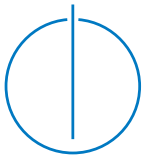
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Deep Learning for Image-Based Localization

Florian Walch



DEPARTMENT OF INFORMATICS

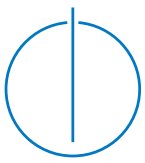
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Deep Learning for Image-Based Localization

Deep Learning für bildbasierte Lokalisierung

Author:	Florian Walch
Supervisor:	Prof. Dr. Daniel Cremers
Advisors:	Dipl.-Ing. (Univ.) Sebastian Hilsenbeck
	Caner Hazırbaş, M.Sc.
	Dr. Laura Leal-Taixé
Submission Date:	15.10.2016



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, _____ _____

Florian Walch

Abstract

"Where was this image taken?" is an intriguing question that has been posed in the literature and is of interest for many applications, from localization of smartphone images to automatic geo-tagging on an online photo platform.

Traditional content-based image retrieval (CBIR) systems typically localize images with a pipeline based on hand-crafted feature descriptors such as SIFT. Convolutional neural networks (CNNs), on the other hand, can be trained end-to-end to learn appropriate features from training data. CNNs have been very successful on many computer vision tasks such as image classification, optical flow prediction, and image superresolution. Recent neural network models such as PoseNet and PlaNet have considered localization of images as regression or classification tasks.

In this thesis, we investigate two different approaches for image-based localization. In the first approach, we improve upon PoseNet's idea of performing pose regression in a specific localization area. In particular, we introduce a neural network model based on PoseNet and LSTMs for single-image regression, and extend this approach to regression from sequences of images. In a second approach, we train a Siamese network on pairs of images that have been taken at nearby locations. The resulting network has the advantage that it is not limited to a specific localization area.

We evaluate our models on the Cambridge Landmarks datasets, an indoor dataset from Deutsches Museum, and the well-known Dubrovnik dataset. We obtain at least competitive, but often outperforming results on most datasets.

Contents

1. Introduction

The widespread use of smartphones equipped with receivers for global navigation satellite systems (GNSS) has enabled a wide range of applications in industry, logistics, sports, and many other fields. Smartphone apps allow users to easily find their way in cities and remote areas alike.

However, life does not only take place outdoors. In the case of navigation, an ideal system would not only find a way to the outside of a building, but to the actual target location of the user, which is often inside the building. For example, while users can find their way to an airport with a GNSS-equipped smartphone, they have to rely on other means of navigation once inside. This is because GNSS systems require line of sight to several satellites and therefore in general cannot operate in indoor environments [54, p. 1].

As smartphones are typically equipped with WiFi and Bluetooth connectivity, it is possible to perform indoor localization based on the signals from WiFi access points or Bluetooth beacons, provided the origins of the signals are known. Especially for large buildings, this requires the setup and maintenance of a significant number of transmitting devices.

Humans mostly rely on visual clues to pinpoint their location. They obtain these clues from e. g. distinctive building façades, logos, or vegetation, but also from infrastructure built explicitly to facilitate human orientation and navigation, such as street- or door signs.

Modern smartphones are also equipped with high-quality cameras. *Image-based localization*, which describes the task of predicting where an image has been taken, can therefore be applied for indoor localization. In contrast to the previous approaches, this does not require the setup and continuous maintenance of infrastructure, but only the creation of a database of georeferenced images.

Devices for the creation of such databases have been described in the literature [24], but a wide range of commercial solutions are also available. This makes it feasible to create such a database even for buildings of tens of thousands of square meters.

Image-based localization also has its use in outdoor environments, as GNSS-based localization is often inaccurate in dense urban environments [54, p. 1].

Using a system that extracts clues, or *features*, from a given image, images containing similar features can be looked up from the georeferenced database. From these, possible locations of the input image can be deduced.

It is important to note that not all features depicted in an image are useful for localization. For example, company logos on a wall typically do not change often, and are therefore distinctive of a certain location. On the other hand, content such as advertisement posters might be removed in the future and thus cannot be reliably used for localization. A feature

extraction system should thus be able to work even if only a subset of features is still present.

This is the basic idea of *content-based image retrieval* (CBIR). To find features, we can use hand-crafted feature extractors such as SIFT [38], or use machine learning to train an appropriate model that extracts features from raw image pixels.

Instead of using a lookup database of reference images, position and/or orientation of an input image can be predicted as continuous values, which constitutes *pose regression*. Alternatively, a *classification* approach can localize an image by finding the most probable out of a number of discrete locations.

Artificial neural networks refer to a class of machine learning tools that have recently become increasingly popular and are considered state-of-the-art for a multitude of tasks, ranging from speech processing to image classification.

Convolutional neural networks (CNNs) [10, 34] are used to classify the content of images into one of multiple categories [33, 57, 61, 17, 60], finding the position of items depicted in an image [17], and many other tasks related to computer vision.

Convolutional neural networks owe their success to their property of end-to-end learning of *hierarchical features* from data, where features of a certain level build upon features of a lower level [29]. For example, simple features like the existence of lines of certain orientations in an image are combined into intermediate features describing simple shapes, which in turn are used to build complex features that detect faces. This is in contrast to hand-crafted feature extractors such as SIFT [38], which compute features directly from an input image.

Deep learning refers to neural networks with a large number of such levels, or *layers*. The term is not well-defined [52], but recent neural network architectures such as [33, 61, 17] can all be considered "deep" [17].

Recurrent neural networks (RNNs) [48] are typically used to work with sequential data such as speech or text. However, they can also be used on sequences of images [67] or on single images by processing them as sequences of pixels [65].

The main contributions of this work are as follows:

- We improve the existing PoseNet [31] model for location regression from a single image by processing extracted features with recurrent neural networks.
- We extend location regression from single images to image sequences.
- We perform Siamese training to obtain neural networks to extract similarity-based features. We use these features in localization based on content-based image retrieval on an indoor dataset.

The remainder of this work is organized as follows. In chapter 2, some background on artificial neural networks is presented. Chapter 3 gives a more detailed motivation for our work and introduces the primary use case we consider for image-based localization. In chapter 4, the concepts of the previous chapters are combined to discuss how to apply neural networks on the task of image-based localization, and introduces several neural

network models. Experiments on these models are discussed in chapter 5, with chapter 6 following up with a discussion of the results. Finally, chapter 7 presents the conclusion and gives an outlook on possible future work. In appendix A, we give some notes on our implementation.

2. Artificial Neural Networks

This chapter presents the theoretical background on artificial neural networks required for the proposed method. Section 2.1 introduces the concepts of individual neurons and arranging them into basic layers to form feedforward neural networks. Other types of layers used to construct networks for processing spatial data, i. e. convolutional neural networks, and sequential data, i. e. recurrent neural networks, are presented in section 2.2 and section 2.3. After section 2.4 discusses supervised training, some important regularization techniques to improve the performance of neural networks are discussed in section 2.5.

Building on all of these concepts, two neural network models originally designed for image classification are presented in section 2.6. Variants of these models adapted for image-based localization are introduced in chapter 4.

2.1. Feedforward Neural Networks

Artificial neurons are loosely inspired by biological neurons, i. e. brain cells, in that they receive input signals on multiple connections and only produce an output signal if a weighted sum of the inputs reaches a certain threshold [29].

Mathematically, a single artificial neuron with K input values represents a nonlinear function $g : \mathbb{R}^K \mapsto \mathbb{R}$, parametrized by a *weight vector* $\boldsymbol{w}$, a *bias* b, and a non-linear *activation function* σ [29]:

$$g(\boldsymbol{x}) = \sigma\left(\sum_{k=0}^{K} w_k x_k + b\right) = \sigma(\boldsymbol{w}^T \boldsymbol{x} + b) \tag{2.1}$$

Figure 2.1 shows an illustration of an artificial neuron.

A set of neurons with a common activation function σ can be arranged into a *layer*. Each neuron of a layer l feeds its outputs only into neurons of layer $l + 1$, constituting the *feedforward* property. A network of multiple consecutive layers connected in this way is called a *feedforward neural network*. The number of neurons in each layer determines the *width* of each layer, while the number of layers determines the *depth* of the network [12, pp. 168–169].

Note that due to the feedforward property, the outputs of all neurons of a layer l can be computed in parallel [12, p. 169]. A layer l with $K^{(l)}$ neurons operating on an input vector $\boldsymbol{x}^{(l-1)}$ thus represents a non-linear function $f^{(l)} : \mathbb{R}^{K^{(l-1)}} \mapsto \mathbb{R}^{K^{(l)}}$, producing an output vector $\boldsymbol{x}^{(l)}$:

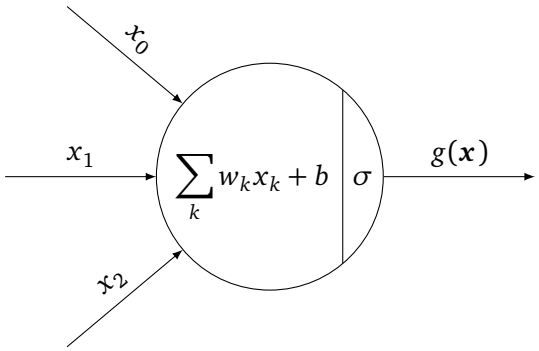


Figure 2.1.: **Artificial neuron.** As described in eq. (2.1), the neuron computes the weighted sum of an input vector $\boldsymbol{x} = (x_0, x_1, x_2)^T$, adds a bias value, applies an activation function σ, and outputs the resulting scalar value. Image based on [29].

$$\boldsymbol{x}^{(l)} = f^{(l)}(\boldsymbol{x}^{(l-1)}) \tag{2.2}$$

The layer function is defined as:

$$f^{(l)}(\boldsymbol{x}) = \sigma^{(l)}(\boldsymbol{W}^{(l)}\boldsymbol{x} + \boldsymbol{b}^{(l)}) \tag{2.3}$$

The *weight matrix* $\boldsymbol{W}^{(l)}$ and the *bias vector* $\boldsymbol{b}^{(l)}$ of a layer are constructed from the weight vectors $\boldsymbol{w}$ and bias values b of the individual neurons comprising the layer.

For a network of L consecutive layers, the overall *network parameters* θ are given by the individual layer weights $\boldsymbol{W}^{(l)}$ and biases $\boldsymbol{b}^{(l)}$. Explicitly specifying the parameters $\boldsymbol{\theta}$, the network thus represents a function $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and can be written as a composition of its layers as $f : \mathbb{R}^{K^{(0)}} \mapsto \mathbb{R}^{K^{(L)}}$ [12, p. 168]:

$$f(\boldsymbol{x}; \boldsymbol{\theta}) = (f^{(L)} * f^{(L-1)} * \ldots * f^{(1)})(\boldsymbol{x}) \tag{2.4}$$

The output of the last layer of a network, $\boldsymbol{x}^{(L)} = f^{(L)}(\boldsymbol{x}^{(L-1)})$, is equivalent to the output of the whole network $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$. This layer is thus called the *output layer* [12, p. 169]. Because these output values are used e. g. for regression or as class scores for classification, usually no activation function σ is applied in the output layer [29]. The network input $\boldsymbol{x}$, equivalent to the first layer's input $\boldsymbol{x}^{(0)}$, can be represented as a separate *input layer* [29]. The intermediate neural layers are called *hidden layers* [12, p. 169].

Figure 2.2 shows an example of a feedforward neural network with two hidden layers, represented as a directed acyclic graph.

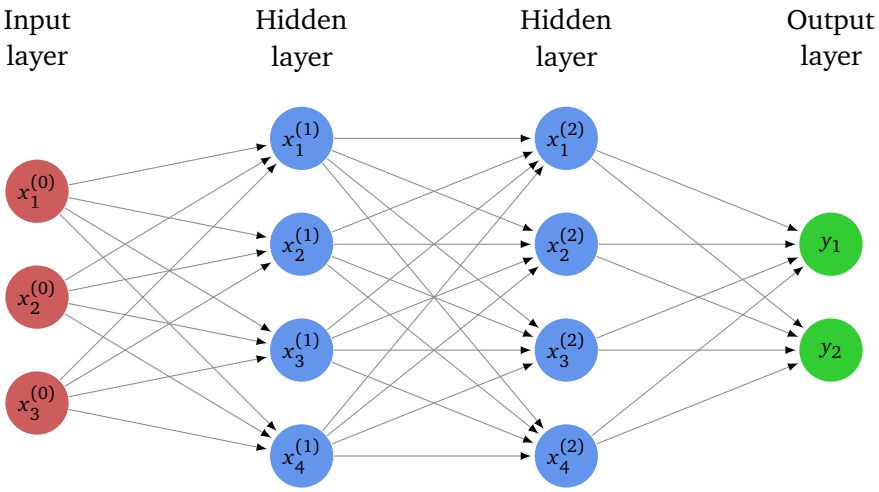
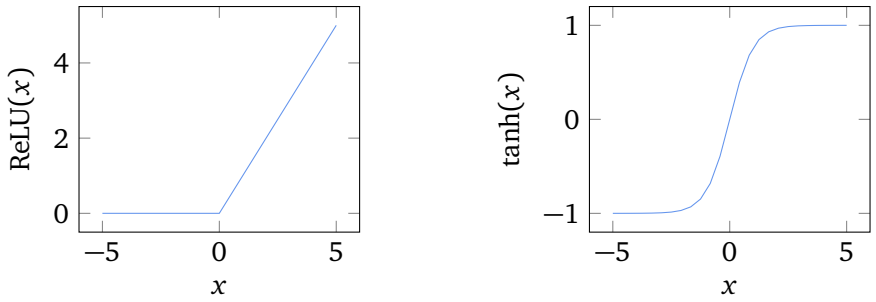


Figure 2.2.: **Feedforward neural network.** This example network consists of an input layer $\boldsymbol{x}^{(0)}$, two hidden layers of width 4, and an output layer $\boldsymbol{y}$ of width 2. Bias values are not shown. Note how neurons of each layer $l-1$ are only connected to neurons of the subsequent layer l, resulting in the *feedforward* property. As described in eq. (2.4), the network function $f : \mathbb{R}^3 \to \mathbb{R}^2$ can be written as a composition of its layer function as $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. Image based on [29].



(a) ReLU activation function.

(b) tanh activation function.

Figure 2.3.: **Examples of activation functions used in artificial neural networks.** The rectified linear unit (ReLU) is typically used by convolutional neural networks [12, p. 174], while tanh is typically used in recurrent neural networks [14, 49].

2.2. Convolutional Neural Networks

The feedforward networks described so far operate on vectors $\boldsymbol{x} \in \mathbb{R}^K$. In *convolutional neural networks*, the input is instead a three-dimensional volume $\boldsymbol{x} \in \mathbb{R}^x \times \mathbb{R}^y \times \mathbb{R}^d$, i. e. of certain width × height × depth. Convolutional neural networks are thus especially suited for processing images [29]. For example, an RGB image of 28 × 28 pixels can be represented as a volume of size 28 × 28 × 3.

In the following sections, different types of layers used in convolutional neural networks are described. An overview is shown in table 2.1.

Table 2.1.: **Overview of basic CNN layers.** Based on [29]. Input size: $x \times y \times d$.

	Convolutional	Pooling	Fully connected
Hyperparameters	filter size x_f, y_f stride s_x, s_y padding p_x, p_y number of filters n	filter size x_f, y_f stride s_x, s_y	number of filters n
# of train. params.	$(x_f \cdot y_f \cdot d + 1) \cdot n$	none	$(x \cdot y \cdot d + 1) \cdot n$
Output size	$x \to \frac{x - x_f + 2p_x}{s_x} + 1$ $y \to \frac{y - y_f + 2p_y}{s_y} + 1$ $d \to n$	$x \to \frac{x - x_f}{s_x} + 1$ $y \to \frac{y - y_f}{s_y} + 1$ $d \to d$	$x \to 1$ $y \to 1$ $d \to n$

As noted in section 2.1, an activation function is used to introduce nonlinearity in neural

networks. The *rectified linear unit* (ReLU, [43]) is currently the default choice for the activation function σ in convolutional neural networks [12, p. 174], although alternatives have been proposed [11, 18]. ReLU is defined as:

$$\mathrm{ReLU}(x) = \max(0, x) \tag{2.5}$$

The ReLU function is visualized in fig. 2.3a.

2.2.1. Convolutional Layers

A *convolutional layer* operates on an input volume of size $x \times y \times d$. It applies a filter of size $x_f \times y_f \times d$, where $x_f \leq x, y_f \leq y$, on each spatial position (x', y') to yield an output volume $x_o \times y_o \times 1$. Two hyperparameters s_x and s_y, called *stride*, determine the spacing in width and height between the positions (x', y'). If $s_x = s_y = 1$, the filter is applied on each spatial position of the input volume [29]. An illustration is shown in fig. 2.4.

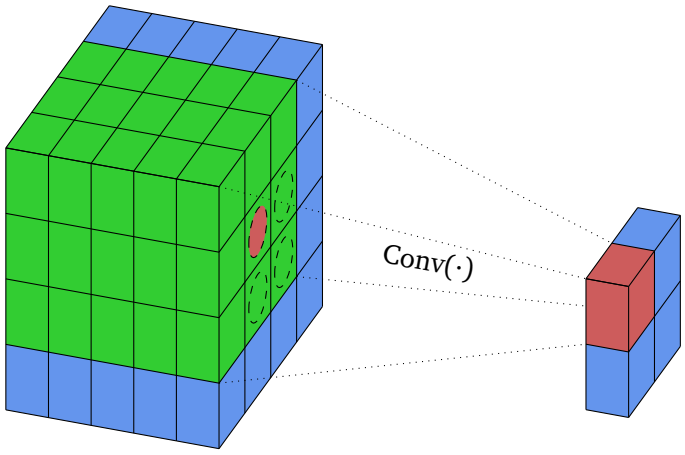


Figure 2.4.: **Example of a convolutional layer.** The layer consists of a single convolutional filter (i. e. $n = 1$) of size $3 \times 3 \times 5$, applied with stride 1 and no padding on a $4 \times 4 \times 5$ input volume. The convolutional filter (in green) operates across the whole depth of the input volume at all four possible spatial positions (x', y'), denoted by circles, resulting in an output volume of size $2 \times 2 \times 1$.

As shown in the illustration, due to the size of the filter, the output volume's spatial size can be smaller than the in input volume, i. e. $x_o \leq x, y_o \leq y$. In particular, $x_o < x$ if $x_f > 1$. *Padding* of size p_x, p_y can be used on the input volume to increase its spatial size x, y before applying the filter, thus again increasing the output volume's size x_o, y_o. Typically, padding is performed with values of zero [29].

In summary, the spatial size of the output is computed as [29]:

$$x_o = \frac{x - x_f + 2p_x}{s_x} + 1 \tag{2.6}$$

$$y_o = \frac{y - y_f + 2p_y}{s_y} + 1 \tag{2.7}$$

Typically, a convolutional layer consists of not only one, but multiple filters. Applying n of these filters and stacking their outputs along the depth dimension results in outputs of shape $x_o \times y_o \times n$. An example of a convolutional layer with multiple three filters is shown in fig. 2.5. This allows to learn d different features from an input volume by applying a convolutional layer. For example, if the input to a convolutional layer is an RGB image, an entry at spatial position (x, y) at depth d in the output volume indicates if a feature like a corner or a blob of a certain color has been detected in the original image at position (x, y) [29]. In chapter 5, we will see what kind of features are detected by convolutional layers in neural networks for image-based localization.

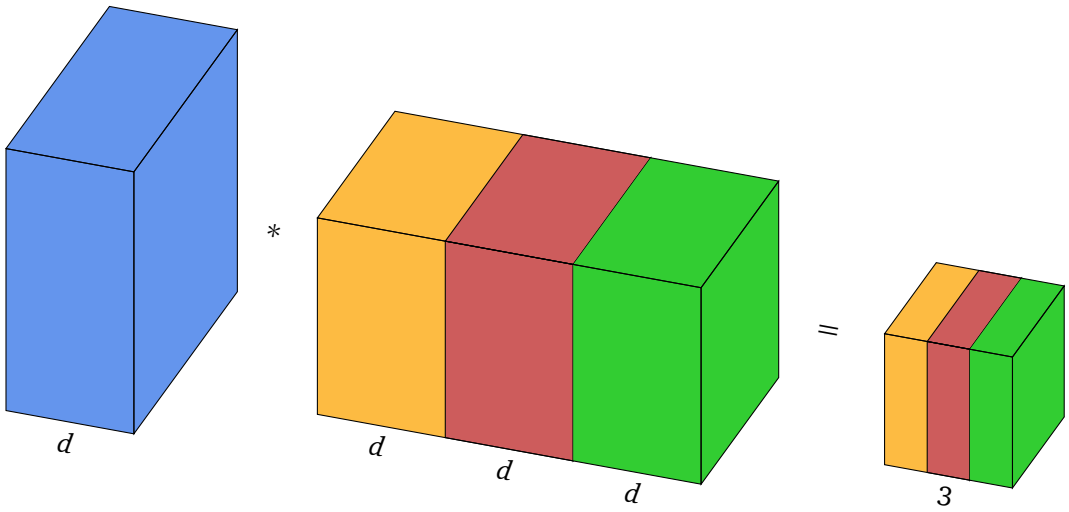


Figure 2.5.: **Example of a convolutional layer with three filters.** As each filter is applied on the whole depth d of the input volume, the output volume has a depth of 3.

Denoting the layer weights and biases as $\boldsymbol{W}$ resp. $\boldsymbol{b}$, the output of a convolutional layer is:

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b}) = \text{ReLU}(\text{Conv}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b})) \tag{2.8}$$

2.2.2. Pooling

Pooling layers are used to reduce the dimensions of an input volume by applying a reduction operation on a small spatial neighborhood. This is done independently for all depth slices

of the input volume. Typical examples are *max pooling* and *average pooling*, both of which operate on a rectangular neighborhood [12, pp. 339–342]. Figure 2.6 illustrates a pooling layer on a 2×2 neighborhood.

Applying pooling introduces an invariance to small translations into the network, i. e. the output of the pooling layer does not change if the input values are spatially shifted by a small amount. It also reduces the computational complexity of the network, as subsequent layers operate on smaller input volumes [12, p. 342].

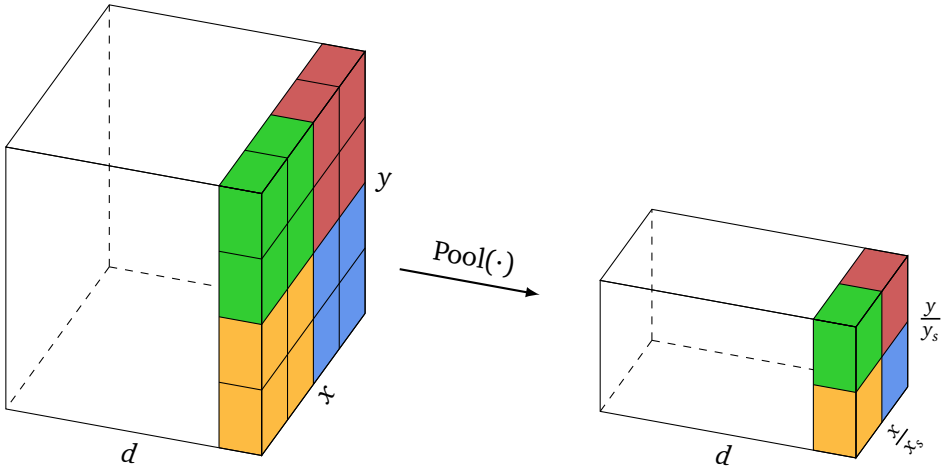


Figure 2.6.: **Pooling layer with stride 2 on a 2×2 window.** Each of the d depth slices of the input volume is spatially reduced by applying a pooling operation on the elements inside the window. The pooling operation can for example compute a composite value such as the average from the window's values, or select a value, e. g. by applying the maximum. As pooling is applied separately on each depth slice, the output volume has the same depth as the input volume.

Contrary to a convolutional layer, a pooling layer is only defined by its hyperparameters and does not contain any network parameters $\boldsymbol{\theta}$. The output of a pooling layer is:

$$f(\boldsymbol{x}) = \text{Pool}(\boldsymbol{x}) \tag{2.9}$$

While pooling layers are used in many neural network architectures [61, 17], it has been proposed to not use pooling layers in convolutional neural networks and instead use convolutional layers with larger stride [58].

2.2.3. Fully Connected Layers

Fully connected layers are an extension of the concepts of section 2.1 to multiple dimensions. In a fully connected layer, each neuron is connected to all $x \times y \times d$ entries of an input volume,

resulting in an output of [29]:

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b}) = \text{ReLU}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{2.10}$$

2.3. Recurrent Neural Networks

The networks presented so far produce a deterministic output for each input they receive, independent from previous inputs or outputs. *Recurrent neural networks* (RNNs, [48]) are a form of neural networks adapted to work with sequential data such as text or videos, and keep an internal state that is updated for each input. This allows RNNs to retain context when processing a sequence of data [12, chapter 10].

Explicit parameters depending on the position of an input in a sequence would mean that only sequences of specific lengths could be supported. Most RNNs support variable-length sequences by sharing the parameters used for processing each input in a sequence [12, chapter 10].

Following [12, chapter 10], sequences of length τ are denoted as $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}$. The updates to the recurrent neural network's internal state can then be described by [12, eq. (10.5)]:

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}) \tag{2.11}$$

$\boldsymbol{h}^{(t)}$ is the *hidden state* of the recurrent neural network after processing input $\boldsymbol{x}^{(t)}$. f is a transition function. The shared parameters $\boldsymbol{\theta}$ are used by the network to process each input at a position t in a sequence.

The formulation in eq. (2.11) defines not only feedforward connections, e. g. between an input $\boldsymbol{x}^{(t)}$ and a hidden state $\boldsymbol{h}^{(t)}$, but also connections between different time steps $t-1$ and t. These are called *recurrent connections* [12, p. 407].

With this formulation, the network continuously updates its hidden state while processing a sequence, but does not produce any output. Output could for example be produced once per sequence or for each input [12, p. 379].

For recurrent neural networks, the tanh activation function is typically used [14, 49]:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.12}$$

The tanh function is visualized in fig. 2.3b.

Instead of only processing the sequence from start to end, it can be beneficial to additionally go into the opposite direction as well. This is called a *bidirectional RNN* [12, sec. 10.3].

2.3.1. Long Short-Term Memory

Long short-term memory (LSTM, [21]) is a type of recurrent neural network designed to be able to learn to accumulate and forget relevant context in its hidden state [12, p. 409].

Compared to the basic RNN formulation, an LSTM introduces *gates* with learnable parameters that decide which values are passed through. Each value h_i of the hidden state $\boldsymbol{h}$ of an LSTM is updated as [12, pp. 410–411]:

$$h_i^{(t)} = f_i^{(t)} h_i^{(t-1)} + g_i^{(t)} l_i(\boldsymbol{x}^{(t)}, \boldsymbol{y}^{(t-1)}) \tag{2.13}$$

The *forget gate* $\boldsymbol{f}^{(t)}$ is multiplied on the previous hidden state $\boldsymbol{h}^{(t-1)}$ to decide which values should be removed. Similarly, the *input gate* $\boldsymbol{g}^{(t)}$ is multiplied on some function l of the current input $\boldsymbol{x}^{(t)}$ and the output of the previous step $\boldsymbol{y}^{(t-1)}$ to decide which values should be used to update the internal state [12, pp. 410-411].

Finally, an *output gate* $\boldsymbol{q}^{(t)}$ is applied to the internal state $\boldsymbol{h}^{(t)}$ to decide which values should be used for producing the output $\boldsymbol{y}^{(t)}$ of the LSTM for each input [12, p. 411]:

$$\boldsymbol{y}_i^{(t)} = \tanh(h_i^{(t)}) q_i^{(t)} \tag{2.14}$$

For more details on how $\boldsymbol{f}^{(t)}$, $\boldsymbol{g}^{(t)}$, $\boldsymbol{q}^{(t)}$, and l are defined, see [12, sec. 10.10.1]. Together, the weights and bias parameters of the gates comprise the LSTM network parameters $\boldsymbol{\theta}$.

2.4. Supervised Training of Neural Networks

So far, the structures of various neural network variants have been described. All of them calculate an output $\boldsymbol{y}$ from an input $\boldsymbol{x}$ given network parameters $\boldsymbol{\theta}$. In this section, we will answer the question of how to obtain $\boldsymbol{\theta}$ from a dataset of labeled examples by *supervised learning*.

In supervised learning for neural networks, the goal is to learn a function $\boldsymbol{y}^* = f^*(\boldsymbol{x})$ by approximating it with a neural network $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ [12, p. 168]. Crucially, the true function f^* is unknown; in the setting of supervised learning, only some example data $\boldsymbol{x}$ and corresponding desired outputs $\boldsymbol{y}^*$, called *labels*, are known [12, p. 105].

The labeled data is divided into a training and a testing set. During training, the network predicts $\boldsymbol{y}$ for inputs $\boldsymbol{x}$ from the training set. The predictions $\boldsymbol{y}$ are compared to their corresponding labels $\boldsymbol{y}^*$ to calculate a training error, or *loss* [29] $J(\boldsymbol{\theta})$, of the network under its current parameters $\boldsymbol{\theta}$. By adjusting the network parameters to minimize the loss, we indirectly try to approximate f^* with the network function f [12, p. 275].

To verify if this goal has been achieved, the network is evaluated on the testing set after training has finished to calculate a *testing error* [29]. It is crucial that training and testing sets are disjoint to verify that the network indeed *generalizes* to previously unseen instances [12, p. 110].

If the loss is low during training, but high during evaluation, this is an indication that the model *overfits* [12, p. 111]. An example is when the model just remembers all the training data and its corresponding labels, which would lead to zero loss during training, but not produce any meaningful predictions on previously unseen testing data.

To optimally update the network's parameters during training, it would be necessary to calculate the training error from the predictions of all training examples $\boldsymbol{x}$. Usually, this is not possible due to the large amount of examples compared to available computing resources (e. g. GPU memory). Instead, a random subset of all data, a *minibatch*, is used to compute the updates to the network parameters $\boldsymbol{\theta}$ [12, p. 278].

Trying to minimize the loss, the network parameters are updated by using *backpropagation* ([48]. For recurrent neural networks, a similar technique called *backpropagation through time* (BPTT) can be employed [12, sec. 10.2.2].

2.5. Regularization for Neural Networks

Regularization is used to improve the generalization properties of neural networks, i. e. avoiding overfitting and helping them perform well on previously unseen data [12, p. 228]. In this section, a few basic regularization strategies are presented. In later sections, these strategies, together with the basic building blocks of the previous sections, are used to build neural network models.

2.5.1. Weight Decay

Weight decay or L^2 regularization is a basic regularization technique also employed for non-neural-network machine learning models. It limits the capacity of a model by adding a penalty on the values of the network parameters $\boldsymbol{\theta}$ [12, p. 230], thereby reducing its potential to overfit [12, p. 112]. Adding a penalty term results in the loss function [12, eq. (7.1)]:

$$J_{\text{decay}}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|_2^2 \tag{2.15}$$

$\lambda > 0$ is a hyperparameter that controls the regularization strength.

2.5.2. Dropout

Dropout [59] is a technique to combat overfitting during training by randomly disabling a neuron and its connections. This can be done by setting the respective network weights to zero. Intuitively, disabling neurons prevents layers from relying on specific inputs too much, thus requiring them to better generalize by utilizing more of its inputs. Each neuron is independently disabled with a probability $p \in (0, 1]$ [12, sec. 7.12]. Figure 2.7 shows a visualization of dropout applied on a small neural network.

Dropout is a computationally cheap way to perform regularization that performs better than other simple regularization techniques such as weight decay [12, p. 265].

Dropout can also be applied during evaluation [30], which is described in more detail in section 4.1.2. In Recurrent neural networks, dropout is typically only applied on the

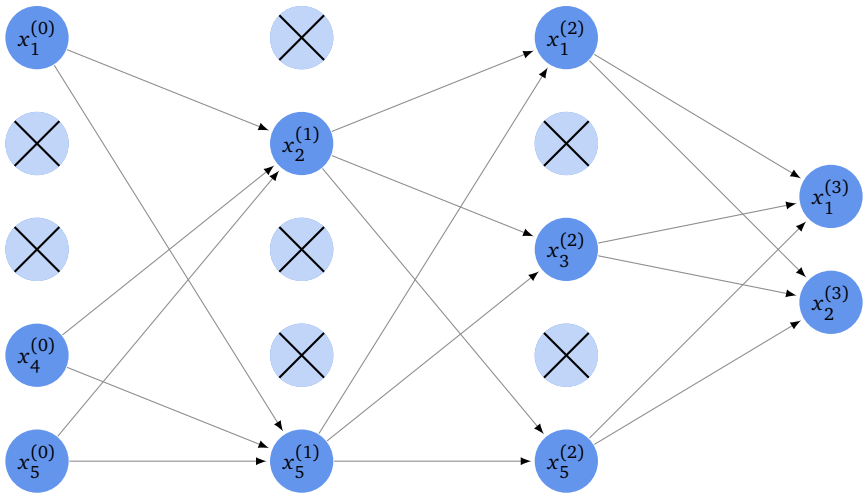


Figure 2.7.: **Visualization of dropout.** In this example, dropout is applied on four hidden layers of an artificial neural network. Applying dropout randomly sets some of the neuron's weights to zero, effectively causing the respective connections to and from these neurons to be disabled. Image based on [59, fig. 1].

feedforward part (i. e. on the connections to the network input or output), not on the recurrent connections between the hidden state for each time step [45, 71].

2.5.3. Local Response Normalization

Local response normalization (LRN) has been introduced in [33, sec. 3.3], where it is applied directly after ReLU nonlinearities and has been found to improve generalization of the network. For an input $\boldsymbol{h}$ of some spatial size and depth d, each value $h^i_{x,y}$ at spatial position (x, y) and depth $i \in \{0, \dots, d-1\}$ is normalized by $\eta^i_{x,y}$, defined as [33, sec. 3.3]:

$$\eta^i_{x,y} = \left(k + \alpha \sum_{j=\max(0, i-\frac{n}{2})}^{\min(d-1, i+\frac{n}{2})} (h^j_{x,y})^2 \right)^{\beta} \tag{2.16}$$

k, α, and β are scalar hyperparameters. $n \in \mathbb{N}$ is a hyperparameter determining the number of depth-wise adjacent values that are used for normalization at each spatial position (x, y).

2.5.4. Batch Normalization

When training a neural network, parameter updates for each layer are computed under the assumption that the parameters of all other layers are constant. However, in practice, all layers are updated at the same time, which can lead to unexpected results [12, p. 317].

Batch normalization [25] is a regularization technique that can be applied to the outputs of any layer in a neural network and helps to reduce this problem [12, p. 318].

For a batch of m outputs of a layer, each value $\boldsymbol{h}_i$ is normalized to $\boldsymbol{y}'_i$ as [12, eq. (8.35)]:

$$\boldsymbol{y}'_i = \frac{\boldsymbol{h}_i - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \tag{2.17}$$

During training, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are vectors of the mean and standard deviation computed element-wise for each value $\boldsymbol{h}_i$ across the batch as [12, eq. (8.36), eq. (8.37)]:

$$\boldsymbol{\mu} = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{h}_i \tag{2.18}$$

$$\boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{m}\sum_{i=1}^{m} (\boldsymbol{h}_i - \boldsymbol{\mu})^2} \tag{2.19}$$

$\delta > 0$ is a small scalar to avoid $\sqrt{0}$.

During evaluation, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are replaced by their average values during training [12, p. 319].

This normalization prevents a layer from updating its parameters merely to increase mean or standard deviation of the outputs. However, it can also reduce the network's expressive power [12, pp. 319]. Therefore, the final batch-normalized value $\boldsymbol{y}_i = \mathrm{BN}(\boldsymbol{h}_i; \gamma, \beta)$ of each entry in a batch is computed as [12, p. 320]:

$$\boldsymbol{y}_i = \gamma \boldsymbol{y}'_i + \beta \tag{2.20}$$

γ and β are learned during training [12, p. 320] and *scale* resp. *shift* the normalized values to restore the original expressive power of the network [25, p. 3]. Instead of producing a specific mean value based on the layers before $\boldsymbol{h}$, the mean only depends on β, making it easier for the network to learn [12, p. 320].

2.6. Neural Network Models

In this section, we discuss the well-known Inception and ResNet architectures and the GoogLeNet and ResNet-50 models based on these architectures.

Both models were originally introduced for classification problems such as ImageNet [7]. However, it has been shown that models that do well on one task, e. g. image classification, can be successfully applied to other tasks such as regression by using *transfer learning*, where model parameters are initialized with the results of training on another dataset [8, 69, 31]. We utilize transfer learning in later sections, where variations of GoogLeNet and ResNet-50 are used for image-based localization.

2.6.1. GoogLeNet

The GoogLeNet model has been proposed in [61]. It is one incarnation of the so-called Inception architecture, named after the Inception building blocks introduced in the same paper. The Inception blocks and the overall Inception network architecture have been improved in subsequent publications [62, 60].

The original Inception blocks are a combination of multiple convolution and pooling operations executed on the same input volumes. The output of all of these operations is concatenated depthwise to form a single output volume. Figure 2.8 shows an illustration of an Inception block.

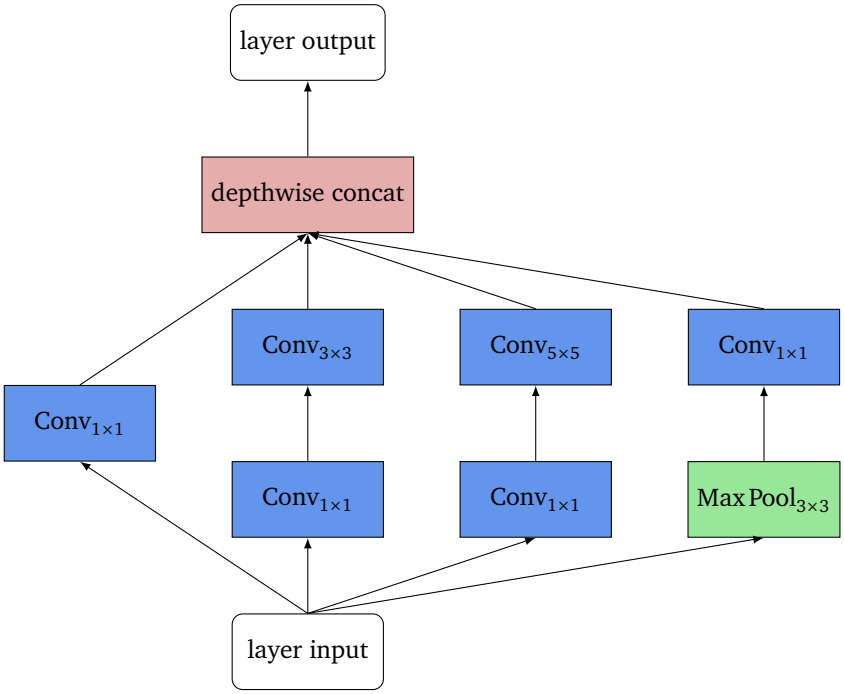


Figure 2.8.: **Inception block.** The input is independently processed by four different pipelines, all of which create an output of the same spatial dimensions. The output of the Inception layer is created from depthwise concatenation of the pipeline outputs. Image based on [61, fig. 2b].

Including the output layer, GoogLeNet consists of 22 layers with trainable network parameters. Each branch of the network outputs classification scores, which are abstracted away as "output" blocks in the GoogLeNet visualization in fig. 2.9. Dropout is applied for each branch immediately before these output blocks.

GoogLeNet is unusual in that it has three distinct output branches that all contribute to the network's loss during training. In addition to the final output at the end of the network, two

auxiliary outputs branch off the main path at intermediate stages. During evaluation, these auxiliary branches are discarded.

Including an additional regularization term on the network parameters $\boldsymbol{\theta}$, the overall loss of GoogLeNet is:

$$J_{\text{GoogLeNet}}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \gamma J^{(1)}_{\text{aux}}(\boldsymbol{\theta}) + \gamma J^{(2)}_{\text{aux}}(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\| \tag{2.21}$$

$\gamma > 0$ is a hyperparameter to weigh the losses of the auxiliary outputs against the final output.

2.6.2. ResNet-50

Residual neural networks, or ResNets for short, have been introduced in [17] and improved in subsequent publications [19, 70, 60]. Their basic building blocks are sequences of convolutions bypassed by *skip connections*, causing the model to learn *residual values* in the convolutional layers. A basic ResNet block with input $\boldsymbol{x}$ and output $\boldsymbol{y}$ can be expressed as:

$$\boldsymbol{y} = \text{ReLU}(f(\boldsymbol{x}) + \boldsymbol{x}) \tag{2.22}$$

$$f(\boldsymbol{x}) = \text{Conv}_{3\times3}(\text{ReLU}(\text{Conv}_{3\times3}(\boldsymbol{x}))) \tag{2.23}$$

Figure 2.10 shows an illustration of a basic ResNet block. The basic block is further improved into a so-called "bottleneck" block with three convolutions:

$$\boldsymbol{y} = \text{ReLU}(f(\boldsymbol{x}) + \boldsymbol{x}) \tag{2.24}$$

$$f(\boldsymbol{x}) = \text{Conv}_{1\times1}(\text{ReLU}(\text{Conv}_{3\times3}(\text{ReLU}(\text{Conv}_{1\times1}(\boldsymbol{x}))))) \tag{2.25}$$

In a bottlenet block, the 1×1 convolutions are used for reducing and restoring the depth dimension, resulting in a smaller input depth for the 3×3 convolution, therefore reducing complexity. Due to this design, inputs to the bottleneck block can have larger depths than inputs to a basic block of similar time complexity [17, p. 6].

The ResNet-50 model, also introduced in [17], consists of 16 bottleneck blocks. Including a convolutional layer after the input layer and a fully connected output layer, the overall model contains 50 layers with trainable parameters.

ResNet-50 also makes heavy use of batch normalization, but does not use dropout. Follow-up work suggests that while applying dropout to the skip connections is harmful [19], applying dropout in between convolutional layers could be beneficial [70].

Contrary to GoogLeNet, ResNet-50 has only one output, leading to the weight-decayed loss function:

$$J_{\text{ResNet-50}}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\| \tag{2.26}$$

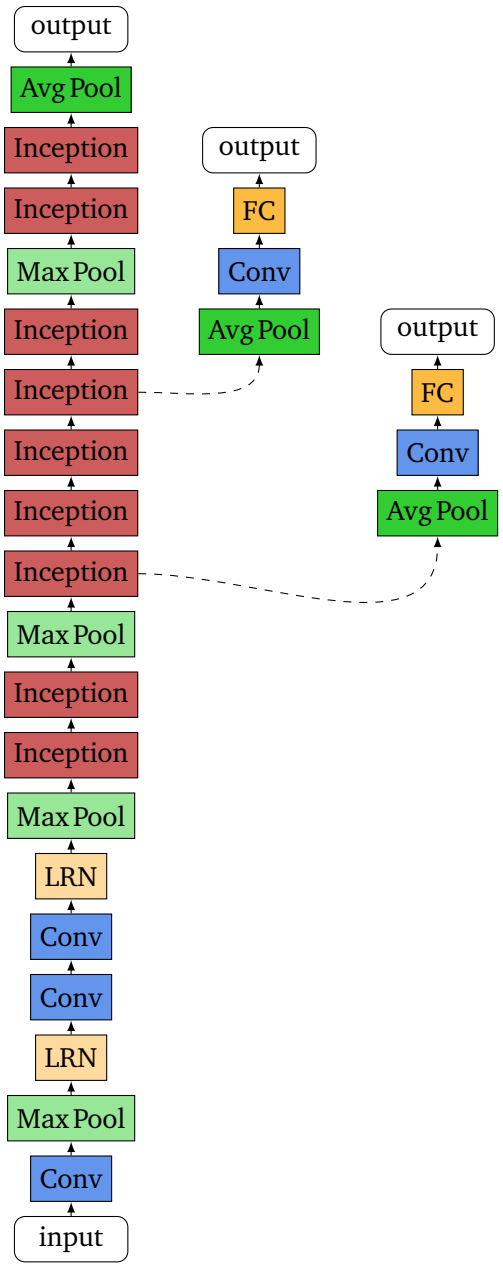


Figure 2.9.: **GoogLeNet model**. This illustration shows all layers of the GoogLeNet model. Inception blocks are shown in more detail in fig. 2.8. The output blocks of the network are only depicted as placeholders, as they are used for classification in the original model, but will be adapted for image-based localization in chapter 4. The auxiliary outputs, denoted by dashed lines, are only used during training and discarded during evaluation. Dropout is applied before each output block. Image based on [61, fig. 3].

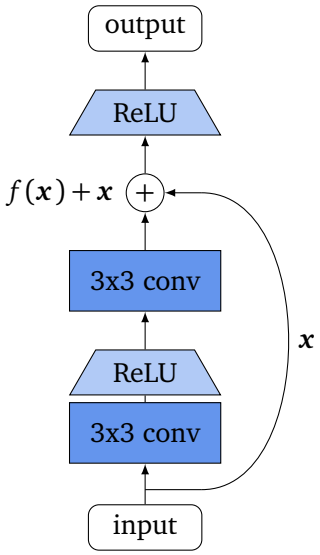


Figure 2.10.: **Basic ResNet block.** The skip connection bypasses the two convolutional layers and causes them to learn a residual $f(\boldsymbol{x})$ added to $\boldsymbol{x}$. Rectified linear units (ReLUs) are depicted separately, as they are not applied after every convolutional layer. Image based on [17, fig. 5].

3. Image-Based Localization

This chapter presents some background on the goal of the proposed method. In section 3.1, we describe the primary use case we consider for an image-based localization system. Based on the use case, section 3.2 introduces the localization task and relates it to similar problems posed in the literature. Finally, section 3.3 discusses three machine-learning-based approaches to solve the localization task and examines existing methods in this context. Neural networks for localization based on these approaches are introduced in chapter 4.

3.1. Primary Use Case

In this work, we consider a localization system that performs online localization from smartphone images. Localization should take place within a well-defined search area, for example a university campus. We consider both outdoor and indoor environments and aim for street- resp. room-level localization accuracy. While applications for e. g. augmented reality or robotic object manipulation often require sub-meter accuracy, we already consider a location prediction within a radius of multiple meters of the true position suitable for our use case, as humans are able to correct for errors within this margin [20]. The use case is illustrated in fig. 3.1.

We do not perform continuous visual localization, but only consider sparsely captured images. This removes the need to continuously record the environment, improving smartphone battery life and making the system more convenient to use.

We assume an existing network connection, so the localization process is not required to run exclusively on the smartphone, but can utilize e. g. GPUs on a server system. Although we are interested in the prediction of both orientation and position, an accurate position is more important. The prediction of a yaw angle resp. compass direction would be enough for the use case, as illustrated in fig. 3.1b.

The only inputs we consider are RGB images; in particular, no depth information must be required for performing a prediction, as current smartphones are not commonly equipped with RGB-D sensors.

We assume that the localization system has access to a sparse database of georeferenced images, obtained for example from a Structure from Motion (SfM) system [36, 1] or an indoor mapping device [24].

Query images to the localization system are expected to not depict completely new scenes, but rather show a previously seen location under different viewing conditions. Depending

(a) RGB image. (b) Associated camera pose.

Figure 3.1.: **Illustration of the localization use case.** The goal of localization is to predict the position and orientation (right) of a single RGB image (left) in a well-known indoor or outdoor environment. Position errors up to multiple meters are acceptable, as humans can still successfully use such a system for localization [20]. Images by NavVis GmbH. Contains maps based on map data copyrighted by OpenStreetMap contributors, available from `http://www.openstreetmap.org`.

on whether scenes are indoors or outdoors, this can include [24, 2]: varying lighting and weather conditions; changing seasons and vegetation; occlusions due to pedestrians, vehicles, furniture, or other objects; smaller environment changes such as opened or closed doors, replaced posters or banners, or moved objects; and using different cameras to capture images.

3.2. The Localization Task

Working towards this use case, we define the *localization task* as determining the location of an image in an arbitrary coordinate frame. More specifically, we want to predict the *camera pose* of an image. This is also known as *camera relocalization* [31].

In the basic localization task, each input image $\boldsymbol{x} \in I$ is processed independently to predict a location $\boldsymbol{y} = f(\boldsymbol{x})$. How this location is represented depends on the used approach and is detailed in section 3.3.

We also consider an extended task, where sequences of t RGB images $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}$ taken in short temporal succession are localized jointly as $\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(t)} = f(\boldsymbol{x}^{(0)}, \ldots, \boldsymbol{x}^{(t)})$.

3.2.1. Related Research Areas

The localization task is related to *Visual Place Recognition*, which talks about a system concerned with the question of whether an "image is of a place it has already seen" and, if so,

which of a number of known places the image depicts. A Visual Place Recognition system can work purely by retrieving similar images of already seen places without having to know their positions [39], and is therefore a too general concept for the localization task, where we need to obtain the explicit location of an image.

Visual Place Recognition is related to *Simultaneous Localization and Mapping* (SLAM) [39], which estimates a model of the environment and the state of a robot in this environment at the same time [4]. The localization task considered here is distinct from SLAM because SLAM systems usually perform continuous tracking and have problems to cope with large viewpoint changes between successive images [30, p. 1]. This makes them unsuitable for processing sparsely captured images as in our use case.

Another related area is *object pose prediction*, which is the inverse of camera pose prediction [9, p. 2]. [3] is an example of a Random-Forest-based system that can be used for both object and camera pose prediction, but has only been evaluated on fine-grained localization tasks, for example for augmented reality.

Interesting work has also been performed trying to understand how neural networks learn from images: in [40], images of locations are reconstructed from features extracted by a convolutional neural network.

3.2.2. Related Use Cases

Our setting is similar to [31], where pose regression is performed on outdoor datasets of at least a few hundred square meters of spatial extent. The resulting predictions are accurate up to a few meters, which is sufficient for our use case.

A lot of recent work on camera pose prediction [63, 3, 42] has only been tested in very small environments, such as rooms of a few square meters size, and achieve position accuracies of a few centimeters. In contrast to the outdoor datasets of [31], these datasets also have a very high image density. For example, the widely used 7 Scenes dataset [55] contains hundreds of images per square meter of floor space, whereas the images used in [31] have been recorded approximately every 1 m [31, p. 4].

Other related work [16, 28, 35, 67] performs localization on a global scale, i. e. *geolocation*. Due to the larger spatial extent of the used datasets, performance decreases proportionately, and the accuracies achieved by such systems are generally not useful for our use case. For example, [67] achieves street-level accuracy for only 3.6 % of input images [67, p. 4]. [35] performs global 6-DOF pose regression by matching 2D image features to 3D points in a globally aligned point cloud, which can be time-consuming: at the time of publication, the matching process took a few seconds for each image.

3.3. Approaches to Localization

In this section, three machine-learning-based approaches to the localization task used in recent work [67, 31, 2] are presented. All of them aim to predict the location $\boldsymbol{y}$ of an input image $\boldsymbol{x} \in I$, but differ not only in how $\boldsymbol{y}$ is obtained, but also in how much information $\boldsymbol{y}$ can convey. We denote the models as functions $f(\boldsymbol{x}; \boldsymbol{\theta})$ with parameters $\boldsymbol{\theta}$. For a machine-learning-based model, parameters $\boldsymbol{\theta}$ are obtained by minimizing a loss function $J(\boldsymbol{\theta})$.

In the approach presented in section 3.3.1, localization is seen as a classification problem, i. e. finding the location of an image is seen as equivalent to assigning the most probable out of a fixed number of classes to an image. In the second approach, presented in section 3.3.2, localization is seen as a regression problem, where a continuous pose in an arbitrary coordinate frame is predicted from an input image.

Both of these approaches can be implemented end-to-end, meaning that models can be directly trained to optimize a loss function comparing expected and actual location prediction. Additionally, the prediction performance and storage requirements of an end-to-end-trainable system do not depend on the number of images, but are constant after training.

Localization can also be approached as a content-based image retrieval (CBIR) problem, where a query image is matched against a georeferenced database created from reference images. A CBIR system computes possible locations for a query image by finding similar-looking images in the database [24, sec. 1]. The performance characteristics of this approach, which is presented in section 3.3.3, thus depend on the size and quality of the reference database. However, computing image similarities in itself can be performed independently of the search area, and can even be agnostic to the localization problem. Furthermore, other location-related information such measured WiFi signals can easily be introduced into the system, as described in e. g. [47].

3.3.1. Localization as Classification Problem

By dividing the search area into fixed number of N non-overlapping cells of arbitrary size, the localization task can be treated as a *classification* problem. Consequently, the best possible absolute accuracy of the system is bounded by the spatial extent of the cells. With this basic approach, only a position, but no orientation is predicted.

A classification model $f : I \to \mathbb{R}^N$ on an input image $\boldsymbol{x} \in I$ computes a class score vector $\boldsymbol{s} \in \mathbb{R}^N$, which can be interpreted as a discrete probability distribution over the considered search area [67, p. 2]. The most probable location $y \in \{1, \ldots, N\}$ is corresponding to the entry with the highest class score in $\boldsymbol{s}$:

$$y = \arg\max_i (s_i) \tag{3.1}$$

The individual scores s_i can also be used to obtain model confidence: class scores can be normalized to form a probability distribution, thus corresponding to the model confidence

about the input image having been taken at a certain location [67, p. 3].

Given an image $\boldsymbol{x} \in I$ with location label $\hat{y} \in \{1, \ldots, N\}$, an end-to-end network for classification can be trained by minimizing the *cross-entropy loss*, defined as [29]:

$$J(\boldsymbol{\theta}) = -\log\left(\frac{\exp(s_{\hat{y}})}{\sum_j \exp(s_j)}\right) \tag{3.2}$$

PlaNet [67] is an example for a geolocation system using the classification approach. It divides earth's surface into cells of variable size and utilizes a convolutional neural network to predict the class scores. The orientation of images as well as different height levels at the same position are not considered. The basic CNN-based model has been extended with an LSTM to more accurately predict locations from sets of multiple images taken from photo albums [67, sec. 4].

3.3.2. Localization as Regression Problem

Localization can be treated as a regression problem, where the output $\boldsymbol{y}$ of a model on an input image $\boldsymbol{x}$ directly describes the location in some reference coordinate frame, i. e. $\boldsymbol{y} = f(\boldsymbol{x})$.

In contrast to previously presented the classification approach, $\boldsymbol{y}$ can not only contain the location, but also the orientation of an image: $\boldsymbol{y} = \{\boldsymbol{p}, \boldsymbol{q}\}$, where $\boldsymbol{p} \in \mathbb{R}^3$ and $\boldsymbol{q} \in SO(3)$. Learning position and orientation at the same time can improve the performance of a model [31, sec. 5.2].

Given location labels $\hat{\boldsymbol{y}} = \{\hat{\boldsymbol{p}}, \hat{\boldsymbol{q}}\}$, a loss function for end-to-end training can be defined as:

$$J(\boldsymbol{\theta}) = d_p(\boldsymbol{p}, \hat{\boldsymbol{p}}) + \beta d_q(\boldsymbol{q}, \hat{\boldsymbol{q}}) \tag{3.3}$$

In this equation, d_q and d_p are distance functions between angular resp. positional predictions and labels. β is a weighting factor between angular and positional errors.

In general, no indication of the confidence of the model can be inferred from a regression model: it is forced to output a pose and has no way of expressing uncertainty about this prediction [67, p. 3]. This is a disadvantage compared to the classification-based approach described in section 3.3.1. However, for particular models, this problem can be remedied: [30] is a regression-based localization system that predicts multiple locations from a single input image. An uncertainty measure is then calculated from the variance of the location predictions. This idea is further explained in section 4.1.2.

Regression-based systems can in general produce arbitrarily fine predictions, while approaches based on classification are limited by the spatial extent defined for each of their classes. Similarly, the performance of systems based on content-based image retrieval is bounded by the minimum distance between the entries of the lookup database. Pose regression has been shown to be able to perform better than a classifier in [31, fig. 6].

[31] uses a convolutional neural network based on the Inception architecture [61] to regress a 6-DOF pose. The network model is explained in more detail in section 4.1.1. Leveraging the idea of *transfer learning*, the layers of the regression network are initialized with weights obtained by training Inception models on the large ImageNet [7] and Places [73] datasets. This reduces training time and prediction performance [31, sec. 5.2] and allows to train on small training datasets without overfitting [31, p. 1].

3.3.3. Content-Based Image Retrieval for Localization

The usual [67, p. 1] approach to the localization problem is to use content-based image retrieval. In this method, a model is used to transform an input image $\boldsymbol{x}$ into some representation $\boldsymbol{z}$, the *feature vector*, as $\boldsymbol{z} = f(\boldsymbol{x})$. Using a reference database containing pairs of feature vectors and location labels $(\boldsymbol{z}', \boldsymbol{y}')$, the nearest neighbor of $\boldsymbol{z}$ can be found as:

$$\boldsymbol{z}^* = \arg\min_{\boldsymbol{z}'}(\|\boldsymbol{z}' - \boldsymbol{z}\|) \tag{3.4}$$

The location $\boldsymbol{y}^*$ associated with the found nearest neighbor $\boldsymbol{z}^*$ can then be used as location prediction $\boldsymbol{y}$ for the input image $\boldsymbol{x}$.

Feature vectors can be extracted by a handcrafted algorithm such as Scale-Invariant Feature Transform (SIFT, [38]), or a machine-learning based approach, where appropriate features are learned during training. Examples of previous work on content-based image retrieval for localization on smartphones with handcrafted features include [54, 64].

Machine-learning-based approaches can be distinguished based on the used training dataset. A model can be pre-trained for a different task on a large dataset and then used to extract features [2]. It is also possible to train for a classification or regression objective on the target dataset, and use the outputs of an intermediate layer as feature vector [37]. Siamese networks, which consist of two weight-sharing networks running on two input images in parallel, can be used to train on pairs of images labeled as similar or dissimilar. This approach can be used with a *contrastive loss* [15] to minimize the distance between similar resp. maximize distance between dissimilar images in feature space. This is explained in more detail in section 4.3.2.

The Siamese network approach has been extended as triplet networks, where an image is given into the network together with one similar and one dissimilar image during each training step [66, 22, 13]. Recently, hybrid Siamese/triplet [68] and Siamese/regression approaches [9] to extract image features for 3D pose estimation have been proposed.

Instead of using just the output of a single layer as feature vector, outputs of multiple layers can be combined. [2] is a very recent approach to image-based localization that uses convolutional neural networks to extract binary features from RGB images. The used network model contains five convolutional layers and is initialized from ImageNet [7] weights. The outputs of the convolutional layers at different depths of the network are combined into a single, high-dimensional feature vector. The size of the feature vector is reduced by a selecting

a randomized subset while ensuring that features from all different depths of the network are included.

4. Deep Learning for Image-Based Localization

In this chapter, we describe how we apply deep neural networks as introduced in chapter 2 to the image-based localization task defined in chapter 3. In particular, we consider neural networks that perform pose regression as described in section 3.3.2 and neural networks that extract features to use in content-based image retrieval for localization, as described in section 3.3.3.

We do not consider a classification approach, because [31, sec. 3.1] suggests that trying to learn position without simultaneously distinguishing different orientations can hinder prediction performance. Intuitively, this could be because images taken at the same location, but facing different directions, can be very dissimilar (e. g. depicting completely different landmarks). Despite these visual dissimilarities, the model is expected to predict the same position for both input images. Conversely, images taken further apart from each other, but facing the same direction, might show the same landmarks.

We consider regression networks and feature extraction networks because they both have unique characteristics: regression networks are end-to-end trainable, but need to be tuned for each dataset; a CBIR-based approach needs a database to perform predictions, but can be trained independently of the localization area it should be deployed for.

In section 4.1, we introduce pose regression from a single image. Section 4.2 extends this approach to sequences of images. Section 4.3 discusses Siamese neural networks for feature extraction.

We introduce neural network models based on GoogLeNet [61] and ResNet-50 [17], both of which have been presented in section 2.6. These models are based on the following assumptions:

1. The training and testing images are either in landscape or portrait format, but not both formats within the same dataset.
2. The training data does not contain large outliers w. r. t. the pose labels.
3. The training data is labeled with 3-D position vectors and normalized quaternions for orientation.
4. Images given to the network have the same field of view.

4.1. Pose Regression from a Single Image

In this section, we introduce neural network models based on GoogLeNet and ResNet-50 to perform pose regression on a single RGB image.

4.1.1. PoseNet

PoseNet [31] is based on the GoogLeNet [61] model introduced in section 2.6.1. Instead of a classification vector, it outputs a 6-DOF pose $\boldsymbol{y} = \{\boldsymbol{p}, \boldsymbol{q}\}$ relative to an arbitrary coordinate frame, consisting of a position vector $\boldsymbol{p} \in \mathbb{R}^3$ and an orientation quaternion $\boldsymbol{q} \in \mathbb{R}^4$.

Compared to the original GoogLeNet, the network is modified as follows [31, sec. 3.2]:

- The three classification output blocks as shown in fig. 2.9 are removed.
- For the final output branch, a fully connected "feature" layer of width 2048 is added. For the two auxiliary outputs, the existing fully connected layers of width 1024 are kept. Dropout is applied after each of the fully connected layers.
- For each of the three output branches, fully connected *pose prediction layers* for position vector and orientation quaternion are added after the respective feature layers. This is illustrated in fig. 4.1.

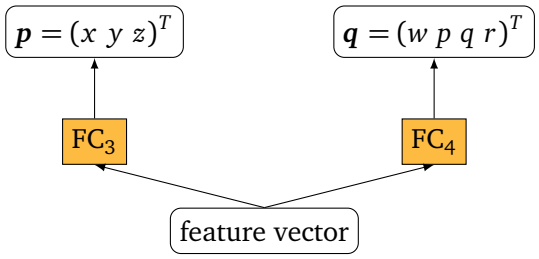


Figure 4.1.: **Pose prediction layers.** The 6-DOF pose output of a pose regression network is produced by two separate fully connected *pose prediction layers*, which are attached to each output branch of a network.

Each output's *pose loss* for a pose prediction $\boldsymbol{y} = \{\boldsymbol{p}, \boldsymbol{q}\}$ under network parameters $\boldsymbol{\theta}$ and a corresponding label $\boldsymbol{y}^* = \{\boldsymbol{p}^*, \boldsymbol{q}^*\}$ is defined as [31, eq. (2)]:

$$J_{\text{pose}}(\boldsymbol{\theta}) = \|\boldsymbol{p} - \boldsymbol{p}^*\|_2 + \beta \left\| \frac{\boldsymbol{q}}{\|\boldsymbol{q}\|_2} - \boldsymbol{q}^* \right\|_2 \tag{4.1}$$

The scalar hyperparameter $\beta \in \mathbb{R}$ determines the relative weight of orientation errors to positional errors and depends on the training dataset. As the network is not guaranteed to output a valid unit quaternion, it is necessary to normalize $\boldsymbol{q}$.

The loss of the overall network $J_{\text{PoseNet}}(\boldsymbol{\theta})$ is a weighted sum of the losses of each output and an additional regularization term, as defined in eq. (2.21).

PoseNet predicts both position and orientation for an image from the fully-connected feature layer. This has been found to be more effective than either training two completely separate networks or branching a single network into separate paths for position and orientation regression, which has been attributed to the network not being able to separate positional and orientation information from each other without having been trained jointly on both [31, sec. 3.1].

4.1.2. Probabilistic PoseNet

Similar to the original GoogLeNet, PoseNet uses dropout to reduce overfitting during training. In the *Probabilistic PoseNet* [30], dropout is also used during evaluation to gain insight into the uncertainty of the network about its predictions. For example, there could be multiple equally probable poses where a highly ambiguous image could have been taken. However, the basic PoseNet as described in section 4.1.1 can only produce a single deterministic output for an input image. In the Probabilistic PoseNet approach, the network is repeatedly evaluated with a fraction of its neurons randomly disabled, resulting in multiple different pose predictions for one single input image [30, sec. 4]. An uncertainty measure can then be obtained from the variance of the pose predictions [30, sec. 4.1].

The evaluation of a Probabilistic PoseNet is described in algorithm 4.1.

Algorithm 4.1 Evaluation of a Probabilistic PoseNet (taken from [30, sec. 4])

function PREDICT POSE(image $\boldsymbol{x}$, learned network parameters θ^*, number of samples N)
    samples ← new List
    for $n \leftarrow 1$ to N **do**
        network parameters $\theta \leftarrow \theta^*$
        for network parameter **in** θ **do** ▷ apply dropout
            with probability p set neuron activation to zero
        end for
        $x \leftarrow$ evaluate network
        samples.append(x)
    end for
    pose prediction ← average of samples
    uncertainty ← COMPUTE UNCERTAINTY(samples) ▷ Details in [30, sec. 4.A]
    return pose prediction, uncertainty
end function

4.1.3. PoseResNet

As ResNets [17] improve the performance on classification datasets such as ImageNet [7] compared to GoogLeNet, we introduce the *PoseResNet* model to investigate how better classification performance translates to our localization task.

PoseResNet is based on the ResNet-50 model, described in detail in section 2.6.2. Similar to the GoogLeNet modifications described in section 4.1.1 that resulted in PoseNet, ResNet-50 is modified for PoseResNet by replacing the classification output layer with a fully connected feature layer. The output of the feature layer is used by pose prediction layers as illustrated in fig. 4.1 to compute a 6-DOF pose.

The loss of the network $J_{\text{PoseResNet}}(\boldsymbol{\theta})$ is the same as the ResNet-50 loss as described in eq. (2.26), but substituting the pose loss $J_{\text{pose}}(\boldsymbol{\theta})$ described in eq. (4.1) for $J(\boldsymbol{\theta})$.

4.1.4. Directional PoseNet

We introduce the *Directional PoseNet* as a modification of the original PoseNet model. In this model, LSTMs are put before the pose prediction layers and thus take some intermediate outputs from either convolutional or fully connected layers as inputs and produce a new feature vector, which is then used for pose prediction with pose prediction layers.

LSTMs work on sequences of inputs, each of which can be a vector. By reshaping an intermediate feature vector into 2-D, we obtain a matrix that can be processed by an LSTM along multiple directions. This is explained in fig. 4.2.

There are multiple way to integrate an LSTM block with the PoseNet architecture. We consider the following modifications to PoseNet, with the specified layers applied after the $1 \times 1 \times 1024$ output of the average pooling layer of the final output branch:

Directional PoseNet A reshape as $32 \times 32 \rightarrow$ LSTM $\rightarrow$ dropout $\rightarrow$ pose prediction layers

Directional PoseNet B fully connected layer of width 2304 $\rightarrow$ reshape as $48 \times 48 \rightarrow$ LSTM $\rightarrow$ dropout $\rightarrow$ pose prediction layers

Directional PoseNet C fully connected layer of width 2048 $\rightarrow$ reshape as $32 \times 64 \rightarrow$ LSTM $\rightarrow$ dropout $\rightarrow$ pose prediction layers

Directional PoseNet D fully connected layer of width 1024 $\rightarrow$ reshape as $32 \times 32 \rightarrow$ LSTM $\rightarrow$ dropout $\rightarrow$ pose prediction layers

The two auxiliary branches are handled as variant D for all models.

For each model variant, we can distinguish between two-way processing (column- and row-wise) and four-way processing (column- and row-wise, forward and backwards) in the LSTM block, illustrated in fig. 4.3.

This approach is inspired by [65], where recurrent neural networks are applied to images and replace convolutional layers in a classification model. In the RNN-based layers proposed

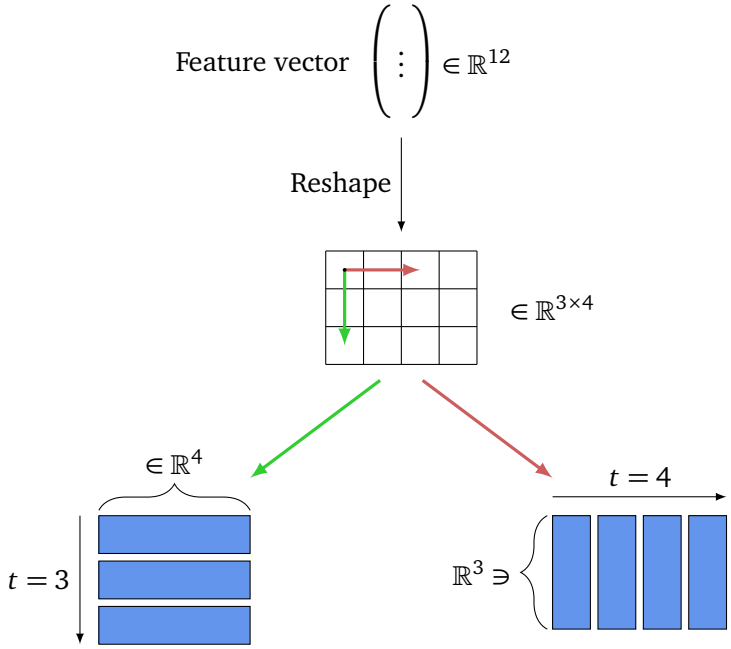
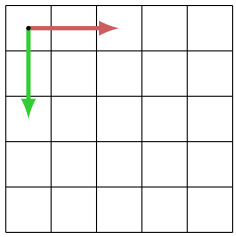
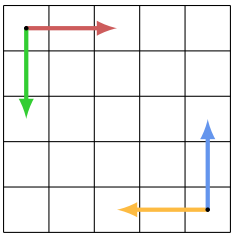


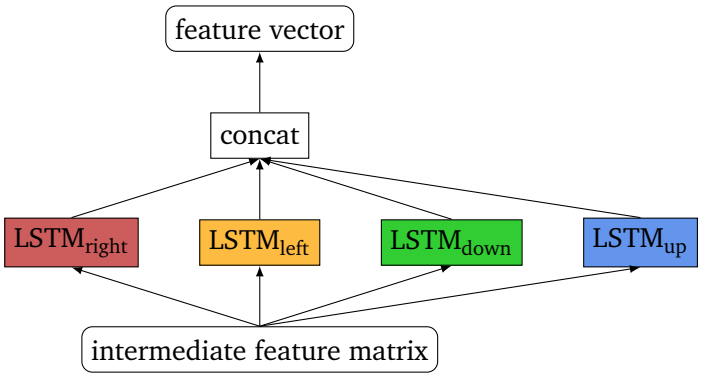
Figure 4.2.: **LSTMs on feature vectors.** To process a feature vector with an LSTM, the vector is reshaped into a matrix, which can then be processed column- or row-wise. In this example, a feature vector in $\mathbb{R}^{12}$ is reshaped as a 3×4 matrix. To process the matrix in y-direction (row-wise), we can see it as a sequence of 4-dimensional vectors of length 3 (left). Similarly, to process it in x-direction (column-wise), we can see the matrix as a sequence of 3-dimensional vectors (right).



(a) Two-way processing.



(b) Four-way processing.



(c) Construction of the final feature vector by four-way LSTM processing of intermediate features.

Figure 4.3.: **Multi-way processing of a feature vector.** After reshaping a feature vector into a matrix, LSTMs can be applied in multiple ways. In two-way processing (left), two LSTMs are applied to a column- and a row-wise sequence. In four-way processing (right) two LSTMs are applied to column- and row-wise sequences *each*. As the resulting feature vector is a concatenation of the final states of each LSTM (bottom), four-way processing results in a feature vector of twice the size of two-way processing.

there, an input image is divided into non-overlapping patches. Patches can be processed column- or row-wise; additionally, each column or row can be processed from start to end or vice versa. This leads to four possible axes along which an RNN can process an input image.

Bringing these ideas of "where to place the LSTM block" and "how to apply an LSTM on a feature vector" together, fig. 4.4 shows a DirectionalPoseNet-D-2, i. e. variant D with two-way processing.

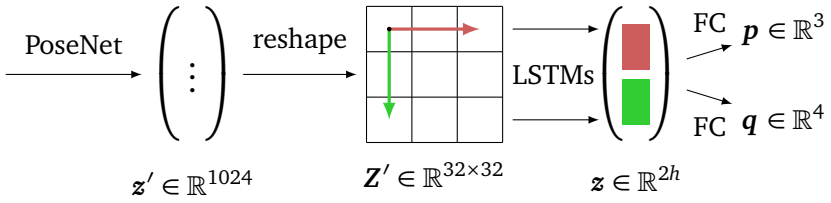


Figure 4.4.: **Directional PoseNet D-2 model.** The 1024-dimensional output of the fully connected feature layer of PoseNet, denoted as $\boldsymbol{z}'$, is reshaped into a matrix $\boldsymbol{Z}'$ of 32×32 elements. Two LSTMs are used to process the matrix in x- and y-direction, i. e. take 32 sequences of vectors in $\mathbb{R}^{32}$ as inputs. This is illustrated in more detail in fig. 4.2. The final states $\boldsymbol{h}_x^{(32)}, \boldsymbol{h}_y^{(32)} \in \mathbb{R}^h$ of both LSTMs after processing the 32 sequences are concatenated into a feature vector $\boldsymbol{z}$ of size $2h$ and used as input for the pose prediction layers, which are illustrated in fig. 4.1.

4.2. Pose Regression from Image Sequences

The pose regression models presented in section 4.1 process a single image at a time. However, when processing successively captured images or images originating from videos, it should be possible to use this information to improve the localization prediction. For example, PlaNet uses LSTMs to predict the locations of multiple pictures from a photo album [67, sec. 4].

In this section, we introduce neural networks to process sequences of images for pose regression. All of these consist of two components: a convolutional neural network to extract a feature vector independently from each image of an image sequence, and an LSTM that predicts a 6-DOF pose from each feature vector in a sequence. This basic architecture is illustrated in fig. 4.5.

4.2.1. Training LSTMs on Feature Sequences

A database of images and 6-DOF poses, such as one used for training single-image pose regression as described in section 4.1, can easily be transformed into a pose-labeled *feature* database by running a feature extractor on the images. If the images resp. features are sorted in a meaningful way, sequences of these features can be used as inputs to an LSTM network

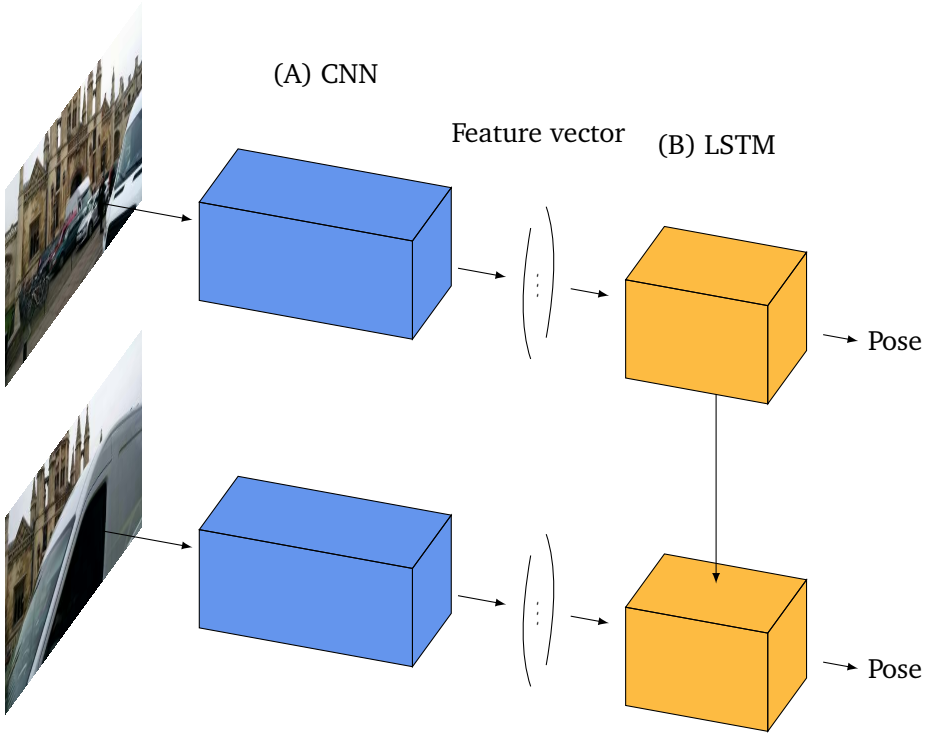


Figure 4.5.: **Pose regression for image sequences.** For each RGB image in an image sequence, the CNN (A) extracts a feature vector. Each feature vector is given into the LSTM (B) in sequence, resulting in a 6-DOF pose prediction for each input image. Example images from King's College dataset [31].

that predicts a pose for each input. Due to the split into two distinct stages, feature extraction (stage A in fig. 4.5) and pose prediction (stage B in fig. 4.5), the system is not end-to-end trainable.

A simple model for pose regression from feature vectors, termed *Feat-LSTM*, is shown in fig. 4.6.

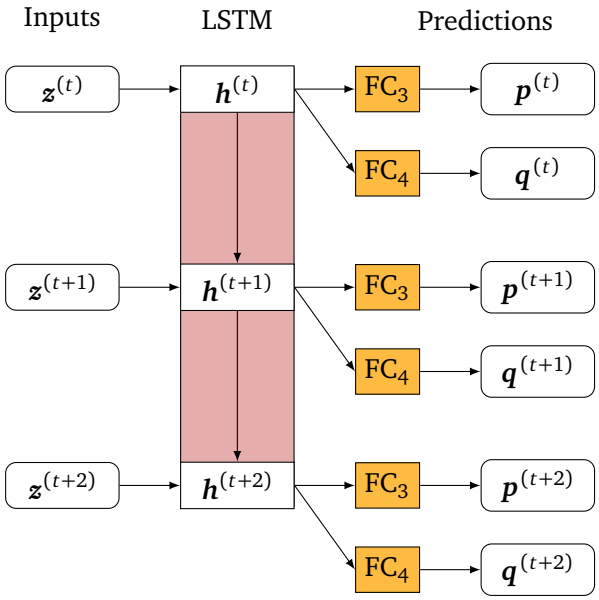


Figure 4.6.: **Feat-LSTM model.** Feat-LSTM regresses 6-DOF poses $\{\boldsymbol{p}, \boldsymbol{q}\}$ from sequences of input features $\boldsymbol{z}$. In this example, poses are predicted for a sequence of length 3.

4.2.2. Training LSTMs on Image Sequences

A natural extension of Feat-LSTM is to train end-to-end from sequences of images instead of distinguishing between the feature extraction and the pose prediction stages, as shown in fig. 4.5. We introduce two models based on this approach:

For the *PoseNet-LSTM* model, we run a PoseNet on each input image and use the output of its feature layer as input for an LSTM. For the second model, the PoseNet used to extract features from each single image is replaced by a Directional PoseNet. As the Directional PoseNet internally uses an LSTM, this model is named *Stacked LSTM*.

For both models, we use bidirectional processing, i. e. image sequences are simultaneously processed by two LSTMs, one operating from start to end and the other one in the opposite direction. Dropout is applied on the outputs of both LSTMs, which are then concatenated and used by pose prediction layers as depicted in fig. 4.1 to output position and orientation for each image in the sequence.

4.3. Feature Extraction with Siamese Neural Networks

In this section, we discuss the use of Siamese networks for feature extraction, with the goal of performing content-based image retrieval for localization.

For this approach, the only training data we consider are pairs of images labeled as similar or dissimilar. In contrast to many other approaches, e. g. the previously presented pose regression approach, the resulting feature extraction network is independent of the localization area and can also be deployed for areas without a common coordinate system. Images from different datasets can be used to train a single network, the only prerequisite being that they are correctly labeled as similar or dissimilar.

One crucial aspect of this approach is thus the way how images are determined to be similar. In section 4.3.1, we describe how to obtain such image similarities from RGB-D data. We do not consider how to extract image pairs from datasets where depth images are not available; however, a similar algorithm could be applied to datasets with sparse 3-D points, such as datasets reconstructed with Structure from Motion techniques. In section 4.3.2, we introduce a feature extraction CNN that can be trained on such a database.

4.3.1. Selecting RGB-D Image Pairs for Training Siamese Networks

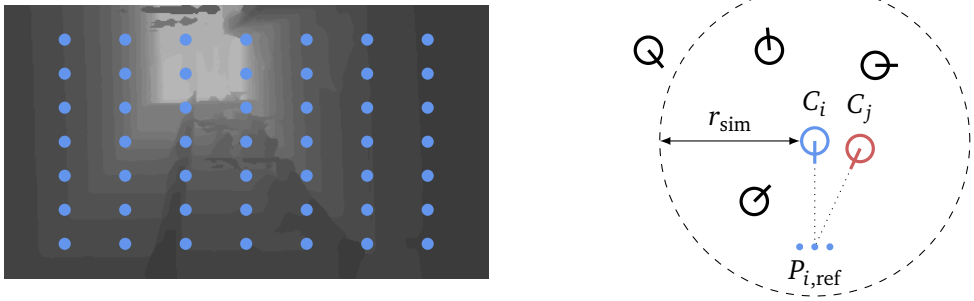
For localization, we only consider images taken at nearby positions *and* viewing many of the same landmarks as similar. We extract these pairs from datasets of RGB-D images $\boldsymbol{I}$, $\boldsymbol{D}$ with known 6-DOF camera poses C as follows:

For each reference camera $C_i \in C$, all other cameras $C_j \in C \setminus \{C_i\}$ within a maximum search radius $r_{\text{sim}} > 0$ around C_i are considered candidate cameras. To compute which candidate cameras C_j show a similar scene as the reference camera C_i, we first extract 3D reference points $\boldsymbol{p}_{i,\text{ref}}$ from C_i. For this, we compute n uniformly spaced points in pixel coordinates, look up the associated depth values in the depth image $\boldsymbol{D}_i$, and project them into world coordinates. See fig. 4.7 for an illustration. This is inspired by the algorithm described in [51, sec. 6.1], but uses multiple reference points instead of calculating only a single characteristic depth per camera.

Using the reference points obtained from C_i, we compute a similarity measure between C_i and C_j as described in algorithm 4.2. If the similarity is above a threshold $s_{\text{sim}} \in [0, 1]$, we store the RGB images $\boldsymbol{I}_i$ and $\boldsymbol{I}_j$ as a pair of similar images. The number of similar image pairs is denoted as $N_{\text{similar pairs}}$.

Finding dissimilar image pairs closely resembles the algorithm for finding similar images, except that candidate cameras are selected according to a *minimum* radius $r_{\text{dissim}} > 0$ and must have a similarity *below* a threshold $s_{\text{dissim}} \in [0, 1]$.

To prevent an unbalanced training database due to a large number of dissimilar image pairs, only $\alpha \cdot N_{\text{similar pairs}}$ for $\alpha < 1$ dissimilar image pairs are randomly selected from all possible dissimilar image pairs.



(a) Selection of reference points.

(b) Selection of similar cameras.

Figure 4.7.: **Finding similar cameras for a reference camera.** For a reference camera C_i, 3-D reference points $P_{i,\text{ref}}$ are extracted by selecting depth values at uniformly spaced pixel coordinates (x, y) in the reference depth image $\boldsymbol{D}_i$ (left). The resulting 3-D points $(x, y, \boldsymbol{D}_i(x, y))$ are projected into world coordinates. A candidate camera C_j (in red) in a specified radius r_{sim} around the reference camera C_i (in blue) is considered similar if it sees more than some specified fraction of all the reference points $P_{i,\text{ref}}$ (right). Image inspired by [51, fig. 6.1].

Algorithm 4.2 Computing similarity of a candidate camera for given reference points.

function COMPUTE SIMILARITY(candidate camera C, candidate depth image $\boldsymbol{X}_d$, reference points $\boldsymbol{P}$)
  seenPoints $\leftarrow 0$
  for $\boldsymbol{p} \leftarrow$ get next reference point from $\boldsymbol{P}$ **do**
    $x, y, z \leftarrow$ project $\boldsymbol{p}$ into coordinate system of candidate camera C
    if x, y outside image dimensions **or** point behind camera ($z < 0$) **then**
      continue
    end if
    depth $\leftarrow \boldsymbol{X}_d(x, y)$
    if $|\boldsymbol{X}_d(x, y) - z| <$ threshold **then**
      seenPoints $\leftarrow$ seenPoints + 1
    end if
  end for
  return $\frac{\text{seenPoints}}{\#\text{ reference points}}$
end function

4.3.2. Siamese ResNet-50

For the Siamese ResNet-50, the model as described in section 2.6.2 is duplicated up to the final output block. The layers of both Siamese branches share their weights, so the number of overall network parameters does not change. For each Siamese branch, a fully connected feature layer is added.

During training, the network is optimized to extract similar features from similar images. This is achieved by applying a contrastive loss on the Siamese feature vectors $\boldsymbol{z}_1 \in \mathbb{R}^z$ and $\boldsymbol{z}_2 \in \mathbb{R}^z$ extracted from the image pairs. The contrastive loss is defined as [15, eq. (4)]:

$$J_{\text{contrastive}}(\boldsymbol{\theta}) = \frac{1}{2} s(\|\boldsymbol{z}_1 - \boldsymbol{z}_2\|_2)^2 + \frac{1}{2}(1-s)\max(0, m - \|\boldsymbol{z}_1 - \boldsymbol{z}_2\|_2)^2 \tag{4.2}$$

$s \in \{0, 1\}$ denotes the similarity of the two training images; if $s = 1$, the images are similar. $m > 0$ is a scalar hyperparameter describing the required minimum margin between the feature vectors of two dissimilar images. Only if the Euclidean distance between $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$ is smaller than m for a pair of dissimilar images, it contributes to the loss of the network. Therefore, dissimilarity between feature vectors is not maximized infinitely, but only up to a certain margin.

An illustration of a Siamese model is shown in fig. 4.8.

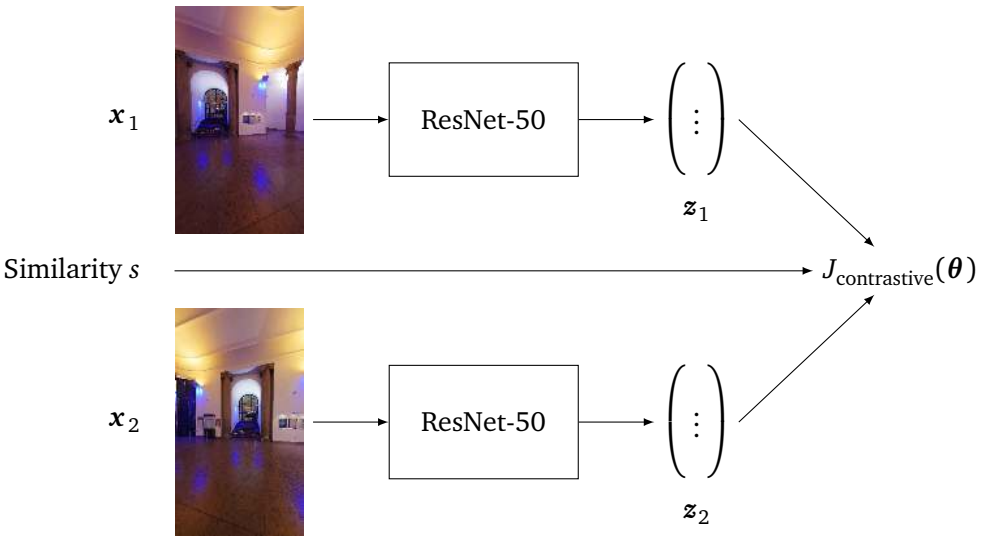


Figure 4.8.: **Siamese ResNet-50 training.** The Siamese ResNet-50 is trained with contrastive loss $J_{\text{contrastive}}(\boldsymbol{\theta})$, described in eq. (4.2). As input, the network requires pairs of images $(\boldsymbol{x}_1, \boldsymbol{x}_2)$ labeled with a similarity value $s \in \{0, 1\}$. The contrastive loss minimizes the distance between images with a similarity value of one and maximizes the distance of feature vectors with a similarity of zero up to a certain margin. Example images by NavVis GmbH.

5. Experimental Results

In section 5.1, the various datasets used in this work are introduced. Next, section 5.2 describes common preprocessing and setup steps utilized for all experiments. In the remaining part of the chapter, we report results for experiments on pose regression from single images in section 5.3, for pose regression from image sequences in section 5.4, and for feature extraction for CBIR in section 5.5.

5.1. Datasets

In this section, the three dataset collections used for the experiments are presented.

Cambridge Landmarks comprises multiple outdoor datasets obtained from smartphone videos. It has been introduced in the original PoseNet publication [31] and is used here to compare the performance of our models to PoseNet.

Deutsches Museum is an indoor dataset consisting of high-resolution camera images taken at discrete locations and orientations. It also contains depth images, which makes it suitable for testing a feature extraction approach based on the algorithm presented in section 4.3.1.

Dubrovnik [36] is a heterogeneous outdoor dataset consisting of images downloaded from the Internet, which makes it a challenging dataset for pose regression.

5.1.1. Cambridge Landmarks

The Cambridge Landmarks datasets [31] comprise five outdoor datasets of RGB images and associated 6-DOF poses of the city center of Cambridge, UK. Each dataset includes multiple video sequences collected by a pedestrian with a smartphone, with each sequence used either as training or testing data as a whole. RGB images were extracted from the video sequences by sub-sampling at a frequency of 2 Hz, resulting in images spaced approximately 1 m from each other. Structure from Motion was used to label each image with a 6-DOF pose, expressed as a position vector $\boldsymbol{x} \in \mathbb{R}^3$ and an orientation quaternion $\boldsymbol{q} \in \mathbb{R}^4$ relative to some coordinate frame [31]. Table 5.1 shows an overview of the five datasets. All images are in landscape format and have a resolution of 1920×1080 pixels. A number of example images are shown in fig. 5.1.

Table 5.1.: **Cambridge Landmarks [31] datasets.**

Dataset	# training images	# testing images
King's College	1220	343
Street	3015	2923
St. Mary's Church	1487	530
Shop Façade	231	103
Old Hospital	895	182

Figure 5.1.: **Example images from King's College [31].** The King's College dataset, part of Cambridge Landmarks, includes images with different weather conditions, occlusions, and pedestrians.

5.1.2. Deutsches Museum

Deutsches Museum is a technical museum in Munich, Germany. RGB images of the interior along with 6-DOF poses have been captured by NavVis GmbH with a mapping device similar to the one presented in [24]. Due to the cameras used on this device, images are available for five different orientations for each position, as illustrated in fig. 5.2. Depth images are available as well.

Table 5.2 shows an overview of the two datasets from Deutsches Museum. All images are in portrait format and have a resolution of 2304×4095 pixels.

Table 5.2.: **Deutsches Museum datasets.** Image pair datasets are used for Siamese training.

Dataset	# training images	# testing images
Ship Exhibition	2675	1610
Ship Exhibition (Pairs)	7035‡	n/a
Museum Subset (Pairs)†	72457‡	n/a

† Does *not* include the ship exhibition.
‡ Number of image pairs, not individual images.

Figure 5.2.: **Example images from Deutsches Museum.** The images for Deutsches Museum have been captured with an indoor mapping device similar to the one presented in [24]. Due to the camera setup of this device, images are available in five different orientations for each position, as shown here. Image by NavVis GmbH.

The Ship Exhibition and the Museum Subset dataset do not overlap, i. e. the two datasets contain images captured at different locations inside the museum. This is evident from fig. 5.3, where the camera positions of the captured images are visualized. Additionally, the rooms captured in the Ship Exhibition dataset are not visible on images of the Museum Subset dataset.

The Ship Exhibition training and testing datasets contain images captured inside the same rooms, but taken nearly 6 months apart. Camera poses of the Ship Exhibition dataset are visualized in fig. 5.4.

As noted in table 5.2, the Museum Subset dataset consists of pairs of images labeled as either similar or dissimilar. These image pairs are selected according to the algorithm detailed in section 4.3.1, using the parameters $s_{\text{sim}} = 0.6$, $s_{\text{dissim}} = 0$, $r_{\text{sim}} = 1\,\text{m}$, $r_{\text{dissim}} = 5\,\text{m}$, and $\alpha = 2$. Figure 5.5 shows an example of similar and dissimilar image pairs.

5.1.3. Dubrovnik

The Dubrovnik dataset [36] contains 6 844 images of the city of Dubrovnik, Croatia. Camera pose labels have been obtained by performing a reconstruction on all of these images. In a later step, the images have been split into training and testing images.

In contrast to the previously presented datasets, the Dubrovnik dataset consists of images downloaded from the Internet and is much more heterogeneous. Some example images are shown in fig. 5.6. In particular, the dataset includes both portrait and landscape images.

The given camera pose labels on Dubrovnik violate the assumption specified in chapter 4 that images given to a model for training or testing have the same field of view. This is evident from fig. 5.7, where two images with virtually the same camera position $\boldsymbol{p}$ are shown. From

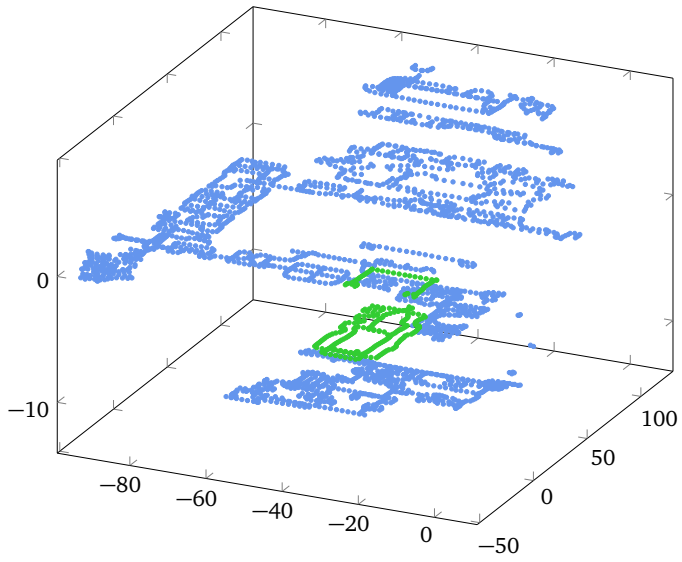


Figure 5.3.: **Poses of the Deutsches Museum datasets.** The Ship Exhibition training dataset (in green) is shown in the context of the Museum Subset dataset (in blue). Note that these datasets do not overlap. The datasets have been subsampled by a factor of 10 for visualization.

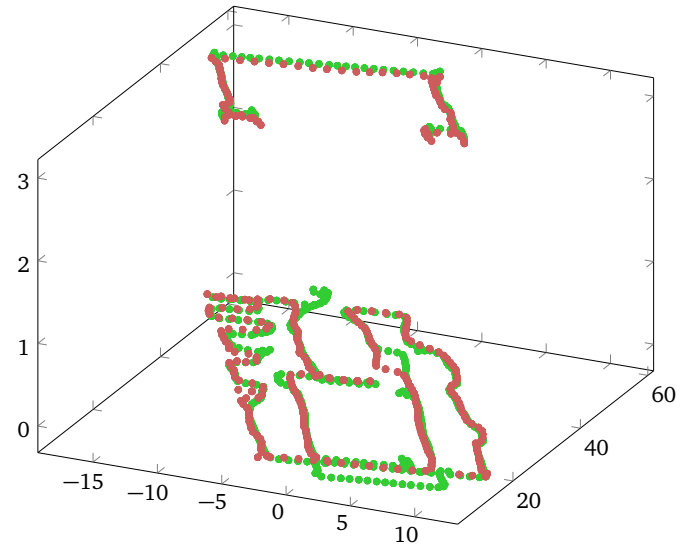


Figure 5.4.: **Poses of the Ship Exhibition datasets of Deutsches Museum.** This image shows the Ship Exhibition training (in green) and test (in red) datasets. These images have been taken at very similar positions on two floors, but approximately 6 months apart from each other.

(a) Reference image. (b) Similar image. (c) Dissimilar image.

Figure 5.5.: **Examples of similar and dissimilar images from Deutsches Museum.** Although all three images are visually similar, only the first two are similar from a localization perspective. The third image is more than 5 m away from the reference image, and does not see any of its reference points. Computing the similarity of images for localization is described by algorithm 4.2. Images by NavVis GmbH.

our point of view, such images therefore constitute outliers in the dataset.

5.1.4. Dataset Filtering

Due to the concern that cropping portrait and landscape images into a common format removes too much spatial information, we only use the landscape pictures, of which there is a larger number in the dataset. We also removed a number of rotated portrait pictures incorrectly saved in landscape format. Finally, we manually removed a small number of images with color effects, images with a large artificial border, composite images showing multiple different locations, images taken with a fisheye lense, and images which were smaller than 455 × 256 pixels. We call the resulting dataset *Dubrovnik Subset*.

Even after these steps, training a neural network on this dataset is inhibited by outliers. To remove a number of the worst outliers, we used a simple box filter to keep only images within camera position boundaries of $(-300 < x < 300, -50 < y < 50, -400 < z < 400)$.

Table 5.3 shows the number of training and testing images before and after the filtering process. fig. 5.8 shows the camera poses in the dataset before and after filtering.

Figure 5.6.: **Example images from Dubrovnik [36].** These example images show the heterogenity of the dataset, including: images with color effects, landscape and portrait formats, and fisheye images.

(a) Training image. (b) Testing image.

Figure 5.7.: **Varying scales on the Dubrovnik [36] dataset.** These images of the training and testing datasets have been labeled with a camera pose within 0.5 m of each other, but show a very different field of view. The red circle indicates where the scene of the testing image is contained in the training image.

Table 5.3.: **Dubrovnik datasets.** "Subset" datasets are created from the full Dubrovnik dataset according to the procedure described in section 5.1.4.

Dataset	# training images	# testing images
Dubrovnik [36]	6 044	800
Dubrovnik Subset	4 221	560
Dubrovnik Subset (filtered)	4 116	548

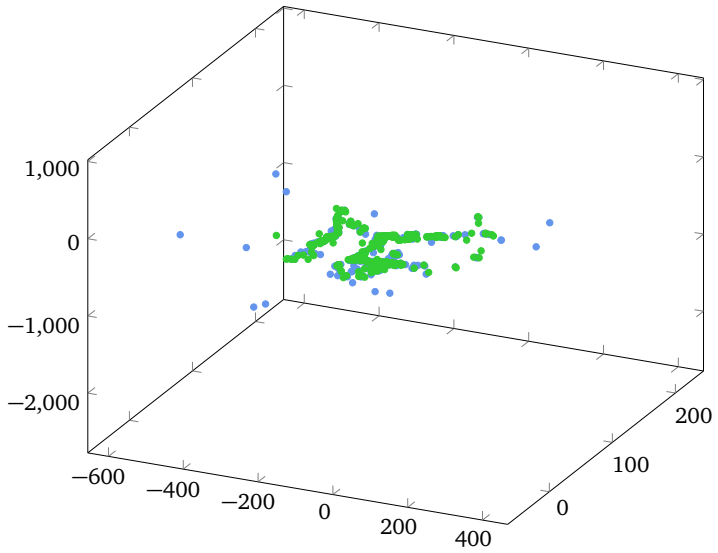


Figure 5.8.: **Poses of the Dubrovnik datasets.** The unfiltered dataset (blue) contains outliers with pose labels of thousands of meters. The filtered dataset (green) is created according to section 5.1.4 and removes these outliers. The datasets have been subsampled by a factor of 10 for visualization.

5.2. Experimental Setup

The original images of each dataset are first cropped to preserve their aspect ratio and then resized to the required dimensions of 455×256 resp. 256×455 pixels, depending on whether they are originally in landscape or portrait format. The resized images are stored in pose-labeled training resp. testing databases.

All networks take images of 224×224 pixels as input. We use random crops during training and central crops during testing. A mean image computed separately for each training database is subtracted from all images.

All experiments are performed on an NVIDIA Titan X and use the TensorFlow [41] framework with Adam [32]. Random shuffling is performed for each batch, and regularization is only applied to weights, not biases. An overview of training hyperparameters is given in table 5.4.

The values β for weighing positional and angular error of the pose loss of eq. (4.1) for each dataset are given in table 5.5.

During training, a snapshot of the network parameters $\boldsymbol{\theta}$ is saved every five epochs. The results are reported from the snapshot with the lowest positional error on the testing dataset.

Table 5.4.: **Overview of training hyperparameters.**

		PoseNet	Dir. PoseNet	PoseResNet
Regularization parameter $\lambda^{\dagger}$		0.0002	0.0002	0.0002
Auxiliary loss weights $\gamma^{\ddagger}$		0.3	0.3	n/a
Dropout probability		0.5	0.5	n/a
Adam [32]	ϵ	0.1	1	1
	β_1	0.9	0.9	0.9
	β_2	0.999	0.999	0.999

† λ is defined in eq. (2.15).
‡ γ is defined in eq. (2.21).

Table 5.5.: **Overview of loss parameters β, as defined in eq. (4.1).** Values for the Cambridge Landmarks datasets been taken from the PoseNet Caffe [27] implementation (see appendix A).

Dataset	β
King's College	500
Street	2000
St. Mary's Church	250
Shop Façade	100
Old Hospital	1000
Dubrovnik	2000
Deutsches Museum	500

5.3. Pose Regression from Single Images

In this section, we examine different models for pose regression on the Cambridge Landmarks and Dubrovnik datasets.

5.3.1. Reproducing PoseNet results

Training with Stochastic Gradient Descent and Momentum [46] of 0.9 and a learning rate of $1e^{-5}$ on batches of size 75 as specified in [31, sec. 3.2] was not successful, as the loss went towards infinity after a few iterations when initializing PoseNet from weights of the Places dataset [73].

Using Adam [32] with a base learning rate of $1e^{-3}$ and a batch size of 75, we were able to achieve similar results as the ones in [31, fig. 6] for most datasets. The results are given in table 5.6. Note that the angular errors reported for PoseNet are exactly twice as large as the ones published in [31, fig. 6]. With this, we follow [30] by the same author, which reports results in the same way.

Interestingly, while we achieve very similar results to [31] for most datasets, we report much better results for King's College. We can only speculate that the original author stopped training too early.

Table 5.6.: **Median localization results for PoseNet on Cambridge Landmarks.** Our reproductions achieve significantly better results on King's College, but do not converge to a competitive result for Street.

Dataset	PoseNet [31]	PoseNet (ours)
King's College	1.92 m, 5.40°	1.25 m, 4.74°
Street	3.67 m, 6.50°	14.19 m, 31.49°
Old Hospital	2.31 m, 5.38°	2.28 m, 5.63°
Shop Facade	1.46 m, 8.08°	1.43 m, 7.86°
St Mary's Church	2.65 m, 8.48°	2.33 m, 9.93°

For the Street dataset, training was more challenging. The Street dataset is unique in that the training database consists of four distinct video sequences, each filmed in a different compass direction. This results in training images at similar positions, but with very different orientations. The other Cambridge Landmarks datasets mostly contain images with only one general orientation for each position. Using the same learning hyperparameters as for the other datasets, training did not converge. We experimented with different base learning rates for Adam [32] and gradient norm clipping [44], but could not achieve competitive results despite a converging training loss. Extensive hyperparameter search was inhibited by the large size of the dataset and the resulting long training time per epoch.

5.3.2. Visualizations

After the initial experiments, we performed some visualizations.

In fig. 5.9, an input image and a corresponding *saliency map* [56] is shown. In fig. 5.10, we show an input image overlaid with a *class activation map* (CAM, [72]). For this visualization, we attach global average pooling to an intermediate convolutional layer of PoseNet and remove the remaining layers. This results in an input size of 14×14 to the CAM block. Due to this small input size, this visualization is much coarser than the saliency map, but similarly highlights how the model concentrates on distinctive elements of buildings to infer the camera pose.

Finally, fig. 5.11 shows a visualization of the weights for each filter of the first convolutional layer of PoseNet and PoseResNet.

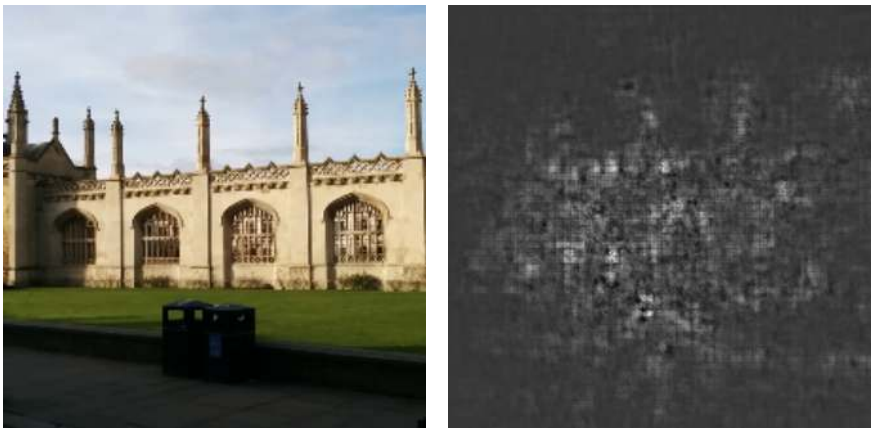


Figure 5.9.: **Saliency map.** A saliency map (right) has been created for an input image (left) from King's College [31]. Bright spots indicate areas the network considers important for pose regression. The visualization shows how the network focuses on the building façade.

5.3.3. Directional PoseNet on Cambridge Landmarks

To test the model proposed in section 4.1.4, we performed a number of experiments on the King's College dataset. We trained all Directional PoseNet variants with two-way processing with the hyperparameters given in table 5.4 for 800 epochs.

The results are given in table 5.7 and visualized in fig. 5.12. As the Directional PoseNet variants A and C showed the most promising results, we additionally trained their four-way variants.

Compared to both [31] and our reproductions in section 5.3.1, all Directional PoseNet variants showed improved results of up to 80 cm.

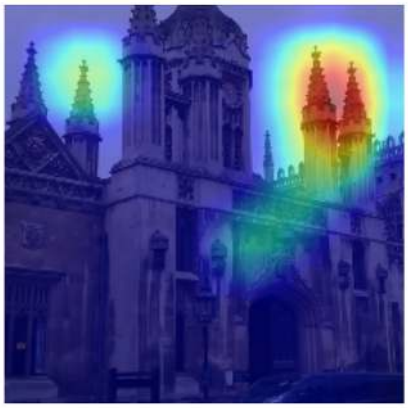
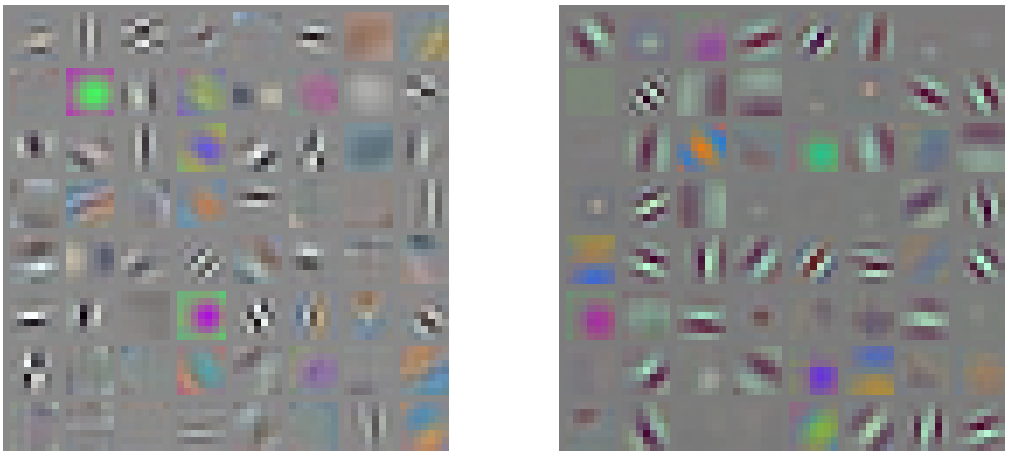
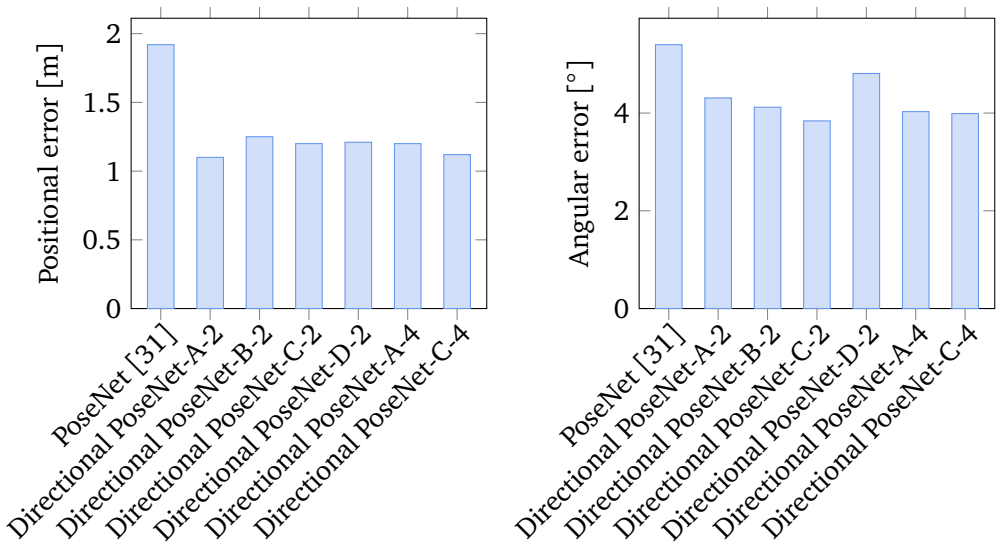


Figure 5.10.: **Class activation map.** The class activation map (CAM) is overlaid on an input image from King's College [31] as a heat-map. Red areas indicate parts of the image the network considers important for pose regression. The visualization shows how the network focuses on distinctive building elements.



(a) PoseNet. (b) PoseResNet.

Figure 5.11.: **Filter weights of the first convolutional layer of PoseNet resp. PoseResNet.** Both networks seem to learn similar features, in particular color blobs and edge-like features.



(a) Positional error.

(b) Angular error.

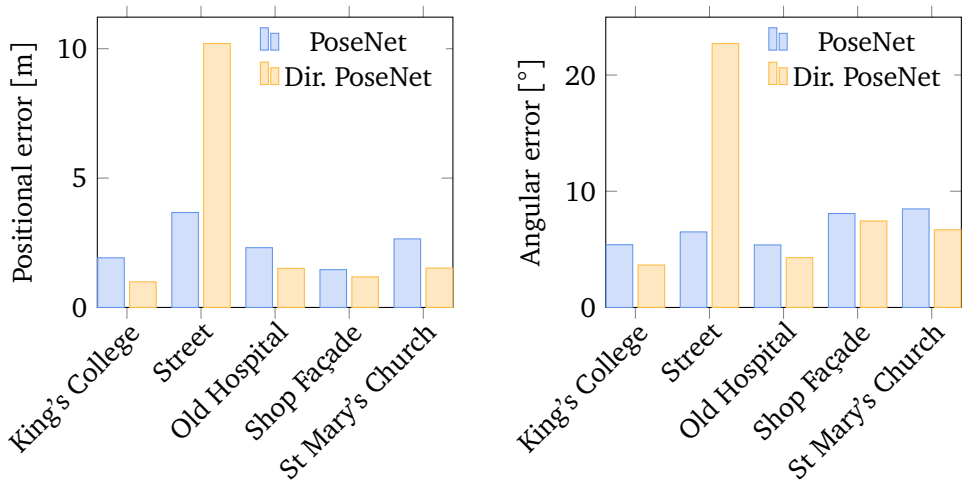
Figure 5.12.: **Median localization results for Directional PoseNet variants on King's College.** Numerical results are given in table 5.7.

Table 5.7.: **Median localization results for Directional PoseNet variants on King's College.** All Directional PoseNet variants perform better than PoseNet by more than half a meter.

Model	Median localization result
PoseNet [31]	1.92 m, 5.40°
Directional PoseNet-A-2	1.10 m, 4.31°
Directional PoseNet-B-2	1.25 m, 4.12°
Directional PoseNet-C-2	1.20 m, 3.84°
Directional PoseNet-D-2	1.21 m, 4.81°
Directional PoseNet-A-4	1.20 m, 4.03°
Directional PoseNet-C-4	1.12 m, 3.99°

As the Directional PoseNet C-4 gave the most balanced results for positional and angular error, we proceeded to run this model for 1200 epochs on all Cambridge Landmarks datasets. The results are given in table 5.8 and visualized in fig. 5.13. The Directional PoseNet consistently improves upon the original PoseNet architecture.

As in section 5.3.1, we could not obtain competitive results for the Street dataset, not even when fine-tuning from weights of other datasets. However, the Directional PoseNet still improved results on Street relative to the results of our own PoseNet training reported in table 5.6.



(a) Positional error.

(b) Angular error.

Figure 5.13.: **Directional PoseNet compared to PoseNet on Cambridge Landmarks.** Numeric results in table 5.8.

Table 5.8.: **Median localization results for PoseNet variants on Cambridge Landmarks.** The Directional PoseNet consistently outperforms other variants by up to 1 m. It did not converge to a competitive result for Street, but nevertheless outperforms our PoseNet results in table 5.6.

Dataset	PoseNet [31]	Bayesian PoseNet [30]	Dir. PoseNet C-4
King's College	1.92 m, 5.40°	1.74 m, 4.06°	0.99 m, 3.65°
Street	3.67 m, 6.50°	2.14 m, 4.96°	10.20 m, 22.71°
Old Hospital	2.31 m, 5.38°	2.57 m, 5.14°	1.51 m, 4.29°
Shop Façade	1.46 m, 8.08°	1.25 m, 7.54°	1.18 m, 7.44°
St Mary's Church	2.65 m, 8.48°	2.11 m, 8.38°	1.52 m, 6.68°

5.3.4. PoseResNet on King's College

For comparison with Directional PoseNet, we trained the PoseResNet model described in section 4.1.3 on King's College. As no weights for the Places dataset [73] are readily available, we initialized PoseResNet from ImageNet [7] weights. The results are given in table 5.9. PoseResNet improves on the results of the Directional PoseNet by 15 cm, but takes more than 3 times longer to train in our implementation.

Table 5.9.: **Median localization results for PoseResNet on King's College.** PoseResNet outperforms the Directional PoseNet by 15 cm.

Model	Median localization result
PoseNet	1.92 m, 5.40°
Directional PoseNet C-4	0.99 m, 3.65°
PoseResNet	0.84 m, 2.44°

5.3.5. Pose Regression on Dubrovnik Subset

For our experiments on Dubrovnik, an unmodified PoseNet was trained for 1200 epochs with a batch size of 75 and a base learning rate of $1e^{-2}$, applying gradient clipping [44] at a value of 10. We also trained a Directional PoseNet with the same parameters. In table 5.10, the results are compared with a recent method [50].

Table 5.10.: **Results for Dubrovnik Subset.** PoseNet and Directional PoseNet can not compete with a state-of-the-art method.

Method	Mean [m]	25 % [m]	50 % [m]	75 % [m]	90 % [m]	95 % [m]
p6p [50]	30.70	0.50	1.30	5.00	19.20	55.30
PoseNet	42.31	5.21	9.78	24.94	91.60	246.19
PoseNet†	26.30	3.83	7.12	17.97	63.35	124.85
D. PoseNet C-4†	25.26	4.16	7.30	19.50	48.10	123.46

† Trained on Dubrovnik Subset (filtered).

The results show that PoseNet and Directional PoseNet can not compete with a state-of-the-art method for the Dubrovnik dataset. In particular, some individual poses are predicted very far from their labels. Investigating these input images revealed that they are instances of the problem described in fig. 5.7, namely that their pose labels can be seen as outliers. In this situation, the Directional PoseNet does not give an advantage over the original PoseNet architecture.

5.4. Pose Regression from Sequences of Images

For these experiments, LSTM-based models as described in section 4.2.2 are trained on sequences of three images with a batch size of 50 for 800 epochs. We used a learning rate of $1e^{-3}$ and gradient clipping [44] at a value of 10. The results are given in table 5.11. We did not train on Street, as we could not get competitive results for this dataset even for single-view regression.

Table 5.11.: **Median localization results of PoseNet variants on Cambridge Landmarks image sequences.** While the use of sequences of images improves over PoseNet on single images, results do not improve compared to Directional PoseNet. The additional LSTM in the Stacked LSTM increases the error, indicating that the architecture of the model is not ideal.

Dataset	PoseNet-LSTM	Stacked LSTM
King's College	1.07 m, 3.28°	1.08 m, 4.42°
Old Hospital	2.03 m, 4.22°	2.51 m, 5.24°
Shop Façade	0.91 m, 6.68°	0.94 m, 6.68°
St Mary's Church	1.56 m, 6.01°	1.83 m, 6.83°

5.5. Feature Extraction Networks on Deutsches Museum

A Siamese ResNet-50 model as described in section 4.3.2 was trained for 20 epochs on the Deutsches Museum Subset dataset with a batch size of 50. A feature layer size of 2048 was used. After training, one half of the Siamese ResNet-50 was used to extract features for the Ship Exhibition training and testing datasets. It is important to use the same mean image that was used to train the Siamese model when extracting these features.

Using the features of the training dataset as lookup database, positional errors when using the best pose of the n nearest neighbors in feature space as prediction for a training image are reported in table 5.12 for $n \in \{1, 3, 5, 10, 20\}$.

To measure the impact of fine-tuning on the target dataset, the Siamese network is fine-tuned for another 20 epochs on the Ship Exhibition (Pairs) dataset. The results, given in table 5.13, show that the median localization error improves by 8 m for the nearest neighbor matching, despite the relatively short training time. In fig. 5.14, the cumulative errors for training on Museum Subset and additional fine-tuning on Ship Exhibition is visualized.

For comparison, we also train a PoseResNet with a batch size of 50 and a base learning rate of $1e-3$ on Ship Exhibition for 600 epochs. The results are shown in table 5.14.

Table 5.12.: **Results for Siamese-trained features on Ship Exhibition.** Using three nearest neighbors, results are good enough to be usable for the localization use case.

# NN†	Mean [m]	25 % [m]	50 % [m]	75 % [m]	90 % [m]	95 % [m]
1	13.69	3.30	11.77	20.80	30.28	37.31
3	6.66	0.55	3.69	10.19	17.51	21.91
5	4.59	0.37	1.79	6.62	13.14	16.83
10	2.71	0.30	0.78	3.72	7.58	10.70
20	1.47	0.25	0.45	1.58	4.26	5.81

† Number of nearest neighbors.

Table 5.13.: **Results for fine-tuned Siamese features on Ship Exhibition.** Fine-tuning for 20 epochs improves the median localization error for nearest-neighbor by 8 m over Siamese training on Museum Subset.

# NN†	Mean [m]	25 % [m]	50 % [m]	75 % [m]	90 % [m]	95 % [m]
1	6.96	0.68	3.32	11.77	18.48	22.37
3	3.36	0.32	0.79	4.15	11.17	15.82
5	2.28	0.27	0.55	2.36	6.46	11.64
10	1.37	0.22	0.38	1.05	3.92	6.27
20	0.81	0.20	0.32	0.60	1.82	3.65

† Number of nearest neighbors.

Table 5.14.: **Results for PoseResNet on Ship Exhibition.** Pose regression with a PoseResNet improves the median localization error by 1 m over fine-tuned Siamese features.

Model	Mean [m]	25 % [m]	50 % [m]	75 % [m]	90 % [m]	95 % [m]
PoseResNet	2.77	1.24	2.10	3.47	5.39	7.09

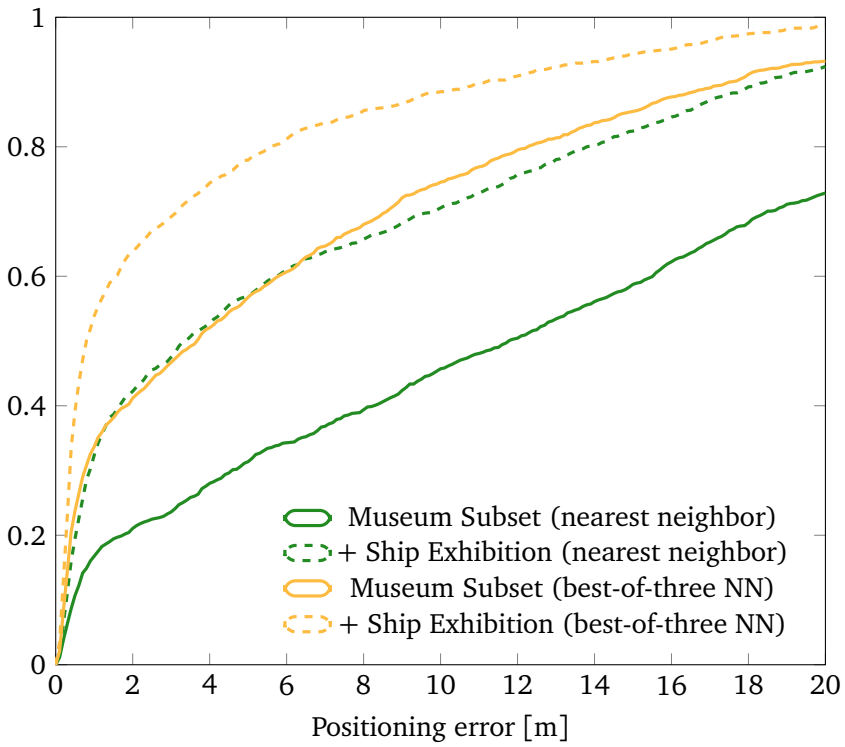


Figure 5.14.: **Cumulative error for Ship Exhibition.** This graph shows the percentage of testing images within a certain radius for nearest-neighbor and best-of-three-nearest-neighbors feature matching. After fine-tuning on the Ship Exhibition dataset for 20 epochs, nearest neighbor feature matching achieves the same results as best-of-three nearest neighbor matching with the network only trained on Museum Subset.

6. Summary

In this work, two approaches to image-based localization with neural networks have been evaluated: pose regression and content-based image retrieval.

For end-to-end pose regression, we extended the PoseNet model by applying LSTMs on a feature vector, improving the median positional error on the Cambridge Landmarks datasets by up to 1 m. We showed that the ResNet architecture, which has been successfully applied to image classification, outperforms the Directional PoseNet by 15 cm on King's College, but is slower to train.

In addition to single-view processing, we also performed pose regression from image sequences on the Cambridge Landmarks datasets. This improves the results by up to 1 m over PoseNet, but does not improve over the single-view Directional PoseNet.

For content-based image retrieval, we evaluated a ResNet model trained solely on pairs of images labeled as similar resp. dissimilar. Even without training on images of the localization area, results were sufficient for the use case of smartphone-based localization. Fine-tuning on images of the localization area greatly improved the results, lowering the nearest-neighbor positional error by 8 m, even though training was performed for only 20 epochs.

During some initial experiments, we experimented with the pose loss of eq. (4.1) on PoseNet, trying to replace the quaternion with an angle-axis representation or a single yaw angle. We did not obtain competitive results, which could possibly be improved by more extensive parameter tuning.

We also performed initial experiments with Feat-LSTM, described in section 4.2.1, on PoseNet and PoseResNet feature databases. However, as this gave no improvements over single-image regression, we abandoned this approach in favor of end-to-end regression from image sequences.

6.1. Discussion

Pose outliers in training data are problematic for regression networks As the presented neural network models are not scale-invariant [26], the heterogeneous Dubrovnik dataset can not be handled by the proposed networks. Additionally, label outliers make it hard for the networks to learn regression, as shown by comparing the results on the box-filtered and the unfiltered Dubrovnik dataset.

LSTMs can help to improve feature vectors Applying an LSTM on top of a PoseNet feature vector results in improved pose regression results. As the rest of the model remains unchanged, the LSTM seems to help the network to learn better features. Similar to how the LSTM-based layers in [65] learn features in the context of the whole image, in contrast to convolutional layers that learn only from local information, applying LSTMs on the feature vector learns new features in the context of all other features.

A ResNet-based model performs better than PoseNet for pose prediction PoseResNet performs better than PoseNet [31], also outperforming dense crops [31] and Probabilistic PoseNet [30]. This was expected, as the underlying architecture ResNet-50 performs better than PoseNet's parent model, GoogLeNet, on image classification tasks, indicating that ResNets are able to learn better features from input images than GoogLeNet.

Training from image sequences improves pose regression Following the results of [67], where LSTMs on image sequences have been shown to improve results for location prediction by classification, our experiments confirm that end-to-end training from image sequences is beneficial for pose regression when compared to PoseNet. However, results do not improve over single-image regression with an LSTM-based Directional PoseNet model, indicating that either the architecture is not ideal or image sequences should be constructed differently, e. g. by using larger spacing between images in a sequence.

7. Conclusion

Pose regression from RGB images remains a challenging task for neural networks when working with homogeneous datasets. As shown in our experiments with the Dubrovnik dataset, careful labeling of the training database is necessary to ensure that models can learn. Recent proposals towards scale-invariant neural network architectures such as [26] could partially remedy this problem.

Siamese training based on image similarity is a valid approach for localization by content-based image retrieval. Fine-tuning on a specific dataset for a small number of epochs gives results comparable to a pose regression network fine-tuned from ImageNet [7] for a much larger number of epochs.

7.1. Future Work

[30] describes a probabilistic approach to single-view pose regression that applies dropout during evaluation, obtaining multiple pose predictions for a single image and a measure of uncertainty of the network about its prediction. This could be applied to image sequences by performing dropout for each image during evaluation.

Further research could investigate how to introduce additional prior information, such as WiFi or cell signal measurements on a smartphone, into a regression network for image localization.

For upcoming smartphones equipped with time-of-flight sensors, depth information is available in addition to RGB images and could be used to improve indoor localization.

Supported by the observations regarding transfer learning in [31] and our own experiments, we expect that neural network architectures that have been shown to improve performance on large classification datasets such as ImageNet [7] will also result in improved localization performance. Recently proposed neural network models for image classification include [60, 70, 23, 6, 5].

Triplet training [53], where triplets of an image, a similar image, and a dissimilar image are given as inputs to the network, could further improve a Siamese feature extraction network.

A. Resources and Notes on Training

In this section, we list the software used for our experiments and give some implementation hints.

For initial experiments with PoseNet, we used the Caffe [27] code available at `https://github.com/alexgkendall/caffe-posenet`. This PoseNet model is based on the GoogLeNet implementation available at `http://vision.princeton.edu/pvt/GoogLeNet`. The same website also provides GoogLeNet weights for Places [73]. Note that this model is not compatible with the weights from `http://places.csail.mit.edu`.

For our own experiments, we used TensorFlow [41] `0.10rc0` on Python 3.4 and 3.5. We ran training on a single NVIDIA Titan X, using CUDA 7.5 and cuDNN 4. We also set `LD_PRELOAD` to use `tcmalloc` to avoid memory issues. We used software available at `https://github.com/ethereon/caffe-tensorflow` to convert Caffe weights to a TensorFlow-compatible format. It is important to note that Caffe works with BGR images, while TensorFlow expects RGB images. When using converted Caffe-trained network parameters to TensorFlow, the depth channel of input images to a network thus needs to be reversed.

For ResNet-50, we used the ImageNet [7] weights available from `https://github.com/KaimingHe/deep-residual-networks` and converted them for use in TensorFlow.

The Cambridge Landmarks datasets, as well as pre-trained PoseNet weights for Caffe, are available from `http://mi.eng.cam.ac.uk/projects/relocalisation`.

The Dubrovnik dataset [36] is available from `http://www.cs.cornell.edu/projects/p2f`.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] S. Agarwal, Y. Furukawa, N. Snavely, I. Simon, B. Curless, S. M. Seitz, and R. Szeliski. "Building rome in a day." In: *Communications of the ACM* 54.10 (2011), pp. 105–112 (cit. on p. 20).

[2] R. Arroyo, P. F. Alcantarilla, L. M. Bergasa, and E. Romera. "Fusion and Binarization of CNN Features for Robust Topological Localization across Seasons." In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2016 (cit. on pp. 21, 23, 25).

[3] E. Brachmann, F. Michel, A. Krull, M. Ying Yang, S. Gumhold, and c. Rother. "Uncertainty-Driven 6D Pose Estimation of Objects and Scenes From a Single RGB Image." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2016, pp. 3364–3372 (cit. on p. 22).

[4] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. D. Reid, and J. J. Leonard. "Past, Present, and Future of Simultaneous Localization And Mapping: Towards the Robust-Perception Age." In: *arXiv preprint arXiv:1606.05830* (2016) (cit. on p. 22).

[5] F. Chollet. "Xception: Deep Learning with Depthwise Separable Convolutions." In: *arXiv preprint arXiv:1610.02357* (2016) (cit. on p. 59).

[6] C. Cortes, X. Gonzalvo, V. Kuznetsov, M. Mohri, and S. Yang. "AdaNet: Adaptive Structural Learning of Artificial Neural Networks." In: *arXiv preprint arXiv:1607.01097* (2016) (cit. on p. 59).

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "ImageNet: A large-scale hierarchical image database." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2009, pp. 248–255 (cit. on pp. 15, 25, 30, 53, 59, 60).

[8] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell. "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition." In: *ICML*. 2014, pp. 647–655 (cit. on p. 15).

[9] A. Doumanoglou, V. Balntas, R. Kouskouridas, and T.-K. Kim. "Siamese Regression Networks with Efficient mid-level Feature Extraction for 3D Object Pose Estimation." In: *arXiv preprint arXiv:1607.02257* (2016) (cit. on pp. 22, 25).

[10] K. Fukushima. “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position.” In: *Biological cybernetics* 36.4 (1980), pp. 193–202 (cit. on p. 2).

[11] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. C. Courville, and Y. Bengio. “Maxout networks.” In: *ICML (3)* 28 (2013), pp. 1319–1327 (cit. on p. 8).

[12] I. Goodfellow, Y. Bengio, and A. Courville. “Deep Learning.” Book in preparation for MIT Press. Online; accessed 01 August 2016. 2016. URL: `http://www.deeplearningbook.org` (cit. on pp. 4, 5, 7, 8, 10–15).

[13] A. Gordo, J. Almazán, J. Revaud, and D. Larlus. “Deep Image Retrieval: Learning Global Representations for Image Search.” In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Springer. 2016, pp. 241–257 (cit. on p. 25).

[14] A. Graves, A.-r. Mohamed, and G. Hinton. “Speech recognition with deep recurrent neural networks.” In: *Proceedings of the IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 6645–6649 (cit. on pp. 7, 11).

[15] R. Hadsell, S. Chopra, and Y. LeCun. “Dimensionality reduction by learning an invariant mapping.” In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Vol. 2. IEEE. 2006, pp. 1735–1742 (cit. on pp. 25, 38).

[16] J. Hays and A. A. Efros. “IM2GPS: estimating geographic information from a single image.” In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2008, pp. 1–8 (cit. on p. 22).

[17] K. He, X. Zhang, S. Ren, and J. Sun. “Deep residual learning for image recognition.” In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2016, pp. 770–778 (cit. on pp. 2, 10, 17, 19, 27, 30).

[18] K. He, X. Zhang, S. Ren, and J. Sun. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.” In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. IEEE. 2015, pp. 1026–1034 (cit. on p. 8).

[19] K. He, X. Zhang, S. Ren, and J. Sun. “Identity mappings in deep residual networks.” In: *arXiv preprint arXiv:1603.05027* (2016) (cit. on p. 17).

[20] S. Hilsenbeck, D. Bobkov, G. Schroth, R. Huitl, and E. Steinbach. “Graph-based data fusion of pedometer and WiFi measurements for mobile indoor positioning.” In: *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM. 2014, pp. 147–158 (cit. on pp. 20, 21).

[21] S. Hochreiter and J. Schmidhuber. “Long short-term memory.” In: *Neural computation* 9.8 (1997), pp. 1735–1780 (cit. on p. 11).

[22] E. Hoffer and N. Ailon. “Deep metric learning using triplet network.” In: *International Workshop on Similarity-Based Pattern Recognition*. Springer. 2015, pp. 84–92 (cit. on p. 25).

[23] G. Huang, Z. Liu, and K. Q. Weinberger. "Densely Connected Convolutional Networks." In: *arXiv preprint arXiv:1608.06993* (2016) (cit. on p. 59).

[24] R. Huitl, G. Schroth, S. Hilsenbeck, F. Schweiger, and E. Steinbach. "TUMindoor: An extensive image and point cloud dataset for visual indoor localization and mapping." In: *2012 19th IEEE International Conference on Image Processing*. IEEE. 2012, pp. 1773–1776 (cit. on pp. 1, 20, 21, 23, 41, 42).

[25] S. Ioffe and C. Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." In: *arXiv preprint arXiv:1502.03167* (2015) (cit. on p. 15).

[26] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu. "Spatial Transformer Networks." In: *Advances in Neural Information Processing Systems (NIPS)*. 2015, pp. 2017–2025 (cit. on pp. 57, 59).

[27] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. "Caffe: Convolutional architecture for fast feature embedding." In: *Proceedings of the 22nd ACM international conference on Multimedia*. ACM. 2014, pp. 675–678 (cit. on pp. 47, 60).

[28] E. Kalogerakis, O. Vesselova, J. Hays, A. A. Efros, and A. Hertzmann. "Image sequence geolocation with human travel priors." In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. IEEE. 2009, pp. 253–260 (cit. on p. 22).

[29] A. Karpathy, J. Johnson, and F.-F. Li. *Course notes on CS231n: Convolutional Neural Networks for Visual Recognition*. Online; accessed 01 August 2016. 2016. URL: `https://cs231n.github.io` (cit. on pp. 2, 4–9, 11, 12, 24).

[30] A. Kendall and R. Cipolla. "Modelling uncertainty in deep learning for camera relocalization." In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2016, pp. 4762–4769 (cit. on pp. 13, 22, 24, 29, 48, 52, 58, 59).

[31] A. Kendall, M. Grimes, and R. Cipolla. "PoseNet: A Convolutional Network for Real-Time 6-DOF Camera Relocalization." In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. IEEE. 2015, pp. 2938–2946 (cit. on pp. 2, 15, 21–25, 27–29, 34, 40, 41, 48–52, 58, 59).

[32] D. Kingma and J. Ba. "Adam: A method for stochastic optimization." In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on pp. 46–48).

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." In: *Advances in Neural Information Processing Systems (NIPS)*. 2012, pp. 1097–1105 (cit. on pp. 2, 14).

[34] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. "Backpropagation applied to handwritten zip code recognition." In: *Neural computation* 1.4 (1989), pp. 541–551 (cit. on p. 2).

[35] Y. Li, N. Snavely, D. Huttenlocher, and P. Fua. "Worldwide pose estimation using 3d point clouds." In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Springer. 2012, pp. 15–29 (cit. on p. 22).

[36] Y. Li, N. Snavely, and D. P. Huttenlocher. "Location Recognition using Prioritized Feature Matching." In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Springer. 2010, pp. 791–804 (cit. on pp. 20, 40, 42, 45, 60).

[37] K. Lin, H.-F. Yang, J.-H. Hsiao, and C.-S. Chen. "Deep learning of binary hash codes for fast image retrieval." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. IEEE. 2015, pp. 27–35 (cit. on p. 25).

[38] D. G. Lowe. "Object recognition from local scale-invariant features." In: *Proceedings of the IEEE International Conference on Computer Vision*. Vol. 2. IEEE. 1999, pp. 1150–1157 (cit. on pp. 2, 25).

[39] S. Lowry, N. Sünderhauf, P. Newman, J. J. Leonard, D. Cox, P. Corke, and M. J. Milford. "Visual place recognition: A survey." In: *IEEE Transactions on Robotics* 32.1 (2016), pp. 1–19 (cit. on p. 22).

[40] A. Mahendran and A. Vedaldi. "Understanding deep image representations by inverting them." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2015, pp. 5188–5196 (cit. on p. 22).

[41] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://tensorflow.org/` (cit. on pp. 46, 60).

[42] D. Massiceti, A. Krull, E. Brachmann, C. Rother, and P. H. Torr. "Random Forests versus Neural Networks-What's Best for Camera Relocalization?" In: *arXiv preprint arXiv:1609.05797* (2016) (cit. on p. 22).

[43] V. Nair and G. E. Hinton. "Rectified linear units improve restricted boltzmann machines." In: *Proceedings of the 27th International Conference on Machine Learning (ICML)*. 2010, pp. 807–814 (cit. on p. 8).

[44] R. Pascanu, T. Mikolov, and Y. Bengio. "On the difficulty of training recurrent neural networks." In: *Proceedings of the International Conference on Machine Learning (ICML)*. 2013, pp. 1310–1318 (cit. on pp. 48, 53, 54).

[45] V. Pham, T. Bluche, C. Kermorvant, and J. Louradour. "Dropout improves recurrent neural networks for handwriting recognition." In: *Proceedings of the IEEE Conference on Frontiers in Handwriting Recognition (ICFHR)*. IEEE. 2014, pp. 285–290 (cit. on p. 14).

[46] N. Qian. "On the momentum term in gradient descent learning algorithms." In: *Neural networks* 12.1 (1999), pp. 145–151 (cit. on p. 48).

[47] M. Quigley, D. Stavens, A. Coates, and S. Thrun. "Sub-meter indoor localization in unmodified environments with inexpensive sensors." In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2010, pp. 2039–2046 (cit. on p. 23).

[48] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors." In: *Nature* 323 (1986), pp. 533–536 (cit. on pp. 2, 11, 13).

[49] H. Sak, A. W. Senior, and F. Beaufays. "Long short-term memory recurrent neural network architectures for large scale acoustic modeling." In: *INTERSPEECH*. 2014, pp. 338–342 (cit. on pp. 7, 11).

[50] T. Sattler, B. Leibe, and L. Kobbelt. "Efficient & Effective Prioritized Matching for Large-Scale Image-Based Localization." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2016) (cit. on p. 53).

[51] M. Schmidhammer. "Deep Learning for Virtual View Generation." MA thesis. Technische Universität München, Mar. 2016 (cit. on pp. 36, 37).

[52] J. Schmidhuber. "Deep learning in neural networks: An overview." In: *Neural Networks* 61 (2015), pp. 85–117 (cit. on p. 2).

[53] F. Schroff, D. Kalenichenko, and J. Philbin. "Facenet: A unified embedding for face recognition and clustering." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2015, pp. 815–823 (cit. on p. 59).

[54] G. Schroth, A. Al-Nuaimi, R. Huitl, F. Schweiger, and E. Steinbach. "Rapid image retrieval for mobile location recognition." In: *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2011, pp. 2320–2323 (cit. on pp. 1, 25).

[55] J. Shotton, B. Glocker, C. Zach, S. Izadi, A. Criminisi, and A. Fitzgibbon. "Scene coordinate regression forests for camera relocalization in RGB-D images." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2013, pp. 2930–2937 (cit. on p. 22).

[56] K. Simonyan, A. Vedaldi, and A. Zisserman. "Deep inside convolutional networks: Visualising image classification models and saliency maps." In: *arXiv:1312.6034* (2013) (cit. on p. 49).

[57] K. Simonyan and A. Zisserman. "Very deep convolutional networks for large-scale image recognition." In: *arXiv preprint arXiv:1409.1556* (2014) (cit. on p. 2).

[58] J. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. “Striving for Simplicity: The All Convolutional Net.” In: *ICLR (workshop track)*. 2015 (cit. on p. 10).

[59] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. “Dropout: a simple way to prevent neural networks from overfitting.” In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958 (cit. on pp. 13, 14).

[60] C. Szegedy, S. Ioffe, and V. Vanhoucke. “Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning.” In: *arXiv preprint arXiv:1602.07261* (2016) (cit. on pp. 2, 16, 17, 59).

[61] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. “Going Deeper With Convolutions.” In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2015, pp. 1–9 (cit. on pp. 2, 10, 16, 18, 25, 27, 28).

[62] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. “Rethinking the inception architecture for computer vision.” In: (2016), pp. 2818–2826 (cit. on p. 16).

[63] J. Valentin, A. Dai, M. Nießner, P. Kohli, P. Torr, S. Izadi, and C. Keskin. “Learning to navigate the energy landscape.” In: *arXiv preprint arXiv:1603.05772* (2016) (cit. on p. 22).

[64] D. Van Opdenbosch, G. Schroth, R. Huitl, S. Hilsenbeck, A. Garcea, and E. Steinbach. “Camera-based indoor positioning using scalable streaming of compressed binary image signatures.” In: *2014 IEEE International Conference on Image Processing (ICIP)*. IEEE. 2014, pp. 2804–2808 (cit. on p. 25).

[65] F. Visin, K. Kastner, K. Cho, M. Matteucci, A. Courville, and Y. Bengio. “Renet: A recurrent neural network based alternative to convolutional networks.” In: *arXiv preprint arXiv:1505.00393* (2015) (cit. on pp. 2, 30, 58).

[66] J. Wang, Y. Song, T. Leung, C. Rosenberg, J. Wang, J. Philbin, B. Chen, and Y. Wu. “Learning fine-grained image similarity with deep ranking.” In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2014, pp. 1386–1393 (cit. on p. 25).

[67] T. Weyand, I. Kostrikov, and J. Philbin. “Planet-photo geolocation with convolutional neural networks.” In: *arXiv preprint arXiv:1602.05314* (2016) (cit. on pp. 2, 22–25, 33, 58).

[68] P. Wohlhart and V. Lepetit. “Learning descriptors for object recognition and 3d pose estimation.” In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2015, pp. 3109–3118 (cit. on p. 25).

[69] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. “How transferable are features in deep neural networks?” In: *Advances in neural information processing systems*. 2014, pp. 3320–3328 (cit. on p. 15).

[70] S. Zagoruyko and N. Komodakis. “Wide Residual Networks.” In: *arXiv:1605.07146* (2016) (cit. on pp. 17, 59).

[71] W. Zaremba, I. Sutskever, and O. Vinyals. “Recurrent neural network regularization.” In: *arXiv preprint arXiv:1409.2329* (2014) (cit. on p. 14).

[72] B. Zhou, A. Khosla, A. Lapedriza, A. Torralba, and A. Oliva. “Places: An Image Database for Deep Scene Understanding.” In: *arXiv preprint (to be released)* (2016) (cit. on p. 49).

[73] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva. “Learning Deep Features for Scene Recognition using Places Database.” In: *Advances in Neural Information Processing Systems (NIPS)*. 2014, pp. 487–495 (cit. on pp. 25, 48, 53, 60).