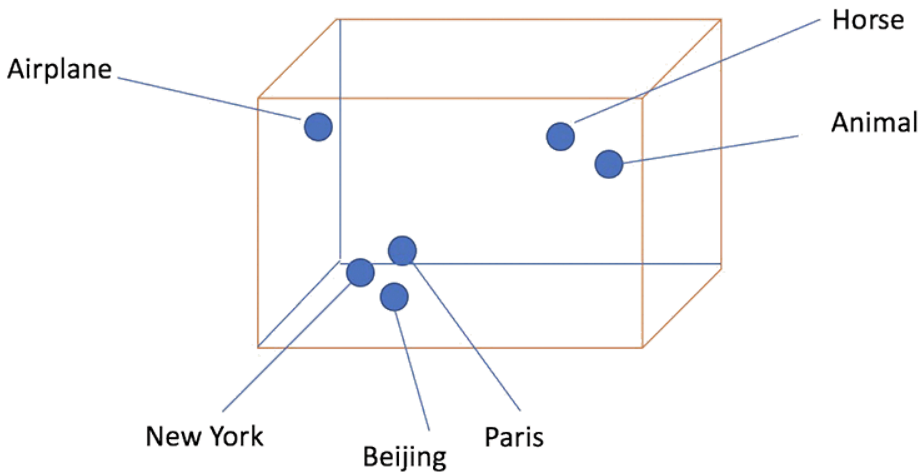

Figure 2.1: Possible three-dimensional embedding space for words [27].

embedding vectors in this space using the original words as labels. Each word is represented by a continuous vector of three numbers, instead of a representation where each word is a discrete variable. This is therefore a dimensionality reduction. Additionally, the embedding vectors are assigned in a smart way, such that similar words are represented by similar vectors. For example, the capitals New York, Beijing and Paris are all close to each other, but relatively far away from the other words, which are not related. The words Animal and Horse are also related and therefore located near each other in the embedding space. The word Airplane is not related to the other words and therefore the distance between its embedding vector and those of the other words is fairly large.

2.3 Artificial neural networks

An *artificial neural network* is a computing concept based on the biological neural networks that can be found in the central nerve system of the human brain. By creating a number of 'neurons' and connecting them using 'synapses', a network is created. In the human body, electrical signals are transferred from neuron to neuron using the synapses. In artificial neural networks, information is propagated from artificial neurons, called *nodes*, to other nodes using *edges*, the artificial version of synapses.

A typical artificial neural network consists of *layers* of nodes. At least two layers are required: the *input layer* and the *output layer*. The input layer of a neural network is a set of nodes that describes the data and serves as input for the network. The input layer is connected to zero or more *hidden* layers. The last hidden layer is connected to the output layer. The output layer produces the result of the computation performed by the neural network, based on the features supplied to the input layer. If there is a single output node, the result is a single number in $\mathbb{R}$, usually in the range $[0, 1]$ or $[-1, 1]$. When there are more output nodes, the output can be viewed as a vector of values in $\mathbb{R}^n$, where n is the number of output nodes.

A neural network node consists of three components: the input function, the activation function and the output. The input of a node is the weighted sum of all its inputs, which are the outputs of nodes in the previous layer. Each edge between nodes of two consecutive layers has a weight which resembles how import the output of the previous layer is for the node in the next layer. The weighted sum is then fed into an *activation function*, which determines the output of the node [7]. In theory, each mathematical function can be used to do this. However, there are some commonly used functions, such as the *sigmoid* function, the *hyperbolic tangent* or, more recently introduced, the *ReLU* [22] and *Leaky ReLU* [18]. The latter is used in this thesis. The activation function maps the input of the node, the weighted sum, to a desired range. This is the output of the node, which is then used as input for the nodes in the next layer. This way, the information is propagated through the network.

The main idea of artificial neural networks is that a network can be *trained* to produce a certain output based on the input. This is done by supplying a *target value* for each set of inputs and adjusting the weights of the network such that its output value gets closer to the target value. By supplying the neural network with many samples of inputs and corresponding target values, the network can 'learn' a function that maps the input samples to the target values by adjusting the weights of the edges.

The most popular algorithm used for training a neural network is the *backpropagation* algorithm [24]. It uses a so called *loss*, which indicates the difference between the target value and the current output value of the network. For each training sample, the algorithm tries to change the weights of the edges in the network in such a way that this loss is minimized. This is done using the *gradient descent* method [29]. By minimizing the loss over all training samples, the network learns what it should output, based on the current inputs.

A common type of artificial neural network is the *feed-forward network* [30], where information only flows forwards, from input layer to output layer, without connections going back. This means that the network has no cycles. A schematic overview of a feed-forward network is shown in Figure 2.2.

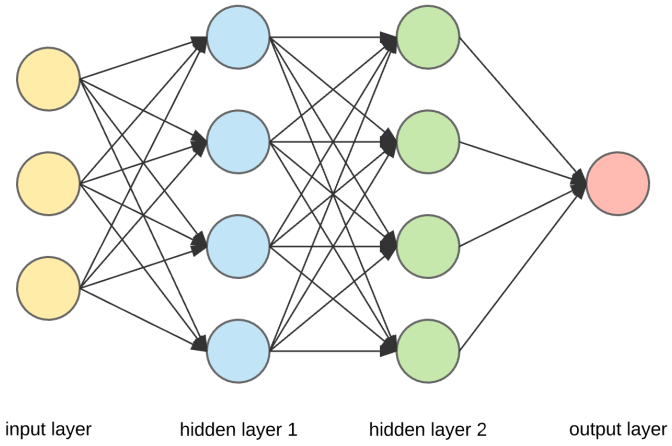

Figure 2.2: A simple feed-forward neural network [8].

In this feed-forward network, there is a single input and output layer, as well as two hidden layers between them. The nodes from the input layer are connected to all nodes of the first hidden layer, therefore the hidden layer is said to be a *dense* layer. The second hidden layer and the output layer are dense layers as well, since every node of the previous layer is connected to all nodes in the next layer. The trainable parameters for a dense layer are the weights of the edges.

Nodes in a neural network layer can be arranged in a special order to represent a certain multidimensional shape. For example, a layer that represents an image can consist of $h \times w \times d$ nodes, each representing a value in a pixel. Here, h and w correspond to the height and width of the image in pixels and d is the number of channels that the image has. We can therefore think of the layers as three dimensional blocks, two-dimensional planes and single-dimensional vectors, illustrated in Figure 2.3.

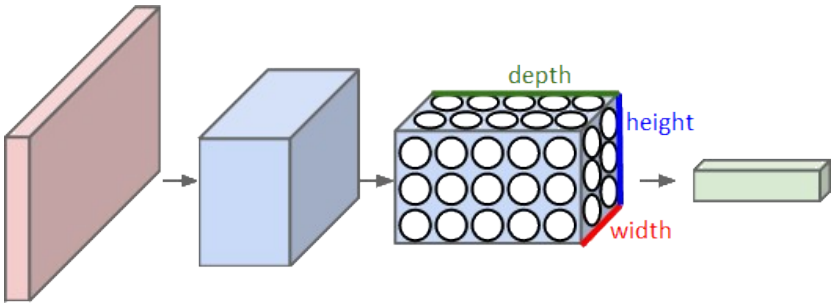

Figure 2.3: A neural network with layers arranged in shapes. Adapted from [16].

A special class of neural network layers are *convolutional* layers. In contrast to dense layers, the nodes of convolutional layers are not connected to all nodes from the previous layer, but to only a few. A convolutional layer consists of *filters*, represented by tensors or matrices. They have the same dimensions as the previous layer, but are generally smaller. Each filter is moved across the height and width of the inputs from the previous layer and at each location the dot product between the values of the inputs and the values of the filter is computed. This generates a *feature map* for each filter, which are then stacked together and an activation function is applied to each entry. This is the output of the convolutional layer. Figure 2.4 shows schematically how a filter is applied to a region of the input, resulting in a value in the feature map of that filter. A neural network with convolutional layers is called a *convolutional neural network*.

A filter in a convolutional layer acts as a *feature detector*. Depending on its values, a filter can react to certain features in the input of the layer, such as edges or straight lines. When the network is trained, the values of the filters are adjusted to detect useful features. Because the values of a filter do not change when it is moved across the inputs, the feature it detects does not depend on the exact location of the feature in the inputs. This makes convolutional layers very useful for image processing. A filter can detect a feature at a location where it had never appeared during training of the network. This property is called *translation invariance*.

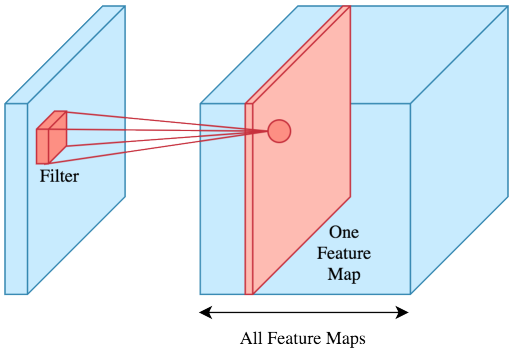

Figure 2.4: Applying a filter to some of the inputs of a convolutional layer [9].

The trainable parameters of a filter are the values in the filter, possibly complemented by the weight of an edge to a bias node. The number of trainable parameters in a convolutional layer therefore depends on the amount and size of the filters. Because many edges between inputs and outputs are shared, this number is very small compared to a dense layer.

Another neural network layer type is the *max-pooling* layer [16]. They are often inserted between convolutional layers to reduce the number of nodes and thereby the complexity of the network. Each max-pooling node is locally connected to a few nodes from the previous layer, but instead of computing the weighted sum, the node simply selects the highest value among its inputs and propagates this value. Since out of all the max-pooling node's inputs only one is propagated, the number of nodes in the next layer is reduced.

The dense, convolutional and max-pooling layers are all used in the convolutional feed-forward network used to implement the location mapping as described in Section 3.3. Its implementation is described in detail in Section 4.2.

2.4 Triplet loss network

The concept of triplet loss networks was proposed by Ailon and Hoffer [2]. It provides a way to train a feed-forward neural network using a dataset that does not contain target values. Instead, it uses a relative measure of similarity between input samples. This is useful when training a network that describes an embedding space, since there are no direct target values available. However, there is a notion of similarity, because 'similar' inputs should have 'similar' output vectors.

In the concept of triplet loss networks, three input samples are processed independently by the feed-forward neural network that is supposed to be trained. These input samples can be denoted as a triplet (x, x^+, x^-), where:

- The *anchor* instance of the triplet x.
- The *positive* instance of the triplet x^+, which is 'similar' to x.
- The *negative* instance of the triplet x^-, which is 'different' from x.

The positive instance x^+ represents a sample that should be close to the anchor x in the embedding space. This is often a slight variation on the anchor instance, making it similar but not equal. When working with images, this variation is often a rotation, shift, change in brightness or sheer transformation. In natural language processing, this might be a shortened or extended version of the anchor sentence and in audio processing the timing, speed and pitch of a sample can be varied to create a positive instance.

In contrast, the negative instance x^- is selected or created to be very different from the anchor x. The goal when training the network is to increase the distance between x and x^- in the embedding space. x^- can be carefully selected from the set of anchors samples or can be created specifically to be different from the anchor.

This is the main idea of triplet loss networks. Section 3.4 describes how this concept can be applied to our location mapping, Section 4.3 describes triplet loss networks in more detail.

2.5 Dimensionality reduction

Neural network outputs are often vectors in high-dimensional space $\mathbb{R}^n$. While single-, two- or three-dimensional vectors can be visualized relatively easily, this gets harder and more difficult to understand for larger values of n. To visualize vectors with a high dimensionality more easily, we can apply a dimensionality reduction algorithm, which maps the high-dimensional vectors to low-dimensional vectors.

One popular algorithm to do this is *t-Distributed Stochastic Neighbor Embedding (t-SNE)* [19]. t-SNE is a variation of *Stochastic Neighbor Embedding (SNE)* [15] that models high-dimensional vectors by a low-dimensional vector in $\mathbb{R}^2$ or $\mathbb{R}^3$. It does this as follows. First, a probability distribution is constructed over all pairs of high-dimensional vectors. Vectors that are close to each other in the vector space have a high probability to be selected, while distant vectors are unlikely to be selected. In the second step, t-SNE creates another probability distribution over the low-dimensional vectors. It then modifies these vectors to minimize the relative entropy between both distributions. This way, the low-dimensional vector space is aligned with the high-dimensional vector space.

3 Overview of our approach

This section describes a general overview of the approach. First, the OpenStreetMap dataset is introduced which provides the data source for the locations. Then, a module is described that converts the objects from the database to location tensors, which can be used by the artificial neural network that implements the mapping function. This network, introduced in Section 3.3, maps a location to a vector in the embedding space. However, this network needs to be trained for this task. The fourth part explains how we do this. The last section, Section 3.5, describes how the trained embedding networks are evaluated.

3.1 OpenStreetMap dataset

In the past, it was hard for end users to obtain geographical data. The agencies that gathered these data were often governmental or commercial. While they did provide data, a high fee and a restrictive licence was demanded in return. Therefore, access to geographical data was limited to a small group of users. In 2004, a project was started to challenge this business model, called the *OpenStreetMap* project. The goal of the project is to provide geographical data to anyone in the world, without any fees and with a very open licence [14, 25]. The project is a success. Many parts of the world have been mapped by volunteers in great detail.

The input of the embedding network is a tensor of features that represents a geographical location. This information is retrieved from a relational database containing OpenStreetMap data obtained from GeoFabrik [12], limited to geographical objects within The Netherlands. Even though it is a small country, this dataset contains over 24 million objects. Table 3.1 shows an overview of the objects in the dataset, ordered by shape type.

Shape	# Objects	Example objects
Points	9.8×10^6	Venues, schools, pubs, shops
Lines	2.1×10^6	Roads, railways, rivers, country borders
Polygons	12.3×10^6	Buildings, forests, grass, lakes

Table 3.1: Summary of the OpenStreetMap extract of The Netherlands.

For each object, the database stores a unique identifier, shape, list of attributes and coordinates. The field with coordinates can be a single (x, y) point, but can also be a list of points, depending on the shape of the object. Locations of point objects are described by a single coordinate, but line and polygon objects have multiple coordinates. An extract from the database is shown in Table 3.2.

ID	Shape	Attributes	Coordinates
295362	Line	path, footway	$(52.080, 5.134), (52.081, 5.126)$
150882	Point	traffic_signals	$(52.081, 5.123)$
691065	Polygon	tennis_court, sports	$(52.080, 5.101), (52.080, 5.100), \ldots$

Table 3.2: Three database entries from the OpenStreetMap dataset.

3.2 Location tensors generation

In this thesis, location tensors of OpenStreetMap objects are used as inputs for the deep learning model. To convert a location to a tensor, a custom *tensor generation module* is implemented.

As input, the module uses a pair of coordinates that specify a point within The Netherlands, which is used to compute the boundaries of the corresponding geographical location. With this bounding box, the module uses a *rendering engine* to convert the location into images. The rendering engine queries the database with OpenStreetMap data for the objects that appear in the location. It converts these objects into image tiles and returns these to the module which converts the images to a three-dimensional tensor. An overview of this process is shown in Figure 3.1. Section 4.1 describes in detail how the tensor generation module works.

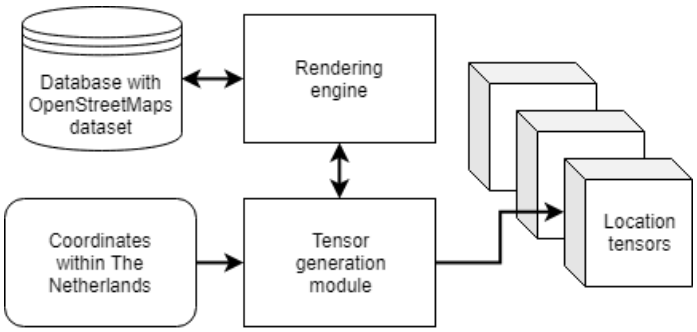

Figure 3.1: Converting a location specified by coordinates to a tensor.

3.3 Mapping locations to vectors

The generated location tensors are used as inputs for the convolutional feed-forward neural network introduced in Section 2.3. The network consists of convolutional, max-pooling and dense layers. The output of the network is a vector, the *embedding vector*, that belongs to the input location.

The convolutional neural network we train maps locations to embedding vectors, with the idea that similar input locations should map to similar embedding vectors. We define similar embedding vectors as vectors that are near each other in the high-dimensional space that they live in. Input locations are similar when they have similar features, but these features should also have similar spatial properties, see Figure 3.2.

Figures 3.2a and 3.2b have similar features and the features have the same spatial properties. Both locations are areas with a lot of grass and long ditches. The embedding network should map these locations to similar embedding vectors. Figure 3.2c is a location that does not have similar features. There is almost no grass or water, but there are a lot of houses. Therefore, its embedding vector should be very different from the vectors of the first two locations.

(a) Grass and ditches

(b) Similar semantics

(c) Different features

(d) Spatially different

Figure 3.2: Four locations in OpenStreetMap representation.

The location of the last figure, Figure 3.2d, contains the same features as the first two figures. The amount of water and grass is approximately the same. However, the spatial properties are quite different. The water in the first two figures resembles long, parallel ditches, but in the location of Figure 3.2d, the water is a river and a small lake with houses. While the features are similar, their semantics are different. The neural network should generate an embedding vector that is somewhat similar to the first two locations because the features are similar, but at the same time it should be different because the spatial properties are different. Therefore, the embedding vector generated by the neural network should be less similar to the vector of the location in Figure 3.2b, but not as distinct as the vector generated for the location of Figure 3.2c.

3.4 Training using a triplet loss network

Triplet loss networks were introduced briefly in Section 2.4. It is a method to train an artificial neural network, when the network cannot be trained directly. This is the case for our embedding network from Section 3.3, since there are no explicit vectors for the input locations. Instead of using direct targets to train the network, the triplet loss network allows us to use relative measures. Section 4.3 describes the concept of triplet loss networks in more detail.

The input of a triplet loss network is a triplet that consists of an *anchor* instance, a *positive* instance and a *negative* instance. The positive instance represents a sample that is 'similar' to the anchor instance, the negative instance should represent a sample that is 'different' from the anchor. Each of these instances is processed by the neural network that is being trained. In our case, the triplet instances are tensors that represent geographical locations.

Anchor locations are selected based on random points within The Netherlands. For each anchor location, a positive and negative location is selected. The locations are converted into tensors using the tensor generation module introduced in Section 3.2. This way, many triplets are created for the triplet loss network that trains the embedding network. Section 4.4 describes in detail how the anchor, positive and negative instances for the triplets are generated.

The triplets are processed by the embedding network and combined by a loss function. Section 4.5 introduces two loss functions: the *margin loss* and *SoftPN loss*. The embedding network is trained twice using two triplet loss networks, once using the margin loss and once using the SoftPN loss function. This results in two copies of the embedding network that have been trained differently.

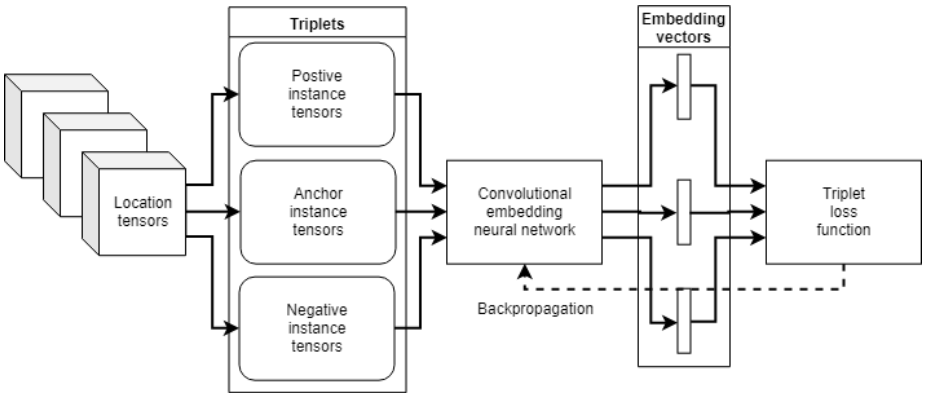

Figure 3.3: Training the embedding networks using a triplet loss network.

An overview of the training process for one embedding network is shown in Figure 3.3. Section 4.6 describes the technical setup and implementation, including the software and hardware used.

3.5 Evaluation

After the copies of the embedding network have been trained using the triplet loss network with the two loss functions, they are evaluated in Section 5. Similar locations should be mapped to similar vectors and different locations should result in two distant embedding vectors. Unfortunately, it is hard to measure the quality of the embedding vectors generated by the networks, since there is no objective way to define 'semantic similarity'. However, it is possible to subjectively check the results using intuition and human knowledge about locations, something the embedding networks should have learned.

In Section 5.1 the quality of the embeddings is assessed using visual inspection. The dimensionality reduction algorithm from Section 2.5 is applied to the embedding vectors generated by the embedding networks. This way, the vectors can be reduced to two dimensions, making it possible to visualize them. By plotting the locations by their vector in the lower dimensional space, the locations embeddings can be compared visually.

An important requirement of the location mapping is that spatial properties should be preserved. In Section 5.2 we check whether this is the case. Given a location, we can find locations that are similar according to their vectors generated by the embedding network that was trained using the SoftPN loss function. These locations should have similar spatial properties. Additionally, we try to find similar locations based on their frequency distribution of features. These similarities are independent on the spatial properties of the features and should therefore yield different results.

Next, we take a look at how useful the embedding networks are. The embedding vectors that they generate should contain enough information about the locations that they represent such that they can be used in practical applications. In Section 6, two applications of the embedding vectors are explored.

In Section 6.1, the vectors that the embedding networks compute are used to find similar locations, based on an input location. The input of the application is a location, of which the embedding vector is computed. Then, the distance from this vector is computed to a large number of known embedding vectors. The vectors that are closest in the embedding space are returned. The locations that they represent should be similar to the input location. By comparing these locations, we can assess whether distances between the vectors in the embedding space provide a good measure of similarity of the actual locations.

As a second application, the embedding vectors are used as an input to a machine learning algorithm. In Section 6.2, a neural network is trained to predict whether a location represents a residential area or not, using as an input the vectors generated by the embedding networks alone. This is possible only if the embedding vectors contain information about housing and other residential features.

4 Experimental setup

Section 3 has provided a general overview of the approach. In this section, the experimental setup is described, including technical details on how location tensors can be generated in Section 4.1. The convolutional embedding network architecture is described in Section 4.2. The next section details how the triplet loss network works and how it can train the embedding networks. To do this, triplets of anchor, positive and negative instances are required, described in Section 4.4, as well as loss functions to update the embedding networks, described in Section 4.5. Finally, in Section 4.6, the technical details on the software and hardware configuration used to run the triplet loss network are provided.

4.1 Generating location tensors

The locations that are used to train the embedding space are randomly selected. This is done as follows. First, a bounding box of size $h \times w$ of the map of The Netherlands is computed. Points within this bounding box can be represented as a pair (x, y), with $(0, 0)$ representing the most south-west point of the bounding box and (h, w) representing the most north-east point. This is shown in Figure 4.1.

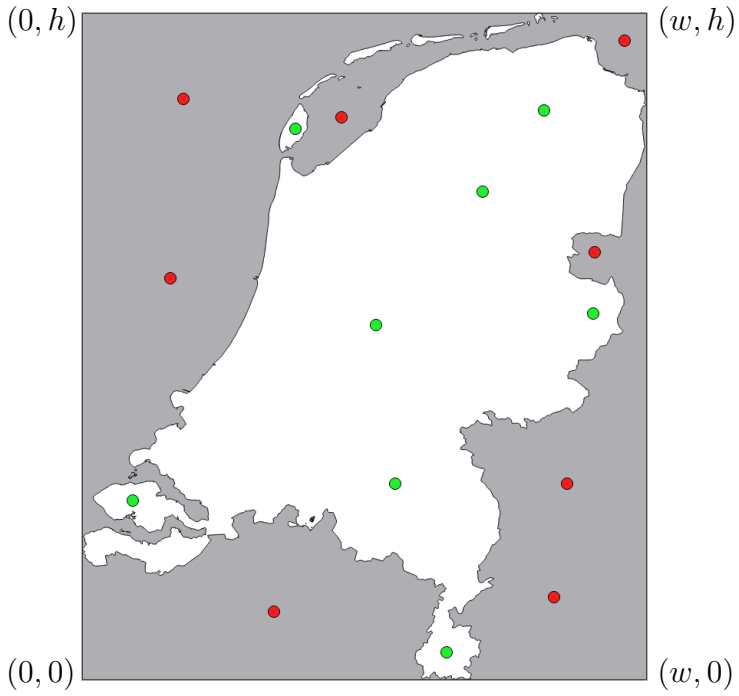

Figure 4.1: The Netherlands (white) within its bounding box (gray rectangle) with valid (green) and invalid (red) points.

Two coordinates, x and y, are sampled from uniform random distributions, $x \sim \mathcal{U}(0, w)$ and $y \sim \mathcal{U}(0, h)$. However, not all points in the bounding box are contained within The Netherlands. Therefore, a sampled point is filtered depending on whether the point is within the boundaries of The Netherlands. Invalid points, outside the boundaries, are discarded. Figure 4.1 shows some valid and invalid points. A valid point is used to generate an anchor location.

A tensor for a location is created using a custom tensor generation module. The input of the module is a valid point within The Netherlands. This coordinate pair specifies the center of a tile with a width and height of 128×128 pixels, which is approximately 275×275 metres in the real world. The OpenStreetMap database from Section 3.1 is queried to retrieve the objects in this tile, their coordinates and their attributes. However, some of the objects in the database are very related. For example, there are objects with the attribute path and footpath, both resembling the same feature. Therefore, attributes are grouped by the module using a manually defined *attribute-feature mapping*, described in Section 4.1.1, creating 136 *tensor features*. These features are used to generate the boolean tensor that represents the location. It has a shape of $128 \times 128 \times 136$, where (x, y, l) is `true` when feature l is present at pixel (x, y) and `false` otherwise. The tensor can be viewed as an image with a height and width of 128 pixels, where each pixel consists of 136 boolean channels.

4.1.1 Attribute-feature mapping

An object in the database has a list of OpenStreetMap attributes that is associated with it. For this thesis, 411 relevant attributes are selected. Each of the 136 tensor features consists of multiple OpenStreetMap attributes. For example, the tensor feature attraction is a combination of the OpenStreetMap attributes attraction, museum, zoo, gallery and theme park. Another tensor feature is parking, which is composed of the attributes parking and parking space. These *attribute-feature-mappings* are visualized in Figure 4.2.

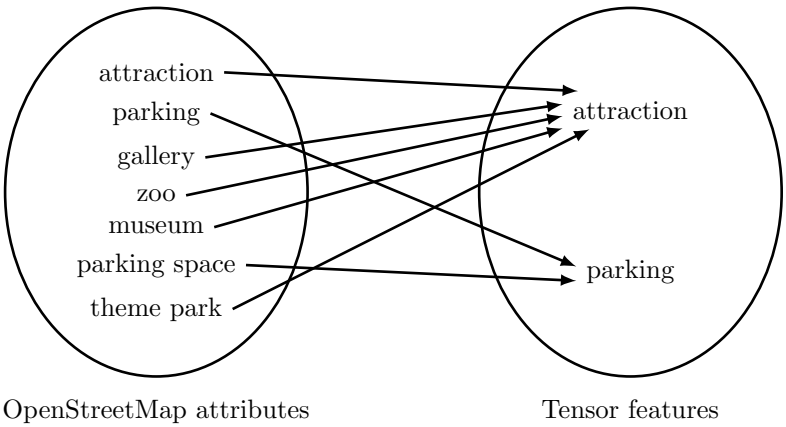

Figure 4.2: Two tensor features composed of seven OpenStreetMap attributes.

The 411 OpenStreetMap attributes that are extracted from the database map to 136 tensor features. The attributes are selected based on the number of appearances in the OpenStreetMap data. Only attributes that appear at least 100 times are used. Therefore, not all OpenStreetMap attributes are mapped to a tensor feature and are not present in any tensor. The attributes that are not mapped are often errors caused by contributors, like fictitious attributes that do not exist in the OpenStreetMap specification, attributes with spelling errors or non-English attributes.

4.1.2 Rendering tensor features

When a location tensor is being created by the tensor generation module, the database needs to be queried to retrieve the objects that describe any of the 136 tensor features in that location. However, the locations of these objects are described by one or multiple coordinates, which need to be converted to a pixel in the tensor. Using a *rendering engine*, the objects of each feature are rendered into a 128×128 binary image, where each pixel indicates the presence or absence of a feature in a pixel. For example, the binary image for the feature motorways, renders only the motorways that appear in the location. When all 136 binary images are rendered, they can be stacked to create the boolean $128 \times 128 \times 136$ location tensor.

However, querying each feature separately and converting them to a binary image is computationally expensive. To reduce the rendering time, multiple non-overlapping features are queried and rendered at the same time. To do this, the features are split into five categories, as listed in Table 4.1.

Category	Features
Roads	All roads: Motorways, residential roads, cycle ways, sidewalks, service roads, etc.
Buildings	All buildings: Industrial and educational buildings, houses, recreational centers, etc.
Residential areas	Areas that are dedicated to housing, opposed to industrial and commercial areas.
Landuse	Terrain usage: grass, forests, playgrounds, water, military terrain, allotments, etc.
Venues	All venues: bus or tram stops, memorials, restaurants, shops, religious venues, schools, etc.

Table 4.1: Feature categories.

These five categories are defined in such a way that the overlap of features in a category is minimized. At most one feature per category is present in each pixel. This way, all features in a category can be queried and rendered at the same time. Since they do not overlap, they do not overwrite one another. Each feature gets rendered in its own color, so they can be distinguished from other colors and converted to a binary plane after rendering. Only five images have to be rendered for each location, one for each category, which is a lot quicker than 136 images. An example is shown in Figure 4.3, where a location's landuse category is rendered.

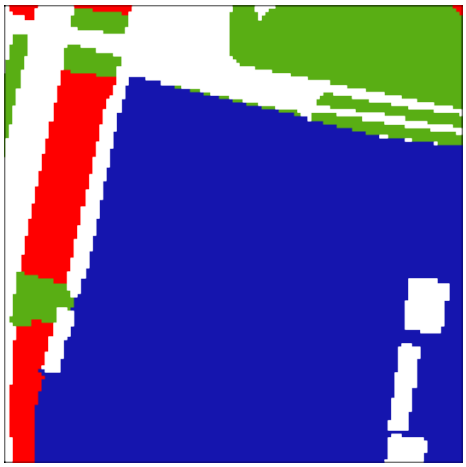

Figure 4.3: A location's landuse render. The three features grass (green), forest (red) and commercial (blue) do not overlap.

This location contains three landuse features, which do not overlap and can therefore be queried and rendered in the image at the same time. Where grass is, there cannot be forest. All features in a category are mutually exclusive. From this image, the presence or absence of these features at a pixel in the location tensor is easily derived by separating the different colors.

By using feature categories, the number of database queries and rendered images for each location is reduced from 136 to 5. Unfortunately, it is not possible to reduce this number further, since there are some combinations of features that can appear in one location at the same pixel. For example, a bakery and commercial area can appear together, which makes it impossible to render them in the same image. Therefore, the five categories are necessary and five renders are required to obtain the features for a location.

4.2 Embedding network configuration

A location tensor is mapped into the embedding space using a mapping function f. Given tensor x as input, the function outputs a vector $f(x)$. In our case, we use an $128 \times 128 \times 136$ boolean tensor of features as input that describes a location, as described in Section 4.1. The function maps this to a vector $f(x) \in \mathbb{R}^{16}$. The choice of a 16-dimensional vector is an arbitrary choice, but it was also used by Sentiance in their venue mapping algorithm [31].

A convolutional feed-forward neural network is used to implement this mapping function. It consists of several convolution layers, each followed by a max-pooling layer. These layers are followed by a fully connected layer which is connected to the output layer. All nodes in the layers use a Leaky ReLU activation function [18], except for the output layer which has a linear activation function. Figure 4.4 shows the embedding network schematically.

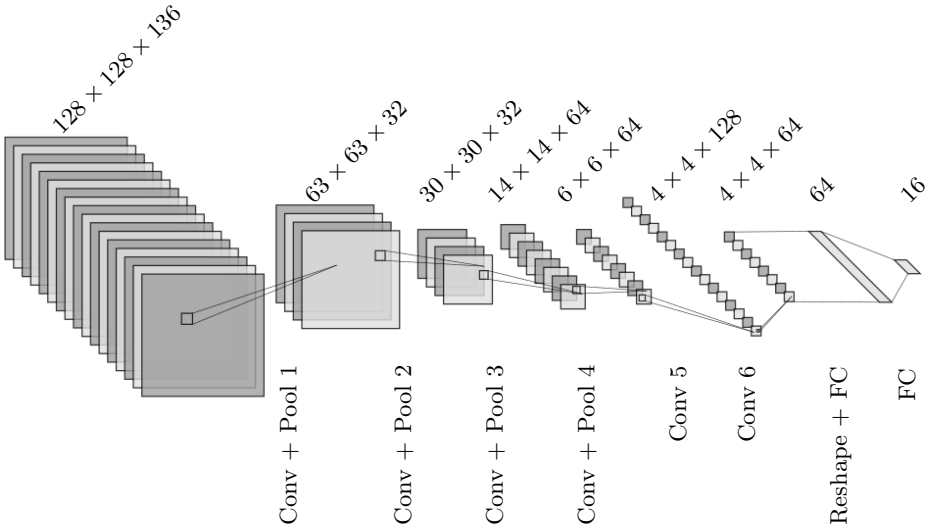

Figure 4.4: Condensed schematic of the embedding network with layers as blocks of nodes, their shapes (top) and applied operations (bottom).

Table 4.2 shows the complete architecture in detail. For each layer, the shape of the input and output tensor is shown. In the case of a convolutional or max-pooling layer, the last dimension specifies the number of filters or strides respectively. The filter size used for the convolutional and max-pooling layers are listed in the fourth column. Convolutional and dense layers have an activation function, which is a Leaky ReLU except for the last dense layer which is the output of the network. The last column shows the number of trainable parameters for each layer.

Layer	Input shape	Output shape	Filter	Act. func.	Params
Conv1	$128 \times 128 \times 136$	$126 \times 126 \times 32$	3×3	L. ReLU	39200
Pooling1	$126 \times 126 \times 32$	$63 \times 63 \times 32$	2×2		0
Conv2	$63 \times 63 \times 32$	$61 \times 61 \times 32$	3×3	L. ReLU	9248
Pooling2	$61 \times 61 \times 32$	$30 \times 30 \times 32$	2×2		0
Conv3	$30 \times 30 \times 32$	$28 \times 28 \times 64$	3×3	L. ReLU	18496
Pooling3	$28 \times 28 \times 64$	$14 \times 14 \times 64$	2×2		0
Conv4	$14 \times 14 \times 64$	$12 \times 12 \times 64$	3×3	L. ReLU	36928
Pooling4	$12 \times 12 \times 64$	$6 \times 6 \times 64$	2×2		0
Conv5	$6 \times 6 \times 64$	$4 \times 4 \times 128$	3×3	L. ReLU	73856
Conv6	$4 \times 4 \times 128$	$4 \times 4 \times 64$	1×1	L. ReLU	8256
Reshape	$4 \times 4 \times 64$	1024			0
FC1	1024	64		L. ReLU	65600
FC2	64	16		Linear	1040

Table 4.2: Embedding network architecture.

The idea behind this architecture that the convolution layers learn distinctive spatial features that are typical for a location. The pooling layer that follows a convolution layer is used to reduce the width and height of the tensor while maintaining a good representation. A fully connected layer combines the output from the last convolution layer. This layer connects to the last layer, which has 16 nodes to reduce the output to $\mathbb{R}^{16}$.

The embedding network is initialized with random weights, using a Glorot uniform initialization scheme [13]. Therefore, the mapping that it represents projects a location tensor to a random vector in $\mathbb{R}^{16}$ initially. The network needs to be trained to produce embedding vectors where similar locations are close to each other, while different locations are further away. One way to do this is using a triplet loss network.

4.3 Training using a triplet loss network

As introduced in Sections 2.4 and 3.4, a triplet loss network can be used to train an artificial neural network on a set of samples that have no clear target value, but where a relative notion of similarity is known among the samples. By providing a triplet (x, x^+, x^-), with an anchor, positive and negative instance respectively, the triplet loss network can train the neural network that is being considered. The positive instance is assumed to be an instance that is 'similar' to the anchor instance and therefore the output vectors should be 'similar' as well. In contrast, the negative instance is assumed to be 'different' from the anchor instance, so the output vectors should be 'different'. The triplet loss networks tries to quantify these properties in a loss function, which is then propagated to the neural network. This way, the neural network is trained to yield similar output vectors for similar inputs and different output vectors for inputs that are different.

Unfortunately, generating location triplets for the training of our embedding network is not easy. While anchor locations can be selected randomly, it is not trivial to select a similar and a non-similar location for the positive and negative instances. Section 4.4 describes our approach to generating triplets.

When a neural network is being trained using a triplet loss network, it computes output vectors $y = f(x)$, $y^+ = f(x^+)$ and $y^- = f(x^-)$ for each of the input samples in a triplet respectively, which combined gives the triplet (y, y^+, y^-). These outputs are used to compute two values: the distance between y and y^+, denoted by $d(y, y^+)$, and the distance between y and y^-, denoted by $d(y, y^-)$. Often, the distance metric used is the Euclidean distance, computed as shown in Equation 1, where $y^a = f(a)$ and $y^b = f(b)$ are output vectors with both n elements. Other distance metrics could be used as well [1], for example the Manhattan distance. In this thesis, the Euclidean distance is used.

$$d(a, b) = ||f(a) - f(b)||_2 = ||y^a - y^b||_2 = \sqrt{\sum_{i=1}^{n} \left(y_i^a - y_i^b\right)^2} \tag{1}$$

The distance measures are combined in a loss function of the triplet loss network, which is used to train the neural network f. The loss is minimized when f minimizes the distance between the anchor and positive instances $d(y, y^+)$, and maximizes the distance between the anchor and the negative instances $d(y, y^-)$. Let Δ^+ denote the anchor-positive distance and Δ^- the anchor-negative distance. Then the loss function is a function $L(\Delta^+, \Delta^-)$ that the triplet loss network tries to minimize, which is used during the backpropagation in the embedding network. There are different approaches to combine the distances in such a loss function. In this thesis, two loss functions are explored, as described in Section 4.5.

The anchor, positive and negative instance are processed by the considered neural network independently. This means that we can implement them in parallel. In that case, three neural networks are used, but they are effectively the same network because their weights are shared. Figure 4.5 shows how the triplets are processed and how their embedding vectors are combined into a single loss value.

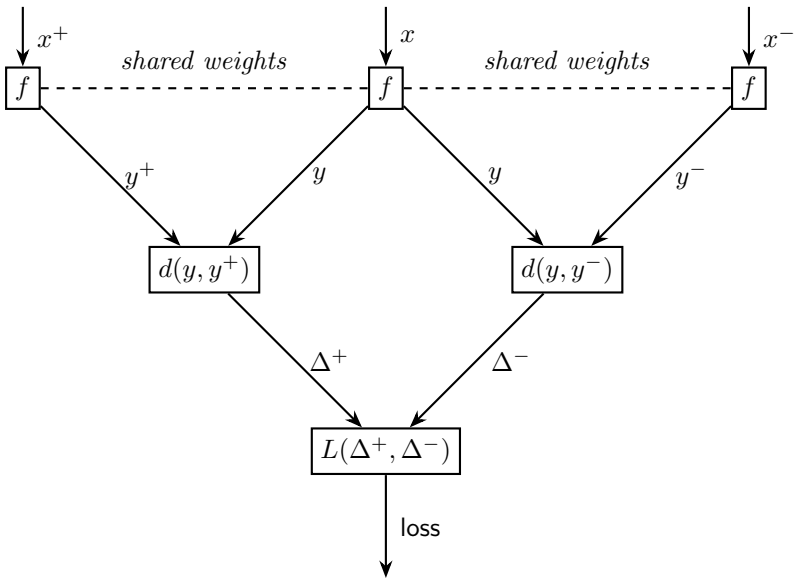

Figure 4.5: Processing triplets and combining them into a single loss value.

In this overview, the triplet (x^+, x, x^-) is processed by the neural network f that is being trained. It computes the output vectors $y^+ = f(x^+)$, $y = f(x)$ and $y^- = f(x^-)$. Then, the Euclidean distance between the output vectors of the anchor-positive pair $\Delta^+ = d(y, y^+)$ and the anchor-negative pair $\Delta^- = d(y, y^-)$ are computed. $L(\Delta^+, \Delta^-)$ is a loss function that combines these distances into a single loss that is used during backpropagation.

4.4 Generating triplets

The triplet loss network from Section 4.3 can be used to train the location embedding network. To do this, triplets of anchors, positives and negative instances are required. The next sections detail how these instances can be obtained and how many were used to train the convolutional embedding networks using the triplet loss network.

4.4.1 Anchor instances

Let x be a $128 \times 128 \times 136$ location tensor that represents a random location within The Netherlands. This is the anchor of a triplet. For training using the triplet loss network, $100,000$ locations were randomly sampled from points within The Netherlands as described in Section 4.1. These 10^5 locations are used as anchor instances x for the triplets.

4.4.2 Positive instances

A positive instance x^+ should have features similar to the corresponding anchor instance. These are locations that are small variations on the anchor instance. Therefore, positive instances can be created by shifting the center of the anchor locations a few metres or by rotating them. This way, x and x^+ are similar and have overlapping features, but the rotation or shift ensures some variation. For each anchor, a positive instance is created by shifting the center the anchor $S^v \sim \mathcal{U}(-80, 80)$ metres vertically (north or south) and then $S^h \sim \mathcal{U}(-80, 80)$ horizontally (east or west). These two shifts are sampled from a continuous uniform distribution. The resulting location is then rotated by $90r$ degrees, with $r \in \{0, 1, 2, 3\}$.

For example, a vertical shift $S^v = 10$, horizontal shift $S^h = -50$ and $r = 2$ results in a positive instance that is created by shifting the anchor location 10 metres to the north, 50 metres to the west and rotating it 180 degrees. For each anchor a, ten positive instances are created this way, resulting in a set of positives $X_a^+ = \{x_{a,1}^+, x_{a,2}^+, \ldots, x_{a,10}^+\}$. Since 10^5 anchors are sampled, the sets of positive instances are X_a^+ with $a \in [0, 10^5]$. Therefore, a total of $10 \cdot 10^5 = 10^6$ positive instances is generated.

4.4.3 Negative instances

Negative instances are locations with features that are different from their corresponding anchor and positive instances. Unfortunately, the locations are not associated with a class, so it is not trivial to select a negative instance x^- for a triplet. Therefore, we make use of an assumption known as Tobler's first law of geography [32]. It tells us that locations that are close to one another are similar, while distant locations are, in general, different. Since the locations are randomly sampled within The Netherlands, it is very likely that a random other location is distant and different. Therefore, the negative instance of a triplet can be generated by selecting a random positive instance. While there

is a probability that the randomly selected location is close and very similar to the anchor and the positive, this is in general not the case and the number of these 'bad' triplets is very slim, which makes them not significantly influence the training process.

4.4.4 Number of triplets

Each triplet consists of an anchor, a positive and a negative instance. There are 10^5 anchor images, with each 10 associated positives. This makes the number of (x, x^+) pairs 10^6. These pairs can be complemented by any of the 10^6 positives that serve as a negative instance. Therefore, $10^5 \cdot 10 \cdot 10^6 = 10^{12}$ (a trillion) distinct triplets can be generated.

4.5 Loss functions

Other than the triplets, the triplet loss network needs a loss function that trains the distances between the instances. There are multiple ways to combine the anchor-positive distance $\Delta^+ = d(y, y^+)$ and the anchor-negative distance $\Delta^- = d(y, y^-)$ in a loss function. We compare the embedding spaces generated by two copies of the embedding network. One of them is trained using a triplet loss network with a *margin loss*, while the other is trained using a *SoftPN loss* function.

4.5.1 Margin loss

The goal of the triplet loss network is to reduce the anchor-positive distance Δ^+ in the embedding space, while increasing the anchor-negative distance Δ^-. The *margin loss* function uses a margin parameter α, which specifies the minimal distance that Δ^- should be greater than Δ^+. Let (x, x^+, x^-) be a single triplet with embedding vectors (y, y^+, y^-), the margin loss L_m is defined as in Equation 2.

$$\begin{aligned} L_m(\Delta^+, \Delta^-) &= \max\left[\Delta^+ - \Delta^- + \alpha, 0\right] \\ &= \max\left[d(y, y^+) - d(y, y^-) + \alpha, 0\right] \end{aligned} \tag{2}$$

First, the difference between the anchor-positive and anchor-negative distances are computed and the margin is added. This is minimized when the anchor-positive distance is pushed to 0 and the anchor-negative distance is as large as possible. Then, the loss is clipped to be at least 0, which is required in the case that $\Delta^- > \Delta^+ + \alpha$. Otherwise, an anchor-negative distance that is already sufficiently large enough would result in a negative loss.

Three types of triplets can be distinguished based on this margin loss definition [21]. *Easy* triplets are triplets where the distance between the anchor and the negative instances is already greater than the distance between the anchor and the positive instances, by at least the margin α. In this case, $\Delta^- \geq \Delta^+ + \alpha$

and the loss will be clipped to 0. Since $L_m = 0$, these triplets do not benefit the training of the embedding network.

Triplets where the negative instance is closer to the anchor than the positive instance are *hard* triplets, so $\Delta^- < \Delta^+$. In this case, the loss is always positive and at least the margin. It contributes to training the embedding network, punishing it for embedding the negative instance closer to the anchor instance than the positive instance, while it should embed it further.

The third triplet type are *semi-hard* triplets. A triplet falls into this category when the positive instance is closer to the anchor instance than the negative instance, but the specified margin is not met. This is the case when $\Delta^+ < \Delta^- < \Delta^+ + \alpha$. Even though the positive-anchor distance is smaller than the negative-anchor distance, the loss is positive. The network is punished since the negative instance is not far enough away from the anchor, or the positive distance is not close enough to the anchor instance.

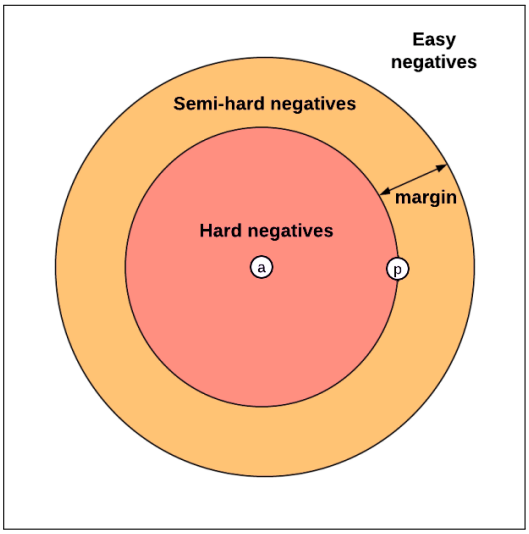

Figure 4.6: Easy, hard and semi-hard triplets. Adapted from [21].

Figure 4.6 shows a schematic overview of the easy, hard and semi-hard triplet types in a two-dimensional space. Here, a represents the anchor, p the positive instance and the colored areas represent the triplet type depending on where the negative instance is located.

4.5.2 SoftPN loss

In their blogpost, Sentiance uses a different loss function [31]. It is a variation on the SoftMax ratio [2]. It uses the *softmax* function, which maps a an arbitrary vector x to a vector y where each element is greater than 0: $y_i > 0$ and the sum of the entries equals 1, so $\sum_i y_i = 1$. For a single triplet and its embedding (y, y^+, y^-), let (Δ^+, Δ^-) be the distance vector. The softmax of the distance vector (S^+, S^-) is then defined as shown in Equation 3.

$$S^+ = \frac{e^{\Delta^+}}{e^{\Delta^+} + e^{\Delta^-}} \text{ and } S^- = \frac{e^{\Delta^-}}{e^{\Delta^+} + e^{\Delta^-}} \tag{3}$$

This results in the softmax distance vector (S^+, S^-). Since $\forall k : e^k > 0$, we get that $e^{\Delta^+} > 0$ and $e^{\Delta^-} > 0$. Then $e^{\Delta^+} + e^{\Delta^-} > e^{\Delta^+}$ and $e^{\Delta^+} + e^{\Delta^-} > e^{\Delta^-}$. Therefore, $0 < S^+ < 1$ and $0 < S^- < 1$ hold. Also, the elements of the softmax distance vector sum to 1, as shown in Equation 4

$$S^+ + S^- = \frac{e^{\Delta^+}}{e^{\Delta^+} + e^{\Delta^-}} + \frac{e^{\Delta^-}}{e^{\Delta^+} + e^{\Delta^-}} = \frac{e^{\Delta^+} + e^{\Delta^-}}{e^{\Delta^+} + e^{\Delta^-}} = 1 \tag{4}$$

The softmax ratio L_r is computed using the softmax distance vector (S^+, S^-) as defined in Equation 5.

$$L_r(S^+, S^-) = (S^+)^2 + (S^- - 1)^2 \tag{5}$$

L_r is the mean squared error on the softmax distance vector, compared to the $(0, 1)$ vector. It can be used as a loss function to train the embedding network. L_r is minimized when $S^+ \to 0$ and $S^- \to 1$, which is the case when $\Delta^+ \to 0$ and $\Delta^- \to \infty$. These are the properties that we want in our embedding space: the anchor-positive distance is minimized, while the anchor-negative distance is maximized.

However, when using L_r loss, there are also *soft* triplets that do not, or barely, contribute to learning. This is the case when the anchor-positive distance is smaller than the anchor-negative distance. In this case, $\Delta^+ < \Delta^-$, which means that the L_r loss is very small. However, it could still be the case that the positive and the negative are very close. This is illustrated in Figure 4.7. Δ^* denotes the positive-negative distance $d(y^+, y^-)$. Ideally, the Δ^+ is small and both Δ^- and Δ^* are large.

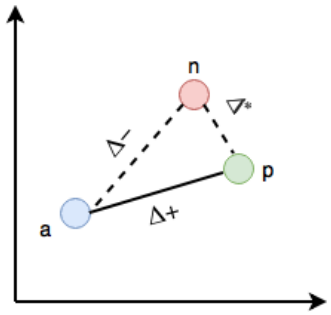

Figure 4.7: 2-dimensional embedding of a triplet with a low L_r loss, even though Δ^* is low. Adapted from [31].

To take into account this positive-negative distance Δ^*, a variation on the softmax ratio was proposed by Balntas et al [3], called the *SoftPN* loss. In the case that the positive-negative distance is smaller than the anchor-negative distance, we compute S^- using the positive-negative distance instead. Effectively, the

roles of the anchor and positive instances are swapped when $\Delta^* < \Delta^-$. The updated softmax distance vector is shown in Equation 6.

$$S_{pn}^+ = \frac{e^{\Delta^+}}{e^{\Delta^+} + e^{\min(\Delta^-, \Delta^*)}} \text{ and } S_{pn}^- = \frac{e^{\min(\Delta^-, \Delta^*)}}{e^{\Delta^+} + e^{\min(\Delta^-, \Delta^*)}} \tag{6}$$

Computing the SoftPN loss L_{pn} is still performed by computing the mean squared error compared to the $(0, 1)$ vector, as illustrated in Equation 7.

$$L_{pn}(S_{pn}^+, S_{pn}^-) = (S_{pn}^+)^2 + (S_{pn}^- - 1)^2 \tag{7}$$

Now, during training of the embedding network, either the anchor or the positive is 'pushed' away from the negative instance, whichever is closer. Still, the anchor and the positive are 'pulled' together. This method fits the triplets quicker in the embedding space and should reduce the convergence of the embedding network [3].

4.6 Technical details

Most components are written from scratch. A tensor generation module, introduced in Section 3.2, is developed to sample the locations, extract their features from the database and converts this information to location tensors. This module is used to produce training triplets automatically. The neural networks and their training are implemented using a library. The code used to assess the quality of the vectors generated by the embedding networks and to build the applications is also custom. This section will cover the software and hardware used to do this, as well as the specific parameters that are used to train the embedding networks.

4.6.1 Software and hardware

The OpenStreetMap data for the location features is retrieved from GeoFabrik [12] and stored in a PostgreSQL database. Sampling location and extracting their features is implemented in Python. The embedding network is implemented in Keras [6], a neural network API for Python. All training was performed on a desktop PC with an AMD Ryzen Threadripper 1950X Processor, 64 GB of RAM memory and an NVIDIA GeForce GTX 1080 Ti GPU. The experiments and applications to assess the embedding network's performance is implemented in Python as well.

4.6.2 Embedding network training

Both embedding networks, one with margin loss, the other with SoftPN loss, are trained using triplet loss networks. As specified in Section 4.4, the anchor instances for the triplets are sampled randomly. For each anchor, 10 positive instances are created. When creating a triplet for training, an anchor and one of its positives are selected randomly. This pair is complemented with a randomly

selected anchor, which serves as a negative instance. This way, triplets are generated 'on the fly'.

Both embedding networks are initialized with random weights using a Glorot uniform initialization scheme [13] and are trained using batches of 128 triplets at a time. Instead of updating the weights in the network after each triplet, they are updated after each batch. This allows all triplets in a batch to be propagated through copies of triplet loss network in parallel. The 128 separate losses are combined to a single loss which is used to update the embedding network weights. The embedding network with the margin loss and the network with the SoftPN loss are both trained using these batches for 48 hours. For the network with margin loss, the margin parameter is set to $\alpha = 0.25$. The SoftPN loss function has no parameters that can be tuned.

After the 48 hours of training, each embedding network had processed approximately 82500 batches, which corresponds to $82500 \times 128 = 1.056 \times 10^7$, or approximately 10 million, triplets. These trained networks and the mappings that they implement are used in Section 5 and 6 to assess the quality of the embedding vectors that they compute.

5 Results

To assess the quality of the trained embedding networks, several experiments are conducted. In the first experiment, dimensionality reduction algorithms are applied to the embedding vectors generated by the embedding networks, which allows visualization of the embedding spaces in two dimensions. In the second experiment, the embedding networks' ability to preserve spatial properties is assessed. This is done by comparing locations that are similar according to their embedding vectors to locations that are similar according to their feature distribution.

5.1 Visualizing embeddings

The embeddings generated by the embedding networks are vectors in $\mathbb{R}^{16}$. To visualize this, we can apply a dimensionality reduction algorithm. One of these algorithms is *t-NSE* [19], introduced in Section 2.5. t-SNE models high-dimensional vectors by low-dimensional vectors.

In our case, the high-dimensional space is the embedding space described by one of the embedding networks. To model this with t-SNE, the embedding vectors of all anchor instances are used. The t-SNE algorithm produces a two-dimensional vector for each anchor, where anchors that were close in the embedding space are also close in the two-dimensional space. Therefore, t-SNE creates a mapping from $\mathbb{R}^{16}$ to $\mathbb{R}^2$, which allows the embeddings to be visualized.

5.1.1 Margin loss

First, the high-dimensional embedding vectors of the anchors are computed using the embedding network with the margin loss function from Section 4.5.1. Then, the anchor vectors in two-dimensional space according to t-SNE are computed. To visualize this space, each anchor is represented by their OpenStreetMap representation, as if the location would have been cut from the OpenStreetMap map. This results in Figure 5.1, which shows all of these representations as small images. Note that many overlap. The horizontal and vertical position of each anchor location is its position in the two-dimensional t-SNE space.

At a first glance, the two-dimensional representation of the embedding space looks decent. There are distinct clusters, which vary in size. The north-west quadrant is mostly occupied by locations with a lot of grass. The locations in the north-east quadrant are mostly farmland. Between these quadrants, there is a reasonably smooth transition: starting in the cluster with grassy locations, moving to the east will increase the amount of farmland until there is no grass left. To the north of the grass and farmland locations, there is a cluster of locations consisting only of water. Since water is the only feature that appears in these locations, there is no transition to other clusters.

In the south-east quadrant, two distinct clusters are shown. More to the east, there is a cluster with inhabited locations. To the east of this cluster, there

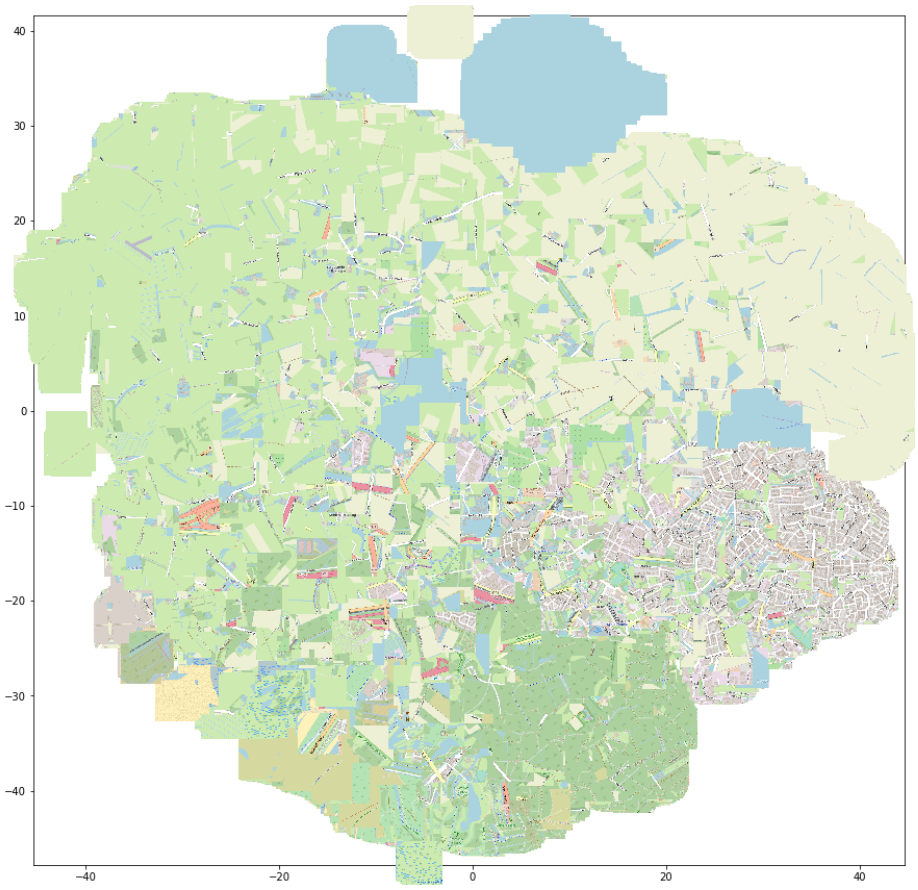

Figure 5.1: Two-dimensional space according to t-SNE of the embedding vectors computed by the embedding network trained using margin loss.

are locations that only consists of roads and residential areas, like city centers. Going to the west of this cluster, more greenery appears and the locations become more rural. The other cluster in the south-east quadrant consists of mostly forestry. Moving towards the center of the t-SNE space, this turns into locations with additional features, like grass, water and roads.

The center of the t-SNE space contains multiple types of locations. The center mainly seems to function as a way to transition between different clusters. It also contains a lot of roads, which can by explained by the fact that roads appear in different types of areas. They appear in residential areas, grasslands, farmlands and can even cross water. Therefore, it is logical that locations with roads are embedded in the embedding space where all types of areas meet.

Other small clusters can be found on the edge of the embedding representation in the south-west quadrant. Along this edge, we can see a group of industrial areas, followed by clusters of cemeteries, sand, swamp and dunes. There does not seem to be a transition among them, although some of these clusters transition fairly smoothly to grassland.

The triplet loss network with margin loss seems to have trained the embedding network fairly well. It is clear that similar locations have been clustered together, while locations that are distinct do usually not appear together. The embedding space often provides a smooth transition between clusters of locations of different features. For example, a location that consists of both grass and farmland appears between the grass and farmland clusters. However, the t-SNE representation does not indicate that this is the case for all clusters and that there might be some hard boundaries between clusters, where no smooth transition is possible.

5.1.2 SoftPN loss

The embedding network with SoftPN loss maps the location tensors differently. The embedding vectors are also computed using this network and t-SNE is applied again to map them to a two-dimensional representation. The resulting visualization is shown in Figure 5.2.

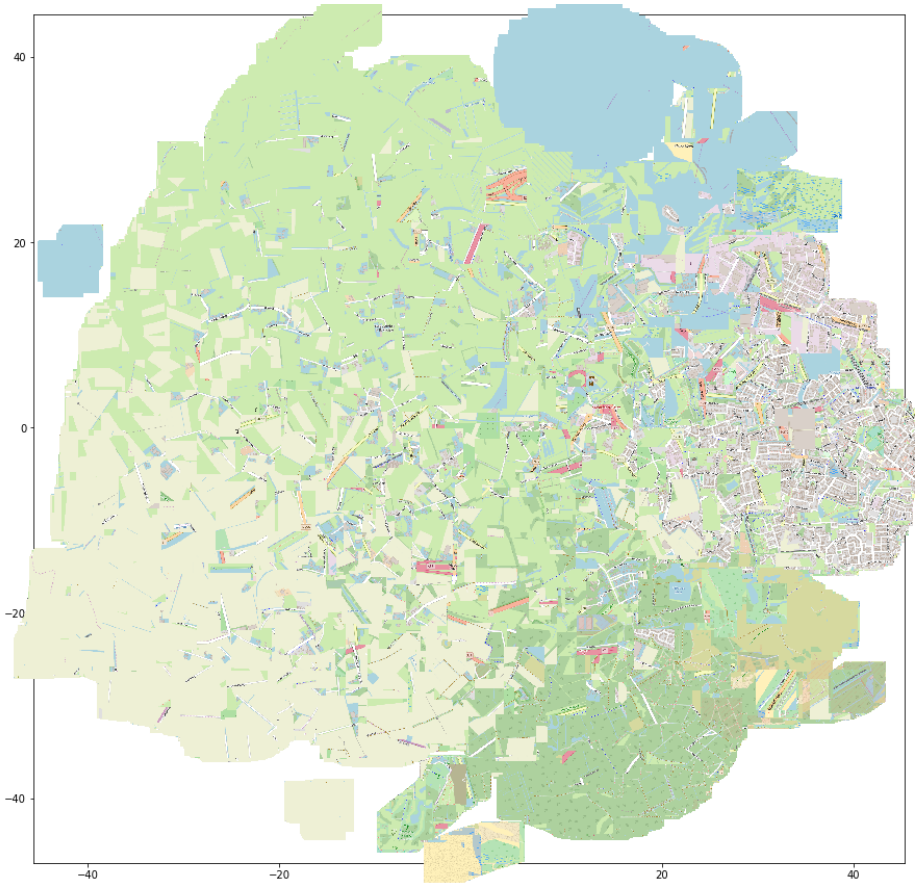

Figure 5.2: Two-dimensional space according to t-SNE of the embedding vectors computed by the embedding network trained using SoftPN loss.

The two-dimensional space of the vectors generated by the embedding network that was trained with SoftPN loss looks somewhat like the one generated by the margin loss network. Again, locations that mostly consist of grass or farmland occupy most of the space. The north-west quadrant is mostly dedicated to grassy areas and moving to the south-west quadrant shows a smooth transition to farmland. This was also the case for the margin embedding space.

The south-east of the SoftPN t-SNE space is occupied by forest. There is a decent transition from this forest cluster to farmland and grass locations. In the margin space, this is not the case. Next to the forest cluster, smaller clusters are present. Again, there is a cluster with cemeteries, sand and swamp without smooth transitions between them. However, there appears to be a transition between swamp and forest.

Where in the margin space the most chaos appeared to be in the center, the most clutter appears in the north-east quadrant of the SoftPN t-SNE space. In the north, there is a large cluster of locations that consist mostly of water. The residential areas are located in the east, where there is somewhat of a transition to grassy areas by going to the west. Between the water and residential clusters, there appears to be a mixture of residential, industrial and aqueous locations.

5.1.3 Comparison

Both two-dimensional representations of the embedding networks trained using the triplet loss network with margin loss and SoftPN loss are decent. Both spaces feature a smooth transition between locations consisting of farmland and grass. Also, both representations show large clusters dedicated to water, forest and residential areas. Smaller clusters are industrial areas, cemeteries, sand and dunes.

There are also differences. In the case of the SoftPN loss, there is a smooth transition between forest and farmland or grass. The margin loss does not show this, as its center is dedicated to transitions between different clusters. Additionally, in the SoftPN representation, the industrial areas are close to the residential areas, with a reasonable transition between them. However, with the margin loss, these clusters appear on opposite sites.

5.2 Preservation of spatial properties

The embedding network architecture consists mostly of convolution and pooling layers, with the idea that these learn to recognize shapes and other spatial properties that characterize a location. This includes for example the distinction between straight and curvy roads. In this experiment, we take a look at some examples to verify whether the spatial properties of the input locations are actually somewhat preserved in the embedding space. We do this by comparing the neighbors of some locations in the embedding space to the neighbors that have the most similar frequency distribution of features.

If the embedding network would take the spatial properties into account, vectors that are close in the embedding space should not only represent locations with the same features, but also the spatial properties of these locations. To test this, we use the anchor locations from the triplets that were used during training. Given one of these locations, the embedding network generates an embedding vector. We compare this to the embedding vectors of all other anchors and generate the OpenStreetMap tiles that correspond to those locations. These can then be visually inspected to compare the similarity in features and their spatial properties.

Additionally, we compare locations without taking the spatial properties into consideration. To do this, we need to represent the locations in a different way, since their tensor representations contain these spatial relations. One way of doing this is by simply counting the occurrences of each feature in the tensor. For each feature, the number of pixels that they appear in is counted. Since there are 136 features, the result is a vector $c = \{c_1, c_2, \ldots, c_{136}\}$ where an element c_i is the number of pixels that contain feature i, or simply the *frequency* of feature i. By creating this frequency distribution, the $128 \times 128 \times 136$ tensor that represents the input location is mapped to a 136-dimensional vector. In this vector, the features and their occurrences are preserved, but their spatial properties are not.

As a distance measure in the embedding spaces and the frequency distributions, the Euclidean distance is used. If this distance between two locations is small, they are considered similar. In the embedding space, this should mean that the locations are similar in the features that appear, but also in their spatial properties. When the distance between two frequency distributions is small, the frequencies of the features that appear in their locations are very similar, but no spatial information is considered.

Figure 5.3 shows an anchor location. This location is characterized by a lot of grass and straight waterways. There is also a bit of track and sidewalk to use for pedestrians. We will use this as the source image and find its neighbors.

Figure 5.3: An anchor location in OpenStreetMap representation.

5.2.1 Embedding network

Figure 5.4 shows the four closest anchor locations of the source location in the embedding space generated using the embedding network that was trained using the SoftPN loss function. They are ranked by distance and the distance in the embedding space is shown below each location as D/Emb.

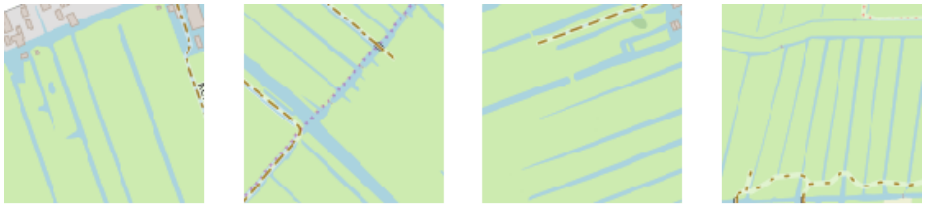

Figure 5.4: Neighbors of the source location in the SoftPN embedding space.

It seems like the neighbors in the embedding space are indeed similar to the source location. All four neighbors show a lot of grass and straight waterways. Additionally, there is some track for pedestrians to walk on. However, there is also a difference. The first neighbor shows some houses in the north, which did not appear in the source location.

5.2.2 Frequency distribution

The source location in Figure 5.3 contains only four features: grass, water, track and a sidewalk. In the frequency distribution of the features, these features have a positive value, all other features are 0. Figure 5.5 is a visual representation of the source's frequency distribution. Only the four occurring features are shown.

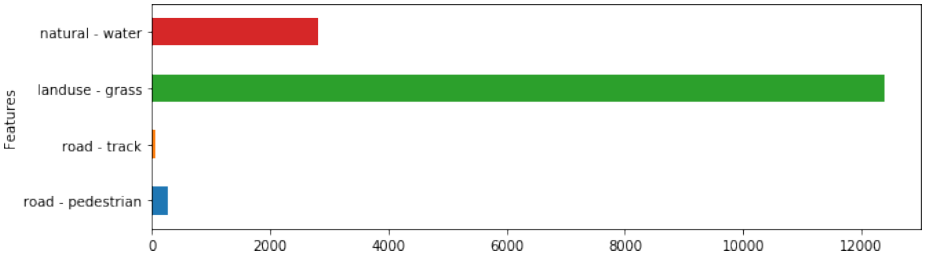

Figure 5.5: Frequency distribution of the source location.

We can now compute the distance between the frequency distribution vector of the source and those of all other anchor locations. Figure 5.6 shows the closest four anchor locations. The location that is ranked first has the most similar frequency distribution. Below each anchor location, the distance to the source's frequency distribution is denoted by D/Freq.

Figure 5.6: Neighbors of the source location by frequency distribution.

The most similar locations by frequency distribution are very different from the locations that were ranked using the SoftPN embedding space. Clearly, the locations are similar to the source location regarding the amount of grass and water. However, the spatial properties of the features are mostly gone. While the source location had straight waterways, the most similar locations do not show these patterns.

This experiment shows that the embedding network successfully maps locations from a $128 \times 128 \times 136$ tensor to a 16 dimensional vector, while maintaining the features and the spatial properties that they have. The most similar locations according to the network have similar features as the input location, comparable to the performance of a 136 dimensional frequency distribution of features. However, the latter is not able to preserve spatial properties, while the network is able to mostly preserve these even though the embedding vector is much smaller.

6 Applications

The two embedding networks that were trained using the triplet loss networks are functions that take a tensor that represents a location and map it to a vector in an embedding space. The goal was to create a mapping where similar locations map to similar vectors. To assess this, two applications were developed that use these embedding vectors as inputs. The first application takes a location as input and finds similar locations using the distance in the embedding space as a similarity measure. In the second application, the embeddings are used as an input to a neural network, which is trained to predict the number of inhabitants that live around the location that the input embedding represents.

6.1 Recommending similar locations

First, an application is developed that can, given a target location, find locations that are similar. Both embedding networks, one trained using margin loss, the other using SoftPN loss, are used independently to generate recommendations. These recommended locations can then be compared.

6.1.1 Approach

The application uses the set of anchor instances from the triplets that were used during training of the embedding networks. For each anchor, the embedding vector was computed using the networks, resulting in a set of 10^5 location vectors for each network. The input, a location specified by the user, is passed through the networks creating two *target vectors*. In each embedding space, a distance metric is used to retrieve the closest n embedding vectors. These suggested locations are ranked by distance in the embedding space and displayed to the user using their OpenStreetMap representation, along with information on their embedding distance and the physical distance.

The similarity of a pair of vectors in the embedding space is measured using the Euclidean distance. This measure is used because it was also used to train the networks. The loss of the triplet loss networks was minimized when the Euclidean distance between the anchor and the positive instances Δ^+ was small and the anchor-negative distance Δ^- (or sometimes the positive-negative distance Δ^* when the SoftPN loss was used) was large.

6.1.2 Results

Consider the location shown in Figure 6.1. This is one of the locations that functions as an anchor in the triplets for the triplet loss network training. The figure shows the OpenStreetMap representation.

We use this image as input for the application to find similar locations. First, the embedding of the input image is computed using the embedding networks. This is the input embedding vector. Then, the Euclidean distance between the input vector and all other embedding vectors are computed. Figure 6.2 shows the

Figure 6.1: An anchor location in OpenStreetMap representation.

closest 8 locations in the embedding space for the embedding network trained using the margin loss. D/Emb denotes the distance in the embedding space to the input location, D/Spat is the spatial distance in the real world.

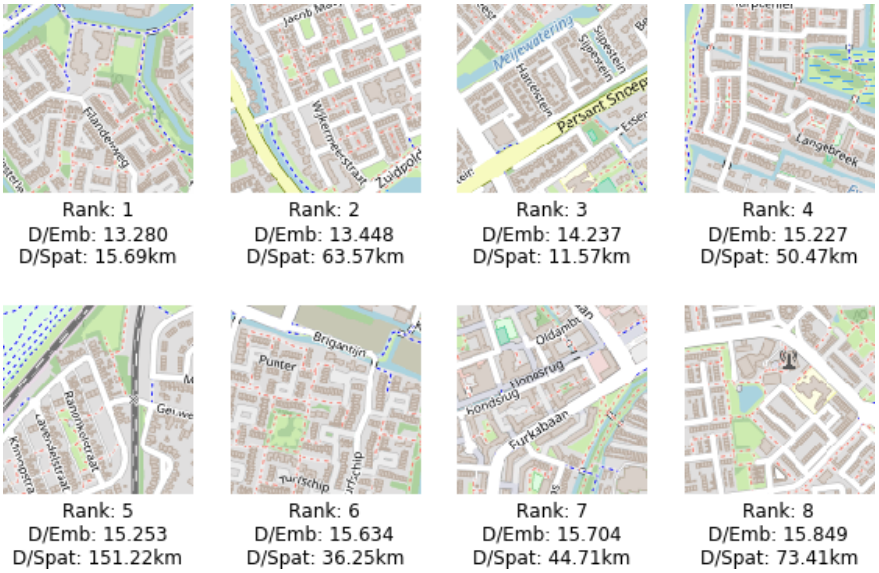

Figure 6.2: Closest 8 locations according to the margin embedding space.

The features in the input locations and the closest 4 locations are very similar. All of these locations contain houses, roads with right angles, some grass and some water. In the embedding spaces, these locations have a pairwise distance of at most 15.227. The locations ranked 5 up to 8 are still similar, but have some different features than appear in the input location. For example the location with rank 5 does not feature any water, but includes some railway.

The real world distances to the input location vary a lot. The closest anchor location in the embedding space, with rank 1 is not the closest location in the real world. This is the location with rank 3. Figure 6.3 shows the real locations of the 8 most similar locations. The green dot is the location of the input location, the red dots show the locations of the 8 closest location in the embedding space. The larger the dot, the lower the rank. There seems to be no relation of the distance or physical locations in relation to the input location.

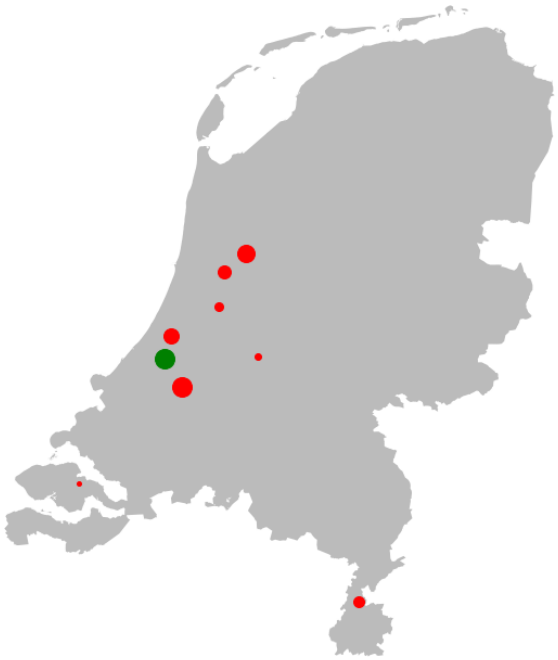

Figure 6.3: Physical location of the closest 8 margin locations.

The recommended places are different for the embedding vectors generated by the embedding network trained with SoftPN loss. The closest 8, ranked by distance, in this embedding space is shown in Figure 6.4.

Figure 6.4: Closest 8 locations according to the SoftPN embedding space.

The most similar locations in the SoftPN embedding space are different from the recommendations from the margin embedding space. The recommended locations are still residential areas, but also show a lot of different features. The location with rank 1 has less houses than the input image and no water, but it is the only recommendation with a communication tower. The other similar locations have similar features to the input location, but in different proportions. The amount of greenery is a lot larger in the locations with rank 6 and 8.

Also a difference is the distance in the embedding space. In the margin embedding space, the most similar location had a distance of 13.280, while for the SoftPN embedding space the distance to the closest location is 3.706. This shows that the embedding spaces have different notions of what is 'close' and what is 'distant'.

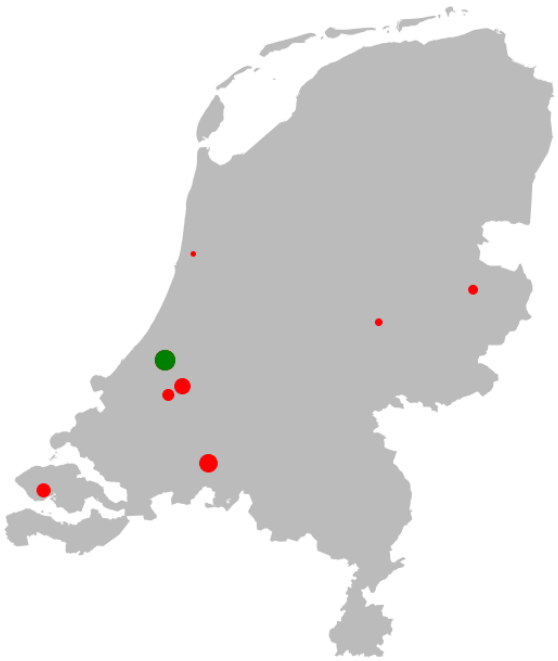

Figure 6.5: Physical location of the closest 8 SoftPN locations.

Again, as Figure 6.5 shows, there does not seem to be a relationship between the input location and the real distance or locations of the 8 closest location in the SoftPN embedding space. Similar locations can be found far away from and close to the input location.

6.1.3 Application summary

In general, the distances in both the embedding space of the embedding network trained with margin loss, as well as the space of the SoftPN trained embedding network, are a measure of similarity. Similar locations live closely together, while the distance between a location pair increases with the amount of features that they differ.

Also, both spaces do not appear to have a bias for physical location. Similar locations are distributed randomly throughout The Netherlands and there does not appear to be a relationship between distances between locations in the embedding space and the real world distances.

6.2 Embedding vectors as machine learning inputs

The embedding vectors computed by the embedding networks are supposed to accurately represent a location. Vectors of similar locations are supposed to be similar and distant from others. This means that they must contain information about the amount of grass, farmland, water, industry and housing. To test this, as a second application, we develop a neural network model with the goal of extracting information about housing from the embedding vectors. The goal of the model is to predict whether a location, given its $\mathbb{R}^{16}$ embedding vector, is residential or not. This is a binary classification problem.

6.2.1 Dataset

The Dutch national statistical office, Statistics Netherlands (CBS), was established to provide independent and reliable information. One of their annual publications is a map of The Netherlands, divided into small areas [4]. Each area is a square of 500×500 metres with information about demography, energy and accessibility of facilities. One of these features is the number of inhabitants, or population, per square. Since people live in residential areas, the population can be used to tell whether a square is residential or not.

The dataset for the classification problem was created as follows. For each anchor and positive location from the triplet loss training set, the corresponding square that this location is located in is determined. The population from this square is used to determine the target t, where $t = 1$ when the population is positive and $t = 0$ otherwise. This generates a dataset consisting of embedding vector-target pairs. Note that not all squares appear in this dataset, because there are squares without any embedding vector locations in them. The locations were randomly sampled within the boundaries of The Netherlands. This also means that a square can appear multiple times in the dataset, which is the case when multiple locations were sampled in that square. Also, not all anchor or positive locations appear in the dataset, since the CBS square dataset does not contain information on all of the sampled locations. The resulting dataset has $1,067,991$ vector-target pairs.

6.2.2 Approach

For both embedding networks, one trained with margin loss, the other with SoftPN loss, a feed-forward network is trained. The input is an embedding vector in $\mathbb{R}^{16}$. The target is a binary value that indicates whether the square that the vector is in is inhabited. The architecture of the networks is simply a stack of 3 fully connected (dense) layers, as shown in Table 6.1.

Layer	Input shape	Output shape	Activation	Params
FC1	16	64	ReLU	1088
FC2	64	32	ReLU	2080
FC3	32	1	Sigmoid	33

Table 6.1: Architecture of the population prediction neural networks.

85% of the anchor and positive locations were used to train the population prediction networks. For each example, the embedding vector was computed according to the embedding networks trained with margin loss and SoftPN loss and fed into the corresponding population prediction networks to train it. The other 15% of locations was used to evaluate the networks. The final prediction of the network is based on the output of the last node, which has a value $0 < p < 1$. When $p > 0.5$, we interpret this as a prediction of $t = 1$ and a prediction of $t = 0$ otherwise. The accuracy A of the network is simply the proportion of correctly predicted targets. When all locations in the test set are correctly predicted to be inhabited or not, the accuracy $A = 1$, which is the goal. When the network predicts all examples incorrectly, $A = 0$. The networks were trained for 30 epochs, with batch sizes of 512 samples.

6.2.3 Results

Before training and after each epoch, the accuracy of the population prediction networks was computed. This is shown in Figure 6.6.

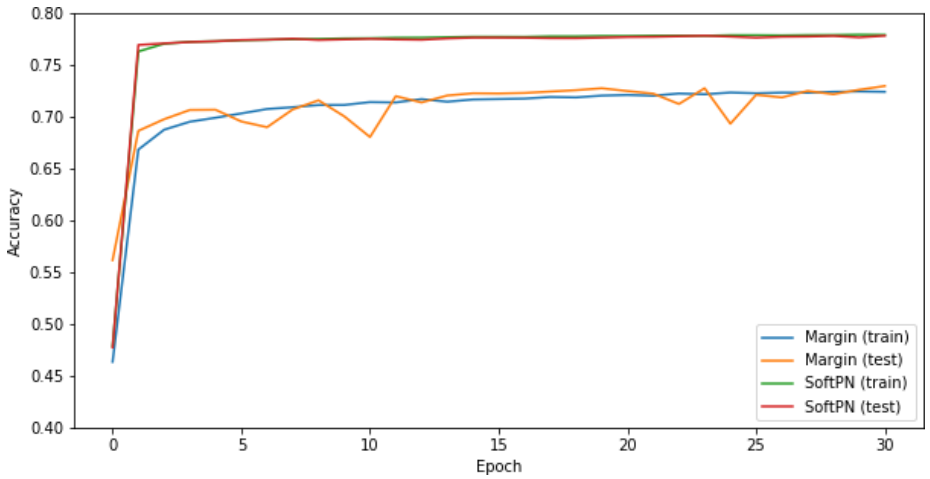

Figure 6.6: Accuracy of the prediction networks at each epoch.

The population prediction network trained using the margin loss embedding vectors has a lower accuracy, of about 73% on the test set. Meanwhile, the network trained using the SoftPN vectors has an accuracy of 78%. Initially, both networks predict randomly, but the accuracy quickly increases, after a single epoch. Then, for both networks, the prediction accuracy hardly increases with the number of epochs.

The lower final accuracy of the network trained with the margin loss embeddings could be explained by the quality of the embedding. The vectors generated by this embedding network may contain less information about residential areas in the locations that it describes. If this is the case, this information is encoded more effectively in the SoftPN embedding vectors.

To see some predictions made by one of the population prediction networks, we can plot some of its predictions. First, we aggregate the predictions by square. When there are no embedding vectors that represent a location in a square, the square is ignored, since there are also no predictions for this square. In case there are multiple embedding vectors for a square, the mean of their predictions is taken. The resulting image, for the network trained with SoftPN embedding vectors, is shown in Figure 6.7.

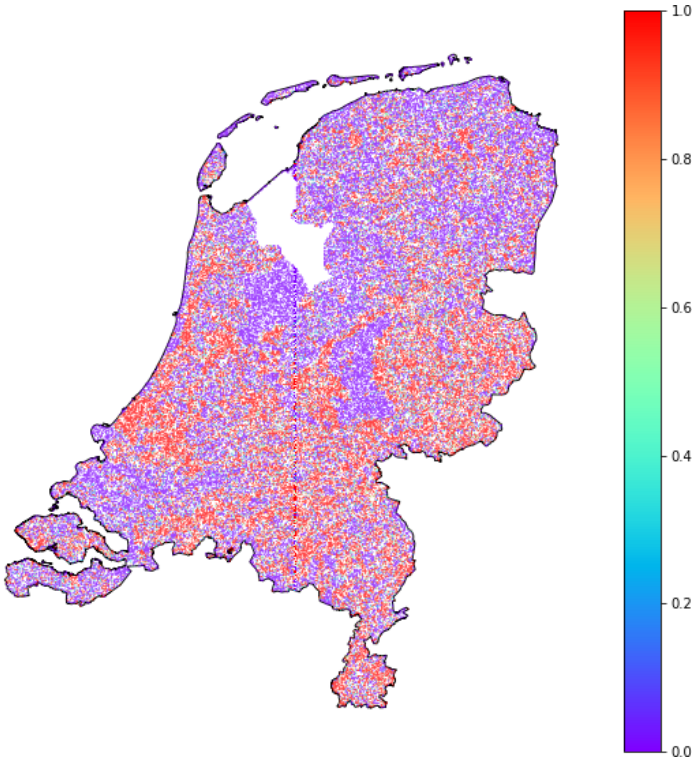

Figure 6.7: Map of predictions by square for the SoftPN prediction network.

The image shows that this population prediction network can predict where people live very well. Large cities, like Amsterdam, Utrecht, The Hague, Rotterdam and Groningen can easily be identified from the predictions. Also places where no people live, nature reserves in Flevoland and the Veluwe are predicted to be free of any population.

6.2.4 Application summary

With results of 73% and 78% on the test sets, it is clear that the vectors generated by the embedding networks contain a vast amount of information about the residential areas in the locations that they describe. However, the SoftPN vectors seem to preserve this information a bit better. From the population network predictions, inhabited areas like cities can easily be identified. It also predicts well that certain areas, like nature reserves, are uninhabited.

7 Discussion

In this section, the research conducted in this thesis is summarized. A recap of the approach is given, followed by the results. Finally, some suggestions are made on how to proceed with further research on this topic.

7.1 Conclusion

Inspired by the application of machine learning models to create embedding spaces for words, graphs, images and other concepts, this thesis shows that a convolutional neural network can be used to map geographical locations to embedding vectors. This mapping meets the requirement that similar locations should be mapped to similar embedding vectors. Since there is no objective measure for describing similarity of geographical locations, the neural network is trained using a triplet loss network, which allows training using relative similarities. Two loss functions for the triplet loss network are explored, each of them generating a different embedding network.

The results show that the two embeddings networks successfully satisfy the requirement. Locations with similar input locations are mapped to similar embedding vectors, while different locations result in distant vectors. This was verified visually by mapping the embedding vectors in two-dimensional space after applying a dimensionality reduction algorithm. The ability of the networks to preserve spatial properties was also shown. Locations were ranked by similarity according to their embedding vectors and using the frequency distribution of their features. While the embedding vector approach was able to take spatial properties into account, the frequency distribution ranking failed to do so.

In the last section, the embedding networks were used in practical applications. First, an application was developed that is able to 'recommend' locations that are similar to an input location. The embedding vector of the input location was used to compute its nearest neighbors in the embedding space, which were then returned to the user. In the second application, the embedding vectors were used as inputs for a machine learning model, with the goal of predicting whether a location is residential or not. This can be done in 73% and 78% of the cases, which shows that the vectors computed by the embedding network can encode vast amounts of information.

7.2 Further research

During design of the embedding network and its training, many arbitrary choices were made. This includes the number of features in the input tensor, the length of the output vector, the number of layers of the embedding network and the margin used by the margin loss triplet loss network. It would be interesting to see whether changing these parameters can increase the embedding network's performance and how this relates to its ability to encode the location's features versus the spatial properties of those features.

Another interesting topic to look into in the future is the use of embeddings network for 'specialized' locations. Instead of selecting random locations, the input data can be specifically selected based on a topic. As an example, the embedding network can be trained on traffic intersections, like roundabouts or roads with traffic lights. Interesting would be to see whether such a network is able to describe similar intersections as similar embedding vectors. Using these vectors, it might be possible to identify dangerous intersections or the vectors can be used for infrastructure planning purposes.

References

[1] Aggarwal, C. C., Hinneburg, A., and Keim, D. A. "On the Surprising Behavior of Distance Metrics in High Dimensional Spaces". In: *Proceedings of the 8th International Conference on Database Theory*. ICDT '01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 420–434.

[2] Ailon, N. and Hoffer, E. "Deep metric learning using Triplet network". In: *CoRR* abs/1412.6622 (2014).

[3] Balntas, V. et al. "PN-Net: Conjoined Triple Deep Network for Learning Local Image Descriptors". In: *CoRR* abs/1601.05030 (2016).

[4] Central Bureau for Statistics (CBS). *Kaart van 500 meter bij 500 meter met statistieken*. 2017. URL: `https://www.cbs.nl/nl-nl/dossier/nederland-regionaal/geografische%20data/kaart-van-500-meter-bij-500-meter-met-statistieken`. In Dutch.

[5] Choi, E. et al. "Multi-layer Representation Learning for Medical Concepts". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*. ACM Press, 2016.

[6] Chollet, F. et al. *Keras*. `https://keras.io`. 2015.

[7] deepai.org. *What is an Activation Function?* 2017. URL: `https://deepai.org/machine-learning-glossary-and-terms/activation-function`.

[8] Dertat, A. *Applied Deep Learning Part 1: Artificial Neural Networks*. 2017. URL: `https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6`.

[9] Dertat, A. *Applied Deep Learning - Part 4: Convolutional Neural Networks*. 2017. URL: `https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2`.

[10] Eisner, B. et al. "emoji2vec: Learning Emoji Representations from their Description". In: *Proceedings of The Fourth International Workshop on Natural Language Processing for Social Media*. Association for Computational Linguistics, 2016.

[11] Garcia-Gasulla, D. et al. "A visual embedding for the unsupervised extraction of abstract semantics". In: *Cognitive Systems Research* 42 (2017), pp. 73–81.

[12] Geofabrik GmbH. *OpenStreetMap Data Extracts*. Karlsruhe, 2018.

[13] Glorot, X. and Bengio, Y. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Teh, Y. W. and Titterington, M. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 2010, pp. 249–256.

[14] Helbich, M. et al. *OpenStreetMap in GIScience Experiences, Research, and Applications*. Springer-Verlag, 2015.

[15] Hinton, G. E. and Roweis, S. T. "Stochastic Neighbor Embedding". In: *Advances in Neural Information Processing Systems 15*. Ed. by Becker, S., Thrun, S., and Obermayer, K. MIT Press, 2003, pp. 857–864.

[16] Karpathy, A. *Convolutional Neural Networks (CNNs / ConvNets)*. 2019. URL: `http://cs231n.github.io/convolutional-networks`.

[17] Koehrsen, W. *Neural Network Embeddings Explained*. 2018. URL: https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526.

[18] Maas, A. L., Hannun, A. Y., and Ng, A. Y. "Rectifier nonlinearities improve neural network acoustic models". In: *Proceedings of the 30th International Conference on Machine Learning*. Atlanta, USA, 2013.

[19] Maaten, L. van der and Hinton, G. E. "Visualizing High Dimensional Data using t-SNE". In: *Journal of Machine Learning Research* 9 (2008), pp. 2579–2605.

[20] Mikolov, T. et al. *Distributed Representations of Words and Phrases and their Compositionality*. 2013.

[21] Moindrot, O. *Triplet Loss and Online Triplet Mining in TensorFlow*. 2018. URL: https://omoindrot.github.io/triplet-loss.

[22] Nair, V. and Hinton, G. E. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *27th International Conference on International Conference on Machine Learning*. ICML'10. Omnipress, 2010, pp. 807–814.

[23] Narayanan, A. et al. *graph2vec: Learning Distributed Representations of Graphs*. 2017.

[24] Nielsen, M. *Neural Networks and Deep Learning*. 2015. URL: http://neuralnetworksanddeeplearning.com/chap2.html.

[25] OpenStreetMap contributors. *OpenStreetMap Map Data*. 2018. URL: www.openstreetmap.org.

[26] Pennington, J., Socher, R., and Manning, C. "Glove: Global Vectors for Word Representation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543.

[27] Ping, D. et al. *Introduction to Amazon SageMaker Object2Vec*. 2018. URL: https://aws.amazon.com/blogs/machine-learning/introduction-to-amazon-sagemaker-object2vec.

[28] Rozemberczki, B. et al. "GEMSEC: Graph Embedding with Self Clustering". In: *CoRR* abs/1802.03997 (2018).

[29] Ruder, S. *An overview of gradient descent optimization algorithms*. 2016. URL: http://ruder.io/optimizing-gradient-descent/.

[30] Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[31] Spruyt, V. *Loc2Vec: Learning location embeddings with triplet-loss networks*. Sentiance, 2018. URL: https://sentiance.com/2018/05/03/venue-mapping.

[32] Tobler, W. R. "A Computer Movie Simulating Urban Growth in the Detroit Region". In: *Economic Geography* 46 (1970), pp. 234–240.