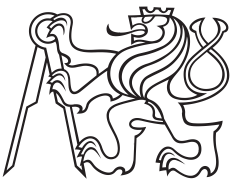
Master's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Estimating Bicycle Route Attractivity from Image Data

Vít Růžička
Master Programme: Open Informatics
Branch of Study: Computer Graphics and Interaction
ruzicvi3@fel.cvut.cz

July 2017
Supervisor: Ing. Jan Drchal, Ph.D.
Department of Computer Science

Acknowledgement / Declaration

I would like to thank ...

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

.....
Vít Růžička

Prague, 16 June 2017

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

.....
Vít Růžička

V Praze, 16. června 2017

Abstrakt / Abstract

tract

Klíčová slova: b

Překlad titulu: Odhadování atraktivity cyklistických tras z obrazových dat

abs

Keywords: deep convolutional neural networks, feature transfer, machine learning, computer vision

Contents /

1 Introduction 1
2 Research 2
2.1 Planning 2
2.1.1 Data collection 3
2.2 History of Convolutional Neural Networks 4
2.2.1 ImageNet dataset 4
2.2.2 ImageNet Large Scale Visual Recognition Competition 4
2.2.2.1 AlexNet, CNN using huge datasets .. 4
2.2.2.2 VGG16, VGG19, going deeper 5
2.2.2.3 ResNet, recurrent connections and residual learning 5
2.2.2.4 Ensemble models 5
2.2.3 Feature transfer 5
2.2.4 Common structures 6
3 The Task 8
3.1 Route planning for bicycles 8
3.2 Available data and collection 8
3.2.1 Initial dataset 8
3.2.2 Google Street View 8
3.2.2.1 Downloading Street View images .. 8
3.2.2.2 Data augmentation introduction ... 10
3.2.3 Neighborhood in Open Street Map data 10
3.2.3.1 OSM neighborhood vector 10
3.2.3.2 Radius choice 12
3.2.3.3 Data transformation 13
4 The Method 14
4.1 Building blocks 14
4.1.1 Model abstraction 14
4.1.2 Fully-connected layers ... 15
4.1.3 Convolutional layers 15
4.1.4 Pooling layers 15
4.1.5 Dropout layers 15
4.2 Open Street Map neighborhood vector model 16
4.3 Street View images model 18
4.3.1 Model architecture 18
4.3.2 Base model 19
4.3.3 Custom top model 19
4.3.4 The final architecture 19
4.4 Mixed model 20
4.5 Data Augmentation 21
4.6 Model Training 22
4.6.1 Data Split 22
4.6.2 Training setting 22
4.6.3 Training stages 23
4.6.4 Feature cooking 23
4.7 Model evaluation 23
4.8 Frameworks and projects 24
5 The Implementation 25
5.1 Project overview 25
5.2 Downloader functionality 25
5.3 OSM Marker 27
5.4 Datasets and DatasetHandler . 29
5.5 Models and ModelHandler 30
5.5.1 Model description in Keras 30
5.5.2 Model building 31
5.5.2.1 OSM only model ... 31
5.5.2.2 Images only model . 31
5.5.2.3 Mixed model 32
5.5.3 ModelHandler Structure 33
5.6 Settings structure 34
5.7 Experiment running 34
5.8 Training 34
5.9 Testing 34
5.10 Reporting and folder structure 34
5.11 Metacentrum scripting 34
6 Results 35
7 Discussion 36
8 Conclusions 37
References 38

Codes / Figures

5.1. RunDownload 26
5.2. Google Street View API url ... 26
5.3. Break down long edges..... 26
5.4. Dataset folder structure 27
5.5. Check downloaded dataset..... 27
5.6. Loading data into PostgreSQL database 28
5.7. Marking data with OSM vector 28
5.9. Data visualization..... 29
5.8. DatasetHandler..... 30
5.10. Building generic model in Keras 31
5.11. Fit generic model to data..... 31
5.12. OSM only model code 32
5.13. Applications 32
5.14. Feature cooking 32
5.15. Building mixed model..... 33
5.16. ModelHandler functions 33

2.1. Abstracted representation of map 2
2.2. Measurable features of real world road segment 3
2.3. Feature transfer illustrated 6
2.4. Generic CNN architecture formula 7
3.1. Sample of the initial dataset.....9
3.2. Initial bearing formula 9
3.3. Google Street View API url generation 9
3.4. Edge splitting and image generation scheme..... 10
3.5. OSM data structure with parameters 10
3.6. Sample of OSM attributes 11
3.7. Sample of OSM objects 11
3.8. Unique locations with neighborhood vectors 11
3.9. Construction of neighborhood vector..... 12
3.10. Generation of data entries from edge segment 12
4.1. Feature extractor and classificator..... 14
4.2. Classificator section formula... 15
4.3. Fully connected layer 15
4.4. Convolutional layer 16
4.5. Pooling layer 16
4.6. Dropout layer 17
4.7. Distinct locations from edge segment..... 17
4.8. OSM neighborhood vector CNN model..... 17
4.9. Reused base model with custom top model 18
4.10. Dimensionality of feature vectors..... 19
4.11. Top model structure 19
4.12. Image model structure 20
4.13. Structure of mixed model input data 20
4.14. Mixed model structure..... 21
4.15. Multiple OSM vector use 22
4.16. Image data augmentation..... 22
4.17. Mean squared error metric 23

4.18. Training stages schematics 23
4.19. Reusing saved image features from a file 24
4.20. K-fold cross validation 24
5.1. Project structure overview 25
5.2. No imagery available on Google Street View 27
5.3. Dataset object 29
5.4. Dataset visualization 29
5.5. ModelHandler Structure 33

Chapter 1
Introduction

Introduction, which chapter contains what.

Chapter 2
Research

2.1 Planning

There are many applications for the task to search for the shortest route on graph representation of map. We can abstract many real world scenario problems into this representation and then use many already existing algorithms commonly used for this class of tasks.

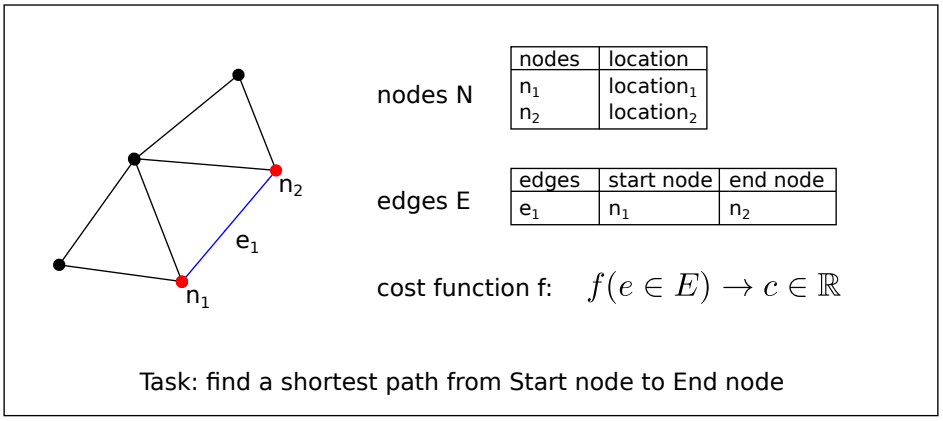


Figure 2.1. Abstracted representation of map, description of real world location via set of nodes and edges. This representation allows us to run generic algorithms over it.

We have a set of nodes N, which can be understood as places on map and edges E, which are the possible paths from one node to another, In order to have measurement of quality of traveling between two nodes, we also need a cost function. This cost function will assign a positive value c ∈ ℝ+ to each edge. Typically we are facing the task of finding the shortest path, in which we are minimizing the aggregated cost. See illustration of Figure 2.1.

In real life scenario, road segments exhibit many different parameters which influence how fast we can traverse them. More or less objective criteria such as surface material, size of the road, time of the day, or criteria which depends solely on the preference of driver. What is the surrounding environment, what is the comfort level of the road.

This tasks gets more interesting, when we look at more complicated examples, where the cost function is multi-criterial. In such case we need more data at our disposal and we can also expect the model to be more computationally difficult. For practical use, we need to use effective algorithm with speed up heuristics (see [1]). Great part of research is also in the area of hardware efficient algorithms, which would work on maps containing continent-sized datasets of nodes and edges, and yet coming up with solution in realistic time. We can expect such task on hand-held devices of car gps.

2.1.1 Data collection

Cost function can be very simple, but in order that it works on real life scenarios, we usually need more complicated one with lots of data recorded. Estimation of how much "cost" we associate with one street (represented by edge e ∈ E connecting two crossroads represented by nodes) should reflect how much time we spend in crossing it.

In case of planning for cars we generally just want to get across as fast as possible, or to cover minimizing distance. When the user is driving a bicycle, more factors become relevant. We need to know the quality of terrain, steepness of the road, amount of traffic in the area and overall pleasantness of the road. In many cases the bikers will not follow the strictly shortest path, choosing their own criteria, such as for example stopping for a rest in a park.

Some of these these criteria are measurable and objectively visible in the real world. For these we need to have highly detailed data available with parameters such as the quality of road and others. Other criteria are based on subjective, personal preference of some routes over other and for these we might need a long period of traces recording which routes have users selected in past. For example the work of [Navigation made personal] makes heavy use of user recorded traces.

See [1] and Figure 2.2 for examples of types of measurements we would likely need to estimate cost of each edge considering the slowdown effect of these features.

surface	∈	[cobblestone, compacted, dirt, glass, gravel, ground, mud, paving stones, sand, unpaved, wood]
obstacle	∈	[elevator, steps, bump]
crossing	∈	[traffic signals, stop, uncontrolled, crossing]
cycleway	∈	[lane, shared busway, shared lane]
highway	∈	[living street, primary, secondary, tertiary]

Figure 2.2. Example of the categories of highly detailed data we would require for multi-criteria cost function formulation as presented by [1]. List of features contributing to a slowdown effect on route segment.

In any case highly qualitative, detailed and annotated dataset is required to begin with and a carefully fitted cost function which would take all these parameters into weighted account is also needed. Large companies are usually protective of their proprietary formulas of evaluating costs for route planners. For example [2] makes use of the road network data of Bing Maps with many parameters related to categories such as speed, delay on segment and turning to another street, however the exact representation remains unpublished.

As we will touch upon this topic in later chapters, its useful to realize that this highly qualitative dataset is not always available. We would like to carry information we can infer from small annotated dataset into different areas, where we lack detailed measurements. We are using visual information of Google Street View images, which is more readily available in certain areas than the highly qualitative dataset.

2.2 History of Convolutional Neural Networks

Initial idea to use Convolutional Neural Networks (CNNs) as model was introduced in [3] by LeCun with his LeNet network design trained on the task of handwritten digits recognition.

In this section we try to trace the important steps in the field of Computer Vision which lead to the widespread use of CNNs in current state of the art research.

2.2.1 ImageNet dataset

Computer Vision research has experienced a great boost in the work of [4] in the form of image database ImageNet. ImageNet contains full resolution images built into the hierarchical structure of WordNet, database of synsets, "synonym sets" linked into a hierarchy.

WordNet is often used as a resource in the tasks of natural language processing, such as word sense disambiguation, spellcheck and other. ImageNet is a project which tries to populate the entries of WordNet with imagery representation of given synset with accurate and diverse enough images illustrating the object in various poses, viewing points and with changing occlusion.

As the work suggests, with internet and social media, the available data is plenty, but qualitative annotation is not, which is why such hierarchical dataset like ImageNet is needed. The argument for choosing WordNet is that the resulting structure of ImageNet is more diverse than any other related database. The goal is to populate the whole structure of WordNet with 500-1000 high quality images per synset, which would roughly total to 50 million images. Even till today, the ImageNet project is not yet finished, however many following articles already take advantage of this database with great benefits.

Effectively ImageNet became the huge qualitative dataset which was needed to properly teach large CNN models.

2.2.2 ImageNet Large Scale Visual Recognition Competition

The ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) [5] fueled the path of progress in the field of Image Recognition. While the tasks for each year slightly differs to encourage new novel approaches, it is considered the most prestigious competition in this field. The victorious strategies, methods and trends of each years of this competition offer a reliable looking glass into the state of art techniques. The success of these works and fast rate of progress has lead to popularization of CNNs into more practical implementations as is the case of Japanese farmer automatizing the sorting procedure of cucumbers [6].

2.2.2.1 AlexNet, CNN using huge datasets

The task of object recognition has been waiting for larger databases with high quality of annotation to move from easier tasks done in relatively controlled environment, such as the MNIST task. In the work of [7] the ImageNet database was used to teach a Deep Convolutional Neural Network (CNN) in a model later referred to as AlexNet. At the time this method has achieved more than 10% improvement over its competitors in an ILSVRC 2012 competition.

Given hardware limitations and limited time available for learning, this work made use of only subsets of the ImageNet database used in ILSVRC-2010 and ILSVRC-2012 competition. Choice of CNN as a machine learning model has been made with the

reasoning that it behaves almost as well as fully connected neural networks, but the amount of parameters of connections is much smaller and the learning is then more efficient.

For the competition an architecture of five convolutional and three fully-connected layers composed of Rectified Linear Units (ReLUs) as neuron models was used. Other alterations on the CNN architecture were also employed to combat overfitting, to fine-tune and increase score and reflect the limitations of hardware. Output of last fully-connected layer feeds to a softmax layer which produces a distribution over 1000 classes as a result of CNN.

The stochastic gradient descent was used for training the model for roughly 90 cycles through training set composed of 1.2 million of images, which took five to six days to train on two NVIDIA GTX 580 3GB GPUs.

2.2.2.2 VGG16, VGG19, going deeper

Following the successful use of CNNs in ILSVRC2012, the submissions of following years tried to tweak the parameters of CNN architecture. Approach chosen by [8] stands out because of its success. It focused on increasing the depth of CNN while altering the structure of network. In their convolutional layers they chose to use very small convolution filter (with 3x3 receptive field), which leads to large decrease of amount of parameters generated by each layer.

This allowed them to build much deeper architectures and acquire second place in the classification task and first place in localization task of ILSVRC 2014.

2.2.2.3 ResNet, recurrent connections and residual learning

The work of [9] introduced a new framework of deep residual learning which allowed them to go even deeper with their CNN models. They encountered the problem of degradation, where accuracy was in fact decreasing with deeper networks. This issue is not caused by overfitting as the error was increased both in the training and validation dataset.

Alternative architecture of model, where a identity shortcut connection was introduced between building blocks of the model, allowing it to combat this degradation issue and in fact gain better results with increasing CNN depth.

Their model ResNet 152 using 152 layers achieved a first place in the classification task of ILSVRC 2015.

2.2.2.4 Ensemble models

The state of the art models as of ILSVRC 2016 made use of the ensemble approach. Multiple models are used for the task and final ensemble model weights their contribution into an aggregated score.

The widespread of using the ensemble technique reflects the democratization and emergence of more platforms and public cloud computing solutions giving more processing power to the competing teams of ILSVRC.

2.2.3 Feature transfer

The success of large CNNs with many parameters trained on large datasets like ImageNet has not only been positive, it also poses a question – will we always need huge datasets like ImageNet to properly teach CNN models? ImageNet has millions of annotated images and it has been gradually growing over time.

Article [10] talks about this issue and proposes a strategy called feature transfer or model finetuning, which composes of using CNN models trained at one task to be retrained to a different task.

They offer a solution, where similar architecture of CNNs can be taught upon one task and then several of its layers can be reused, effectively transferring the mid-level feature representations to different tasks.

This can be used, when the source task has a rich annotated dataset available (for example ImageNet), whereas the target one doesn't. When talking about two tasks, the source and target classes might differ, when for example the labeling is different. Furthermore the whole domain of class labels can be also different – for example two datasets of images, where first mostly exhibits single objects and the second one rather contains more objects composed into scenes. This issue is referred to as a "dataset capture bias".

The issue of different class labels is combated by adding two new adaptation layers which are retrained on the new set of classes. The problems of different positions and distributions of objects in image is addressed by employing a strategy of sliding window decomposition of the original image, continuing with sensible subsamples and finally having the result classifying all objects in the source image separately.

The article also works with a special target dataset Pascal VOC 2012 containing difficult class categories of activities described like "taking photos" or "playing instrument". In this case the source dataset of ImageNet doesn't contain labels which could overlap with these activities, yet the result of this "action recognition" task achieves best average precision on this dataset.

This article gives us hope, that we can similarly transfer layers of Deep CNN trained on the ImageNet visual recognition source task to a different target task of Google Street View imagery analysis for cost estimation over each edge segment.

Articles being submitted to the ILSVRC competition often include a section dedicated on using their designed models on different tasks then what they were trained upon. Effectively they should the models suitability for feature transfering.

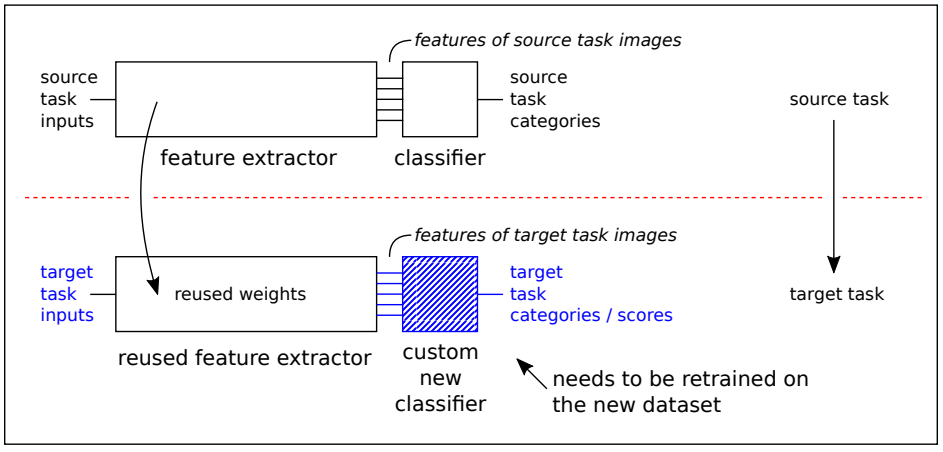


Figure 2.3. Illustration of the basic idea of feature transfer between source and target task. Imagine the source task source task being classification task on ImageNet and the target task as a new problem without large dataset at its disposal.

2.2.4 Common structures

When designing the architecture of custom CNN model, we are using certain layers and building block schemes established as common practice in the ILSVRC competition. For a new unresearched task it has been suggested (by lecturers and online sources such as [11]) to stick to an established way of designing the overall architecture.

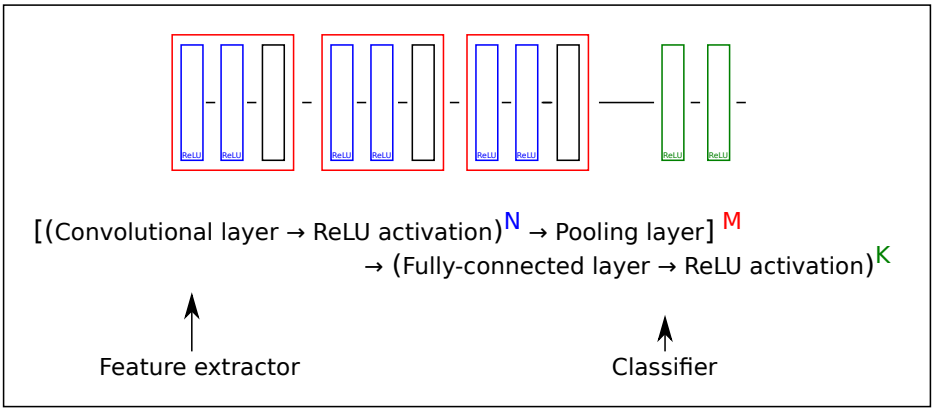


Figure 2.4. Formula describing a generic CNN architecture used with image data.

Refer to Figure 2.4. For illustration of this recommended architecture. Naturally for custom tasks this architecture is later adapted and tweaked to serve well in its specific situation. We will return to this suggested architecture scheme when building our own custom CNN models in 4.1.

Chapter 3
The Task

3.1 Route planning for bicycles

The task we are faced with consists of planning a route for bicycle on a map of nodes and edges. We are designing an evaluation method, which will give each edge segment appropriate cost. In such a way we are building one part of route planner, which will use our model for cost evaluation and fit into a larger scheme mentioned in 2.1.

As has been stated in 2.1.1 a cost function can be an explicitly defined formula depending on many measured variables. Similar formula has been used by the ATG research group, which produced a partially annotated section of map with scores of bike attractivity. We want to enrich this dataset with additional visual information from Google Street View and with vector data of Open Street Map.

We want to train a model on the small annotated map segment and later use it in areas where such detailed information is not available. We argue that Google Street View and Open Street Map data are more readily obtainable, than supply of highly qualitative measurements.

3.2 Available data and collection

3.2.1 Initial dataset

We are given a dataset from the ATG research group of nodes and edges with score ranking ranging from 0 to 1. Score of 0 denotes, that in simulation this route segment was not used and value 1 means that it was a highly attractive road to take.

Each node in supplied with longitude, latitude location, which gives us the option to enrich them with additional real world information. Figure 3.1. shows the structure of initial data source.

3.2.2 Google Street View

As each edge segment connecting two nodes is representing a real world street connecting two crossroads, we can get additional information from the location. We can download one or more images alongside the road and associate it with the edge and it's score from the initial dataset.

We are using a Google Street View API which allows us to generate links of images at specific locations and facing specific ways.

3.2.2.1 Downloading Street View images

Google Street View API uses the parameters of location which is the latitude and longitude and heading, which is the deviation angle from the North Pole in degrees. See Figure 3.3. In calculation of heading we are making a simplification of Earth being spherical using formula for initial bearing in Figure 3.2.

Edges.geojson

```
{
  "type":"FeatureCollection",
  "features":[
  {
    "type":"Feature",
    "geometry":{
      "type":"LineString",
      "coordinates":[
        [14.434785, 50.07245],
        [14.434735, 50.07255]
      ]                    ←location
    },
    "properties":{
      "length":13,
      "roadtype":"RESIDENTIAL",
      "attractivity":24    ←score
    }
  },
  ... more edges ...
}
```

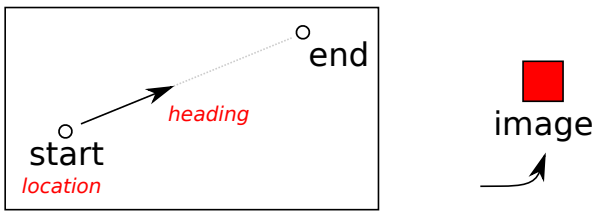
Nodes.geojson

```
{
  "type":"FeatureCollection",
  "features":[
  {
    "type":"Feature",
    "geometry":{
      "type":"Point",
      "coordinates":
        [14.434785,50.07245]
    },
    "properties":{
      "id":1109,
      "ele":250365
    }
  },
  ... more nodes ...
}
```

Figure 3.1. Sample of the structure of initial dataset.

$$y = \sin(lon_2 - lon_1) * \cos(lat_2)$$
$$x = \cos(lat_1) * \sin(lat_2) - \sin(lat_1) * \cos(lat_2) * \cos(lon_2 - lon_1)$$
$$\Theta = \operatorname{atan2}(y, x)$$

Figure 3.2. Formula for calculation of initial bearing when looking from one location (lon_1, lat_1) to another (lon_2, lat_2).



url: maps.googleapis.com/.../...&location=*location*&heading=*heading*

Figure 3.3. Illustration of Google Street View API url generation

In order to make good use of the location and collect enough data, we decided to break down longer edges into smaller segments maintaining the minimal edge size fixed. We also select both of the starting and ending locations of each segment. In each position we also rotate around the spot.

We collect total of 6 images from each segment, 3 in each of its corners while rotating 120 ° degrees around the spot. This allows us to get enough distinct images from each location and we don't overlap with neighboring edges. See the illustration in Figure 7. Note that all of these images will correspond to one edge and thus to one shared score value.

Also note that we are limited to downloading images of maximal size 640x640 pixels as per the limitation of free use of Google Street View API.

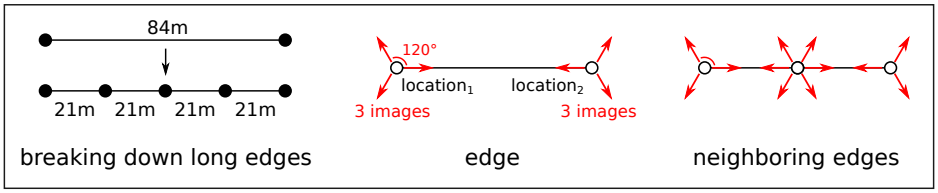


Figure 3.4. Splitting of initial possibly large edge segments into sections not smaller than the minimal length limit for edge segment. Each of these generates six images, which are usually not overlapping even with neighboring edges.

3.2.2.2 Data augmentation introduction

In CNNs we often use the method of data augmentation to extend datasets by simple image transformations to overcome limitations of small datasets. We can generate crops of the original images, flip and rotate them or alter their colors.

We will return to the issue of dataset augmentation in 4.5.

3.2.3 Neighborhood in Open Street Map data

We are looking for another source of information about an edge segment in the neighborhood surrounding its location in Open Street Map data.

Open Street Map data is structured as vector objects placed with location parameters and large array of attributes and values. We can encounter point objects, line objects and polygon objects, which represent points of interest, streets and roads, building, parks and other landmarks.

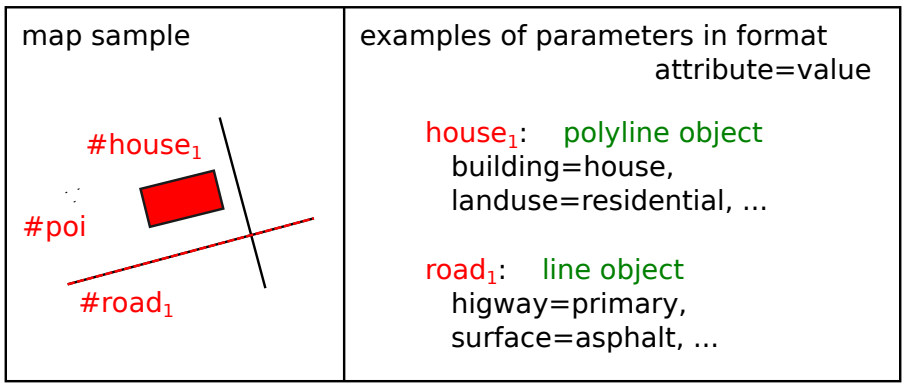


Figure 3.5. Example of structure of OSM data with parameters showing its properties.

From implementation standpoint, we have downloaded the OSM data covering map of our location and loaded it into a PostgreSQL database. In this way we can send queries for lists of objects in the vicinity of queried location. We will get into more detail about implementation in 5.3.

3.2.3.1 OSM neighborhood vector

The structure of OSM data consists of geometrical objects with attributes describing their properties. In the PostgreSQL database each row represents object and attributes are kept as table columns.

Depending on the object type, different attributes will have sensible values while the rest will be empty. For better understanding consult table in Figure 3.6 with examples of attributes and their values and table in Figure 3.7. for examples of objects in OSM dataset.

attribute	possible values
highway	residential, service, track, primary, secondary, tertiary, ...
building	house, residential, garage, apartments, hut, industrial, ...
natural	tree, water, wood, scrub, wetland, coastline, tree_row, ...
surface	asphalt, unpaved, paved, ground, gravel, concrete, dirt, ...
landuse	residential, farmland, forest, grass, meadow, farmyard, ...

Figure 3.6. Sample of interesting attributes and their possible values in OSM dataset.

objects	attribute 1	attribute 2	...
object1	val 1	val 2	...
object2	val 1	val 2	...
object3	val 1	val 2	...

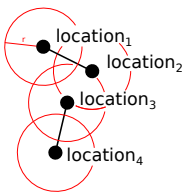
Figure 3.7. Example of values given to a selection of objects from OSM dataset. IMAGE IS STILL TEMPORARY, WILL BE FILLED WITH REAL DATA.

For example attribute “highway” will be empty for most objects, unless they are representing roads, in which case it reflects its size. We will be interested in counting attribute-value pairs, for example the number of residential roads in the area, which will have “highway=residential”.

Out of these pairs, we can build a vector of their occurrences. The only remaining issue is to determine which attribute-value pairs we select into our vector. If we used every pair possible, the vector would be rather large and more importantly mostly filled with zeros.

In order to select which pairs are important, we chose to look at OSM data statistics available at webpage taginfo.openstreetmap.org. From an ordered list of most commonly used attribute-value pairs, we have selected relevant pairs and generated our own list of pairs which we consider important.

Then for each distinct location of edge segment, we look into its neighborhood and count the number of occurrences of each pair from the list.



	0	1	2	3	...
	higway=primary	building=residential	natural=tree	surface=asphalt	...
$location_1$	1	3	0	2	...
$location_2$	2	7	0	5	...
$location_3$	2	12	1	5	...
$location_4$	1	20	6	0	...

Figure 3.8. Four unique locations with their corresponding neighborhood vectors. Note that these can be very similar for close enough locations.

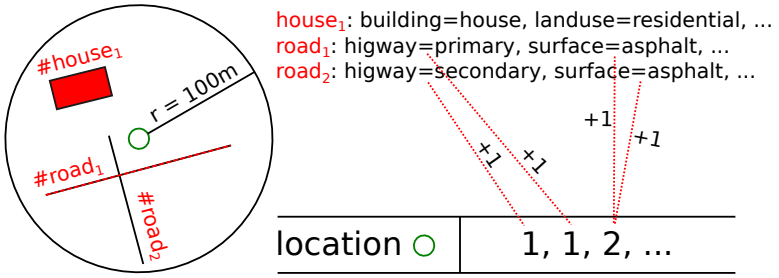


Figure 3.9. The construction of neighborhood vector from collection of nearby objects.

Each distinct location of each edge will end up with same sized OSM vector marking their neighborhood. Note that due to the method of downloading multiple images per one edge, for example by using the same location and rotating around the spot, some of these will have the same OSM vectors.

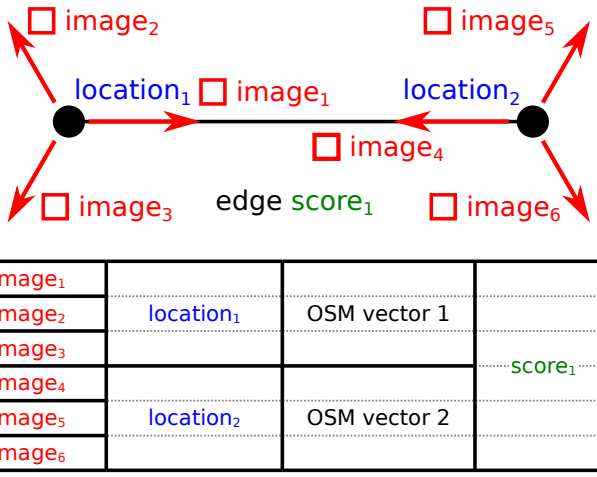


Figure 3.10. Example of further not divided edge segment, which generates data entries, where certain values overlap. For example the score is shared among all images produced from one segment.

One further undivided edge segment contains 6 images which share the same score, and some of which will share location and therefore also their OSM vectors. For illustration see Figure 3.10.

3.2.3.2 Radius choice

Depending on the radius we choose different area will be considered as neighborhood. If we were to choose too small radius, the occurrences would mostly result in zero OSM vector. On the other hand selecting too high radius would lead many OSM vectors to be indistinguishable from each other as they would share the exact same values.

Our final choice for radius was value of 100 meters, which was empirically tested in 6 Results.

3.2.3.3 Data transformation

Similar to the spirit of data augmentation for images, we can try editing the OSM vectors in order that they will be more easily used by CNN models.

Instead of raw data of occurrences, we can convert this information into one-hot categorical representations or reduce them into Boolean values. Multiple readings of varying radius size can also be used for better insight of the neighborhood area.

See more about data augmentation in 4.5.

Chapter 4
The Method

Our method will rely upon using Convolutional Neural Networks (CNNs) mentioned in 2.2 on an annotated dataset described in 3. As we have enriched our original dataset with multiple types of data, particularly imagery Street View data and the neighborhood vectors, we have an option to build more or less sophisticated models, depending on which data will they be using. We can build a model which uses only relatively simple OSM data, or big dataset of images, or finally the combination of both.

Depending on which data we choose to use, different model architecture will be selected. Furthermore we can slightly modify each of these models to tweak its performance.

4.1 Building blocks

Regardless of the model type or purpose, there are certain construction blocks, which are repeated in the architecture used by most CNN models.

4.1.1 Model abstraction

When building a CNN model, we can observe an abstracted view of such model in terms of its design. Whereas at the input side of the model we want to extract general features from the dataset, at the output side we stride for a clear classification of the image.

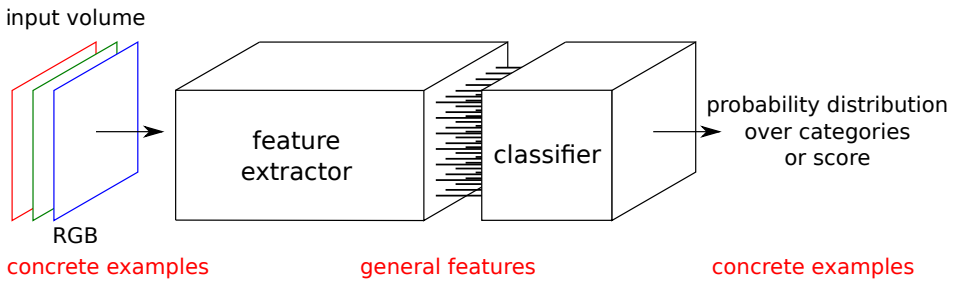


Figure 4.1. Abstraction of CNN model into feature extractor section and classificator section.

Each of these segments will require different sets of building blocks and will prioritize different behavior. Good model design will lead to generalization of concrete task-specific data into general features, which will then be again converted into concrete categories or scores. The deeper the generic abstraction is, the better the model behavior in terms of overfitting will be.

Classification segment transforms the internal feature representation back into the realm of concrete data related to our task. In our tasks we are interested in score in range from 0 to 1 as illustrated by Figure 4.2.

output = sigmoid(dot(input, weights) + bias)
output ∈ <0,1>
parameters

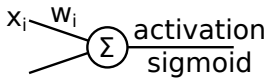


Figure 4.2. Classificator section formula describing how we generate one value at the end of CNN model ranging from 0 to 1 as a score estimate.

4.1.2 Fully-connected layers

The fully-connected layer denoted as "Dense" in Keras stands for a structure of neurons connected with every input and output by weighed connections. In Neural Networks these are named as hidden layers.

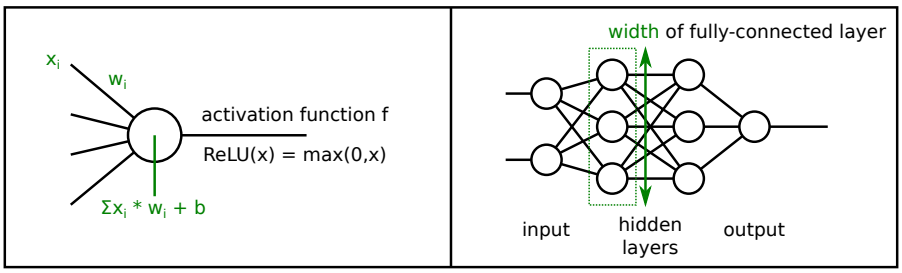


Figure 4.3. Shows the model of connections of neurons in fully connected layer.

The fully-connected layer suffers from a large amount of parameters it generates – weights in each connection between neurons and biases in individual neuron units. Fully-connected layers are usually present in the classification section of the model.

4.1.3 Convolutional layers

Convolutional layers are trying to circumvent the large amount of parameters of fully-connected layers by localized connectivity. Each neuron looks only at certain area of the previous layer.

For their property to use considerably less parameters while at the same time to focus on features present in particular sections of image, they are often used as the main workforce in the feature extractor section of CNN models.

4.1.4 Pooling layers

Pooling layers are put in between convolutional layers in order to decrease the size of data effectively by downsampling the volume. This forces the model to reduce its number of parameters and to generalize better over the data. Pooling layers can apply different functions while they are downsamplig the data – max, average, or some other type of normalization function.

4.1.5 Dropout layers

Dropout layer is special layer suggested by [12] which has since been widely used on the design of CNN architectures as a tool to prevent model overfitting.

The dropout layer placed between two fully-connected layers functions randomly drops connections between neurons with certain probability during the training period.

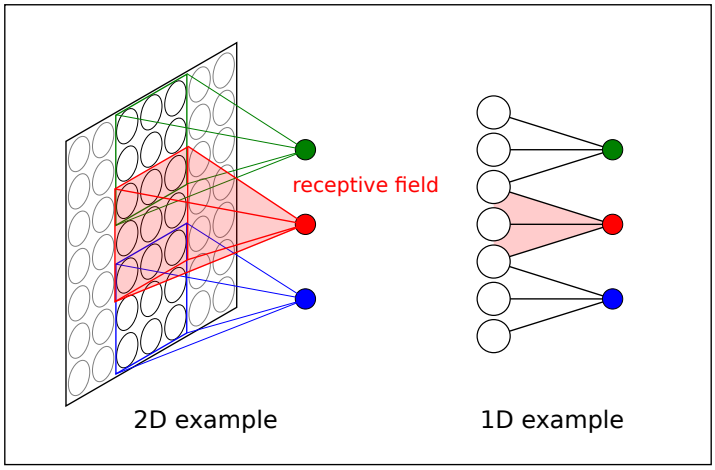


Figure 4.4. Schematic illustration of connectivity of convolutional layer. Connections are limited to the scope of receptive field.

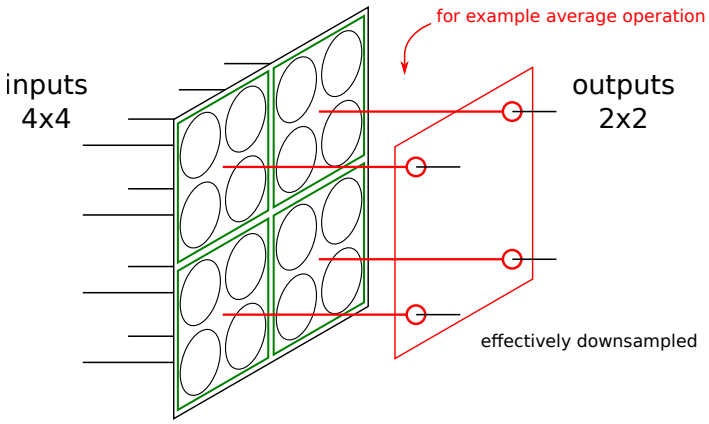


Figure 4.5. Pooling layer structure effectively downsampling the volume of input data.

Instead of fully-connected network of connections we are left with a thinned network with only some of the connections remaining. This thinned networks is used during training and prevents neurons to rely too much on co-adaptation. They are instead forced to develop more ways to fit the data as there is the effect of connection dropping. During test evaluation, the full model is used with its weights changed by the probability of dropout probability.

4.2 Open Street Map neighborhood vector model

In this version of model, we broke down edge segments formerly representing streets in real world into regularly sized sections each containing two locations of its beginning and ending location. These locations were enriched with OSM neighborhood vectors in 3.2.3.

Single unit of data is therefore a neighborhood vector linked to each distinct location of the original dataset. We have designed a model which takes these vectors as inputs and scores as outputs.

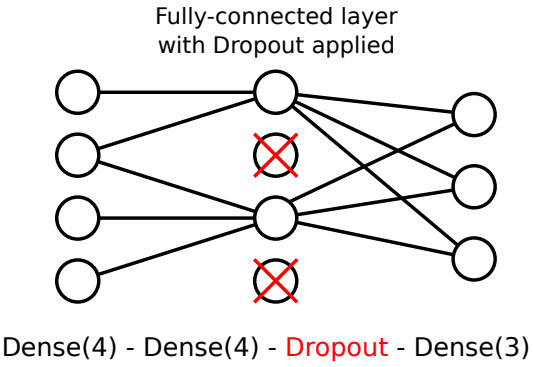


Figure 4.6. Illustration of dropout layers effect during the training period, which renders certain connection invalid with set probability. This prevents the network to depend too much on overly complicated formations of neurons.

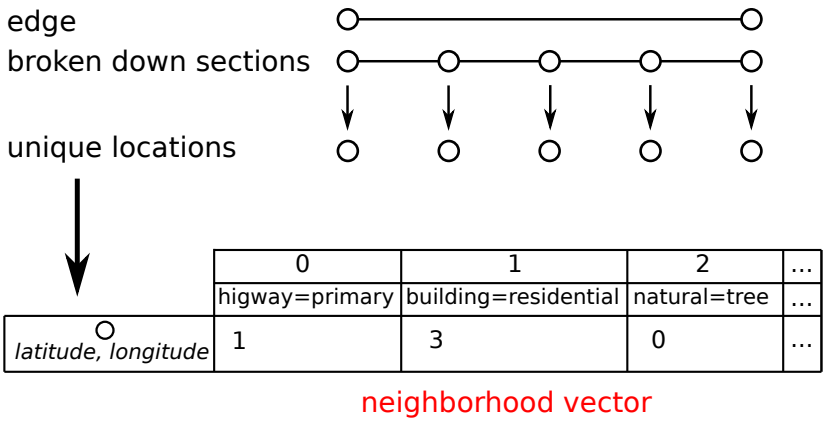


Figure 4.7. Edge segment broken down into set of distinct locations, where each of these locations is assigned its own neighborhood vector.

The OSM vector model is built from repeated building blocks of fully-connected layer followed by dropout layer. Fully connected layer of width 1 with sigmoid activation function is used as the final classification segment. Figure 4.8 shows the model alongside its dimensions.

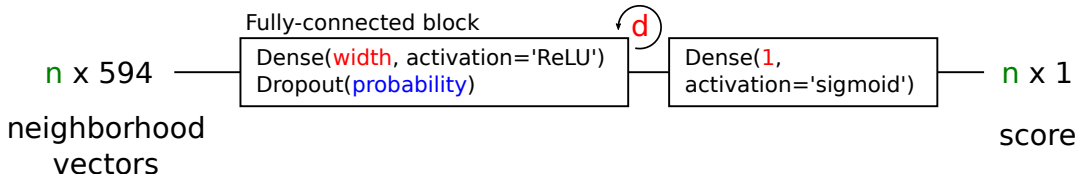


Figure 4.8. CNN model making use of only the neighborhood OSM vector data.

4.3 Street View images model

Each edge segment is represented by multiple images captured via the Street View API. Images generated from the same edge segment will share the same score label, however the individual images will differ. We can understand one image-score pair as a single unit of data.

The image data can be augmented in order to achieve richer dataset, see 4.5.

As discussed in 2.2 we are using a CNN model which has been trained on ImageNet dataset. We reuse parts of the original model keeping its weights and attach a new custom classification segment architecture at the top of the model.

We can generally divide even the more complicated CNN models into two abstract segments as mentioned in 4.1.1. The beginning of the model, which usually comprises of repeated structure of convolutional layers, followed by a classification section usually made of fully-connected layers.

The former works in extracting high dimensional features of incoming imagery data, whereas the classification section translates those features into a probability distribution over categories or score. In our case we are considering a regression problem model, which works with score instead of categories.

As was mentioned in 2.2.3 we reuse the model trained on large dataset of ImageNet, separate it from its classificatory and instead provide our own custom made top model. See Figure 4.9.

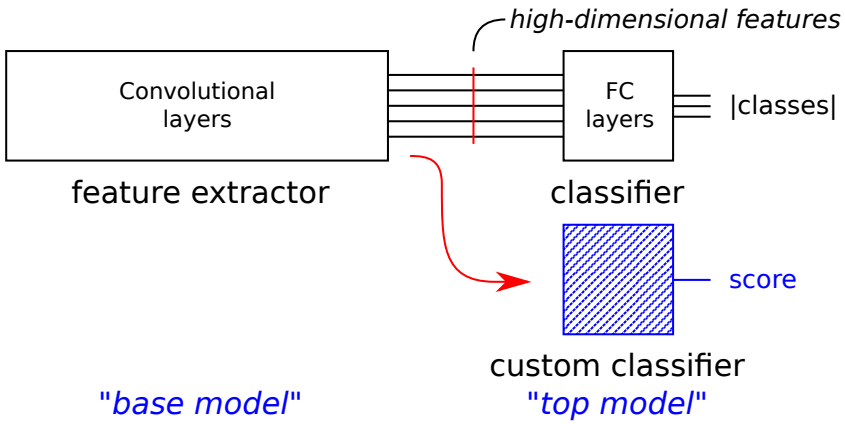


Figure 4.9. Reusing part of already trained CNN model as a base model and adding our custom top model.

We prevent the layers of base model from changing their weights and train only the newly attached top model for the new task. There are certain specifics connected with this approach which we will explore in 4.6.4 section.

4.3.1 Model architecture

The final model architecture is determined by two major choices: which CNN to choose as its base model and how to design the custom top model so it's able to transfer the base models features to our task.

4.3.2 Base model

The framework we are working with, Keras, allows us to simply load many of the successful CNN models and by empirical experiments asses, which one is best suited for our task. More about Keras in the appropriate section ??.

The output of what remains of the base CNN model is data in feature space. The dimension will vary depending on the type of model we choose, the size of images we feed the base models and also the depth in which we chose to cut the base CNN model.

The remains of base CNN model are followed by a Flatten layer which converts the possibly multidimensional feature data into a one dimensional vector, which we can feed into the custom top model.

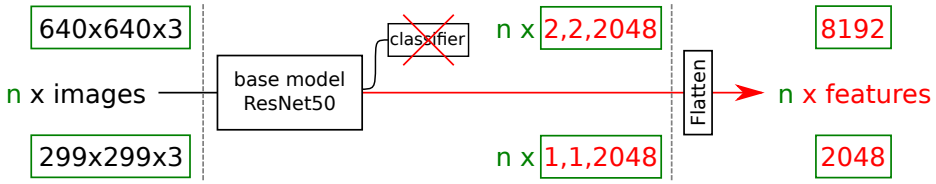


Figure 4.10. Example of differently sized images on the input which result in different feature vector size.

4.3.3 Custom top model

We feed the feature vector into a custom model built of repeated blocks of fully-connected neuron layers interlaced with dropout layers. The number of neurons used in each of the layers influences the so called model "width" and the number of used layers influences the model "depth". Both of these attributes influence the amount of parameters of our model. We can try various combinations of these parameters to explore the models optimal shape.

The final layer of the classification section consists of fully-connected layer of width 1 with sigmoid activation function which weighs in all neurons of the previous layer. See Figure 4.11. Note that in the final model we chose to interlace individual fully-connected layers with dropout layers.

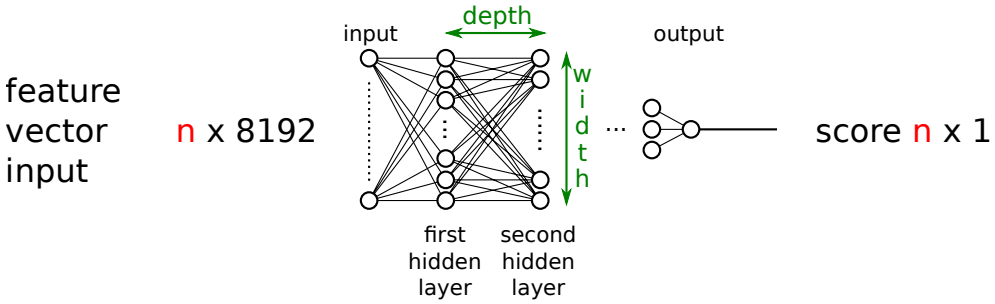


Figure 4.11. Top model structure which takes in the feature vector and follows with fully connected layers which are retrained on our own task.

4.3.4 The final architecture

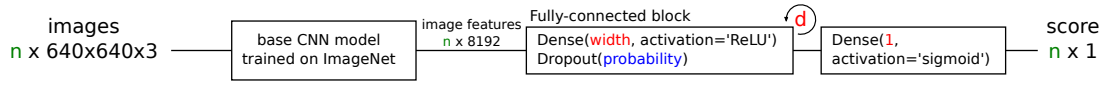


Figure 4.12. Final schematic representation of CNN model using Google Street View images. Note that parameters of width and depth can change its performance.

The final architecture composes of base CNN model with its weights trained on the ImageNet dataset and of custom classification top model trained to fit the base model for our task.

4.4 Mixed model

After discussing the architecture of two models making only a partial use of the data collected in our dataset, we would like to propose a model combining the two previous ones.

In this case one segment again generates multiple images, which share the same score and depending on how they are created they could also share the same neighborhood OSM vector representing the occurrences of interesting structures in its proximity. Different edges will generate not only different images, but also different scores and OSM vectors.

As a single unit of data we can consider the triplet of image, OSM vector and score. It's useful to note that later in designing the evaluation method of models, we should take into account, that the neighborhood vector and score can be repeated across data. When splitting the dataset into training and validation sets, we should be careful and place images from one edge into only one of these sets. Otherwise data with distinct images, but possibly the same neighborhood vector and score could end up in both of these sets. Figure 4.13 illustrates how single data units are generated from one edge.

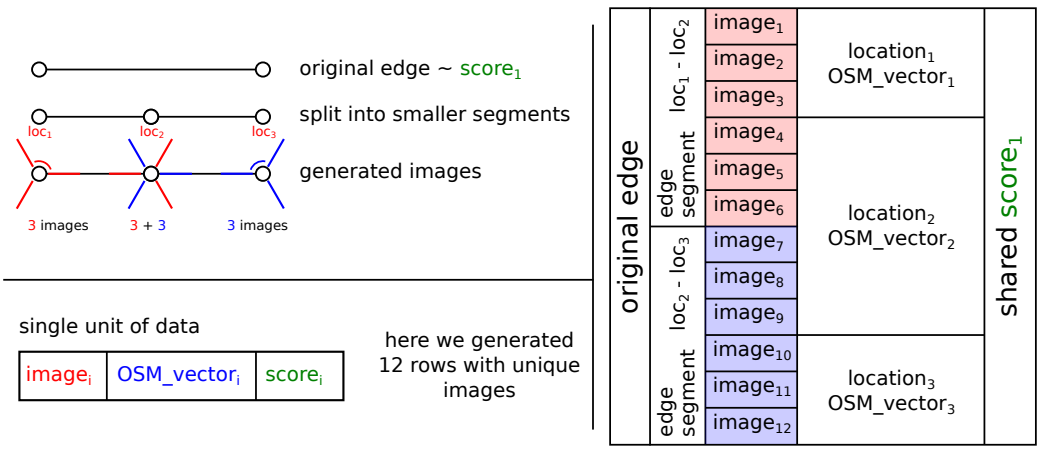


Figure 4.13. Example of possible decomposition of original edge into rows of data accepted by the mixed model. We use both the image data and the OSM neighborhood vectors.

We can join the architectures designed in previous steps, or we can design a new model. We chose to join the models in their classification segment. We propose a basic idea for a simple model architecture, which concatenates feature vectors obtained

in previous models and follows with structure of repeated fully-connected layers with dropout layers in between. Concatenation joins the two differently sized one dimensional vectors into one. As is observable on Figure 4.14 we use several parameters to describe the models width and depth.

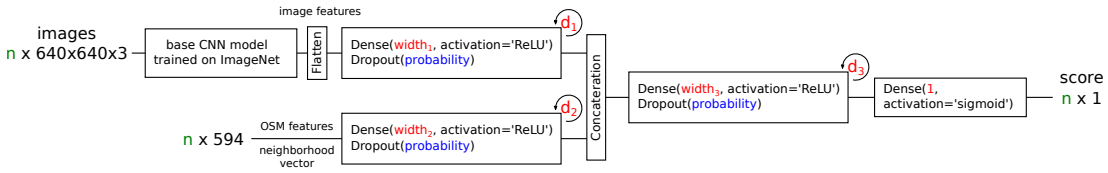


Figure 4.14. Final architecture of the mixed model, which uses both OSM vector and the image data.

4.5 Data Augmentation

When using complicated models with high number of parameters on relatively small datasets, the danger of overfitting is always present. We would like to combat this by expanding our dataset with the help of data augmentation.

Overfitting occurs when the model is basically able to remember all the samples of the training dataset perfectly and incorporate them into its structure. It achieves very low error on the training data, but looses its ability to generalize and results in comparably worse results on the validation set.

The idea of data augmentation is to transform the data we have in order to get more samples and a model which in turn behaves better on more generalized cases.

This cannot be done just blindly, as some of these transformations could mislead our model (for example left to right vertical flip makes sense in our case, but a up side down horizontal flip wouldn't).

There are multiple ways we can approach the problem of generating as many images from our initial dataset as we can. Before getting to the data augmentation aspect, please note, that this is also the reason, why we are generating multiple images per segment. We stand in two corners of each edge segment and rotate 120 ° degrees to get three images on each side. We have also employed a technique, which splits long edges into as many small segments as possible, while not hitting the self imposed minimal edge length.

It could be debated, that we could rotate for smaller angle or split edges to even smaller segments in order to take advantage of the initial dataset fully. However we came across an issue, that with too small minimal edge length or with different rotation scheme, we obtain very similar images, which actually do not improve the overall performance. This occurs when we don't generate differing enough images during downloading. For actual performance change see chapter 6.

We face similar issue when choosing a radius for obtaining OSM neighborhood vector as specified in the section 3.2.3. Instead of selecting one particular radius setting, we can make use of results of multiple queries. When building the OSM vector we would effectively multiply its length by concatenating it with other versions of OSM vectors. We could concatenate the vectors acquired with one fixed radius setting with another version with different radius. See Figure 4.15 for illustration.

Finally we also come across the method of data augmentation by transformation of the original image dataset. Certain operation, such as vertically flipping the image

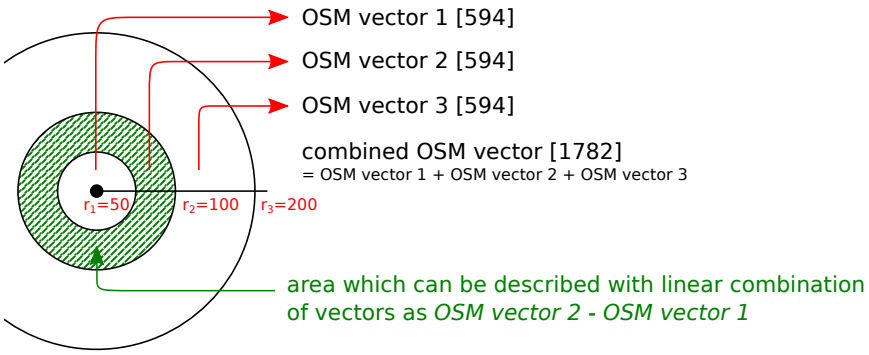


Figure 4.15. Illustration of the construction of combined OSM vector and the expected result of more specific area targetting by the model.

make sense for our dataset. (Also the normalization of color palete can be used in our task.) We show an example of images undergoing such transformation on Figure 4.16. Note that in this particular example we chose vertical flipping alongside with clipping of 90% of the image while making up for the lost 10%. These operations are random and the resulting images are added to create a larger dataset. For the sake of repetition of experiments with the same data, we save these generated images into an expanded dataset.

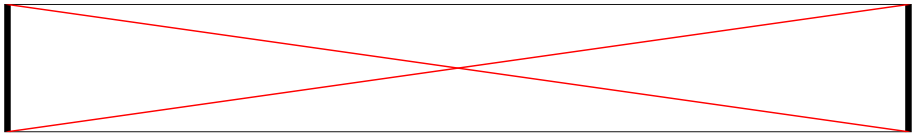


Figure 4.16. Data augmentation example.

4.6 Model Training

4.6.1 Data Split

Traditionally we split our dataset into two sets – training set, which we use for training of model and so called validation set, which is used only for models performance evaluation. As we discuss in 4.7, we employ more complex strategy of k-fold cross validation test to obtain more precise results.

The difference between the error achieved on training data and on validation data can be used as a measure of our model overfitting.

4.6.2 Training setting

We are using backpropagation algorithm to train our models as is supported by the selected Keras framework. We can choose from a selection of optimizers, which control the learning process. We made use of the more automatic optimizers supported by Keras, such as 'rmsprop' and 'adam' (see [13]). For greater parametric control we can also select the SGD optimizer.

Given the nature of our task, we are solving a regression problem, trying to minimize deviation from scored data. In most models mentioned in 2.2.2, the task revolves around selecting the correct category to classify objects.

Accordingly we have to select appropriate loss function. We have selected the mean squared error metric with the formula given by Figure 4.17.

$$MSE = \frac{1}{n}\sum_{i=0}^{n}(score_i - estimate_i)^2$$

Figure 4.17. Mean squared error metric used as loss function when training models.

4.6.3 Training stages

As has been discussed in 4.3.1, we are building models by reusing base of other already trained CNN models. In our case we make use of weights loaded from model trained on the ImageNet dataset.

We attach a custom classifier section to base model and try to train it on a new task. In order to preserve the information stored in connections of the base model, we lock its weights and prevent it from retraining. The only weight values which are changing are in the custom top model. This can be understood as a first stage of training the model as illustrated on Figure 4.18.

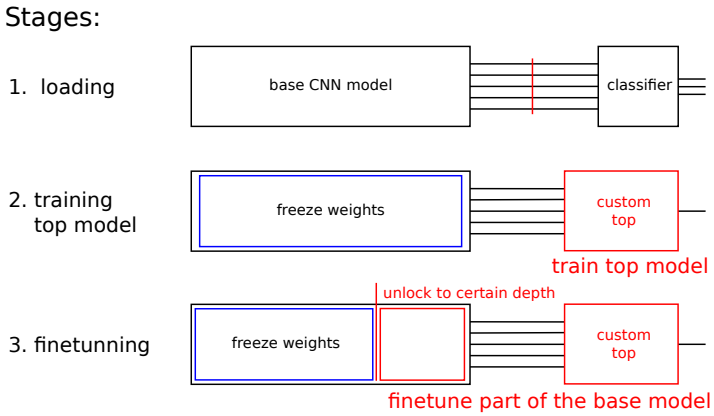


Figure 4.18. Training stages illustrated for feature transfer approach.

This is sometimes followed with a finetunning period, where we unlock certain levels of the base CNN model for training. However this is commonly done with customized optimizer setting, so that the changes to the whole model weights are not too drastic. We are also usually not retraining the whole model, because of the computational load this would take.

4.6.4 Feature cooking

This method is specific to situation when we are training a model with parts, which are frozen, as is in the case of 4.3 image and 4.4 mixed model design. We can take advantage of the fact that certain section of the model will never change and precompute the image features for a fixed dataset.

In this way we can save computational costs associated with training the model. In the end the original model can be rebuilt by loading obtained weight values.

This allows for fast prototyping of the custom top model, even if the whole model composes of many parameters in the frozen base model. For illustration see figure 4.19.

4.7 Model evaluation

As was mentioned in 4.6 we are using the practice of splitting dataset into training and validation dataset, with the k-fold cross validation technique.

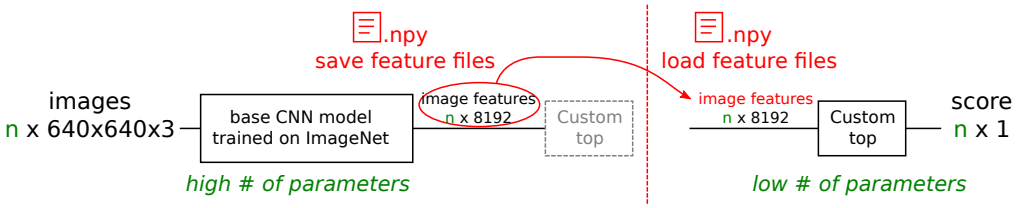


Figure 4.19. Reusing saved image features from a file instead of costly computations.

In order to prevent from being influenced by the selection bias, we split our entire dataset into k folds and then in sequence we use these to build training datasets and validation datasets. Every fold will take role of validation set for a model, which will train on data composed from all the remaining folds. Each of these will run a full training ended by an evaluation giving us score. Eventually we can calculate the average score with standard deviation.

This approach obviously increases the computational requirements, because it repeats the whole experiment for each fold. It is not used while prototyping models, but as a reliable method to later generate score. We have chosen the number of folds to be 10, as is a traditionally recommended approach.

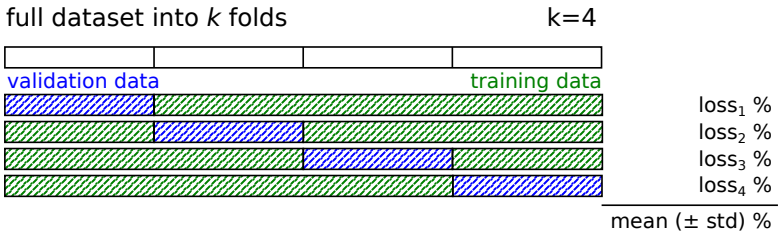


Figure 4.20. Splitting schema employed in the k-fold cross validation. We are given average score alongside with its standard deviation.

4.8 Frameworks and projects

The existence of this thesis depends on couple of frameworks which helped it greatly. Framework called “Keras” allowed for fast prototyping of Deep CNN models as well as efficient training environment. Thanks to the Metacentrum project we had access to machines with high enough computational power needed to teach these models.

Chapter 5
The Implementation

5.1 Project overview

In planning of the composition of project code, we have somewhat separated the sections responsible for downloading data, from those managing the dataset and modeling and running experiments.

Downloader is tasked to download images from Google Street View API from the initial dataset of edges and nodes mentioned in 3.2.1. Segment object works as a unit holding information about edge and its corresponding images and score. See 5.2.

However for later working with data, we have created a DatasetHandler which contains all the necessary functions. See 5.4.

To run more instances of models and later evaluate them, we have chosen to build individual experiments from custom written setting files in 5.6. Experiment Runner provides the common framework for all these more complicated computations in 5.7. In Settings folder we hold setting files defining each experiment.

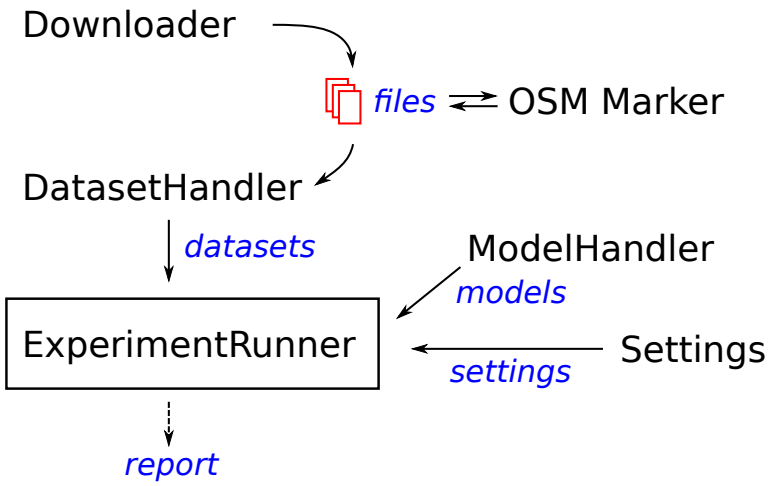


Figure 5.1. Project structure in a schema.

5.2 Downloader functionality

The Downloader is the main method of acquiring imagery data and preparation of dataset. It first creates necessary folders according to the directory path and custom name we provide it with.

Defaults.py contains default settings for the downloader, such as default number of times the code should try downloading each image and internal codes with which

```
def RunDownload(segments_filename, json_file_path):
# Main downloading function
  Segments <- PrepSegments(json_file_path):
    # parses JSON file
    Segments <- Segment(Start, End, Score, Id)
    return Segments

  FilenameMap <- GenListOfUrls(Segments):
    for Segment in Segments:
      split long Segment into fractions
      for [point_start, point_end] in fractions:
        urls, filenames <- betweenPoints(point_start, point_end)
        # generates three images from standing at point_start
        # and three images from standing at point_end
        return urls, filenames
    # build list of urls and filenames to download

  DownloadUrlFilenameMap(FilenameMap, Segments):
    for [url, filename, segment_id, nth_image] in FilenameMap:
      loaded, error <- url_retrieve_with_retry(url, filename)
      Segments[segment_id].HasLoadedImages[nth_image] = loaded
      Segments[segment_id].ErrorMessage[nth_image] = error

  SaveDataFile(Segments, segments_filename) # save pickled array
```

Code 5.1. RunDownload code sample.

to mark unsuccessfully downloaded segments. Besides these internal representation settings, it also controls the pixel size of downloaded images.

Downloading procedure is initiated in RunDownload() method (see Code 5.1), which parses node and edge data in provided GeoJSON files and build an array of Segments.

For each segment in this array, we generate list of images to download. Long segments will be split by interpolating the start and end location to create smaller edge sections (see Code 5.3). We generate a url link an shape corresponding to Google Street View API discussed in Code 5.2 alongside with unique file name.

```
http://maps.googleapis.com/maps/api/streetview?size=600x400
   &location=<lat>,<long>&heading=<angle from north>&key=<api>
```

Code 5.2. Exact Google Street View API url we need to generate

```
def getGoogleViewUrls(self, min_allowed_distance = 30)
        edge_length = 1000*distance_between_two_points(self.Start, self.End)
        number_of_fractions = int(max(floor(edge_length / min_allowed_distance),1.0))

        for (last_fraction, current_fraction) in fractions:
            # runs for example with 0.0 - 0.2, 0.2 - 0.4, ..., 0.8 - 1.0
            PointA = interpolation(self.Start, self.End, fraction=last_fraction)
            PointB = interpolation(self.Start, self.End, fraction=current_fraction)

            urls, filenames <- self.betweenPoints(PointA, PointB)

  return urls, filenames
```

Code 5.3. Code segment which breaks down long edges and generates lists of urls and filenames Segment.getGoogleViewUrls().

One by one we try to download each of these images, marking segments with error flag, if we were unable to access them. The code attempts to retry each request for default number of times, to prevent temporary instability of network to hinder the downloading process.

Figure 5.2. Location outside of streets with available photographic imagery.

```
dataset folder: 5556x_example_dataset_299px/
SegmentsData.dump
images/0000_0.jpg
images/0000_1.jpg
…
images/5546_5.jpg
```

Code 5.4. Folder structure of dataset.

```
1  def RunCheck(segments_filename):
2  # Check downloaded segments and fix them
3    Segments = LoadDataFile(segments_filename)
4    if (HasErroneousData(Segments, ERROR)):
5      Segments = FixDataFile_FailedDownloads(Segments, ERROR)
6    SaveDataFile(Segments, segments_filename)
7
8  def FixDataFile_FailedDownloads(Segments):
9    BrokenSegments = []
10   for (i, Segment) in Segments:
11     if Segment.ErrorMesages[i] == ERROR
12       BrokenSegments.append(Segment)
13
14   FilenameMapOfBroken <- GenListOfUrls(BrokenSegments)
15   DownloadUrlFilenameMap(FilenameMapOfBroken, BrokenSegments)
16   return Segments
```

Code 5.5. RunCheck code sample checks the downloaded dataset and redownloads missing images.

We also check for invalid images in areas where Google Street View doesn't have any data available (see Figure 5.2).

Eventually we save the pickled array into file for further processing. Folder structure is as follows in Figure 5.4

Function RunCheck() in 5.5 allows us to load previously downloaded Segments file and check for erroneous data – for example in case of sudden network failure we can download only the missing files and then save the fixed Segments file.

5.3 OSM Marker

Our code works with data downloaded from Open Street Maps to estimate what is the neighborhood of each segment. We have downloaded the .osm data file of corresponding location and exported it into a PostgreSQL database following commands represented in Code 5.6.

Having a PostgreSQL database is advantageous, because it allows us sending repeated queries to database of objects with geographical location while using PostgreSQL macros and functions for getting distance and intersections between areas. PostgreSQL

```
createdb gisdb
psql -d gisdb -c 'CREATE EXTENSION postgis; CREATE EXTENSION
    postgis_topology;'
osm2pgsql --create --database gisdb example_data.osm.pbf -U example_user
```

Code 5.6. Loading data into PostgreSQL database.

```
def Marker(Segments, radius = 100):
# OSM Marker
  global connection
  connection = ConnectionHandler() # handles Python <> PostgreSQL DB

  for Segment in Segments:
    MarkSegment(Segment, radius)
  return Segments

def MarkSegment(Segment, radius):
  for (i, distinct_location) in Segment.DistinctLocations:
    nearby_vector = connection.query_location(distinct_location, radius)
    Segment.mark_with_vector(nearby_vector, i)

class ConnectionHandler:
def query_location(self, location, radius):
  sql_command = self.sql_cmd_radius(location, radius)
  # builds the query
  # SELECT <interesting_columns> FROM table WHERE distance < radius

  rows, column_names = self.run_command( sql_command )

  pairs = extract_all_pairs(rows, column_names)
  # returns pairs of column names and their values
  # for example "highway=primary", ...

  nearby_vector = [0] * number_of_observed_pairs

  for pair in pairs
    if pair in observed_pairs:
      index = indice_dictionary(pair)
      nearby_vector[ index ] += 1
  return nearby_vector
```

Code 5.7. Marking data with OSM vector.

database has hierarchical structure of data storage which is necessary for fast data access. Our python code can access this database with simple queries and then process its responses.

The code 5.7 loads array of Segments and one by one accesses all the distinct locations stored in each Segment. This could be only two locations for segments too short to be broken down, or more if the original segment was split. Minimally these are the starting and the ending locations of the segment.

We generate a SQL command, which targets particular columns which we chose to observe and also filters the data with "WHERE" clause for objects in distance smaller than chosen radius from segments location.

OSM data we imported into PostgreSQL database takes structure of four tables representing "point", "line", "polyline" and "road" objects. We look into each of these tables and combine the results.

Result of query gives us list of rows, each representing one object in vicinity of our location and columns containing attributes. We can produce a "attribute=value" pair from the values in each row alongside with the count of their occurrences.

Final neighborhood vector counts all these occurrences in fixed positions as was discussed in 3.2.3.1 .

Each segment stores multiple neighborhood vectors, one for each distinct location. Marked array of Segments is saved into pickled .dump file. We can store multiple versions of these files, each with different radius setting, while reusing all the downloaded images.

5.4 Datasets and DatasetHandler

DatasetObject is a structure shielding us from low level manipulation with the segments stored in .dump file, which can then remain unchanged and be reused for many experiments.

Figure 5.3 shows us the structure of DatasetObject, with its most important functions and variables exposed.

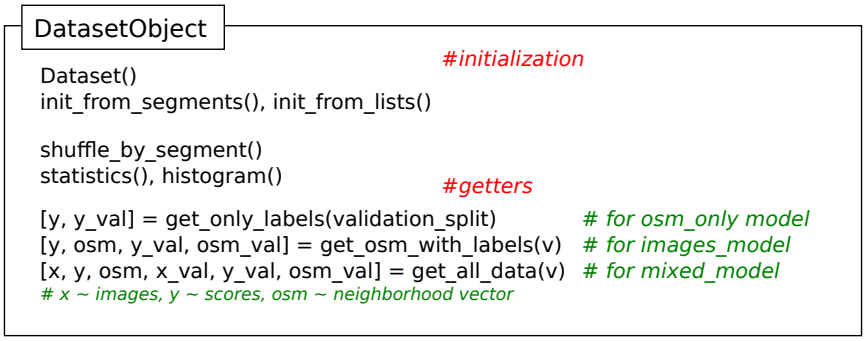


Figure 5.3. Dataset object.

Dataset is initialized by reading a SegmentsData.dump file, however it stores data in its own four internal arrays. These are: list of urls to images, labels marking the scores, OSM vectors if we have loaded a marked Segments file and unique IDs of original segments. Note that we are not directly loading all the images yet, rather we are keeping only their filenames. We can access the images when necessary, saving us from wasteful memory allocation.

We also provide multiple getter functions which split data into training and validation datasets for the purpose of testing experiments. Each model type will require access to different data – image only model will for example omit all the OSM neighborhood vectors.

```
1 Dataset.statistics()
2 # prints statistics over data scores
3 # min |-- 25th_percentile { mean } 75th_percentile --| max
4
5 Dataset.histogram()
6 # generates histogram of data scores
```

Code 5.9. Functions for data statistics visualization.

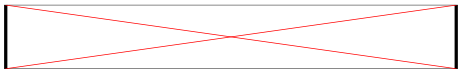


Figure 5.4. Example of dataset visualization.

```
def load_dataset(path_to_segments_file):
# part of DatasetHandler which creates the Dataset object
  dataset = Dataset()
  dataset.init_from_segments(path_to_segments_file):
    # loads 4 important lists:
    Segments = LoadDataFile(path_to_segments_file)
    list_of_images, labels, osm, segment_ids = self.load_data_from_segments(
        Segments)
    self.init_from_lists(list_of_images, labels, osm, segment_ids)

  dataset.shuffle_by_segments()
  # shuffles data while respecting that the data from one segment should be next
      to each other

  return dataset

def load_data_from_segments(Segments):
  for Segment in Segments:
    for i_th_image in Segment.number_of_images:
      index = Segment.location_index[i_th_image]

      list_of_images <- Segment.get_image_filename(i_th_image)
      labels         <- Segment.get_score()
      osm            <- Segment.distinct_nearby_vector[index]
      segment_ids    <- Segment.get_id()
  return list_of_images, labels, osm, segment_ids
```

Code 5.8. DatasetHandler.

5.5 Models and ModelHandler

ModelHandler is composed of three functional sections. ModelOI provides the functions handling input and output – including interaction with DatasetHandler and reporting functions. ModelGenerator is responsible for building models in Keras syntax depending on settings we choose for current model. ModelTester provides functions controlling the training and testing capabilities of model.

These individual blocks are controlled from ExperimentRunner, which ties them together and manages shared Settings information.

5.5.1 Model description in Keras

Generic code for building models in Keras works in two modes – Sequencial and Functional. We will use Functional in these particular examples as it is convenient both for writing and understanding.

We will mention some of the layers we can use to build Keras models.

```
Out = Dense(256, activation='relu')(In)
Out = Dense(1, activation='sigmoid')(In)
```

Dense layers stand for fully connected layers mentioned in 4.1.2. Here we can see two possible uses for this type of layer. With width set to 1, this layer represents one neuron which can be used as the final output of our classifier.

```
Out = Dropout(probability)(In)
```

Dropout layer is used accompanying the Dense layers and follows the process mentioned in 4.1.5. With selected probability the connection of neurons will be dropped during the training period.

We are using these mentioned layers in our models, for more detailed survey check ?? APPENDIX KERAS SYNTAX.

After building the model we need to compile it specifying settings which will be used for training such as the optimizer, loss function and additional measured metrics.

```
Model.compile()
```

```
from keras.models import Model
from keras.layers import Input, Dense, Dropout

def build_generic_model(input_shape):
# part of ModelHandler
  input = Input(shape=input_shape)
  x = Dense(256, activation='relu')(input)
  x = Dropout(0.5)(x)
  ... # more layers defined

  output = Dense(1, activation='sigmoid')(x)
  model = Model(inputs=input, outputs=output)
  return model
```

Code 5.10. Building generic model in Keras.

For details about which settings we use for training, check 4.6.
Finally our model is trained on dataset with fit command.

```
model.fit(data, labels, ...)
```

The full code to build and train a generic Keras model on a dataset can be seen in Figure 5.11.

```
def run_generic_experiment():
  dataset = DatasetHandler.load_dataset(path_to_segments_file)
  [images, score, images_val, score_val] = dataset.get_imgs_with_labels(...)

  input_shape = dataset.get_input_shape()
  model = ModelHandler.build_generic_model(input_shape)

  model.compile(optimizer, loss, metrics)
  # optimizer, loss, metrics,
  # epochs, batch_size are loaded from Settings

  history = model.fit(images, score, epochs, batch_size,
                      validation_data=(images_val, score_val))

  ModelOI.save_history( history )
  ModelOI.visualize_history( history )
```

Code 5.11. Fit generic model to data in Keras.

5.5.2 Model building

Particular models described in section 4 are shown here with functions which generate them. We have stated that their structure can be parametrized, and that we can alter these parameters to change their shape and performance. We can change the depth and width of these models.

5.5.2.1 OSM only model

See Code sample 5.12, which builds the model discussed in 4.2.

5.5.2.2 Images only model

When we are working with model described in 4.3, we make use of the feature transfer method and as such the training process can be made easier as was discussed in 4.6.4.

Also the Keras syntax will be correspondingly altered. We create two models instead of a single whole model. One represents the feature extractor section of model and is mostly created by the reused base of pretrained CNN model. We can load these as base of our models with weights initialized from model trained on ImageNet dataset, such as is done in 5.13.

```
from keras.models import Model
from keras.layers import Input, Dense, Dropout

def build_osm_only_model(input_shape, model_depth, model_width=256):
    osm_features_input = Input(shape=input_shape)
    top = Dense(model_width, activation='relu')(osm_features_input)
    top = Dropout(0.5)(top)
    for i in range(0,model_depth-1):
        top = Dense(model_width, activation='relu')(top)
        top = Dropout(0.5)(top)
    output = Dense(1, activation='sigmoid')(top)

    model = Model(inputs=osm_features_input, outputs=output)
    return model
```

Code 5.12. Function producing a Keras model described in 4.2.

```
# Keras Applications, deep CNN models with weights trained on ImageNet

model = ResNet50(weights='imagenet', include_top=False, input_shape=input_shape)

model = VGG16(weights='imagenet', include_top=False, input_shape=input_shape)

model = VGG19(weights='imagenet', include_top=False, input_shape=input_shape)

model = InceptionV3(weights='imagenet', include_top=False, input_shape=input_shape)

model = Xception(weights='imagenet', include_top=False, input_shape=input_shape)
```

Code 5.13. Applications

Our second model is then made of just our custom top architecture discussed in 4.3.3. We will feed it with outputs of features from the first model. Note that for repeated experiments with the same dataset and model setting, we can save these features into files and spare a lot of valuable computation time. Code snippet in 5.14 shows us an example of the use of two models.

```
def img_model_with_feature_cooking():
  dataset = DatasetHandler.load_dataset(path_to_segments_file)
  [images, score, images_val, score_val] = dataset.get_imgs_with_labels(...)

  input_shape = dataset.get_input_shape()
  model_base = ResNet50(weights='imagenet', include_top=False, input_shape=input_shape)

  training_features = model_base.predict(images)
  validation_features = model_base.predict(images_val)

  # here it's possible to split the computation
  # save the training_features, validation_features
  # and load them repeatedly after

  model_top = build_simple_top_model( input_shape=training_features.shape[1:] )
  model_top.compile(...)

  history = model_top.fit(training_features, score, ...
      validation_data=(validation_features, score_val))
```

Code 5.14. Feature cooking

5.5.2.3 Mixed model

Mixed model uses the same techniques of feature transfer and is fully described in 4.4.

Code in 5.15 shows how we build this model. Note that we can even reuse the saved feature files from image only model, as long as we maintain the same order in our dataset. DatasetHandler can shuffle data in deterministic manner which allows us to do so.

```
from keras.models import Model
from keras.layers import Input, Dense, Dropout, Flatten, concatenate
from keras.applications.resnet50 import ResNet50

def build_full_mixed_model(input_shape_img, input_shape_osm, number_of_repeats):
  model_cnn = ResNet50(input_tensor=input_shape_img, weights='imagenet', include_top=False)
  img_features = Flatten()(model_cnn.output)

  osm_features_input = Input(shape=input_shape_osm)
  osm_features = Dense(256, activation='relu')(osm_features_input)
  osm_features = Dropout(0.5)(osm_features)

  top = concatenate([osm_features, img_features])
  for i in range(0,number_of_repeats):
    top = Dense(256, activation='relu')(top)
    top = Dropout(0.5)(top)
  top = Dense(1, activation='sigmoid')(top)

  model = Model(inputs=[model_cnn.input, osm_features_input], outputs=top)
  return model
```

Code 5.15. Building mixed model.

5.5.3 ModelHandler Structure

Figure 5.5 illustrates the various functions which are part of the ModelHandler. In 5.16 we can see snippets of code dealing with building and testing of models. Note that in one run we are handling multiple instances of models and datasets, as is explained in 5.7.

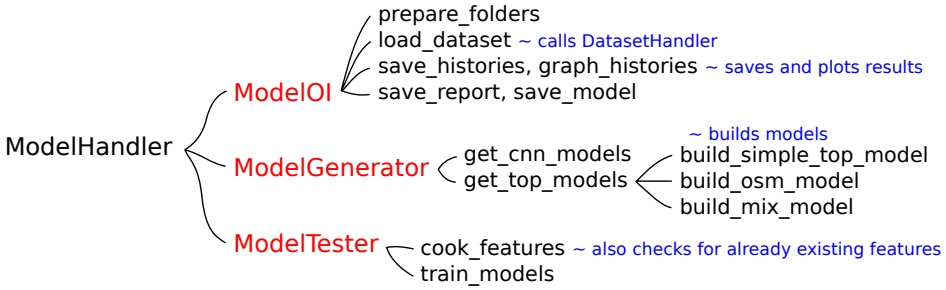


Figure 5.5. ModelHandler Structure.

```
def get_top_models():
  models = []

  for model_setting in Settings["models"]:
    model_type = model_setting["model_type"]
    model = build_img_only / build_osm_model / build_mix_model

    models.append( model )
  return models


def train_models():
  histories = []

  for model_setting in Settings["models"]:
    model <- models
    dataset <- datasets

    history = train_model(model, dataset, model_setting)
    histories.append( history )
  return histories
```

Code 5.16. ModelHandler functions.

5.6 Settings structure

A

5.7 Experiment running

A

5.8 Training

A

5.9 Testing

A

5.10 Reporting and folder structure

A

5.11 Metacentrum scripting

A

Chapter 6
Results

C

Chapter 7
Discussion

B

Chapter 8
Conclusions

V

References

[1] Jan Hrnčíř, Pavol Žilecký, Qing Song, and Michal Jakob. Practical multicriteria urban bicycle routing. *IEEE Transactions on Intelligent Transportation Systems*, 2016.

[2] Daniel Delling, Andrew V. Goldberg, Moises Goldszmidt, John Krumm, Kunal Talwar, and Renato F. Werneck. Navigation made personal: Inferring driving preferences from gps traces. November 2015.

[3] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[4] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[5] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014.

[6] Kaz Sato. How a japanese cucumber farmer is using deep learning and tensorflow. `https://cloud.google.com/blog/big-data/2016/08/how-a-japanese-cucumber-farmer-is-using-deep-learning-and-tensorflow`, 2016.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[8] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[10] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[11] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition. `http://cs231n.github.io/convolutional-networks/`, 2017.

[12] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[13] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.