Master Thesis

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Triplet Network for Self-Supervised Location Embedding

March 2019

Author

Raphaël Steinmann

Supervisors

Olivier Verscheure
Executive Director, *Swiss Data Science Center* (SDSC — EPFL)

Jean-Philippe Nantel
TESD Team Director, *Computer Research Institute of Montréal* (CRIM)

Michel Savard
Senior Data Scientist, *Computer Research Institute of Montréal* (CRIM)

Abstract

This work presents a method to generate embeddings that capture the semantic characteristics of geographic locations and their surroundings. To this end, a triplet network is trained in a self-supervised manner and with unlabelled raster data to learn a similarity metric between locations and encode them in vectors of floating point numbers. Combining raster data generated from OpenStreetMap and automated features engineering allows the embeddings to capture complex semantic information hidden in a latent dimension. It is demonstrated that our model can generate meaningful embeddings for tens of thousands of locations all over the province of Québec. Finally, a typical use-case is presented where these embeddings are used as additional features in a properties price prediction problem.

Acknowledgements

This master thesis rules off my academic studies in the School of Computer and Communication Sciences at EPFL.

First of all, I would like to thank Olivier Verscheure, executive director at the *Swiss Data Science Center* (SDSC), for supervising my thesis remotely from Switzerland during the last six months. I also want to express my gratitude to Dr. Radhakrishna Achanta from SDSC for his feedback and advice.

This master thesis was conducted in the *Computer Research Institute of Montréal* (CRIM) in Montréal, QC. I would like to thank Jean-Philippe Nantel, director of the Emerging Technologies and Data Science team for his trust and for giving me the opportunity to carry out my thesis at the CRIM. Special thanks go to Michel Savard, my supervisor at the CRIM for his precious advice and guidance throughout the realization of this work. I also want to express my gratitude to Martin Sotir for his valuable inputs and feedback.

Last but not least I would like to thank my parents, without whom none of this would have been possible.

Contents

1 Introduction

1.1 Context

Why is it that some neighborhoods are more pleasant than others? What are the attributes that make people want to settle down somewhere and why are some areas more sought than others? This is hardly an easy question, as there can be dozens of different explanatory factors for the quality of life in a neighborhood.

Location classification is a very popular topic, both in research and in the industry. In this paper, an approach to encode the semantic attributes of geographic locations in a meaningful representation is proposed. As opposed to classical machine learning models based on categorical attributes and manual features engineering, our approach uses the power of deep learning and automated features engineering to capture information from latent dimensions that define the semantic nature of a place.

1.2 Problem at hand

The aim of this work is to identify the attributes that best represent a location and define its nature, and embed them in a metric space that defines semantic similarity between places. In other words, the goal is to build a neural network that is capable of encoding any location in a vector of floating point numbers.

These embeddings reside in a metric space, which means the distance between two vectors can be used as an indicator of the similarity between the two locations they represent. It is also important that the network captures not only basic features like the distribution of colors on tiles, but also complex semantic information hidden in latent dimensions like the shapes of roads or buildings. This problem relies on informative features to learn a good generalizable model.

1.3 Motivation

There are several motivations for this work. As an applied research center, the CRIM is interested both in research topics and applied problems from the industry. From the research point of view, the CRIM has an interest in building a knowledge base about trending topics such as location embedding and triplet networks. From the industrial point of view, there are many possible applications for location embeddings, as they can easily be used as features in regression or classification models. We could imagine for example a price prediction model for a real estate company or a neighborhood recommender system to help people find the place of residence that suits them the most.

One of the motivations for these embeddings, if we think end product, is that the resulting embedding space is generic, and thus can be used for any application. By not pre-selecting features or targets with a certain task in mind (for example predicting the price of houses or the popularity of restaurants), we ensure that the embeddings remain generic and can be used arbitrarily for any application.

It is worth mentioning that the CRIM is currently contracted by a land title solution company called JLR to develop advanced AI solutions using the data they collected over the years to provide better services. This work has several possibilities of applications for JLR which are discussed in sections 6 and 9.

This work is inspired from a blog post from Sentiance (cf. [Spr18]), a tech and risk profiling company that extracts behavioral insights from smartphone data (accelerometer, gyroscope, location). The blog post explains how Sentiance used a similar solution for the purpose of venue mapping, i.e. figure out the exact location visited by an individual given inaccurate smartphone location data. This article describes what was done in a higher level perspective, however the code is not open source and Sentiance does not share any implementation of its algorithms. The CRIM is highly interested in the precise implementation process and in-depth technical knowledge of a location embedding system, which is why a part of this work is dedicated to *reverse-engineering* Sentiance's article.

1.4 Problem formulation

When we think of a place's semantic context, we think shops, restaurants, public transports, traffic, roads, houses, buildings, parcs, green areas, etc. All of this information is duly documented and accessible on platforms like Google Maps or OpenStreetMap. By looking at the map, all the icons, layers, shapes, contours and colors instantly give a great deal of relevant information and insights on any location. The underlying idea behind this work is to give the network a similar sense of *human intuition* in order to capture and encode this latent information inside the embeddings.

That is the reason why, instead of manually selecting each and every feature that we think might be relevant and approximating them in a structured fashion, we decide to represent the data in a different manner, that is in the form of *tiles*. A tile is a square image directly rasterized from the map (think *screenshot*). This way, all the information that could be relevant is kept as is, and will be embedded by the network.

In order to generate embeddings from tiles, we use a triplet network, which is a neural network derived from the well-known siamese network. A triplet network takes as input triplets of samples and is trained to turn them into meaningful encodings. The first reason behind this choice is that

the embeddings produced by a triplet network reside in a metric space. This concept is detailed in section 3.2. The second reason is that a triplet network can be used in a self-supervised manner. Concretely, this means that as long as triplets can be properly constructed, no labeled data is required. This is particularly indicated for the problem at hand, as our tiles dataset consists of unlabeled data.

2 Datasets

The main dataset used in this work is composed of 71'982 tiles from inhabited areas of the province of Québec. This dataset has been thought, designed and generated in the scope of the work. To have more information on the data generation process, refer to section 5.

Originally, each tile has a size of 256 per 256 pixels and depicts a square area of 1000 per 1000 meters. When processed by the network, the tiles are subject to a number of transformations and eventually result in 128 per 128 pixels and 800 per 800 meters tiles. The details and reasons behind these transformations are explained in section 5.2. The last-mentioned dimensions roughly correspond to a circle of 400 meters radius, which amounts to a five minutes walking distance (as the crow flies) at average walking speed.

Each tile's location corresponds to the coordinates of a property somewhere in Québec, and the centers of each pair of tiles is distant from at least 500 meters. At first, the tiles centers were distant from at least 800 meters in order to completely avoid overlap between tiles. However, due to properties being closer to one another in dense (urban) areas, this resulted in an unbalanced dataset where the vast majority of tiles was from rural areas, and cities like Montréal were highly underrepresented. The locations of properties were provided by JLR, who kindly accepted to share their dataset in the scope of this work. However, note that this dataset is to remain confidential, that is why although it is mentioned in the report, the data itself does not appear anywhere. This dataset is also used later in section 6 as an example of application for the embeddings.

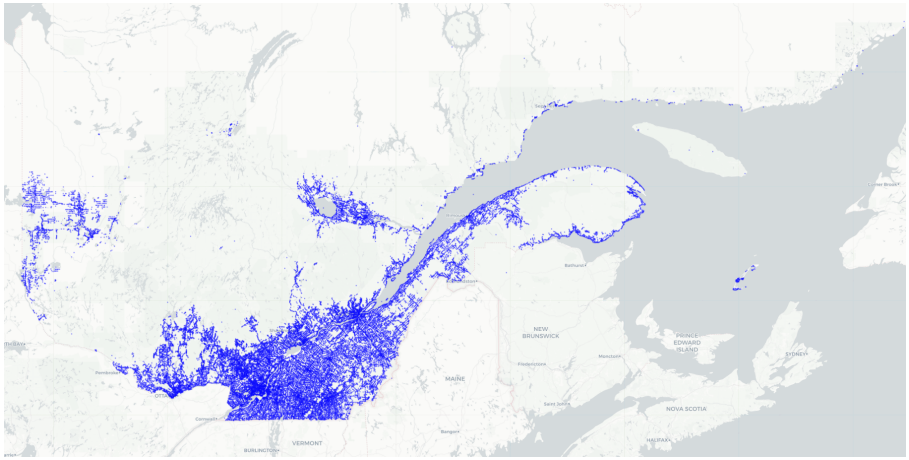

Figure 1: Locations of the tiles. The coordinates of each tile center correspond to the location of a property in the province of Québec.

The tiles dataset has been randomly sampled and split into train, eval and test subsets. Setting a minimum distance between the tile centers prevents any overlapping between neighboring tiles, as shown on figure 3.

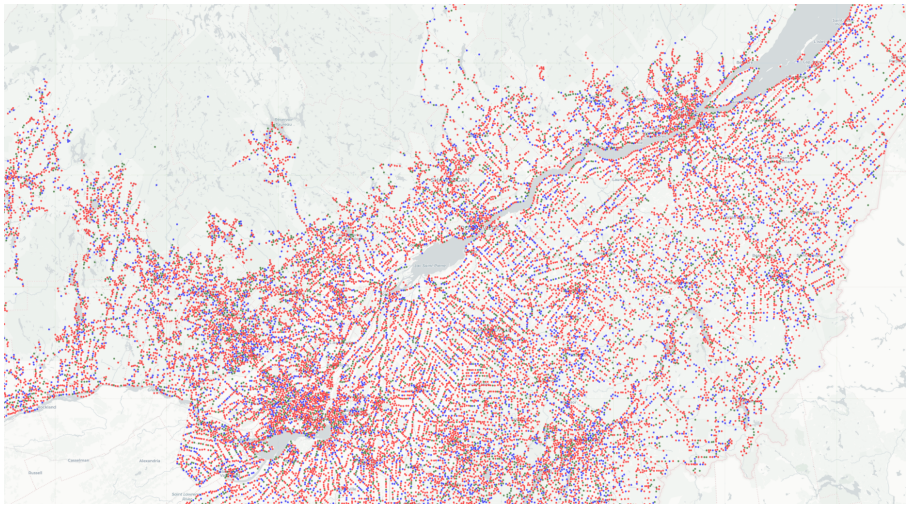

Figure 2: Random sampling of the dataset. Red corresponds to train set, green to eval set and blue to test set.

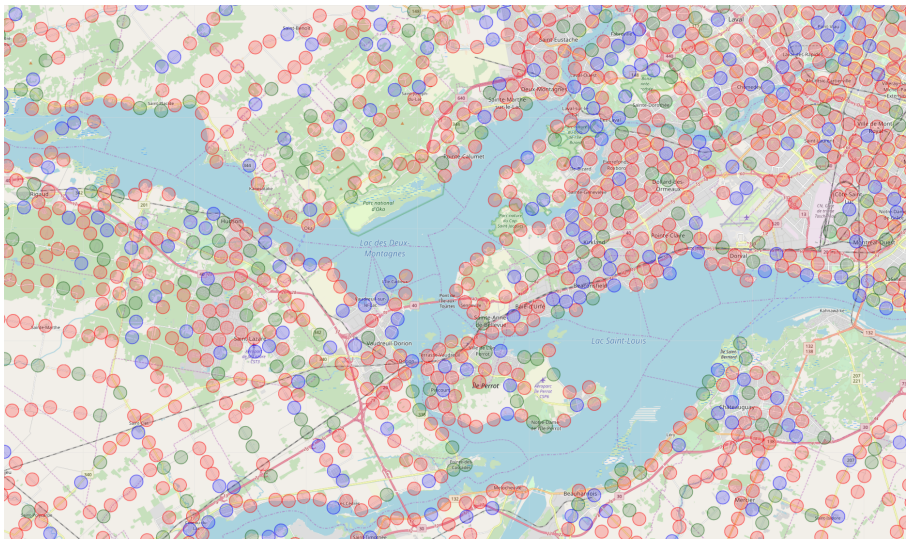

Figure 3: Random sampling of the dataset. Red corresponds to train set, green to eval set and blue to test set.

One important aspect of the tiles dataset is the segmentation of the tiles per layer. This means that instead of a single-layer grayscale image or a 3-layers RGB image, a tile consists of a 10-layers tensor. Each of these layers represents a distinct semantic concept, for example buildings, roads,

green spaces or commercial areas. The different layers are illustrated on figure 5 and described in section 3.1. The point of this segmentation is to help the neural network distinguish the different semantic concepts that he is trying to learn so he can treat each one separately. Note that using a single channel image containing all the information is also a viable solution, however not only do we lose a great deal of information due to all the features of the tile being merged together and overlapping each other, but the task is also a lot harder for the encoder. Indeed, in this configuration the network has no idea what the different shapes, symbols, areas or colors correspond to and it must learn to properly segment the information by itself, which is a fairly complicated work. It is also important that each layer always has the same index in all the tensors to ensure consistency for the network. The process used to achieve layer segmentation is described in section 5.1.4. Note that for the sake of clarity and ease of reading, a tile will often be represented as a single RGB image in the different illustrations present throughout the paper. Yet keep in mind that a tile is actually a tensor composed of not three RGB layers, but ten semantic layers.

Figure 4: Tile segmentation. OSM maps are built in a layered fashion, where each layer contains a distinct type of information (road networks, landcover, buildings, ...). Thanks to this structure, each tile can be represented as a 10-channel tensor where each channel contains a specific type of information.

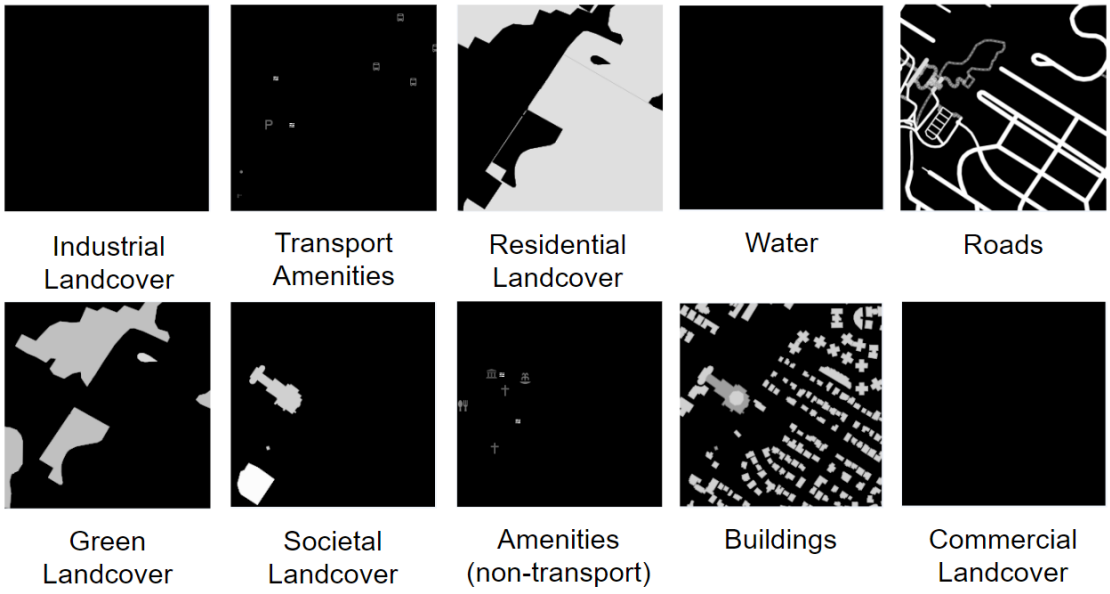

Figure 5: Segmented layers of a tile. Each layer contains a specific type of semantic information. The different information types are described in section 3.1.

Concerning the dataset provided by JLR, it consists of several million transactions of properties that took place between 1981 and 2015. Each transaction comes with a certain number of typical housing features.

3 Location embeddings

The goal of the project is to design a neural network capable of generating an embedding from a location given as input in the form of a tile. This network acts as an encoder, representing the relevant semantic data comprised in a stack of images (the tile) into a vector of floating point numbers.

3.1 Location semantic similarity

So what exactly does *semantic similarity* mean? And what does it mean for two places to be *semantically similar*? We consider that two locations are semantically close when they share similar semantic attributes. These attributes can be divided in several classes, in a similar manner as the tiles are segmented in distinct semantic layers. The following subsections give an overview of the main classes of attributes taken into account, to help have a better understanding of the information we attempt to capture.

In essence, the semantic nature of a place boils down to the different attributes that can be found there. Two tiles are considered to be semantically similar when they share a certain number of similar attributes among those mentioned in the sections below. It is also important to keep in mind that it is not essentially the presence of certain features that define the semantic nature of a place, but also their density, shape, repartition, etc. As mentioned before, the underlying assumption behind the choice of feeding the network with rasterized tiles instead of structured data is that there is a latent dimension in these images that reflects a sense of human intuition.

3.1.1 Amenities

Amenities are described by OpenStreetMap as an assortment of community facilities. This includes sustenance amenities (bars, pubs, restaurants, cafes, food courts), education amenities (schools, colleges, universities, libraries), financial amenities (banks, ATMs), healthcare amenities (clinics, hospitals, pharmacies), entertainment and culture amenities (cinemas, casinos, museums, theatres).

Each amenity is generally represented on the map by an icon of distinct shape and color. Therefore, even if they share the same tile layer, two distinct amenities might still be perceived and processed differently by the network.

3.1.2 Transport amenities

The transport amenities basically represent all the amenities related to public transports, like bus or subway stations but also parkings, bicycle rental terminals, taxis stations or gas stations. Public transports serving in a location is a typical example of the type of semantic information

we wish to capture. Indeed, it is the kind of factor that shape the nature of a place.

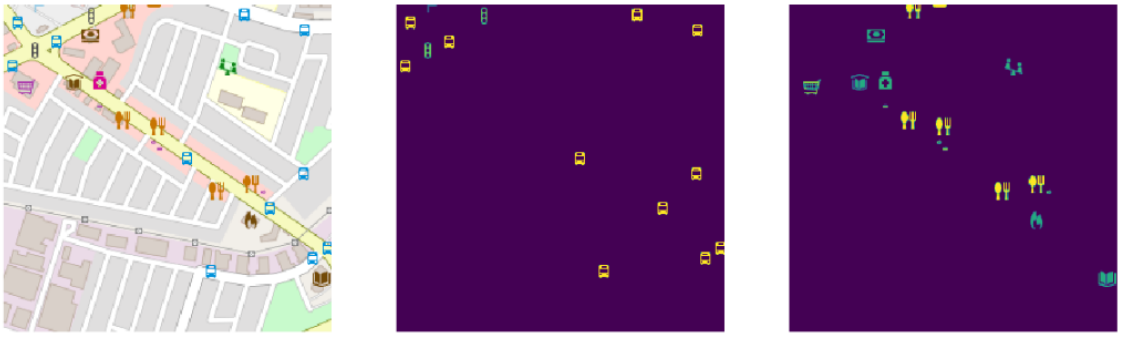

Figure 6: Left: original tile (RGB). **Center**: transport amenities layer. **Right**: amenities layer.

3.1.3 Landcovers

A landcover is basically a portion of land that can be attributed to one of several categories. Among these categories are green areas (parcs, forests, nature), water (lakes, oceans, sees, rivers, ponds, ...), residential areas, industrial areas, commercial areas or societal areas (libraries, hospitals, churches, ...). The presence of one or more of the aforementioned areas in a location also influences its semantic nature.

3.1.4 Buildings

Buildings also play an important role in the task of defining a place. Indeed, both the quantity and the shapes of the buildings can tell a lot about their location. The density and the disposition of the buildings in a neighborhood are equally important, as they can provide for instance information about the type of accommodations people live in. Unfortunately, as OpenStreetMap is open-source and constantly updated, buildings are not always present in the data. An example of well-documented buildings layer can be observed on figure 4.

3.1.5 Roads

In a similar way, the presence of roads on a tile influences its nature. In particular, the size of the roads (highways, national roads, driveways, ...), their shape and the proximity between them give some valuable information. For instance, a small residential road with many junctions, leading to a traffic roundabout surrounded by properties and a long straight road reflect very different semantic information, as illustrated on figure 7.

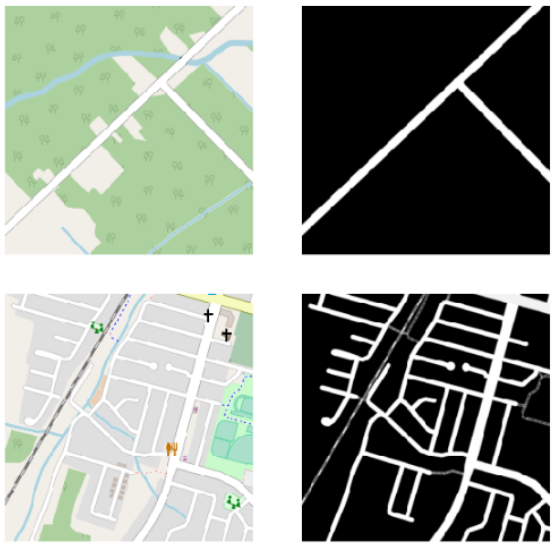

Figure 7: Road network layer. Different road patterns reflect different semantic information. As discussed in section 3.3, capturing this kind of information requires automatic feature selection through deep learning and unstructured raster data.

3.2 Metric space

As stated before, the network produces embeddings of the tiles, and therefore acts as an encoder. Due to the nature of the triplet loss (the triplet loss uses the L2 norm as a metric to compute the distance between two embeddings, see section 4 for implementation details), these embeddings reside in a euclidean metric space. The network aims to learn a similarity space where two tiles that are semantically akin produce two embeddings that are close in the metric space, whereas the embeddings of two different tiles will be far away from each other.

One interesting property of this metric space is that the embeddings respect the basic operations of algebra, like addition or subtraction. Therefore, adding up two encoded vectors produces a resulting vector that shares the properties of both of them. Several examples of algebraic operations on the embeddings are presented in the discussion (see section 8.2). A comparison can be established with the Word2Vec model by Mikolov&al. In the same way word embeddings can be added or subtracted (think for example *king - man + woman = queen*), the embeddings of two tiles can be added in order to produce an new vector that corresponds to a tile sharing the attributes of both tiles.

However, note that the network acts as an encoder but not as a decoder, which means it cannot turn a embedding back into a tile. Therefore, finding the tile corresponding to the result of an addition or a subtraction is performed by picking the embedding that is the closest to the resulting vector in the metric space and returning its associated tile.

3.3 Manual vs automatic feature selection

As described before, the data consists of tiles (images) rasterized from OpenStreetMap. One could argue that structured data would be more appropriate, more convenient to extract and store and easier for the network to interpret. Indeed, we could imagine for instance extracting a list of the attributes present on a tile and computing their distance to the center of tile or simply boolean values indicating their presence.

However this schema boils down to manual feature selection and a lot of features, such as the shape of a road or the disposition of buildings would not be captured. We believe that many features from a latent dimension would be missed with categorical data. This problem is overcome thanks to the automatic feature selection properties of deep neural networks. By feeding it with a visual representation of a place and all its the attributes in the form of an image, we give the network the freedom to consider all kind of information and determine what is important or not.

Furthermore, segmenting the tiles into distinct semantic layers improves the network's capacity to differentiate the nature of each attribute. The segmentation also prevents a loss of information due to the great number of elements that usually overlap on a map.

4 Triplet networks

4.1 Definition

The structure of the triplet network is inspired by the well-known siamese network architecture originally presented in [Bro+94]. It consists of three instances of the same network, sharing the same parameters whereas the siamese network only has two. Let us first define the siamese network to better understand the triplet network.

4.1.1 The siamese network and contrastive loss

The siamese network aims at quantifying similarity between two samples of similar or different nature. It takes as input pairs of samples, encodes them both and outputs a distance between the two encodings. The goal of the network is to minimize this distance if the two input samples have the same label, or to maximize it if they have different labels. This is usually done with a contrastive loss function:

$$\ell = (1 - Y)\frac{1}{2}D_w^2 + Y\frac{1}{2}[max(0, m - D_w)]^2$$

Where D_w is the euclidean distance between the two encodings and Y is a binary value set to 0 if the two inputs have the same label, 1 otherwise. m is a margin value meant to ignore pairs that have different labels, but the network already considers as fairly dissimilar.

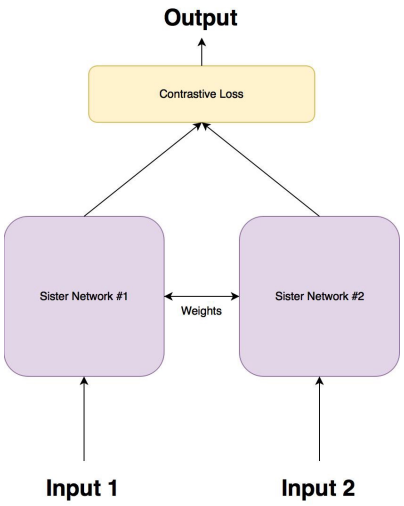

Figure 8: The siamese network architecture can be seen as two identical networks with shared parameters. Each network encodes one of the two input samples. The distance between the two encodings is computed with the contrastive loss and the parameters are updated to minimize or maximize the distance, depending if the input samples share the same label. Source: [Gup17]

4.1.2 The triplet network and triplet loss

Starting 2014, new architectures involving not pairs but triplets of input samples were investigated by Hoffer&Ailon and Wang&al in [HA14] and [Wan+14], then improved in the following years notably by Balntas&al in [Bal+16b] (PN-Net) and [Bal+16a] (T-Feat) or Schroff&al. in [SKP15] (*Google FaceNet*). Note that these papers use slightly different formulations. The losses presented and used in this paper are mostly based on Balntas&al.'s T-Feat paper, which is described by the authors as an improved version of their previous, very similar PN-Net paper.

A triplet consists of three samples, namely an anchor x, a positive instance x^+ and a negative instance x^-. The positive instance is a sample that is of the same class as the anchor, as opposed to the negative instance which is of a different class. The three samples are fed to three identical networks with shared parameters, which produce three embeddings: $f(x)$, $f(x^+)$ and $f(x^-)$. These embeddings are then merged in two intermediate values, δ^+ and δ^- which represent the euclidean distance between the anchor and the positive (resp. negative) instance:

$$\delta^+ = \|f(x) - f(x^+)\|_2$$

$$\delta^- = \|f(x) - f(x^-)\|_2$$

The goal of the network is to minimize δ^+ and maximize δ^- in order to encode inputs in a metric space in which $f(x)$ is close to $f(x^+)$ and distant from $f(x^-)$.

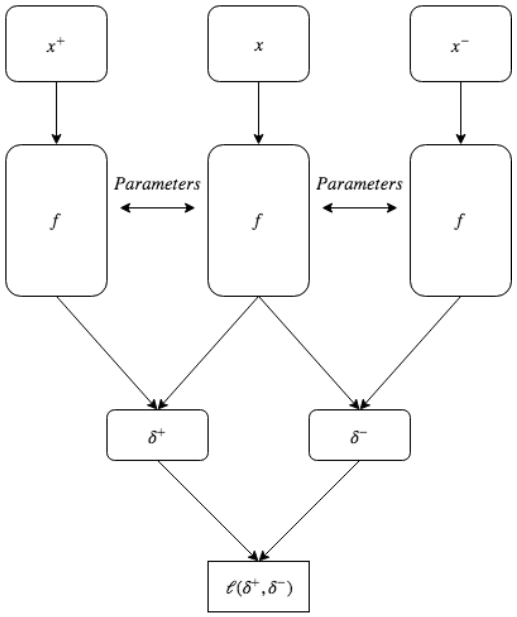

Figure 9: Triplet network architecture. The input triplet is represented by (x, x^+, x^-), f is the encoder and δ^+ (resp. δ^-) represents the positive (resp. negative) distance. The loss is computed from the two distances.

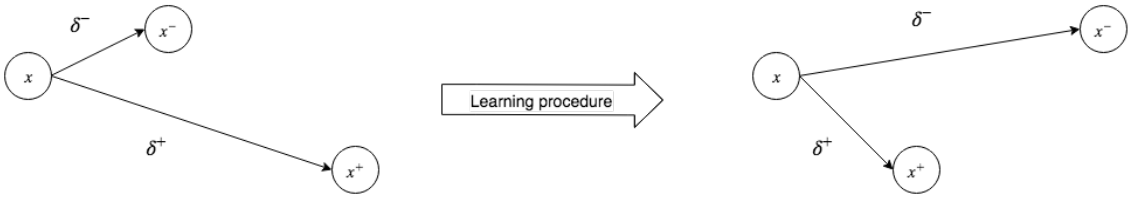

Figure 10: Triplet loss learning procedure. The goal of the network is to minimize δ^+ and maximize δ^- in order to encode inputs in a metric space in which $f(x)$ is close to $f(x^+)$ and distant from $f(x^-)$.

This optimization process is achieved through the triplet loss function. There are two main variants of the triplet loss, the margin ranking loss and the ratio loss.

4.1.3 The margin ranking loss

The margin ranking loss is given by:

$$\ell(\delta^+, \delta^-) = max(0, \mu + \delta^+ - \delta^-)$$

Where μ is a margin parameter with the same purpose as the contrastive loss' margin parameter. If $\delta^- > \delta^+ + \mu$, then the margin is respected and the weights will not be updated, because the max function will set the loss to 0. In other words, if the network already performs good enough on a given triplet, it will not learn anything from it.

This loss function is pretty self-explanatory. It measures how wrong a triplet is by adding the positive distance and subtracting the negative distance. Minimizing the loss boils down to minimizing δ^+ and maximizing δ^-, which is exactly what we want.

4.1.4 The ratio loss

The second main variant of the triplet loss function is the ratio loss. As its name suggests, instead of ensuring a minimal margin between the negative and positive distances, it computes a ratio between them:

$$\ell(\delta^+, \delta^-) = (\frac{e^{\delta^+}}{e^{\delta^+} + e^{\delta^-}})^2 + (1 - \frac{e^{\delta^-}}{e^{\delta^+} + e^{\delta^-}})^2$$

The ratio loss function basically tries to set $(\frac{e^{\delta^+}}{e^{\delta^+}+e^{\delta^-}})^2$ to 0 and $(\frac{e^{\delta^-}}{e^{\delta^+}+e^{\delta^-}})^2$ to 1, which boils down to minimize δ^+ and maximize δ^-. A softmax function is applied to map the non-normalized distances to the interval [0,1]. Note that as opposed to the margin ranking loss, the loss cannot be negative and there is no margin.

4.2 Use case: the triplet loss for facial recognition

This section presents a typical use-case of triplet networks as an illustrative example in order to provide a better understanding of how triplets are used in a neural network. The next section describes how triplets can be used for the purpose of location embedding, i.e. the subject of this paper.

Facial recognition consists of identifying and verifying a person's identity from a picture of this person. With many pictures of several different person, a neural network can be trained to perform this task. The use of the triplet loss to perform facial recognition was investigated by Schroff&al. (Google FaceNet).

The main idea here is to consider a dataset of pictures of different persons where each image is labelled with the identity of the person it depicts. From this dataset, triplets of images can be built. As stated earlier, each triplet consists of an anchor, a positive instance and a negative instance. In this particular case, the anchor x is a picture of a person, the positive instance x^+ is a different picture of the same person and the negative instance x^- is a third picture of a different person. The network is then trained to learn that x is of the same class as x^+ (they are the same person) and of a different class than x^-.

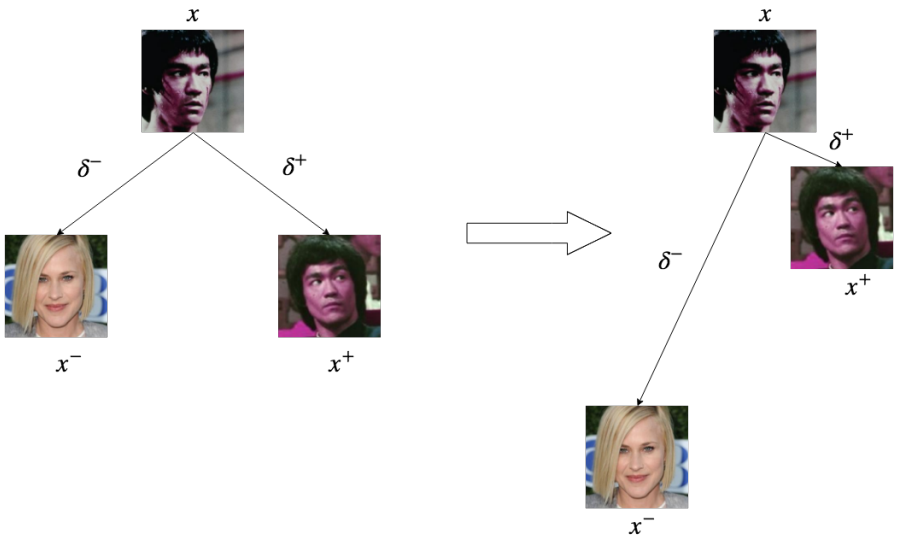

Figure 11: Triplet loss for facial recognition. A triplet consists of two picture of the same person (x and x^+) and one picture of a different person (x^-).

4.3 Use case: the triplet network for location embedding

This second use-case is the subject of this report and consists of using a triplet network for the purpose of location embedding. As stated in sections 2 and 3, our dataset consists of tiles rasterized from OpenStreetMap. The objective is to generate embeddings from these tiles in order to represent each tile with a vector of floating point numbers that encodes its semantic features. As stated in section 3.2 these vectors reside in a metric space, which means that the semantic similarity between two tiles (i.e. two locations) is inversely proportional to the euclidean distance between their respective embeddings.

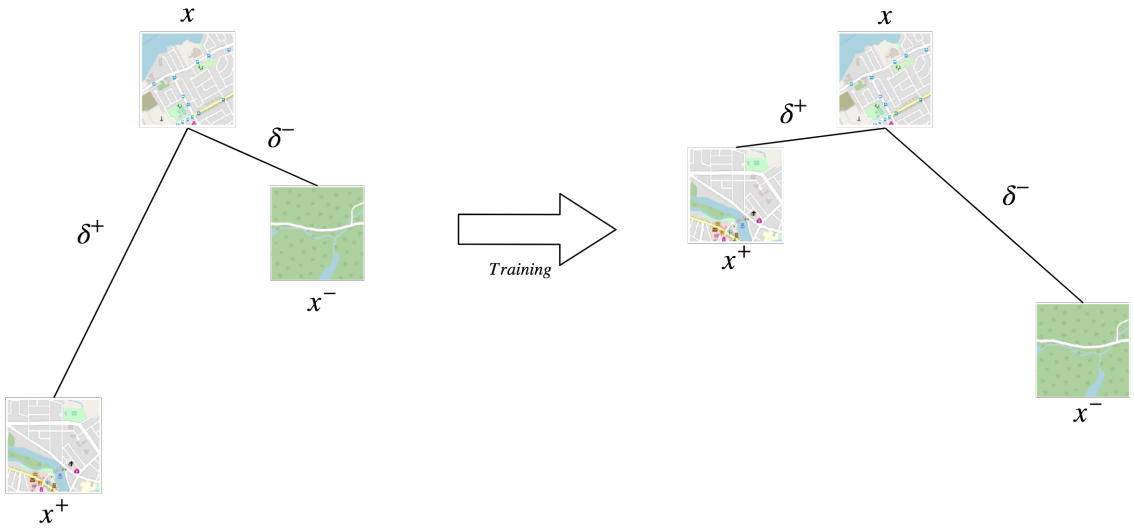

Figure 12: Triplet loss for location embedding. The positive instance is a tile whose location is semantically similar to the anchor's, while the location of the negative instance is semantically different. The selection of positive and negative instances is explained in sections 4.3.1 and 4.3.2.

In order to achieve this, triplets of tile images are generated from the dataset of tiles described in section 2 and fed to the network to teach it the notion of similarity between tiles and produce meaningful embeddings. As explained in section 5.2, each triplet is generated on the fly during training.

4.3.1 Triplets generation

A triplet consists of three tiles: an anchor, a positive instance that is semantically similar to the anchor and a negative instance that is semantically different from the anchor. The generation of triplets relies on the assumption that two neighboring locations are more similar than two distant locations. Hence, the triplets are generated as follows:

- The anchor x simply consists of a tile from the tiles dataset

- The positive instance x^+ consists of the same location as the anchor but randomly shifted by a few tens of meters, and subject to a certain number of geometrical transformations.
- The negative instance x^- is a tile of a randomly chosen location.

One of the reasons that justify the use of a triplet network is the unlabelled nature of our data. Indeed, there is no label that defines the nature of a tile such as we had in the previous section's facial recognition task. It would actually be very complicated to label this data since we don't even precisely know what kind of information the network is learning. The input data is unstructured and the features that the network is capturing are, in a certain way, abstract.

In order to get around the lack of labels, we use a triplet network in an self-supervised fashion. Building a triplet of tiles only requires knowledge about what is similar to the anchor and what is different, which does not necessarily involves labeled data. Ideally, the tiles would be labelled and we could generate the triplets in the same fashion as for facial recognition. Unfortunately this is not the case, hence the aforementioned workaround solution.

This alternative triplet generation method is directly inspired by Sentiance's previously mentioned article on venue mapping and relies on Tobler's first law of geography:

"Everything is related to everything else, but near things are more related than distant things".

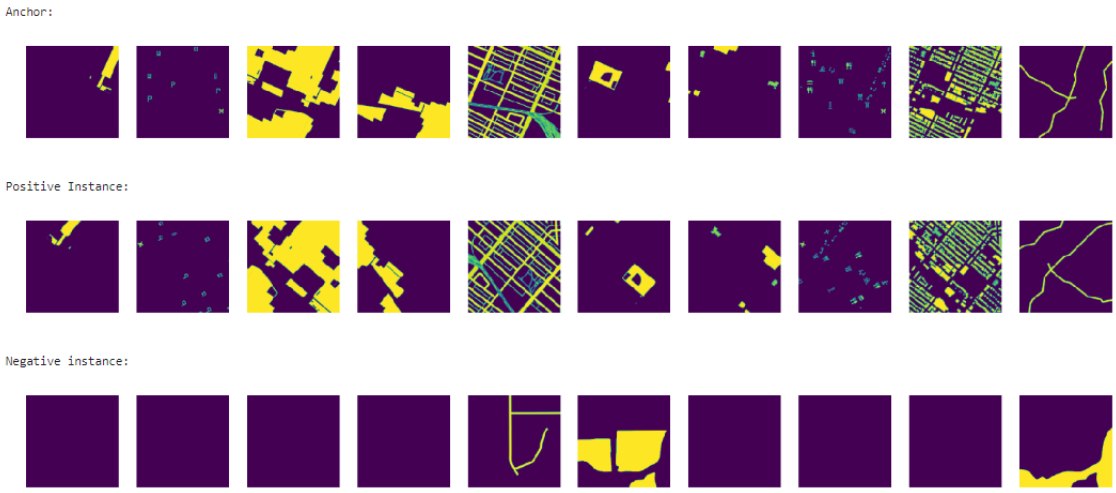

Figure 13: Example of a triplet of tiles. The positive instance is generated by shifting the anchor's location by a small distance and applying some transformations to the tile. This results in a unique tile that is semantically similar to the anchor.

4.3.2 Positive instances generation

As stated above, positive instances are generated from the anchor, through a series of transformations. By doing that, we hope to create a unique tile that is different enough from the anchor, but still shares the same semantic attributes. In other words, a tile that the networks interprets as a different place that resembles the anchor.

Typical examples of transformations used to generate positive instances are symmetric padding, random rotation, random shift and random flip. The exact nature of these transformations and the positive instance generation process are described more precisely in section 5.2. Note that although this process is not the ideal way of crafting triplets, it is a robust workaround to overcome the unlabelled nature of the dataset.

4.4 Advanced triplet selection methods

One common issue with triplet networks is characterized by a network that stops learning after a few iterations because it is only fed with examples that are too easy. In the case of the margin ranking loss, this is portrayed by a loss of 0 due to the margin, and hence no weights update. Therefore, it is important to carefully chose complicated examples, i.e. examples that violate the triplet constraint the most and produce high losses in order to keep the network learning for many iterations. This subsection describes some advanced triplet selection methods, and how they can be adapted to the use-case of location embedding.

4.4.1 Hard negative mining

Hard negative mining is a triplet selection method that consists of selecting the most difficult negative instance possible for an anchor. By "difficult", we mean a negative instance that violates the triplet constraint as much as possible. In the context of facial recognition, we could for example imagine as a negative instance a person that looks a lot like the person on the anchor image.

The first question that arises at this point is *"How do we assess the difficulty of a negative instance?"*. A difficult negative tile is an image that the network cannot easily identify as being different from the anchor. This is reflected in the distance between the anchor and the negative instance:

$$\delta^- = \|f(x) - f(x^-)\|_2$$

As mentioned before, the triplets are generated on the fly during training and a negative instance is a randomly chosen tile whose location is different than the anchor's. Therefore, in order to select a hard negative instance, a simple solution consists of generating several negative tiles for a single anchor, computing δ^- for each of them and keeping the negative instance that produces

the smallest δ^- (i.e. the tile that the network has the most trouble differentiating from the anchor). Note that there is a trade-off in the choice of the number of negative candidates: the more candidates, the higher the probability to find a hard example, but the longer the processing time. Indeed, as the triplets are generated on the fly and the negative candidates are randomly chosen, each candidate entails a random I/O operation to the disk, which increases computation time.

In the same idea, one could also consider *hard positive mining*, i.e. generating several positive instances for a given anchor and keeping the one that produces the highest positive distance δ^+. However, as explained in sections 4.3.2 and 5.2, positive instance generation requires many geometrical transformations which come at a cost and represent one of the main bottlenecks in terms of network performance. The next section presents another advanced triplet selection method that avoids these performance issues.

4.4.2 Anchor swap

Anchor swap is a triplet selection method presented by Balntas&al. It was first introduced in [Bal+16b] as the *SoftPN* loss function, and was later reformulated as *anchor swap* in [Bal+16a]. Note that the two papers actually describe a very similar concept, only with a slightly different formulation.

Anchor swap is a computationally cheaper way of performing hard negative mining. The idea behind it is to consider the anchor as a second positive instance. The anchor and the positive instance are then swapped if the distance between the negative instance and the positive instance is smaller than the distance between the negative instance and the anchor. In other words, swapping the anchor and positive increases the distance between the anchor and negative which makes for a more difficult triplet, potentially helping the network learn. Formally we now consider not two but three distances in a triplet, defined as

$$\delta^+ = \|f(x) - f(x^+)\|_2$$

$$\delta_1^- = \|f(x) - f(x^-)\|_2$$

$$\delta_2^- = \|f(x^+) - f(x^-)\|_2$$

and the negative distance δ^- becomes

$$\delta^- = min(\delta_1^-, \delta_2^-)$$

This ensures that the hardest negative distance within a triplet is chosen for backpropagation by considering the three possible distances in a triplet. Note that anchor swap can be applied both

for the margin ranking loss and the ratio loss, and that it figures in Pytorch's implementation of the margin ranking loss function.

One could argue that anchor swap replaces the need for hard negative mining as described in section 4.4.1. Nonetheless it is important to keep in mind that in our case the positive instances are generated directly from the anchors, which intuitively implies that if δ_1^- provides poor learning (i.e. if it is high), then δ_2^- will be high as well. Consequently, hard negative mining as described in the previous section is still justifiable even with anchor swap.

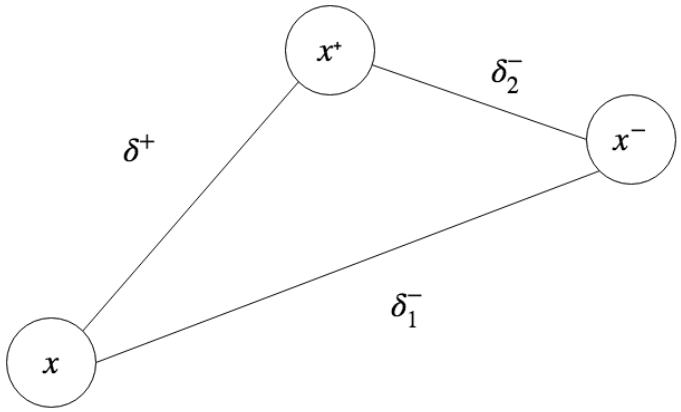

Figure 14: Illustration of a triplet that can benefit from anchor swap. Swapping the anchor and the positive instance results in a smaller negative distance ($\delta_2^- < \delta_1^-$), and therefore in a more difficult triplet.

5 Data generation process

This section first explains how the tiles dataset was generated, and more precisely how a tile can be split in semantic layers. The tile generation process is non-trivial and requires a very specific setup. The second half of this section addresses the details of triplets generation through some illustrative examples. The methodology and underlying assumptions of the triplet generation process are presented in section 5.2.

5.1 Rasterization of the tiles

The generation of the tiles dataset is possible thanks to OpenStreetMap's open source policy. OSM is a collaborative project, and all its geographic data is available online for free so anyone can use it to build its own custom application.

The typical use of a tile rendering server is a web-based map application, like OpenStreetMap itself. Let's say a user opens the OpenStreetMap website to see the city of Montréal. The client will send a query to the server with the bounding box of the area to display, the zoom level and other information. The server splits the area into square images called *tiles*, queries a GIS database to get Montréal's geospatial data and rasterizes the tiles. The tiles are then sent to the client, and displayed side by side on the user's screen to form the map.

5.1.1 Personalized tiles rendering

The easiest way to get tiles from OpenStreetMap is to use a third-party provider, as this requires almost no set-up. A list of tile servers is available on `https://wiki.openstreetmap.org/wiki/Tile_servers`. However, these tile providers have their own server which cannot be customized. This project's requirements are very specific, as we need a server capable of rasterizing a tile's layers independently. In order to have full control on the way tiles are rasterized, the solution is to build our own database and tiles server to render and serve our own tiles.

Rendering and serving tiles is a computationally intensive task and may require substantial storage space depending on the size of the regions covered, or high processing power depending on the traffic. In our case, we only need data for the province of Québec and there will never be more than one client at a time, as the goal is to generate a dataset. The different components necessary to build an independent tile generation service are described in the following sections and the pipeline is illustrated on figure 15.

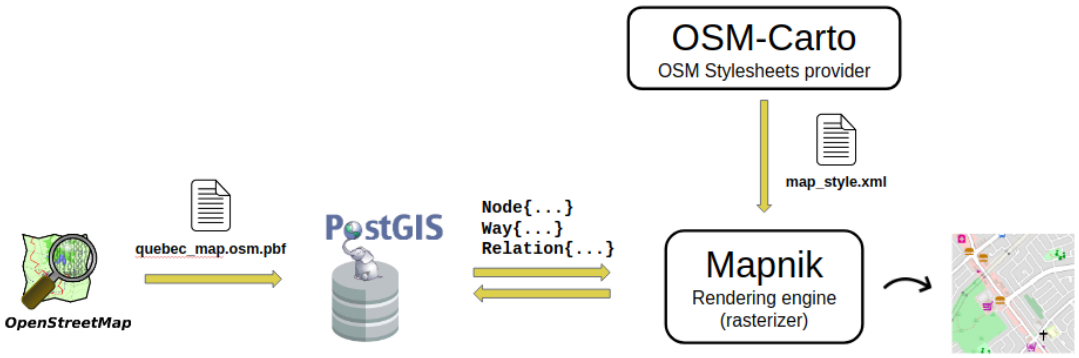

Figure 15: Tiles rasterization process. The data from OpenStreetMap is downloaded and stored in a GIS database. A server queries the data from the database and rasterizes the tiles with Mapnik and OpenStreetMap-Carto stylesheets.

5.1.2 Data storage

The first step of the process is to download the data. The data is available on different places and for different locations. For instance, *Geofabrik* provides OpenStreetMap data for countries and province, and *bbbike.org* for some major cities. The data for the desired area is to be downloaded in a single compressed .osm.pbf file. Once this is done, a PostgreSQL database with the spatial database extender *PostGIS* must be set up and filled with the data contained in the .osm.pbf archive.

The easiest way to extract the archive and fill the postgreSQL database is to use a tool called *osm2pgsql*. As its name suggests, its purpose is to convert OpenStreetMap data to PostgreSQL data. This helper tool is part of the OpenStreetMap project and is available on their github page.

5.1.3 Tile server

Once the database is up and running, the next task is to build a server that can query the database, render the tiles and save them as images. This can be done with Mapnik, an open source toolkit specially designed to develop mapping applications. Among other things, Mapnik can be used as a rendering engine. The tiles rendering system implements a queue in order to efficiently use all the resources available in a parallel fashion and accelerate the rendering process.

The server basically takes as input a list containing the coordinates of the centers of the tiles, renders for each of these tiles every semantic layer defined by the project files (see section 5.1.4) and concatenates these layers in a single .tiff image file. In the same way a classic RGB image has three channels, each of these tiff files has ten channels. These images will then be imported in PyTorch as tensors. The dimension of a tile (both in meters and pixels) are also defined in the server.

The tiles are stored in subdirectories of one thousand tiles each. For each tile, two image files are stored: the ten layers tiff file containing the actual data that will be used to train the neural network, and a second RGB image of the same tile. This second file is used for ease of visualization when presenting the results. Along with these files, a metadata CSV file (path and coordinates of each tile) is generated.

The data that the server gets from the database describes *what* elements are displayed on the map, but not *how* they are displayed. For instance, Mapnik knows there is road on a tile, but it does not know what color or how thick this road should be. This information is contained the map's project file and stylesheets.

5.1.4 Stylesheets and layer separation

The project file and the stylesheets used in this work are modified versions of the official stylesheets of the standard map on OpenStreetMap's website which can be found on [gra12]. Note that aside from the official style used here, there are many other sets of stylesheets designed by different people for different purposes, for instance ibikecph-carto, a CartoCSS map style used to render a map of bicycle paths in Copenhagen (see `http://www.ibikecph.dk`).

In order to define the style of a map, carto provides a project file (.mml) and some stylesheets (.mss). Without getting too much into technical details, here is how this works:

The project file basically contains global settings (the bounds of the map, the maximum and minimum zoom values, the name of the database, ...), the list of the stylesheets that are used for rendering the map and the list of all the layers that will be rendered. This list is of particular interest here, as removing any layer from it will simply not display this layer. Therefore, in order to generate a layer, let's say the roads layer for instance, we can simply remove from the project file all the layers that have nothing to do with roads. Repeat this process and keep a different project file for each layer. Then, during the generation, each tile is rendered ten times, once for each project file.

Note that OpenStreetMap has more than eighty different layers, and almost no documentation about them. Sorting them is an error-prone process and must be done with care. The project file uses the YAML format, which is very convenient because it can easily be parsed using Ruamel's YAML library.

The stylesheets are .mss files that define the style of each element displayed on the map. Basically, they are to maps what CSS files are to websites. Although the documentation is very sparse here again, the syntax and variable names are quite easy to interpret. These stylesheets

can be edited to resize, reshape or remove elements from the map. For example, all the text displayed on the standard OSM maps is removed. In a similar fashion, some amenities, (bus stops, shops, restaurants, ...) are usually displayed only at certain zoom levels. Editing the appropriate stylesheets allows to display and resize them as desired. For instance, you can notice that the trees are way too big on figure 16 (top image), and are represented by brown dots, surrounded by transparent green circles. By simply editing the appropriate stylesheet, we can replace them by small green dots (bottom image), which is much more consistent and accurate regarding the rest of the green areas.

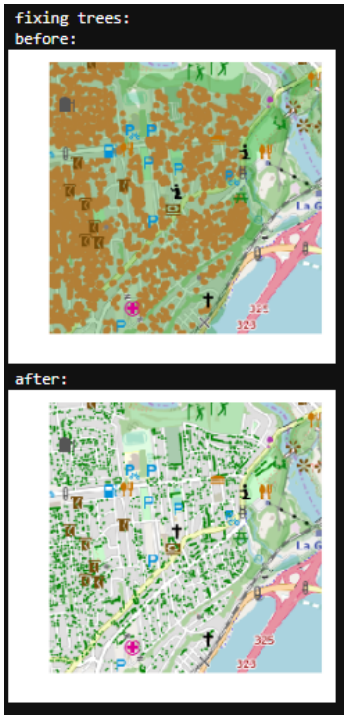

Figure 16: Example of MSS stylesheets editing. Any graphical element can be displayed as desired by editing the appropriate stylesheet. This figure illustrates the modification of the trees' size and shape for consistent green areas.

The following code snippet is an example of how a layer (in this case the *green areas* layer) of the resulting tiles can be generated.

Listing 1: Generation of the *green areas* tile layer

```
############################################################
# 7. Landcover (Green spaces)
############################################################

# First, filter out all layers that are unrelated to nature or green areas
landcover_layers = []
for layer in filtered_layers:
  if any(keyword in layer['id'] for keyword in ['landcover',
      'nature-reserve-boundaries', 'tree']):
    if 'symbol' in layer['id']: continue # discard symbols in landcover layers
    landcover_layers.append(layer)

# discard original landcover stylesheet
stylesheets = [sheet for sheet in original_stylesheets if sheet !=
   'landcover.mss']
# replace it by custom stylesheet to keep only nature-related landcover items
stylesheets.append('landcover_green.mss')

# Construct a new project.mml file with custom layers and stylesheets
data['Stylesheet'] = stylesheets
data['Layer'] = landcover_layers
yaml.dump(data, Path(mml_path + 'landcover_green_layers.mml'))
```

This process basically boils down to parsing the original mml project file (all layers displayed together, as in the standard OSM map) and turning it into another tailor-made project file. The first step is the filtering of unwanted layers: we only keep in the project file the layers that are related to nature. For aesthetic purpose, all layers responsible for drawing symbols on the map are discarded as well. Then, the list of stylesheets present in the project file is filtered. The original landcover stylesheet is removed from the project and replaced by a new version of it, customized for green areas. Once this is done, a new project file is saved for later use during the tiles generation.

5.2 Triplets generation

The tiles dataset is static, in the sense that it is meant to be generated only once, and is not subject to any modification after its generation. The triplets on the other hand, are generated at training time. At each epoch, minibatches of 128 triplets are generated and fed to the triplet

network. An instance of the class `TripletOSMDataset` is responsible for generating and serving triplets. This class inherits from the `torch.utils.data.Dataset` superclass and declares an instance of `OSMDataset` at instantiation.

When the triplet dataset is queried, it first gets the anchor by querying the `OSMDataset` instance, which will load the requested tile from disk in a tensor. Then, in order to have a negative instance, the triplet dataset will query a second tile from the tiles dataset, this time at a randomly chosen index (i.e. different from the anchor's).

The generation of positive instances is slightly more complicated. As stated in section 5.3 and due the the unlabelled nature of the tiles, a positive instance is generated by applying a series of transformations to its anchor. Based on the assumption that close things are more related than distant things (Tobler's *first law of geography*), this results in a tile that is unique but semantically similar to the anchor.

Figure 17: Generation of a triplet. The positive instance is generated from the same tile as the anchor. The negative instance is generated from a randomly chosen tile in the dataset. In order to keep consistent dimensions, the anchor and the negative instance are center-cropped and resized. The input side dimension of a tile is 256 pixels (1000 meters), and its ouput dimension is 128 pixels (800 meters).

The different transformations applied to the anchor tile are the following:

1. **Symmetric padding:** This transformation serves two purposes. First, it allows to perform some kind of data augmentation on the anchor tile, which is exactly what we want for the positive instance. Second, by making the tile temporarily bigger, this is a way to get rid of the black corners generated by the rotation.

2. **Random rotation:** The positive tile is then rotated by a randomly chosen angle in [0,360] degrees. This rotation is followed by a center crop transformation in order to avoid the black corners mentioned above and go back to the original zoom level of the tile.

3. **Random shift:** A shift of the center of the tile is performed. The tile is shifted both vertically and horizontally by randomly chosen distances in [-150,150] meters. Since the tiles are pre-raterized, this is actually achieved by randomly cropping a square image in the input tile. In order to keep consistent dimensions, a margin is present on the raw rasterized tiles and each anchor and negative tile is center cropped.

4. **Random flip**: The tile is then flipped on the vertical (resp. horizontal) dimension with a probability of 0.5.

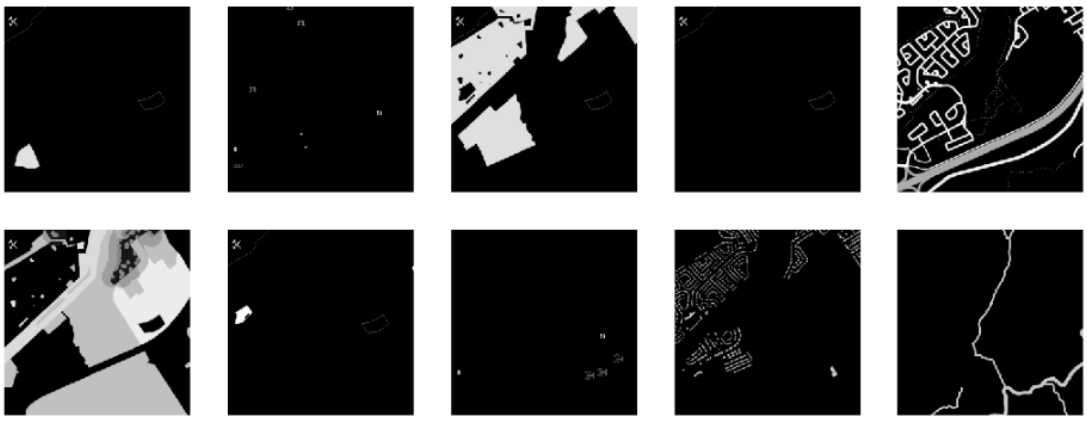

Figure 18: Positive instance generation Step #0: input tile with 10 layers. Each image is 256x236 pixels.

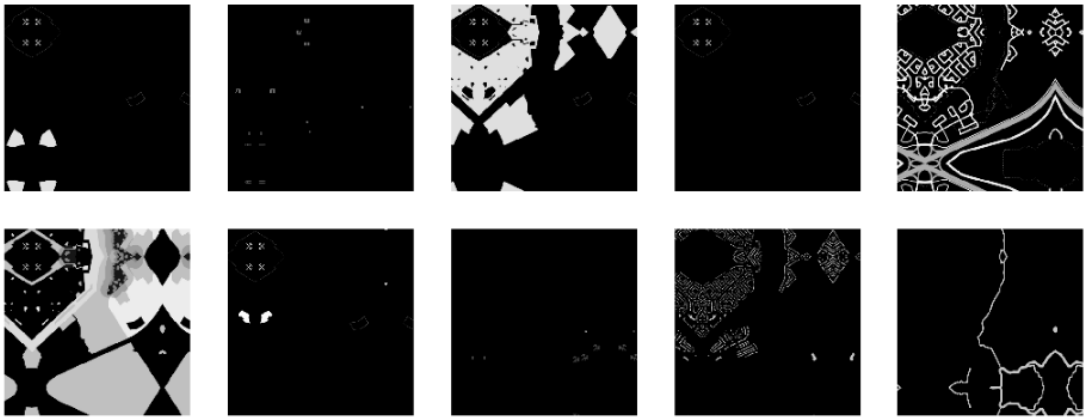

Figure 19: Positive instance generation Step #1: symmetric padding where we see the repeating patterns of streets from the 5th layer.

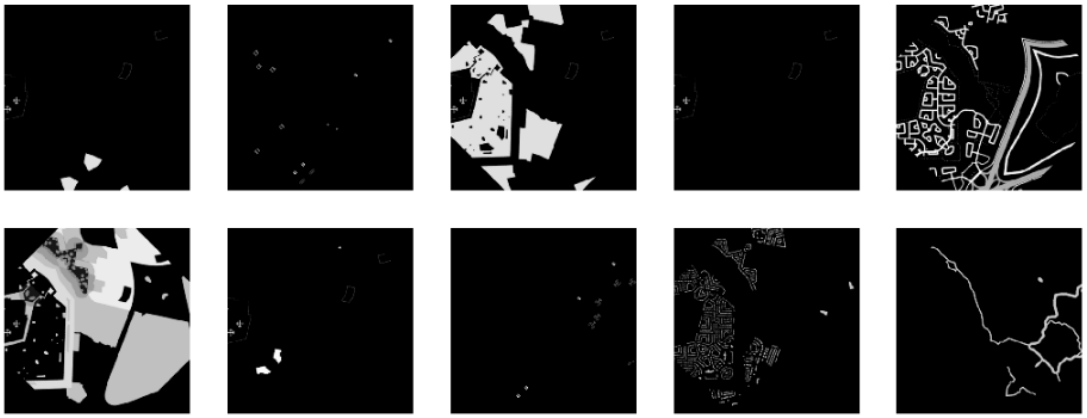

Figure 20: Positive instance generation Step #2: random rotation

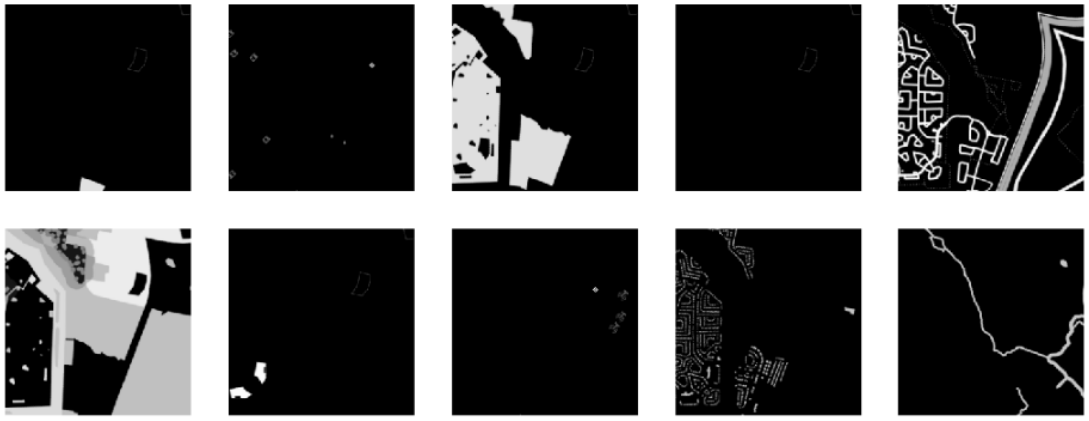

Figure 21: Positive instance generation Step #3a: random shift (center crop)

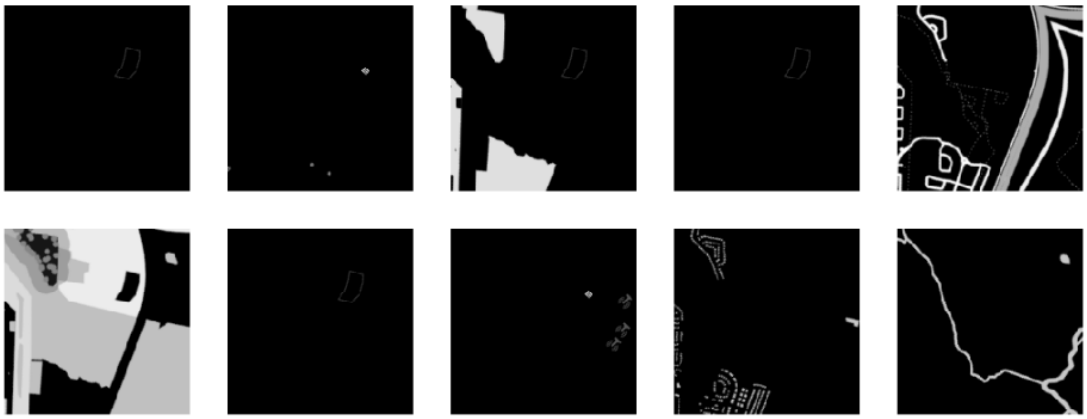

Figure 22: Positive instance generation Step #3b: random shift (random crop)

Figure 23: Positive instance generation Step #4: random flip

Note that it is important that the zoom level (in other words the pixels per meters ratio) remains consistent throughout the triplets and instances.

Generating the triplets on the fly results in an almost infinite dataset, as the positive instances generated from the same anchor tile are different at each epoch due to the random parameters of the transforms. New negative instances are also picked again at each epoch. This has the advantage of keeping the network learning for more epochs. Note that this is only the case during training: for a matter of consistency, the evaluation dataset generates the same exact triplets at each epoch. Also note that when a transformation is applied to a tile, the same transformation is applied to all of its ten layers. This ensures that the layers will remain consistent with one another.

6 Use case: housing price prediction

There are many possible applications for the location embeddings learned by our triplet network. The locations are encoded as vectors of floating point numbers, which means they can easily be used as features for any classification or regression problem. This section presents a use case where the encoded tiles are used as additional features in a housing price prediction model.

6.1 Motivation

The motivation for this use-case comes from a project led by the CRIM. As mentioned before, JLR, a land title solution company in Québec is currently contracting the CRIM to develop different AI-powered real estate solutions. One of them is a house price prediction solution, i.e. a model capable of predicting the price of a house or apartment using different features provided by JLR.

6.2 Problem description

The goal of this project is to implement a model that can predict a price for any house or apartment given typical housing features. Although this model may seem straight-forward, it is actually more complicated than it seems. First of all, the features available are informative but likely not sufficient to determine the price of a property. This is why some researchers at the CRIM are working on new features that could help the price prediction model.

In a similar manner, the location embeddings addressed in this work can serve as additional features for this regression problem. The underlying assumption behind this is that the information encoded in the tiles embeddings could have an influence on housing prices. For example, the presence of green spaces, shops, restaurants or schools near a house could influence its price. The same reasoning applies for more complex features hidden in a latent dimension such as the shape of nearby roads or the surrounding landcovers. For instance, a building located in a residential landcover, on a small traffic circle is likely to be more quiet than another building located on a big avenue.

6.3 Scope

This problem has a non-negligible temporal dimension, since real estate market prices in Québec are in constant evolution. There are clear price augmentation patterns over time that must be taken into account when designing the regression model. This is actually a complex time series problem currently investigated by a team of researchers at the CRIM and goes beyond the scope of this work.

Nonetheless, there is still a big interest here for the problem of location embedding. Indeed, if we can show in a simple manner that adding the tiles encodings as extra features improves

the performance of the regression model, this would be additional proof that the triplet network learned substantial and informative features (as a reminder, our triplet network is trained in a self-supervised manner, which makes it hard to evaluate). This would also justify a potential use of the tiles encodings in a more complex model developed by the team working on the JLR project. In short, the point here is not to build the optimal price prediction model for JLR, but to evaluate the impact of the embeddings on a baseline regression model and to provide additional proof that the embeddings contain valuable information.

6.4 Problem formulation

In order to simplify the problem and get around the temporal dimension, the dataset is filtered and only contains transactions from a small time span. This avoids any bias due to variations of the market prices over time. The experiment is straight-forward: first, a regression model is trained and tested with selling prices as targets and the original attributes from JLR as features. Then, the same model is trained and tested again, but this time each transaction has its corresponding encoded tile as new features. If the model performs better with the embeddings, it means that the embedding space learned by the triplet network has an influence on the transaction prices. This experiment is repeated for different models, and several different evaluation metrics are used as well.

A second experiment in also carried out in order to determine if the embeddings contain valuable information regarding the price prediction problem. This experiment consists of comparing a baseline prediction model in which every price prediction is equal to the average price (or the median price) of the training set against a regression model with the embeddings as only features.

6.5 Adding the embeddings

In order to generate the embeddings, a triplet network previously trained with the dataset described in section 2 is used. A forward pass is run on a new set of tiles, where each tile is centered on a property from the dataset used for regression. This means that each property from the price prediction dataset gets an encoded vector from a tile centered on its location, but also that these tiles are different than the ones used to train or test the triplet network.

6.6 Confidentiality

Different models were used for these experiments. These models are inspired by the models that gave the best results in the previous experiments led by the CRIM and therefore cannot be cited in this paper for confidentiality reasons. Unfortunately, the results of the first experiment mentioned in the previous section cannot be shared either as they involve confidential material. The results of the second experiment (using the embeddings as only features against a median predictor) are not confidential and are presented in section 7.3.2 and discussed in section 8.

6.7 Evaluation metrics

The evaluation metrics considered in this regression problem are the Mean Absolute Error (MAE) and the Mean Absolute Percentage Error (MAPE):

$$MAE = \sum_{i=1}^{n} \frac{\mid y_i - x_i \mid}{n}$$

$$MAPE = \frac{1}{n} \sum_{i=1}^{n} \frac{\mid y_i - x_i \mid}{y_i}$$

Where x_i is a price prediction, y_i the ground truth and n the number of properties. The MAE is basically the average of the differences between ground truth and prediction. The MAPE on the other hand divides each error by the true price before computing the average. This means that each error is computed relatively to the prediction price. In other words, the same error will have less impact on the total loss if the price is big.

7 Experiments and results

7.1 Hardware and software specifications

The implementation of the triplet network was done in Python using Pytorch 0.4.1. The training was accomplished on GPU, using a Nvidia GeForce GTX 1080 Ti graphics card.

7.2 Network architectures

In this section, the different network architectures of the triplet network are presented. In total, three architectures with different characteristics have been retained.

7.2.1 TNet 1

The first architecture is inspired from Balntas&al.'s TFeat network as described in [Bal+16a]. TNet 1 uses Stochastic Gradient Descent for optimization, with an initial learning rate of 0.01 and a weight decay of 10^{-4}. The learning rate is adaptive and decreases statically. It is multiplied by 0.5 every 10 epochs.

Layer	Input size	Output size	Kernel	# params
InstanceNorm2d	$10 \times 128 \times 128$	$10 \times 128 \times 128$	-	0
Conv2d	$10 \times 128 \times 128$	$32 \times 122 \times 122$	$7 \times 7, 1$	15712
Tanh	$32 \times 122 \times 122$	$32 \times 122 \times 122$	-	0
MaxPool2d	$32 \times 122 \times 122$	$32 \times 61 \times 61$	$2 \times 2, 2$	0
Conv2d	$32 \times 61 \times 61$	$64 \times 56 \times 56$	$6 \times 6, 1$	73792
Tanh	$64 \times 56 \times 56$	$64 \times 56 \times 56$	-	0
Linear	200704	128	-	25690240
Tanh	128	128	-	0

Table 1: TNet 1

7.2.2 TNet 2

This second architecture is inspired by the network presented by Sentiance in [Spr18]. It is fairly deeper than TNet 1 and uses batch normalization after each hidden layer in order to improve the performance and stability of the network, but also to encourage each layer of the network to learn more independently of other layers.

TNet 2 uses the Adam optimizer with an initial learning rate of 0.001 and a weight decay of 10^{-4}. The learning rate is adaptive and decreases statically by a factor 10 every 10 epochs.

Layer	Input size	Output size	Kernel	# params
Conv2d	$10 \times 128 \times 128$	$32 \times 126 \times 126$	$3 \times 3, 1, 0$	2912
LRelu	$32 \times 126 \times 126$	$32 \times 126 \times 126$	-	0
BatchNorm2d	$32 \times 126 \times 126$	$32 \times 126 \times 126$	-	64
MaxPool2d	$32 \times 126 \times 126$	$32 \times 63 \times 63$	$2 \times 2, 2, 0$	0
Conv2d	$32 \times 63 \times 63$	$32 \times 61 \times 61$	$3 \times 3, 1, 0$	9248
LReLU	$32 \times 61 \times 61$	$32 \times 61 \times 61$	-	0
BatchNorm2d	$32 \times 61 \times 61$	$32 \times 61 \times 61$	-	64
MaxPool2d	$32 \times 61 \times 61$	$32 \times 30 \times 30$	$2 \times 2, 2, 0$	0
Conv2d	$32 \times 30 \times 30$	$64 \times 28 \times 28$	$3 \times 3, 1, 0$	18496
LReLU	$64 \times 28 \times 28$	$64 \times 28 \times 28$	-	0
BatchNorm2d	$64 \times 28 \times 28$	$64 \times 28 \times 28$	-	128
MaxPool2d	$64 \times 28 \times 28$	$64 \times 14 \times 14$	$2 \times 2, 2, 0$	0
Conv2d	$64 \times 14 \times 14$	$64 \times 12 \times 12$	$3 \times 3, 1, 0$	36928
LReLU	$64 \times 12 \times 12$	$64 \times 12 \times 12$	-	0
BatchNorm2d	$64 \times 12 \times 12$	$64 \times 12 \times 12$	-	128
MaxPool2d	$64 \times 12 \times 12$	$64 \times 6 \times 6$	$2 \times 2, 2, 0$	0
Conv2d	$64 \times 6 \times 6$	$128 \times 4 \times 4$	$3 \times 3, 1, 0$	73856
LReLU	$128 \times 4 \times 4$	$128 \times 4 \times 4$	-	0
BatchNorm2d	$128 \times 4 \times 4$	$128 \times 4 \times 4$	-	256
Conv2d	$128 \times 4 \times 4$	$64 \times 4 \times 4$	$1 \times 1, 1, 0$	8256
LReLU	$64 \times 4 \times 4$	$64 \times 4 \times 4$	-	0
BatchNorm2d	$64 \times 4 \times 4$	$64 \times 4 \times 4$	-	128
Conv2d	$64 \times 4 \times 4$	$64 \times 4 \times 4$	$1 \times 1, 1, 0$	4160
LReLU	$64 \times 4 \times 4$	$64 \times 4 \times 4$	-	0
BatchNorm2d	$64 \times 4 \times 4$	$64 \times 4 \times 4$	-	128
Linear	1024	64	-	65600
LReLU	64	64	-	0
Linear	64	16	-	1040

Table 2: TNet 2

7.2.3 TNet 3

The last architecture tested for the triplet network is partially inspired by [SKP15] (Google FaceNet). This network consists of six hidden layers. This is less than TNet 2, but there are more parameters. As opposed to TNet 2, TNet 3 does not make use of batch normalization but aggressively uses dropout. This design choice is discussed in section 8.

TNet 3 uses the Adam optimizer with an initial learning rate of 0.001 and a weight decay of 10^{-4}. The learning rate is adaptive and decreases dynamically by a factor 10 as soon as the loss does not improve for at least three epochs.

Layer	Input size	Output size	Kernel	# params
Conv2d	$10 \times 128 \times 128$	$64 \times 61 \times 61$	$7 \times 7, 2$	31424
MaxPool2d	$64 \times 61 \times 61$	$64 \times 30 \times 30$	3×3,2	0
LeakyReLU	$64 \times 30 \times 30$	$64 \times 30 \times 30$	-	0
Dropout (0.05)	$64 \times 30 \times 30$	$64 \times 30 \times 30$	-	0
Conv2d	$64 \times 30 \times 30$	$192 \times 28 \times 28$	3×3,1	110784
MaxPool2d	$192 \times 28 \times 28$	$192 \times 13 \times 13$	3×3,2	0
LeakyReLU	$192 \times 13 \times 13$	$192 \times 13 \times 13$	-	0
Dropout (0.05)	$192 \times 13 \times 13$	$192 \times 13 \times 13$	-	0
Conv2d	$192 \times 13 \times 13$	$384 \times 11 \times 11$	3×3,1	663936
MaxPool2d	$384 \times 11 \times 11$	$384 \times 5 \times 5$	3×3,2	0
LeakyReLU	$384 \times 5 \times 5$	$384 \times 5 \times 5$	-	0
Dropout (0.05)	$384 \times 5 \times 5$	$384 \times 5 \times 5$	-	0
Conv2d	$384 \times 5 \times 5$	$256 \times 3 \times 3$	3×3,1	884992
LeakyReLU	$256 \times 3 \times 3$	$256 \times 3 \times 3$	-	0
Dropout (0.05)	$256 \times 3 \times 3$	$256 \times 3 \times 3$	-	0
Conv2d	$256 \times 3 \times 3$	$256 \times 5 \times 5$	3×3,1,2	590080
LeakyReLU	$256 \times 5 \times 5$	$256 \times 5 \times 5$	-	0
Dropout (0.05)	$256 \times 5 \times 5$	$256 \times 5 \times 5$	-	0
Conv2d	$256 \times 5 \times 5$	$256 \times 1 \times 3$	3×3,1	590080
MaxPool2d	$256 \times 3 \times 3$	$256 \times 1 \times 1$	3×3,2	0
LeakyReLU	$256 \times 1 \times 1$	$256 \times 1 \times 1$	-	0
Dropout (0.05)	$256 \times 1 \times 1$	$256 \times 1 \times 1$	-	0
Linear	256	128	-	32896
Linear	128	64	-	8256
Linear	64	16	-	1040
Dropout (0.2)	16	16	-	0

Table 3: TNet 3

7.3 Results

7.3.1 Triplet network

	Test loss	# epochs before convergence
TNet 1	0.0654	65
TNet 2	**0.0105**	56
TNet 3	0.0238	50

Table 4: Triplet network results. The loss values are computed with the margin ranking loss, as presented in section 4.1.3

7.3.2 Housing price prediction

	MAE	MAPE
Mean price	93709	55%
Median price	92802	52%
Embeddings only	**80898**	**45%**

Table 5: Median price prediction against embeddings alone. This table compares the accuracy of a trivial predictor where all the predictions correspond the the mean or median value of all prices against the accuracy of a predictor where the only features are the tiles embeddings. The model used to predict prices from the embeddings is confidential.

7.4 Visualization of the embeddings

This section presents some visualizations of the embeddings in their metric space. In order to display the tiles, the dimensionality of the embeddings is reduced to three using PCA or t-SNE and each tile is displayed in the resulting 3D space at the coordinates defined by its embedding. Figures 24, 25 and 26 (resp. 27, 28 and 29) are screenshots of the same plot, only from different points of view. Taking two-dimensional screen captures of a three-dimensional space is not ideal as a lot of visual insight comes from seeing the 3D plot in movement. To this end, some video captures are available on `https://bit.ly/2HgC2Ob`. More visualizations of the metric space are presented in section 8.2.

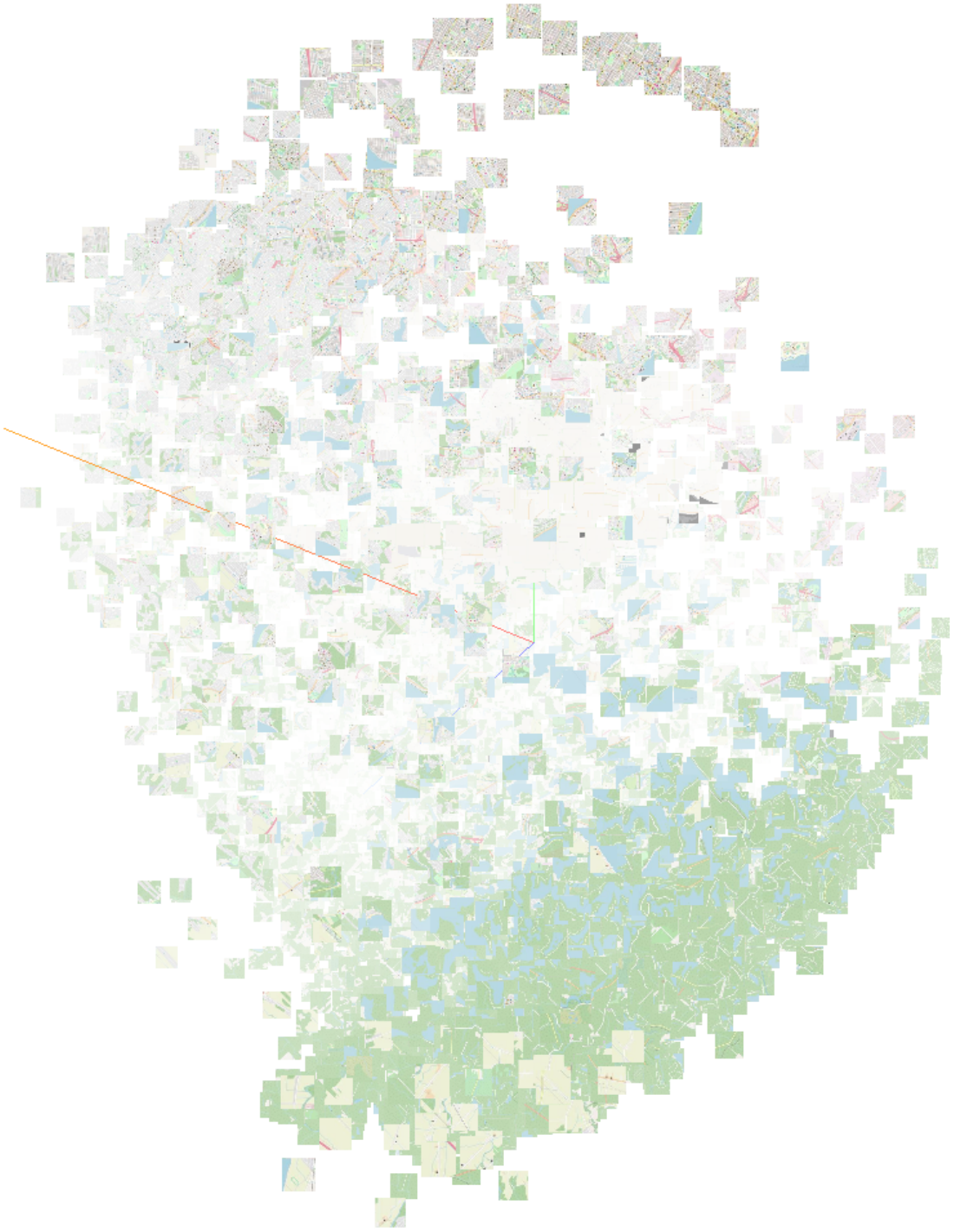

Figure 24: PCA cluster visualization

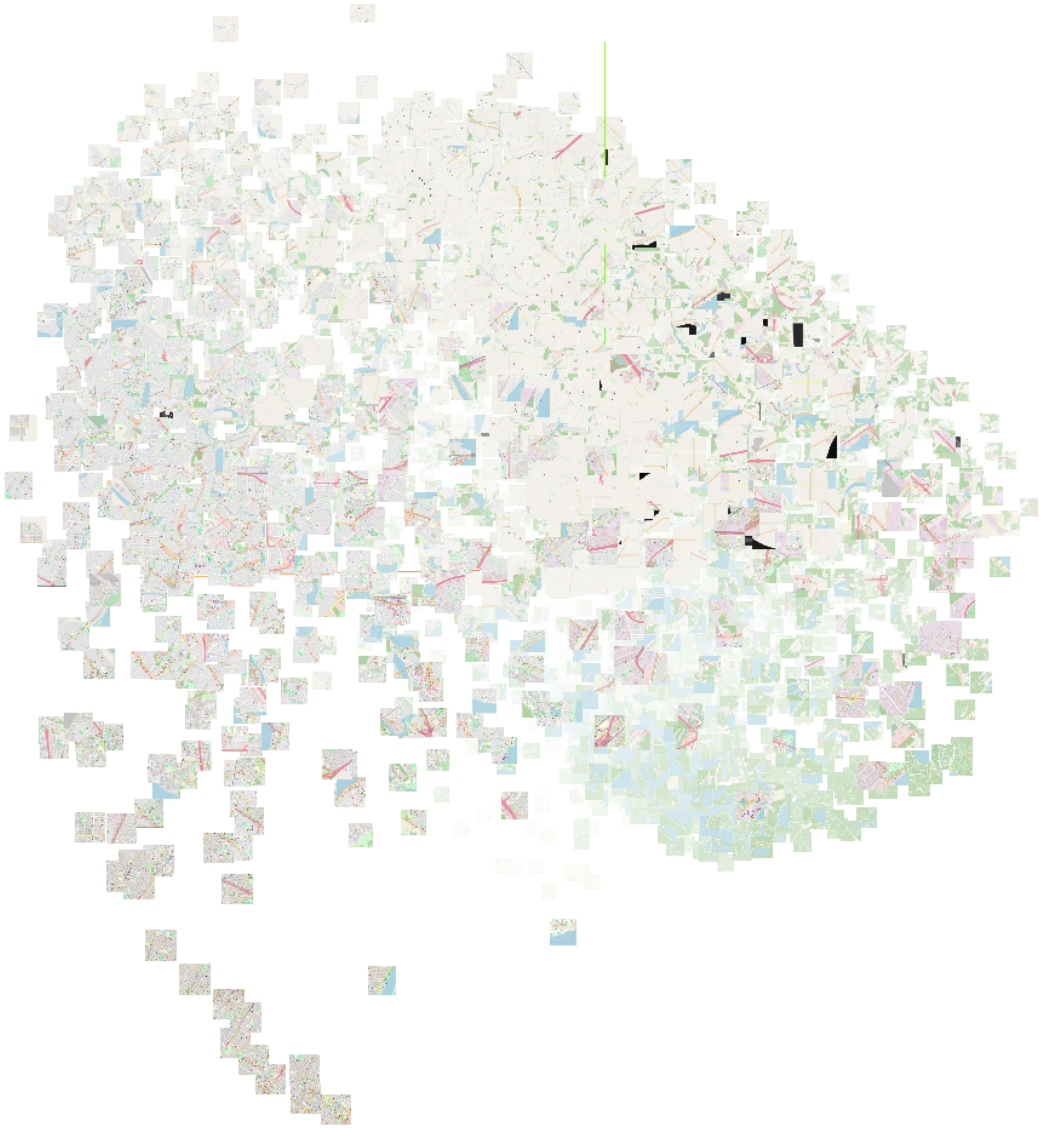

Figure 25: PCA cluster visualization

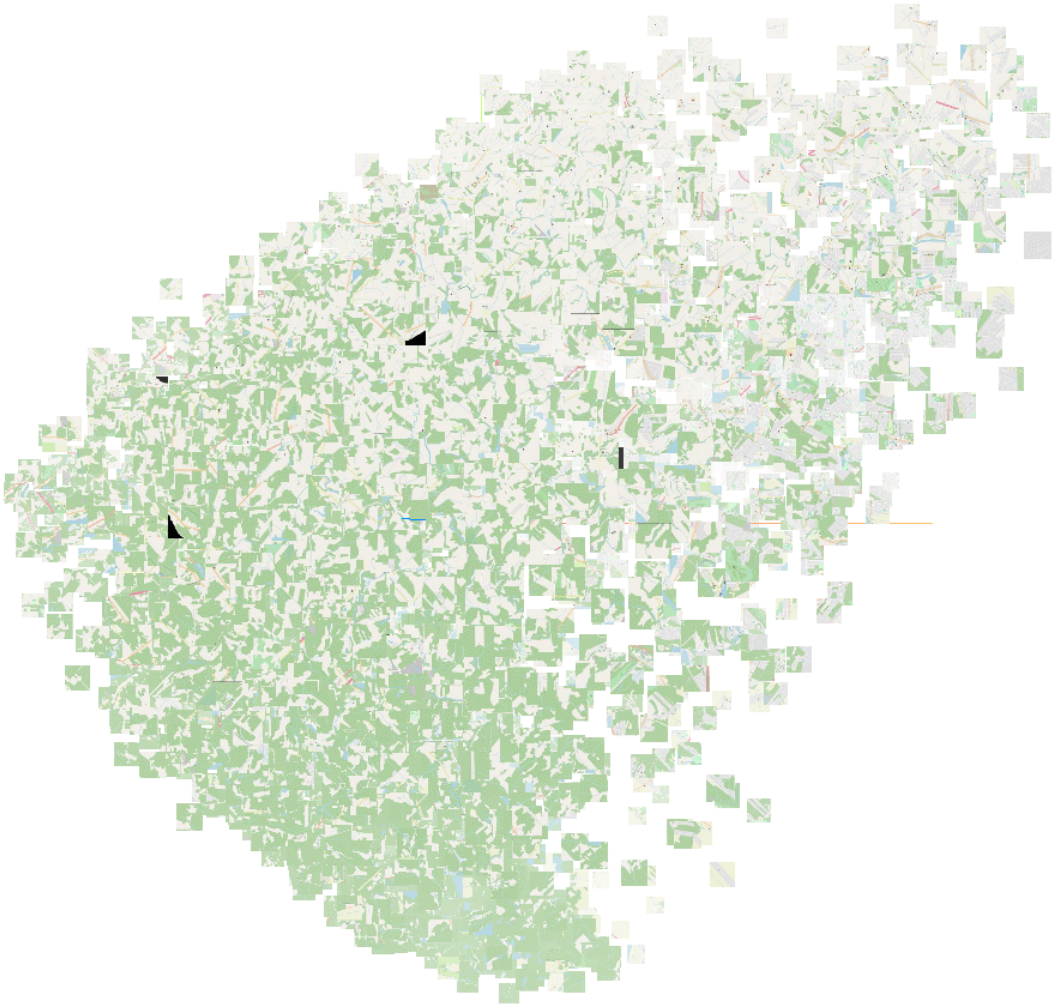

Figure 26: PCA cluster visualization

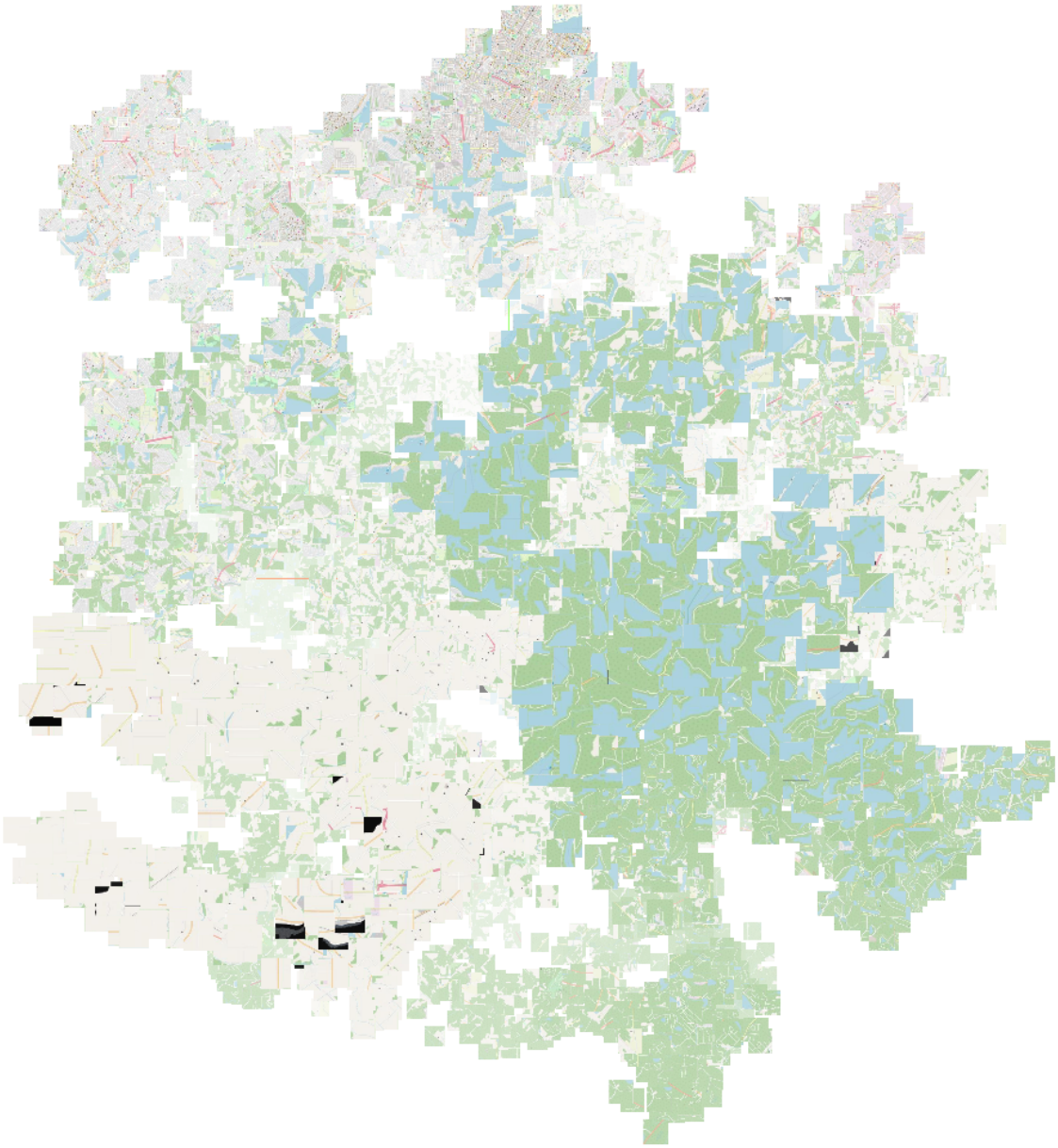

Figure 27: t-SNE cluster visualization

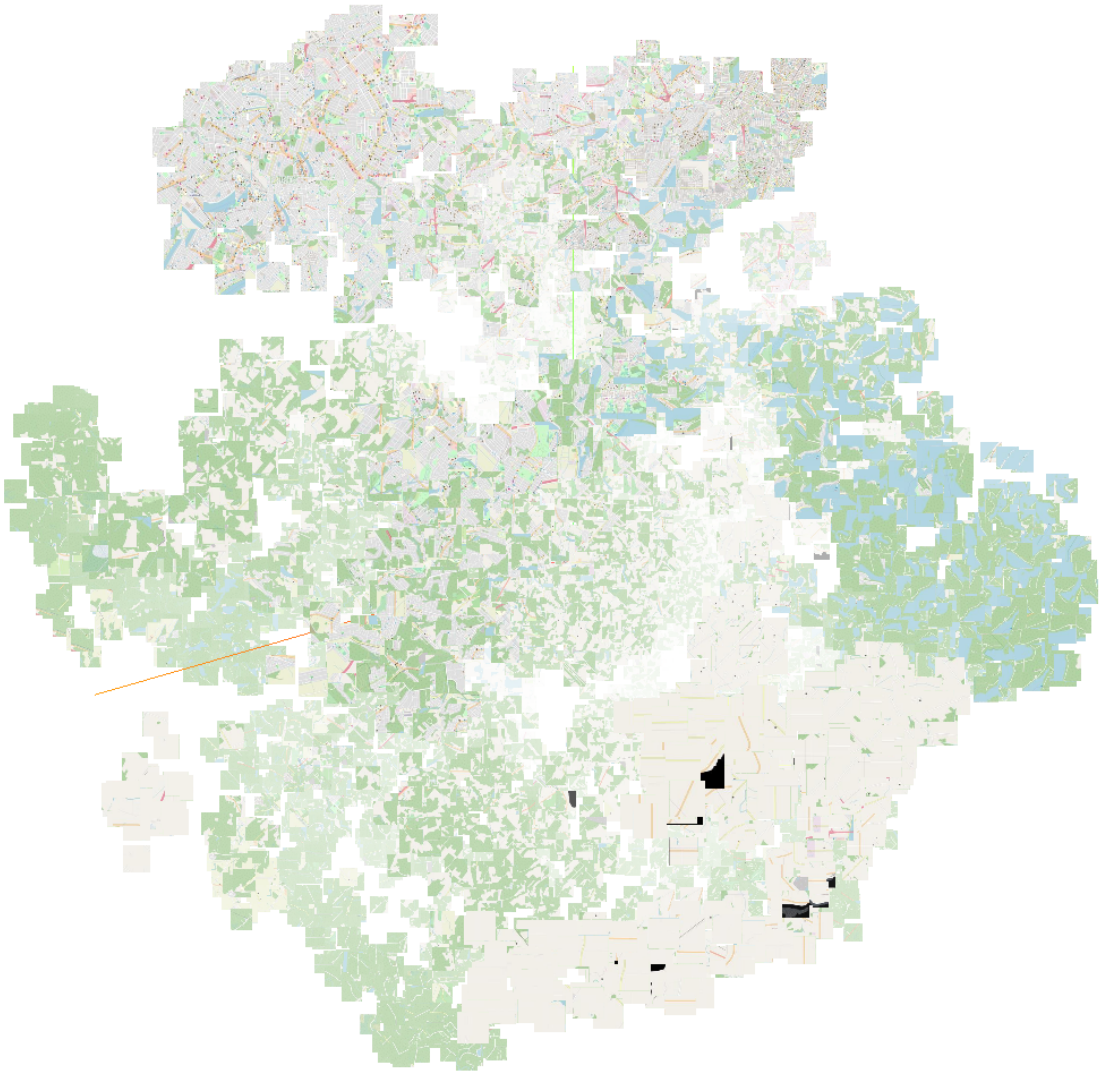

Figure 28: t-SNE cluster visualization

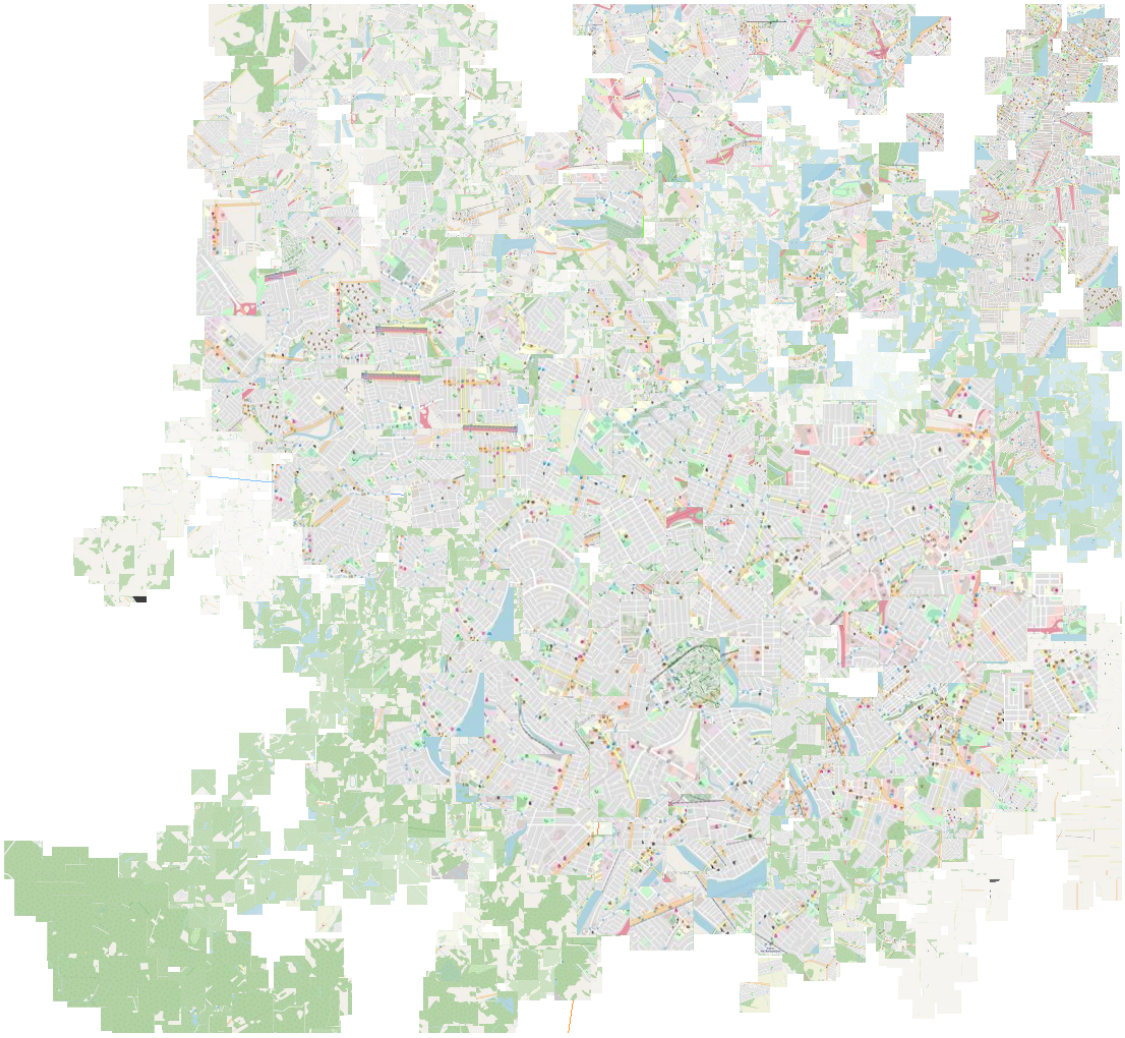

Figure 29: t-SNE cluster visualization

8 Discussion

This section is meant to discuss the results of the triplet network and the price prediction model, but also the problems encountered during the realization of this work and how these problems were tackled, or why they could not be solved.

8.1 Results and model evaluation

As stated before, the training of the triplet network is self-supervised and the data is not labelled, which makes the results difficult to asses. In order to overcome these issues and have insights on what the network learned and on the type of features captured by the embeddings, several visualizations are presented in sections 7.4 and 8.2.

The repartition of the tiles in the plots from section 7.4 is promising as it shows that the network learned something intuitively meaningful. Tiles with green areas, water, urban areas, lot of amenities, etc are clearly separated in distinct clusters. These plots are indicators that the three principal components of the embeddings contain informative and meaningful features.

The results of the second price prediction experiment presented in table 5 also tend to confirm that the embeddings learned meaningful information. Indeed, when using the embeddings as only features, the MAPE is seven percent better than the median predictor's error and ten percent better than the mean predictor's error. However, the results of the first price prediction experiment (which cannot be stated in this paper for confidentiality reasons) are only slightly better (the MAPE is reduced by one percent) when appending the embeddings to the original housing features. A deeper analysis of these results shows that there is a substantial correlation between the embeddings and some of the original house feature, which suggests that a part of the information explained by the embeddings is already explained by these features.

The price prediction models presented in section 6 give valuable insight on the information captured by the triplet network. However, it is not a reliable index of the embeddings' quality and should not be used as an evaluation metric for the embedding space, as the information learned by the triplet network is meant to be generic and not exclusively focused on real estate market prices. In other words, the performance of the embeddings in a particular use-case is not necessarily related to their performance in other use-cases.

8.2 Metric space visualization

As stated in section 3.2, the metric used in the triplet loss to compute the distance between two encoded vectors is the L2 norm. This implies that the embeddings reside in a metric space where the distance between two vectors is a direct indicator of the similarity between their respective tiles.

In this section, we explore the potential of this metric space through several visual representations.

8.2.1 Tiles algebra

Since the embeddings reside in a euclidean metric space, they obey to the basic operations of algebra. Concretely speaking, this means that the semantic features of two tiles can be combined by adding or subtracting their embeddings. Adding two embeddings result in a new vector. Since the triplet network does not act as a decoder, we cannot generate a new tile from the resulting vector. Therefore, the tile corresponding to the embedding that is the closest to the resulting vector is associated with it. Some examples of embeddings addition and subtraction operations are presented below. Of course, these are real examples: the embeddings used for these operations were generated by our triplet network. Instead of the embeddings, the vectors' associated tiles are displayed on the figures for ease of interpretation.

Figures 31, 33 and 34 give the impression that road types are properly assimilated by the network. In a similar fashion, it seems like features such as water, green areas, road patterns and amenities are also beautifully captured in the embedding space. In figure 30, it looks like the green areas and water of the left tile are combined with the transport amenities and the roads and buildings patterns of the right tile. Fifure 32 illustrates the combination of different landcovers and particular road patterns.

Figure 30: Tiles addition example

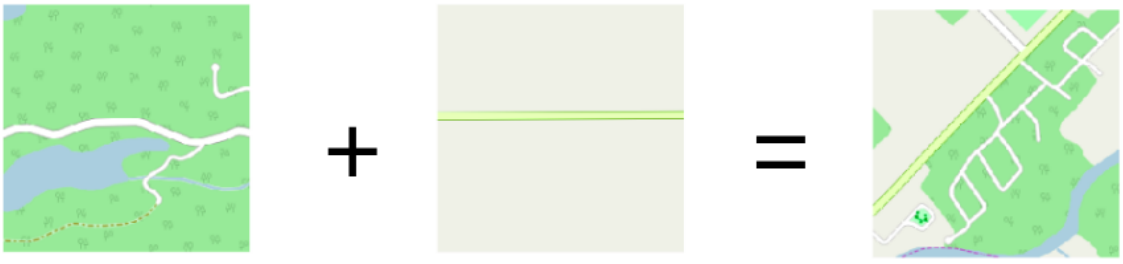

Figure 31: Tiles addition example

Figure 32: Tiles addition example

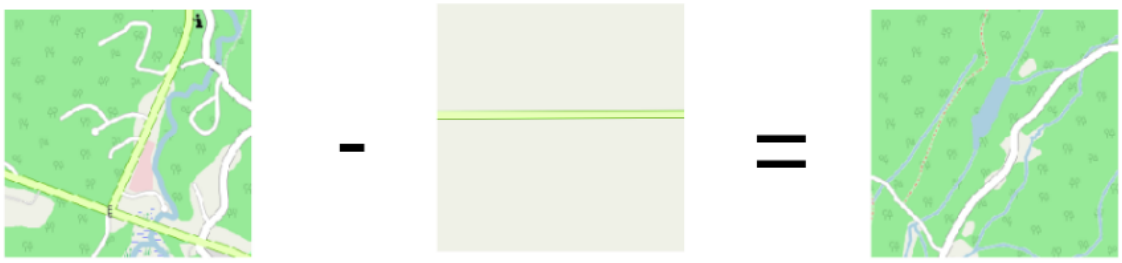

Figure 33: Tiles subtraction example

Figure 34: Tiles subtraction example

8.2.2 Random walk

Another illustrative operation that can be performed on the tiles embeddings is a simple random walk through the metric space. It is performed by selecting a random tile from the dataset and jumping to one of the n closest tiles. This procedure is then repeated 10 times in a loop in order to perform a random walk inside the metric space, jumping from tile to tile. Again, two tiles are considered *close* if the euclidean distance between their respective embeddings is small.

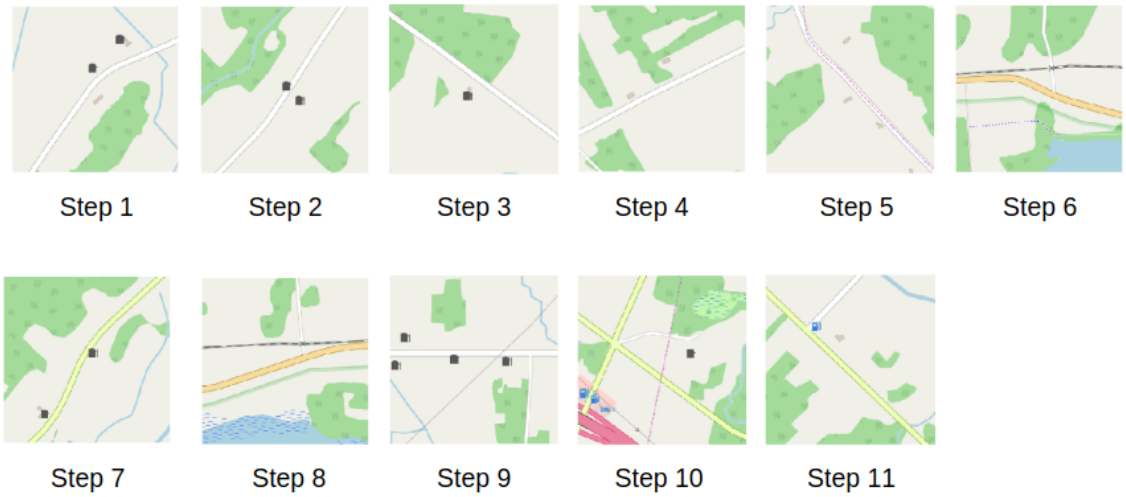

Figure 35: Random walk through the metric space

As expected, each randomly selected nearby tile presents attributes that are similar to its preceding tile's. Indeed, all the tiles have the same landcovers, road patterns, etc. It is also interesting to see on figure 35 that the same amenity (silos) appear on half of the tiles.

8.2.3 PCA to RGB

This section presents a way to detect a potential correlation between the semantic features of a place and its geographic location. This experiment is achieved by reducing the dimensionality of the tiles embeddings to three and scale the values of the embeddings to the [0,255] domain, such that each embedding becomes a tuple of three values comprised in [0,255] and can be interpreted as an RGB color. A colored dot is displayed on the map at each tile's location and its color corresponds to the tile embedding translated to RGB. To put it simply, two dots that have a similar color in the RGB space are also close in the metric space learned by the triplet network, and are therefore semantically similar.

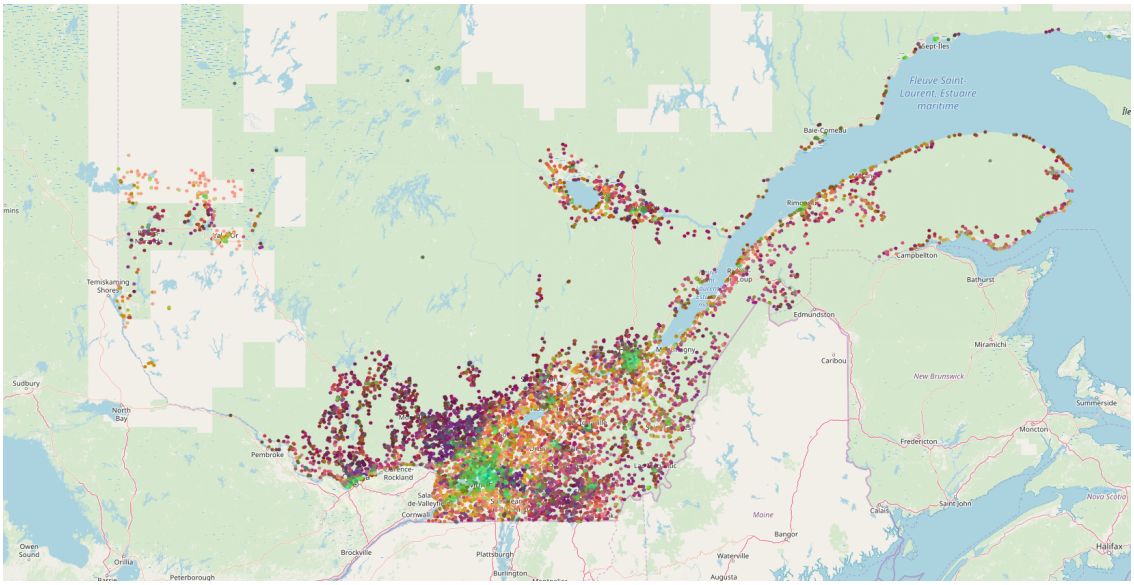

Figure 36: PCA to RGB map. Each point represents a tile from the test dataset. The RGB color code of each tile corresponds to its embedding's three principal components scaled to [0,255]. Therefore, similar colors indicate semantic similarity.

Interestingly, the map clearly presents three major categories of locations. Light green positions mainly correspond to densely populated urban areas (downtown areas). The main stacks correspond to the island of Montréal, Québec City and Ottawa. Smaller clusters can be seen around smaller cities like Trois-Rivières, Sherbrooke, Saguenay, Rimouski or Val-d'Or. The orange dots seem to represent periphery, i.e. urban areas around big cities, not as much populated as town centers. Last, the dark purple points correspond to rural areas and green spaces, as opposed to urban places.

8.3 Architectures and design choices

8.3.1 Depth

The three network architectures experimented in this work present different characteristics. TNet 1 is not very deep and does not have a lot of parameters compared to the two other architectures. Its purpose is to demonstrate that a shallow neural network (only two convolutional layers) is capable of learning a reasonably good embedding space for the task at hand. The shallow nature of this network makes it interesting for practical applications that require running the network with limited computing power.

TNet 2 and TNet 3 are pretty similar in terms of depth. The main difference between them resides in the use of batch normalization and dropout, as discussed in the following section.

8.3.2 Dropout and batchnorm

As stated in the results section, TNet 2 uses batch normalization after each hidden layer in order to improve the performance and stability of the network, but also to encourage each layer of the network to learn more independently of other layers.

TNet 2 is composed of seven hidden layers, and an output layer. Each convolutional layer applies batch normalization and max pool after the activation function. Note that in the original BatchNorm paper by Ioffe&Szegedy (cf. [IS15]), the batch normalization is applied *before* the activation function. However, in practice there are many examples where batch normalization is applied *after* the activation. In fact, François Chollet, author of Keras and currently AI researcher at Google declared:

"... I can guarantee that recent code written by Christian [Christian Szegedy, co-author of [IS15]] applies ReLU before BN. It is still occasionally a topic of debate, though.".

In his implementation of [Spr18], Vincent Spruyt supposedly used both batch normalization and dropout in the convolutional layers. In our case however, the use of dropout combined to batch normalization resulted in worse results. This phenomena could be explained by a possible disharmony between batch normalization and dropout, as stated by Li, Hu&Yang in [Li+18b].

As opposed to TNet 2, TNet 3 does not make use of batch normalization but aggressively uses dropout with a probability of 0.05 after each hidden layer, and with a probability of 0.2 after the output layer. Note that in the original Dropout paper by Hinton&al ([Hin+12]), dropout is only applied after fully connected layers but not after hidden layers. Although this became the state of the art for dropout in neural networks, Park&Kwak showed in [PK16] that using dropout both after fully connected and hidden layers, but with a lower probability in hidden layers can improve the stability of the network.

8.3.3 Two-dimensional convolutions and max-pooling

The convolutional layers used in all the architectures are two-dimensional convolutions. Basically this means that for any sample, the same convolution operation is applied by a 2D kernel on all the channels of the tensor. The reason behind this design choice is that the information contained in the different layers is categorical and not sequential, which is why we want a convolution along the two spatial dimensions, but not across channels. The same reasoning applies for max-pooling operations. The order of the channels within a tensor is not important, however it must remain consistent across all tiles.

8.4 Computational bottleneck

As mentioned in section 7.1, the training of the triplet network was performed on a Nvidia GeForce GTX 1080 Ti graphics card using Pytorch. Due to the amount of data used for training, each batch of tiles is loaded on the fly before being processed by the network. Before a tile can be processed by the GPU to update the parameters of the model the following operations must be performed:

- Loading the anchor tile and the negative tile from the disk to the CPU,
- Performing a certain number of image transformation on the anchor tile to generate the positive instance. These operations take place on the CPU,
- Loading the triplet of tiles from the CPU to the GPU.

This pipeline results in a performance bottleneck in the CPU, as these operations are computationally expensive. In short, the bottleneck is due to data loading and data augmentation (i.e. positive instance generation). This results in a clearly under-utilized GPU and ultimately in long training times (about 30 hours for 100 epochs), which makes benchmarks and grid searches difficult and time-consuming.

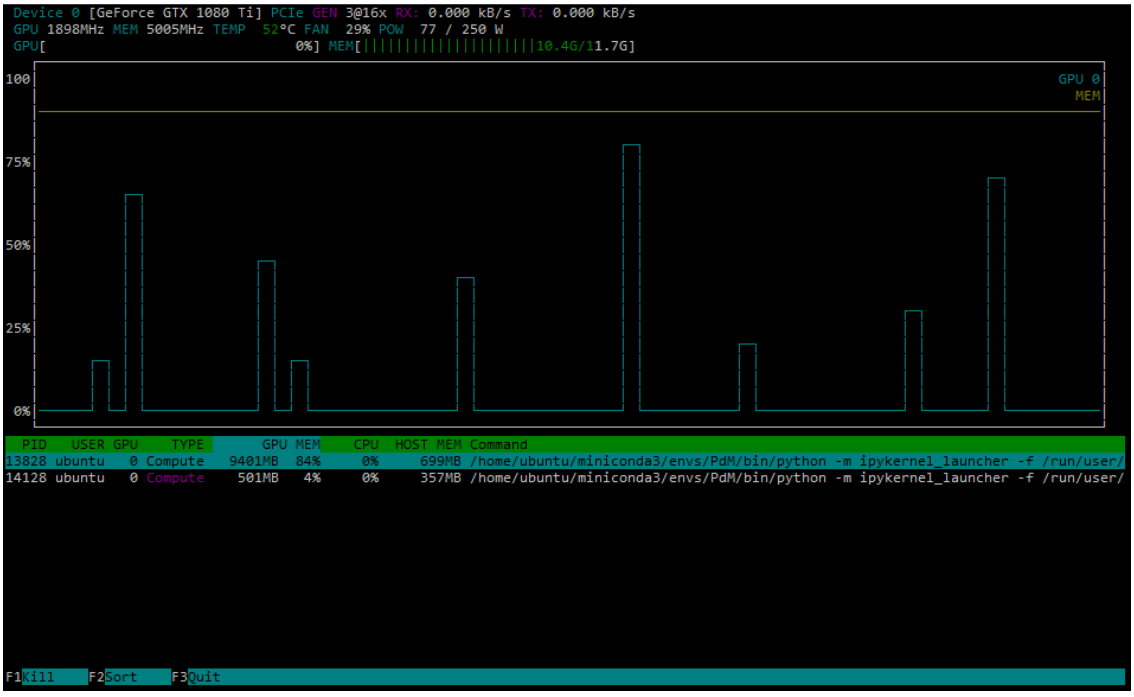

Figure 37: GPU utilization during training. The GPU is under-utilized due to a bottleneck during data loading and positive instances generation.

The cost of positive instance generation is partly due to the size of the tensors, as each transformation (padding, rotating, cropping, etc) is applied to each layer of a tensor. The multi-layered nature of the tensors also poses another problem, as Pytorch's torchvision.transforms package is only meant to perform transformations on images of at most four channels (i.e. RGBA images). This limitation is due to the PIL dependencies of the torchvision package. To overcome this issue, Github user *sshuair* implemented *torchvision-enhance* (see [ssh18]), a Pytorch-compatible plugin that implements transforms on tiff images with an arbitrary number of channels. As handy as this plugin is, it is unfortunately not optimal in terms of computational cost. Performance monitoring showed that this is due (among other things) to the scikit-image dependencies, and more particularly to the *warp* method.

A potential solution would be to perform positive instances generation on the GPU, as it is clearly under-utilized and the CPU is overloaded. However, this would require to re-implement all the necessary transformation matrices and methods from scrath and goes beyond the scope of this work. Hopefully torchvision will provide transform operations for multichannel images at some point in the future.

It is also worth noting that this work was conducted on Openstack machines in a research center. This implies that the bandwidth is shared with other people as well, which slows down the loading of the data from disk to memory (the disks are virtual).

8.5 Hard negative mining and hard positive mining

This section addresses the use of hard negative mining in this work. As stated before, hard negative mining is a triplet selection method that forces the network to keep learning for many iterations. Refer to section 4.4.1 for a formal description of the process.

A customized version of hard negative mining has been implemented and evaluated in order to see if the network keeps learning for a bigger number of epochs. The main difference between our implementation and the classical version of hard negative mining is that the original version generates n negative instances and selects the one that has the biggest negative distance (i.e. the hardest example), whereas our implementation generates a new negative instance until it results in a non-zero loss value, and gives up after n attempts. As a remainder, the triplet margin loss function is given by

$$\ell(\delta^+, \delta^-) = max(0, \mu + \delta^+ - \delta^-)$$

and a network that does not learn anything from a triplet is characterized by $\delta^- > \mu + \delta^+$, and therefore a loss of zero. To put it simple, it means that the network already performs good enough on the triplet and does not have anything new to learn from it.

The main reason for this customized way of mining negative examples is its cost. Indeed, because we don't necessarily need to load all n negative candidates, it's computationally cheaper than traditional hard negative mining and as stated in section 8.4, there is already a considerable bottleneck due to tiles loading and triplets generation.

In order to measure the impact of negative instances mining on the training procedure and determine its usefulness, a counter is implemented to compute the proportion of minibatches that benefit from this process. It turns out that only 0.5% of the minibatches benefit from hard negative mining. This brings to the conclusion that the small gain in training performance is not worth the computational overload caused by the additional I/O operations.

Hard positive mining might also help keep the network learning, however as explained in section 4.4.1, hard positive mining is extremely expensive in terms of CPU power due to the transformations applied on the tiles to generate positive instances and is therefore avoided in this work.

9 Conclusion and future work

As a conclusion, this work shows the embeddings capture meaningful and informative features as illustrated on all the visualizations from sections 7.4 and 8.2. The tiles algebra operations and the random walks provide evidence that the metric space created by the triplet network expresses a meaningful notion of distance between the tiles and the distinguishable clusters shown in section 7.4 uphold the idea that the embeddings contain semantic concepts that are closely related to human intuition. The PCA to RGB map (fig. 36) confirms the intuition of a correlation between geographical position and semantic location attributes.

Nevertheless, this evaluation rests solely on visual insights and is not as reliable as a true performance indicator. Although the visualizations presented in the results and discussion sections are strong indicators of meaningful semantic knowledge, a real evaluation metric provided by labelled data would be ideal. However, the tiles dataset is unlabelled and the triplet network learns in a self-supervised fashion. Although it is too specific to serve as an evaluation metric for the embeddings, the price prediction model presented in section 6 gives valuable insight on the information captured by the network. The triplet network was trained to produce generic embeddings, therefore it would be interesting to see how they perform in other use-cases from various domains.

The interest of this work lies not only in the exploration process but also in the great potential of applications for the embeddings. One of these applications was presented in section 6. We could also imagine a similar neighborhood recommender system, i.e. some kind of recommender system that, given a location as input, outputs semantically similar locations. This might be a useful application for real estate market companies. We could also imagine creating custom tiles to find a neighborhood that presents particular characteristics. Imagine for example a neighborhood that presents all the features that we want, except transport stations. We could create a custom tile, with all its channel zeroed-out except the *transport amenities* channel, which possesses a lot of amenities. This tile could be added with a tile from the aforementioned neighborhood in order to find a location that presents all the desired attributes.

References

[Bro+94] Jane Bromley et al. *Signature Verification using a "Siamese" Time Delay Neural Network*. 1994. URL: https://papers.nips.cc/paper/769-signature-verification-using-a-siamese-time-delay-neural-network.pdf.

[gra12] gravitystorm. *openstreetmap-carto*. https://github.com/gravitystorm/openstreetmap-carto. 2012.

[Hin+12] Geoffrey E. Hinton et al. *Improving neural networks by preventing co-adaptation of feature detectors*. 2012. eprint: arXiv:1207.0580.

[HA14] Elad Hoffer and Nir Ailon. *Deep metric learning using Triplet network*. 2014. eprint: arXiv:1412.6622.

[Wan+14] Jiang Wang et al. *Learning Fine-grained Image Similarity with Deep Ranking*. 2014. eprint: arXiv:1404.4661.

[IS15] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. eprint: arXiv:1502.03167.

[SKP15] Florian Schroff, Dmitry Kalenichenko, and James Philbin. "FaceNet: A Unified Embedding for Face Recognition and Clustering". In: (2015). DOI: 10.1109/CVPR.2015.7298682. eprint: arXiv:1503.03832.

[Bal16] Vassileios Balntas. *PN-Net: Conjoined Triple Deep Network for Learning Local Image Descriptors*. https://github.com/vbalnt/pnnet. 2016.

[Bal+16a] Vassileios Balntas et al. "Learning local feature descriptors with triplets and shallow convolutional neural networks". In: *Proceedings of the British Machine Vision Conference (BMVC)*. Ed. by Edwin R. Hancock Richard C. Wilson and William A. P. Smith. BMVA Press, Sept. 2016, pp. 119.1–119.11. ISBN: 1-901725-59-6. DOI: 10.5244/C.30.119. URL: https://dx.doi.org/10.5244/C.30.119.

[Bal+16b] Vassileios Balntas et al. *PN-Net: Conjoined Triple Deep Network for Learning Local Image Descriptors*. Jan. 2016. eprint: arXiv:1601.05030.

[PK16] Sungheon Park and Nojun Kwak. *Analysis on the Dropout Effect in Convolutional Neural Networks*. 2016. URL: http://mipal.snu.ac.kr/images/1/16/Dropout_ACCV2016.pdf.

[Bal17] Vassileios Balntas. *TFeat shallow convolutional patch descriptor*. https://github.com/vbalnt/tfeat. 2017.

[Dou17] Firdaouss Doukkali. "Batch normalization in Neural Networks". In: (Oct. 20, 2017). URL: https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c.

[Gup17] Harshvardhan Gupta. “One Shot Learning with Siamese Networks in PyTorch”. In: (July 15, 2017). URL: https://hackernoon.com/one-shot-learning-with-siamese-networks-in-pytorch-8ddaab10340e.

[Rib17] Edgar Riba. *TFeat shallow convolutional patch descriptor*. https://github.com/edgarriba/examples/tree/c3376d13/triplet. 2017.

[Vei17] Andreas Veit. *A PyTorch Implementation for Triplet Networks*. https://github.com/andreasveit/triplet-network-pytorch. 2017.

[AA18] Afshine Amidi and Shervine Amidi. “A detailed example of how to generate your data in parallel with PyTorch”. In: (Mar. 2018). URL: https://stanford.edu/~shervine/blog/pytorch-how-to-generate-data-parallel.

[Bie18] Adam Bielski. *Siamese and triplet learning with online pair/triplet mining*. https://github.com/adambielski/siamese-triplet. 2018.

[Isc+18] Ahmet Iscen et al. *Mining on Manifolds: Metric Learning without Labels*. 2018. eprint: arXiv:1803.11095.

[IHR18] Haque Ishfaq, Assaf Hoogi, and Daniel Rubin. *TVAE: Triplet-Based Variational Autoencoder using Metric Learning*. 2018. eprint: arXiv:1802.04403.

[Li+18a] Wenbin Li et al. *Online Deep Metric Learning*. 2018. eprint: arXiv:1805.05510.

[Li+18b] Xiang Li et al. *Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift*. 2018. eprint: arXiv:1801.05134.

[Spr18] Vincent Spruyt. “Loc2Vec: Learning location embeddings with triplet-loss networks”. In: (May 3, 2018). URL: https://www.sentiance.com/2018/05/03/venue-mapping/.

[ssh18] sshuair. *torchvision-enhance*. https://github.com/sshuair/torchvision-enhance. 2018.