UNIVERSITÉ DU LUXEMBOURG

Faculty of Science, Technology and Communication

Deep Convolutional Neural Networks for Camera Relocalization

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master in Information and Computer Sciences

Author:
Maciej Marcin Żurad

Supervisor:
Prof. Dr.-Ing. Holger Voos

Reviewer:
Prof. Dr. Christoph Schommer

Advisor:
Dr. Miguel A. Olivares-Mendez

November 2017

Abstract

The rise of inexpensive image sensors has greatly transformed numerous industries; from mobile devices, through surveillance to robotics; opening doors for novel application. However, processing images and retrieving valuable information from them has proven to be an utterly difficult task for computers, leaving many problems unsolved. An example of such problem is image-based localization, also known as camera relocalization, which given an image answers the question *"Where am I?"*. Until recently, images were localized using content-based image retrieval (CBIR) systems based on hand-crafted feature descriptors such as SIFT or ORB. However, they fail under various circumstances and generally do not scale well with the spatial extent of the environment we operate in.

The advent of Deep Learning and Convolutional Neural Networks (CNNs) has disrupted the entire field of Computer Vision as they proved themselves successful on many classical problems such as image classification and segmentation, human-pose estimation and optical-flow prediction. The main advantage of CNNs is the ability to learn appropriate features from the data by training the whole system end-to-end. PoseNet, PlaNet and VidLoc are recent neural network models that aim to solve image-based localization using regression or classification.

The motivation behind this thesis is to further investigate different approaches for image-based localization. Similarly to PoseNet, we formulate the problem as pose regression and further improve upon it by introducing quaternion algebra for proper attitude representation. In addition, we combine two recently developed approaches: (1) a multi-task loss function that learns the optimal weighting between position and orientation regression tasks, (2) a CNN followed by a spatial LSTM network for better structured feature correlation. Furthermore, we only finetune a small portion of the pretrained CNN feature extractor. Lastly, we extend the problem to videos and employ sequence-to-sequence regression model based on LSTMs. We evaluate the models on the 7Scenes dataset and introduce a new Airframe dataset, where localization is performed with respect to an object that changes position and orientation in the environment.

We achieve at least competitive, but sometimes outperforming results, while requiring considerably less computational power for training the models.

Abbreviations

ML	**M**achine **L**earning
DL	**D**eep **L**earning
NN	**N**eural **N**etwork
FC	**F**ully **C**onnected Layer
CNN	**C**onvolutional **N**eural **N**etwork
RNN	**R**ecurrent **N**eural **N**etwork
LSTM	**L**ong **S**hort **T**erm **M**emory
GRU	**G**ated **R**ecurrent **U**nit
ROS	**R**obot **O**perating **S**ystem
GPU	**G**raphics **P**rocessing **U**nit
DOF	**D**egree **Of** **F**reedom
UAV	**U**nmanned **A**erial **V**ehicle
UGV	**U**nmanned **G**round **V**ehicle
CBIR	**C**ontent **B**ased **I**mage **R**etrieval
SfM	**S**tructure **f**rom **M**otion

Introduction

Localization is a fundamental problem in robotics. It is often solved with a Global Positioning System (GPS), which is not suitable for urban or indoor scenarios due to satellite signal blockage or reflection. In so called GPS-denied environments, one must use other sensors for localization such as LIDARs, depth and image sensors. LIDARs are very expensive and often heavy, while depth sensors have short operational range. Image senors, on the other hand, are inexpensive, lightweight and provide high resolution images, but processing information from these sensors is a very challenging task.

Current state-of-the-art visual localization systems are based on hand-crafted feature descriptors such as SIFT [1], SURF [2] and ORB [3]. These systems perform very well in controlled environments, but are susceptible to varying illumination, different weather conditions, dynamic objects such as people, animals or the opening and closing of doors. This is due to the fact that these features fail to capture global context. Moreover, they also do not scale well with the spatial extent of environment and usually required massive amount of computational power, thus making it extremely difficult for real-time processing on embedded devices, such as unmanned ground vehicles (UGVs), unmanned aerial vehicles (UAVs) and virtual reality headsets.

In order to address these problems, we use deep learning to regress the full 6-DOF pose directly from images without the need to hand-craft features. The goal of deep learning is to learn a deep representation of the input space. Convolutional neural networks (CNNs) are an example of deep learning applied to image processing. For instance, images of an indoor environment can be decomposed into objects such as walls, desks and chairs, which further decompose into polygons and textures. Eventually, the basic concepts of this deep hierarchical representation are edges, corners and color blobs. Deep also refers to the fact that the number of layers used in a neural network is large.

We base our work on previously developed methods such as PoseNet [4] and VidLoc [5], as well as their further improvements published in [6–10]. The main contributions of this thesis are:

(1) We propose quaternion algebra for proper computation of the orientation loss based on quaternion angular error.

(2) We combine a multi-task loss function for learning optimal weighting between position and orientation regression tasks [7] with spatial LSTM architecture developed in [11].

(3) We introduce a new indoor dataset called **Airframe** containing approximately 19000 images of a reduced airframe model for learning localization with respect to that airframe. The dataset contains 6 subsets, each with a different position and orientation of the airframe model.

(4) We present a temporal GRU model for predicting 6-DOF poses from sequences of images and combine it with the multi-task loss function from (2).

(5) We evaluate our models on **Airframe** and **7Scenes** [12] datasets using 4 different CNNs as base feature extractors: *VGG16-Hybrid1365*, *GoogLeNet-ImageNet*, *GoogLeNet-Places365*, *InceptionResNetV2-ImageNet*

The remainder of the thesis is organized in the following way. Chapter 1 gives some necessary background on machine learning. In chapter 2, deep learning is described in detail, starting from basic neural networks, through convolutional neural networks to recurrent neural networks. Chapter 3 presents the proposed models for solving image-based localization using pose regression. Finally, experiments using those models on different datasets are described in chapter 4.

Contents

Chapter 1

Machine Learning

Methods proposed in this thesis are categorized as deep learning, a subset of machine learning. A solid understanding of machine learning fundamentals is crucial, when it comes to designing efficient deep learning models. This chapter aims to provide an overview of machine learning, its common pitfalls and best practices for designing and training models.

1.1 Overview

Machine learning is a field of Computer Science that allows computers to learn without explicitly programming them. The most widely quoted definition of machine learning is given by Tom Mitchel [13].

Definition 1.1.1 (Machine Learning)**.** A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E

In case of image classification problem, experience E can be viewed as the act of presenting the machine learning algorithm pairs of images and their corresponding labels of objects present in each image. Task T is simply guessing which object is present in a given image and performance measure P is the probability that the guessed object is actually present in that image.

Designing an algorithm for these types of problems in a way we design algorithms for classical problems, like sorting a list, is not feasible. This is because images of objects suffer from viewpoint and scale variation, deformation, occlusion and illumination condition to name a few. Instead, a data-driven approach has to be leveraged, where an algorithm (model) learns from looking at examples, hoping it will generalize well-enough to maintain its accuracy even if it is presented with images it has never seen before.

Machine learning systems are commonly classified into two categories based on whether learning feedback is given to the learning system.

- **Supervised learning** - the system is given a set of N training samples in form of $\{(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}), \ldots, (\boldsymbol{x}^{(N)}, \boldsymbol{y}^{(N)})\}$, where each $\boldsymbol{x}^{(i)}$ is a feature vector and $\boldsymbol{y}^{(i)}$ its corresponding label. Learning algorithm explores the hypothesis space $\boldsymbol{G}$ to find a function $g : X \mapsto Y$, where X is the input space ($\boldsymbol{x}^{(i)} \in X$) and Y is the output space ($\boldsymbol{y}^{(i)} \in Y$). The function g is found using a scoring function $f : X \times Y \mapsto \mathbb{R}$, such that $g(\boldsymbol{x}) = \arg\max_y f(\boldsymbol{x}, \mathbf{y})$
- **Unsupervised learning** - the system is given a set of N training samples in form of $\{\boldsymbol{x_1}, \ldots, \boldsymbol{x_N}\}$ and is asked to retrieve some latent structure of the data. This is a much more challenging problem than supervised learning as the goal is not clearly defined. Unsupervised learning is usually further divided into clustering (k-means), anomaly detection (One-class SVMs) and latent-space analysis (VAEs [14], GANs [15])

Supervised learning can also be divided into classification and regression. Classification corresponds to a discrete output space Y, while regression allows for a continuous space. Examples of classification include spam detection, scene recognition and face detection. On the other hand, object localization, human pose estimation and speech synthesis are instances of regression. This thesis focuses on camera relocalization from monocular images, which is another example of supervised regression.

1.2 Common pitfalls and best practices

The field of machine learning has proven to be very successful across a number of applications. From fraud detection systems [16], through recommender systems [17, 18] and applications in medicine [19] to classical problems in computer vision [20–22]. However, machine learning systems suffer from many pitfalls.

1.2.1 Interpretability

Interpretability is usually a concern, due to the black-box nature of machine learning systems. The problem frequently arises when these systems are deployed in production and report a significant decrease in performance. It is often difficult to give an exact explanation why the system behaves in that manner. There is a lot of effort in the machine learning research community to gain a better understanding about the inner workings of these systems [23, 24]. In section 2.3.3, we explore various techniques for understanding and visualizing Convolutional Networks.

1.2.2 No free lunch theorem

The famous no free lunch theorem for machine learning [25] states that every classification algorithm has the same out-of-sample error rate, if we average it over all possible data-generating distributions. It means, that there is no single machine learning algorithm that is universally better than any other.

Fortunately, the theorem holds only if we average over all possible distributions. In real-world applications, we make assumptions about the data generating distributions, therefore we can design algorithms, which perform exceedingly well. It is important to remember, that the goal of machine learning is not to pursue a universal algorithm, but to understand what types of machine learning algorithms achieve great results with data-generating distributions we care about.

1.2.3 Underfitting, overfitting and model's capacity

The difference between machine learning and optimization is that the resulting model has to perform well on both training and test datasets. These datasets must satisfy the i.i.d. assumptions, meaning that the samples from each dataset are independent from each other and that the training and test sets are identically distributed. A well performing model will have a low training error and the gap between training (in-sample) and test (out-of-sample) error will also be small. These two components are crucial in machine learning and have a direct correspondence with underfitting and overfitting.

Overfitting takes places, when the gap between training and test error is substantial. It is often explained that an overfit model merely learned the training data by heart in some convoluted way. On the other hand, **underfitting** occurs, when the model struggles to obtain low training error. These two concepts are directly related to model's capacity and controlling it can help avoiding these two problems.

Model's capacity is usually described as the ability to fit a set of functions. If the capacity is too low, then the model cannot fit a function and therefore underfits. One example of such scenario is if the training data represents some non-linear relationship, but we are trying to learn it with a model whose hypothesis space H is a set of all linear functions (e.g. linear regression). Furthermore, if the model's capacity is too high, then there is little chance for choosing a solution that generalizes well and the model overfits. Figure 1.1 depicts a common relationship between capacity and error. The left-hand side of the figure illustrates the underfitting zone, where the model capacity is too low. As the capacity increases, the training error decreases, but the generalization gap increases. Finding an optimal capacity (one that balances training and generalization error) is a difficult task.

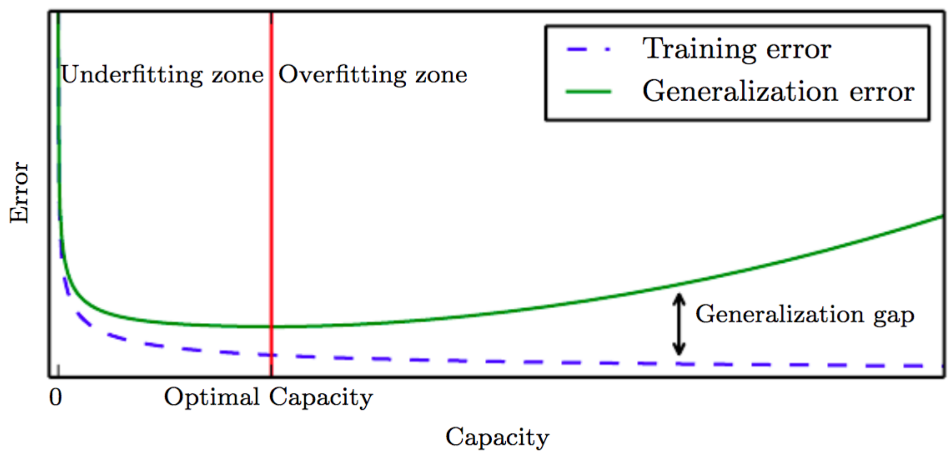


FIGURE 1.1: Relationship between model capacity and error rate [26, Figure 5.3]

Regularization is a technique that can prevent overfitting even if model's capacity is too large. The learning algorithm can favor one solution from another, despite both of them being eligible. The most widely used regularization is **weight decay** described in section 2.2.5.

1.2.4 Hyperparameter optimization and validation sets

Nearly all machine learning algorithms have **hyperparameters**. These are parameters (knobs) that control the behavior of the learning algorithm, but are not modified or learned by the learning algorithm itself. A good setting of hyperparameters is crucial to model's performance, while a poor one can hinder the ability to learn. Hyperparameters require tuning, which is expensive as models usually take days or even weeks to train. This process is often described as "black-art", as it requires expert experience, involves unwritten rules of thumb, along with some brute-force search [27]. Some of the best-practices [28] for performing hyperparameter search are:

- **Random search over grid search**. Grid search is the usual method for finding hyperparameters. Given a finite set of possible values for each hyperparameter, we make a Cartesian product of these sets and validate the performance on each element. However, this approach suffers from curse of dimensionality as the number of hyperparameters increases and often becomes infeasible. Moreover, in the case where some hyperparameters are not as important as other, we effectively waste computational time by trying these settings.

  On the other hand, in **random search,** sampling from a joint distribution of all hyperparameters is performed n times. Fig 1.2 illustrates an example why random search is usually better than grid search. In both cases, 9 different hyperparameter settings were examined. Grid search fails due to an unimportant hyperparameter, which forces the other hyperparameter to be checked at only 3 different places instead of 9, therefore increasing the chance to miss important regions.

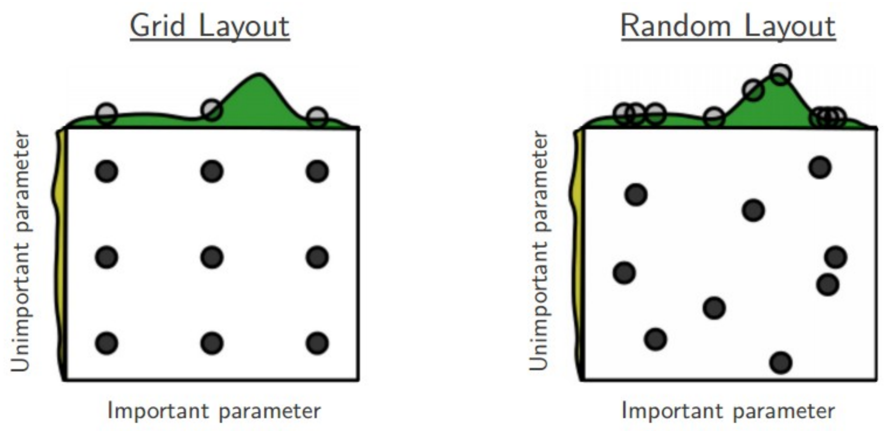


FIGURE 1.2: Grid search vs random search [29, Figure 1]

- **Choosing scale.** Some hyperparameters such as learning rate and weight decay (section 2.2.5, 2.2.6), should be searched on a log scale, meaning sampled from $c^{\mathcal{U}(a,b)}$, where c is the exponent, and a and b are the boundaries of the uniform distribution. Other hyperparameter can be sampled from the uniform distribution directly e.g. dropout.

- **From coarse to fine.** In reality, it is best to first search in a coarse range and narrow down the range as best results turn up. Being careful with best values at the boundaries is also important.

- **Bayesian Optimization.** There exists an entire area of research dedicated to finding algorithms that traverse the space of hyperparameter more efficiently. Spearmint [30] and SMAC [31] are examples, but it is still an active area of research with many unsolved problems such as parallelization and meta-learning [32].

It is crucial to realize that we cannot tweak model's hyperparameters and evaluate the performance on test set. Doing so may lead to an overly optimistic generalization error and if the model was to be deployed in production, there would be a significant performance reduction. This is due to the fact, we effectively used test set as training set and thus, overfitted the test set. In order to solve this problem, we need an additional set called: **validation set**, which is obtained by splitting the training set into training and validation sets. We can now tune the hyperparameters by evaluating the performance on validation set and once we find the best setting, train on both validation and training set and evaluate a *single* time on the test set.

When the training set is too small to afford splitting it, **cross-validation** is an option. It splits the training set into k equal fold, uses $k - 1$ for training and 1 for validation. This is done k times such that every single fold becomes the validation fold and the rest is used for training. The performance is averaged over all fold selections. In reality, people avoid using k-fold cross-validation as it increases computational time by a factor of k. The ratio of training to validation split generally varies from 50% to 90%. However, it is safer to use a larger validation set if the number of hyperparameters to tweak is large.

1.2.5 Data leakage

Data leakage is a serious problem in machine learning and is considered ”one of the top ten data mining mistakes” [33, 34]. It takes place when external information (not part of training set) is used to train the model and leads to overly optimistic or invalid models. In practice, this is often unintentional, subtle and indirect, thus making it very hard to detect. Some examples of data leakage are [35]: (1) leaking test data into the training data, (2) leaking knowledge about future into the past, (3) addition of data that is not given in model's operational environment and (4) leaking ground truth data into the test set.

Chapter 2

Deep Learning

Traditional machine learning algorithms such as decision trees [36], support vector machines [37] and k-nearest neighbors methods accomplish good results on a wide range of problems. However, they have failed on fundamental problems in Artificial Intelligence (AI), including speech and vision. The reason behind their poor performance is due to inherently high-dimensional structure of the data in which it becomes exponentially more difficult to achieve generalization. This is known as curse of dimensionality.

The rise of deep learning was partially motivated by the lack of success in these difficult AI problems, where humans excel well, but computers fall short. Deep learning aims to solve this problem by focusing on representation learning. The goal is to allow the learning algorithm construct a structure, where concepts are expressed in terms of other simpler concepts. A primary example of this is CNN (short for Convolutional Neural Network), where the bottom concepts in hierarchy describe edges and corners, higher-order concepts can represent honeycomb or wheel-like structures and eventually the top-level concepts might represent faces or animals. CNNs are explained in detail in section 2.3.

2.1 Manifold hypothesis

When dealing with high-dimensional spaces $\mathbb{R}^n$ such as images or sound waves, one might become pessimistic that the learning algorithm can actually learn functions where samples are distributed across the entire space. However, **manifold hypothesis** states that the data exists on low-dimensional manifold embedded in a high-dimensional space. This assumption is at least partially correct for images, text and sound [26]. Figure 2.1 illustrates a learned manifold from MNIST dataset using Variational Auto Encoder (VAE) [14]. It shows that hand-written characters lie on a 2-D manifold.

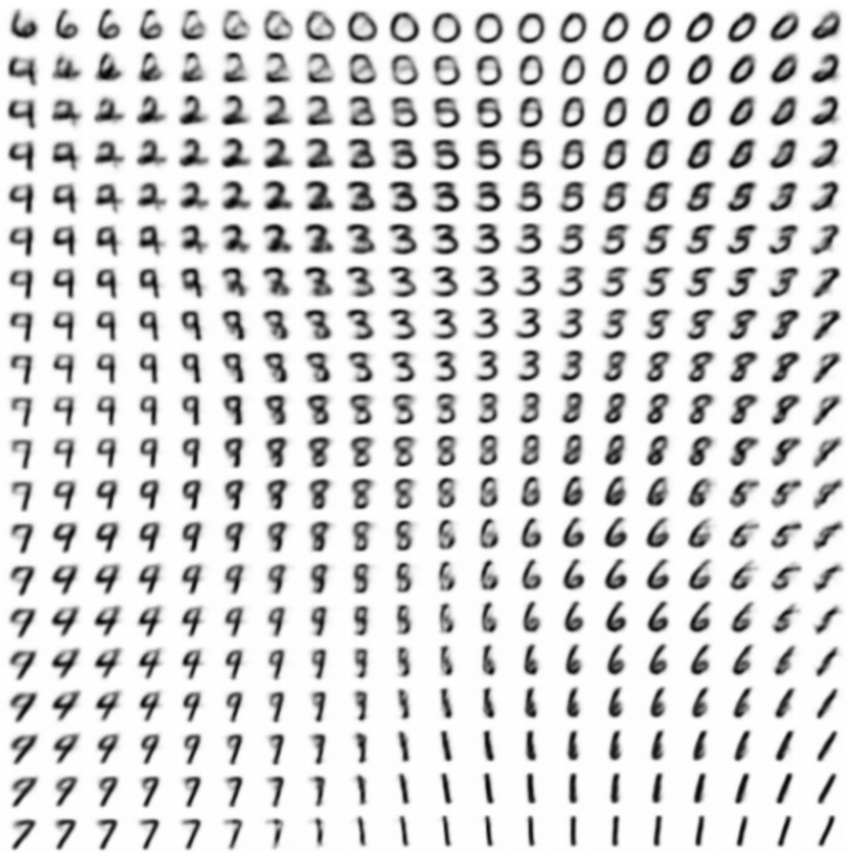


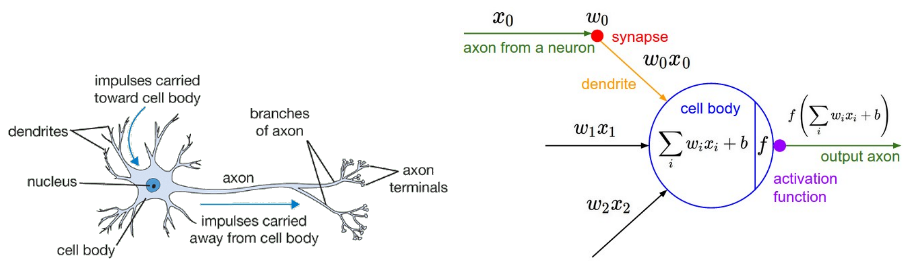
FIGURE 2.1: MNIST manifold [14, Figure 4]

2.2 Neural Networks

The field of Neural Networks, also called Artificial Neural Networks (ANNs) in literature, has be influenced by the objective of modeling biological neural systems.

2.2.1 Perceptron

Perceptron is the simplest neural network, which consists of a single neuron. Figure 2.2 depicts a biological neuron on the left and its mathematical model on the right. A neuron is a basic computational unit, which receives signals from its **dendrites** (inputs) and generates an output signal ahead its **axon**. In a human brain with approximately 86 billion neurons, axons are connected to dendrites of other nearby neurons through synapses. The mathematical model is an overly simplified model of a real neuron, in which signals traveling along axons (x_i) interact with the dendrites of nearby neurons. This interaction is multiplicative and based on synaptic strength (w_i), such that signal entering the cell body from each axon is in form $w_i x_i$. The signals are summed up and if a threshold is reached, then the neuron fires a signal along its axon. In our mathematical model, using the **code rate** interpretation, the firing rate represents the frequency at which neuron fires and it is modeled using **activation function** f. The weights $\mathbf{w}$ are parameters, which are obtained from training the perceptron.



(A) Biological neuron (B) Mathematical model

FIGURE 2.2: Comparison of biological and mathematical neurons

Definition 2.2.1 (Perceptron)**.** Given n number of inputs in vector form $\boldsymbol{x} = (x_1, x_2, \dots, x_n) \in \mathbb{R}$, weights $\boldsymbol{w} = (w_1, w_2, \dots, w_n) \in \mathbb{R}$, bias $b \in \mathbb{R}$ and an activation function f. Perceptron is a function $g : \mathbb{R}^n \mapsto \mathbb{R}$, such that:

$$g(\mathbf{x}) = f(\sum w_i x_i + b) \tag{2.1}$$

2.2.2 Multilayer perceptron

Multilayer perceptron (MLP) is a generalization of a perceptron, where multiple neurons are combined together to form layers and those layers are stacked on top of each other. Figure 2.3 shows a multilayer perceptron with 2 hidden layers. The input layer does not perform any processing and simply forwards signals to the first hidden layer. Each neuron's output signal is forwarded to each neuron in the next layer. It is important to notice, that hidden layers cannot have linear activation functions, because it does not increase model's capacity. Missing connection between two neurons can be simply modeled with a weight $w = 0$. Layer computation can be represented as matrix-vector product ($W_l x$) followed by an element-wise application of activation function. This operation is very fast and can be easily computed on a GPU, which is why deep learning has seen such a rapid expansion. Layers inside a MLP are often called fully-connected layers (FC) or dense layers.

Multilayer perceptrons (with at least 1 hidden layer) are **universal function approximators**, meaning that for any continuous function $f(x)$ on compact subsets of $x \in \mathbb{R}^n$ and some $\epsilon > 0$, there exist a function $g(x)$ such that: $\forall_x, \|f(x) - g(x)\| < \epsilon$, regardless of the choice of non-linear activation function [38]. This signifies that a simple MLP can, in theory, learn any function given appropriate parameters, but usually 2 or 3-layer networks are utilized as learning becomes easier.

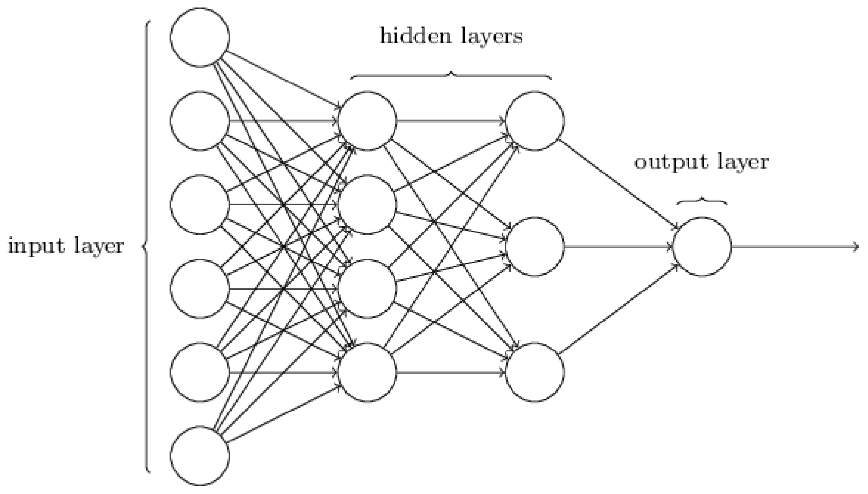


Figure 2.3: Multilayer perceptron with 2 hidden layers

2.2.3 Backpropagation

Backpropagation is the most common technique for training neural networks. The analogy behind backpropagation is that of blindfolded hiker, who wants to reach the bottom of the valley. However, the valley is high-dimensional and at any given time the hiker can only feel, underneath his feet, the steepness of the slope in all directions. He has to be careful not to take too short steps as it will take him long to reach the bottom and not too long steps as he might overstep and miss an important path.

Formally backpropagation is defined as follows. Given the input training data $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(N)}\}$ with its corresponding ground truth labels $\{\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, \ldots, \boldsymbol{y}^{(N)}\}$, we want to obtain the output of the network (forward pass), such that: $\hat{\boldsymbol{y}}^{(i)} = f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})$ where f is the function computed by the network using model parameters $\boldsymbol{\theta}$. Afterwards, we compute the error between the predicted output and the ground truth data and finally backpropagate this error through the network to improve the parameters (backward pass). The error function $\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$, where $\hat{\boldsymbol{y}}$ is the predicted output and $\boldsymbol{y}$ is the ground truth, is called **loss function** in literature and is problem specific. Classification problems frequently use cross-entropy function, while regression problems adopt mean-squared error (**L2**) or mean-absolute error (**L1**).

Backpropagation of the error involves computing gradients of the loss function with respect to each model parameter, $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)})$. In order to obtain these partial derivatives, recursive application of the chain-rule [1] is employed [28]. Once the gradients are retrieved, the update step is performed:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\alpha}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}_t} \mathcal{L}(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)}) \tag{2.2}$$

where $\boldsymbol{\theta}_t$ are all parameters of the network at iteration t and α is the learning rate. It considers all samples from the training dataset, which can be computationally expensive as present datasets tend to have millions of samples. Equation 2.2 is the simplest update, known as vanilla update, which moves the parameters closer to a local minima of the loss function, decreasing the training error and consequently test error. In order to train the network, we have to repeat this step until some stopping criterion. In section 2.2.6, we describe a general algorithm for training neural networks as well as more sophisticated update rules that yield far superior performance.

It is important to remember, that all operations inside neural network have to be fully-differentiable for backpropagation to make sense. In addition, deriving the gradient formulas for network can be tedious and error-prone. Fortunately, there exist multiple libraries, which can do it automatically by inferring them from the computational graph [39–42].

[1] In practice neural networks are represented as computational graphs, where nodes denote operations and edges data flowing through it. Therefore, the chain-rule ($\frac{\partial f}{\partial g} = \frac{\partial f}{\partial h}\frac{\partial h}{\partial g}$) makes it easy to compute gradients of the output w.r.t to any parameter contained in the nodes of the graph.

2.2.4 Activation functions

Activation functions provide non-linearity to the model, which gives it to power to express complex functions. They all must take a single number and perform a fixed (differentiable) mathematical operation. Different activation functions illustrated in figure 2.4 have been proposed over the years:

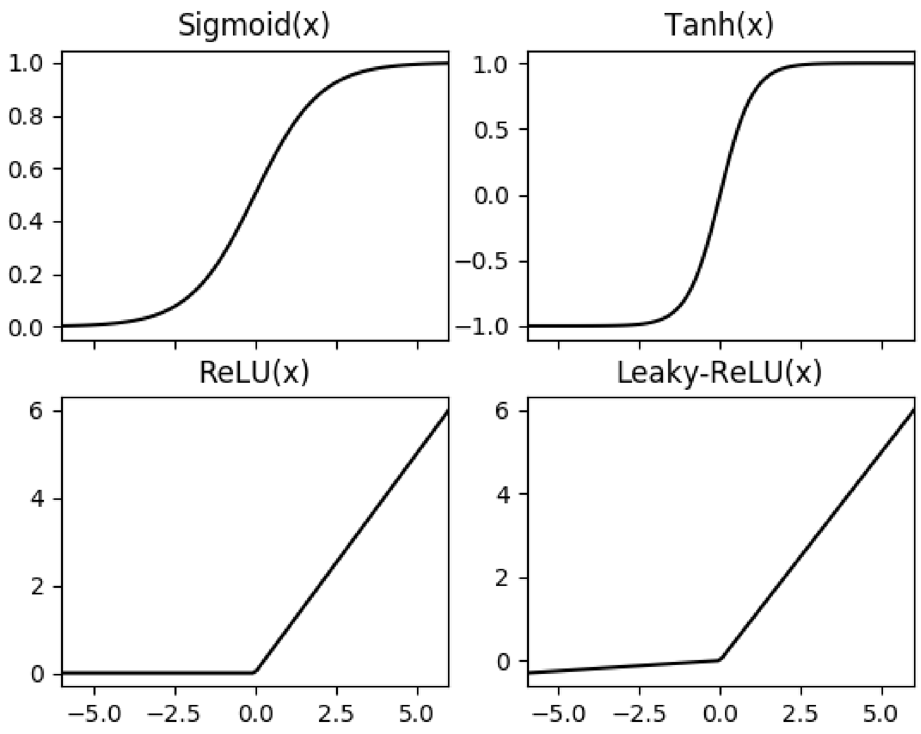


Figure 2.4: 4 different activation functions: sigmoid, tanh, ReLu and Leaky ReLU

- **Sigmoid** is an old activation function defined as, $\sigma(x) = \frac{1}{1+e^{-x}}$, which squashes values into range between 0 and 1. Large negative numbers become 0 and large positive numbers become 1. It was widely used, because of its firing neuron interpretation, where 0 stands for not firing at all and 1 is assumed as maximum frequency. However, it has two considerable disadvantages:
  - *saturation* - the output of sigmoid saturates at both tails (0 or 1) causing the gradient to be almost zero. In this case, the local gradient will effectively kill the gradient flowing through that neuron and block the error signal from going further back. This is widely known as the vanishing gradient problem [43].

  - *output is not zero-centered* - sigmoid activation will produce only positive values, making the gradient of weights **w** all positive or all negative, thus prohibiting from increasing one weight and decreasing another in the same neuron. This will produce zig-zagging dynamics in the weight updates. Nevertheless, in practice gradient is computed over a batch of data, where variable signs can appear, making it less of a problem than saturated activation. [28]

- **Hyperbolic tangent (tanh)** is a slight improvement over sigmoid function as the values are now zero-centered. It can actually be defined in terms of sigmoid: $tanh(x) = 2\sigma(2x) - 1$. However, it still suffers from the saturation problem.

- **Rectified linear unit (ReLU)** computes function $f(x) = max(x, 0)$, which is a very inexpensive operation to perform compared to other activation functions. It was shown by Krizhevsky et al [20] to give $6x$ speed-up in convergence. One big drawback ReLUs have is called "dying ReLU" problem, where large gradients can inadvertently cause neurons to only produce zeros and never activate again. This can be mitigated by carefully setting the learning rate.

- **Leaky ReLU** attempts to fix the "dying ReLU" problem by making the function have a small negative slope for $x < 0$ (see figure 2.4). However, this approach and other complex activation functions are still a very active area of research and the results are not always consistent [44].

2.2.5 Regularization

As mentioned in section 1.2.3, regularization is an important concept of controlling model's capacity in order to reduce chance of overfitting. Possibly the most widely used method of regularization is **weight decay** of which there exist two types:

- **L2 regularization** is achieved by penalizing the square magnitude of all parameters directly in the loss function. Term $\frac{1}{2}\lambda w^2$ is added for each parameter w, where λ is the regularization strength hyperparameter. The factor $\frac{1}{2}$ appears, because the gradient of the penalizing term becomes λw instead of $2\lambda w$. The intuition behind this method is that diffused weight matrices are preferred over sparse, encouraging the network to use all of its input instead of focusing on a small subset.

- **L1 regularization** follows the same methodology as L2 regularization, but differs in penalizing term, which turns into $\lambda\|w\|$. This technique affects weights differently compared to L2 regularization, because it leads to very small weights, essentially performing feature selection and ignoring noisy inputs. In practice, L1 performance is usually inferior to L2.

One of the most important regularization techniques that has been recently developed is certainly **dropout** [45]. It is a surprisingly simple method that complements weight decay. During training phase at each iteration randomly selected neurons are dropped along with their connections. The hyperparameter p is the probability, whether a neuron should be kept. Dropping a neuron simply means that we set it to zero.

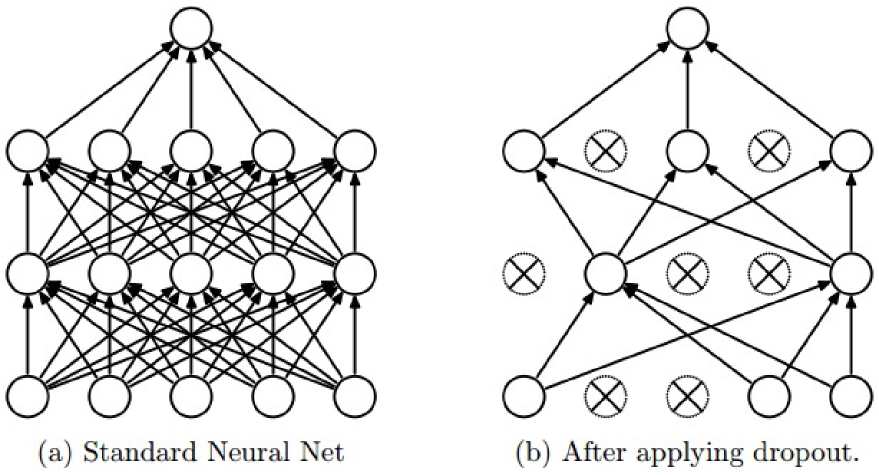


FIGURE 2.5: Dropout applied to a 2-layer multilayer perceptron [45, Figure 1]

Figure 2.5 depicts an example of a 2-layer MLP with and without dropout. The right-hand side of the figure shows that a dropout may be interpreted as sampling a smaller neural network from the full neural network, where merely parameters of the sampled network are updated. During test-time dropout is not applied, but instead layers where dropout was applied have to be scaled by p in order to have the same outputs in expectation. Moreover, at test-time all neurons are active, effectively performing an average over all sampled networks and form a model ensemble. Finally, dropout prevents overfitting by forcing the network to learn multiple independent representations and decreases co-adaptation of neurons.

2.2.6 Optimization for training neural networks

In section 2.2.3, we described a general backpropagation method, where the update step (equation 2.2) used the full training dataset, commonly referred to as **gradient descent (GD)**. In practice, datasets are too large to use this method. Hence, a common approach is to compute the gradient only over a randomly selected **batch** of training data. These methods are called **stochastic gradient descent (SGD)** or **mini-batch gradient descent (MGD)** in literature. There are multiple advantages resulting from the use of such stochastic approximation [46]:

- **faster convergence** - gradient descent has to examine the entire training dataset to update parameters $\boldsymbol{\theta}$ a single time, while stochastic gradient descent can make progress much more often.
- **escaping local minimum** - gradient descent will always converge to a local minimum of the loss function. On the other hand, each step in SGD will be a noisy estimate of GD, allowing it to escape these situations.

Algorithm 1 Stochastic Gradient Descent

```
Require: Training dataset $T = \{(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}), (\boldsymbol{x}^{(2)}, \boldsymbol{y}^{(2)}), \dots, (\boldsymbol{x}^{(N)}, \boldsymbol{y}^{(N)})\}$
Require: Neural network as function $f(\boldsymbol{x}; \boldsymbol{\theta}_t)$ with $\boldsymbol{\theta}_0$ as initial parameters
Require: Loss function $\mathcal{L}(\boldsymbol{x}, \boldsymbol{y})$
Require: Learning rate $\alpha$ and batch-size $m$
1: $t \Leftarrow 1$
2: while stopping criterion not met do
3:     Shuffle $T$
4:     $i \Leftarrow 1$
5:     while $i \leq N$ do
6:         Compute network output for $j = i, \dots, max(i + m - 1, N)$: $\hat{\boldsymbol{y}}^{(j)} = f(\boldsymbol{x}^{(j)}; \boldsymbol{\theta}_t)$
7:         Compute gradient estimate: $\hat{\boldsymbol{g}} \Leftarrow \frac{1}{m} \sum^{j} \nabla_{\boldsymbol{\theta}_t} \mathcal{L}(\hat{\boldsymbol{y}}^{(j)}, \boldsymbol{y}^{(j)})$
8:         Update network parameters: $\boldsymbol{\theta}_t \Leftarrow \boldsymbol{\theta}_{t-1} - \alpha \hat{\boldsymbol{g}}$
9:         $t \Leftarrow t + 1$
10:        $i \Leftarrow i + m$
11:    end while
12: end while
```

Algorithm 1 shows standard operation of SGD. Given a training set T with N samples, initial network parameters $\boldsymbol{\theta}_0$, learning rate α and batch-size m, we shuffle the dataset and yield batches used to update the network parameters (weights) using backpropagation. Lines (5-11) describe an **epoch**, during which the network sees the entire training data once. The stopping criterion from line (2) is usually number of epochs, or sometimes a technique called **early stopping**[2]. The algorithm does not specify how the parameters should be initialized, which is an important problem and also an active area of research. We describe **weight initialization** in detail in section 2.2.7. Lastly, line (8) shows the update step with static learning rate α. There is an abundance of update schemes with their respective advantages and disadvantages. They can be classified into two main categories: **global methods**, i.e. those that manipulate learning rate globally and **adaptive methods**, which manipulate learning rate in per-parameter fashion.

SGD with momentum is an example of a global method and an improvement over vanilla SGD update rule, which almost always results in faster and better convergence [47]. The idea is to look at the optimization from physics perspective, where we are simulating a particle roll on the landscape. In vanilla SGD, the gradient directly influences the position of this particle, but in momentum approach the gradient directly influences the velocity of the particle, which then acts on its position. The momentum update formula is the following:

$$\begin{aligned} \boldsymbol{v}_t &= \mu \boldsymbol{v}_{t-1} - \alpha \hat{\boldsymbol{g}} \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} + \boldsymbol{v}_t \end{aligned} \tag{2.3}$$

In equation 2.3, μ is a hyperparameter, usually ranging from 0.5 to 0.99, which is a damping factor. In cases, where the slope of the landscape is very steep and the particle starts oscillating around a local minimum, the damping factor acts a friction coefficient, thus making it gradually lose its kinetic energy and converge. On the other hand, if the landscape is very shallow in a given direction, then velocity v_t gets built up allowing it to converge faster along those shallow dimensions.

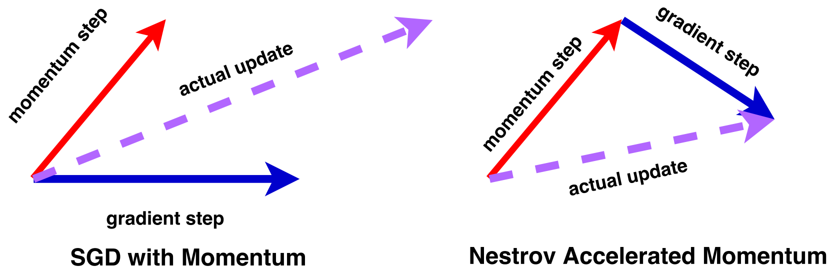


Figure 2.6: SGD with Momentum and NAG

[2]**Early stopping** is another form of regularization, which avoids overfitting by monitoring generalization error during training and stopping the training phase when the model starts overfitting.

Nesterov Accelerated Gradient (NAG) further improves upon SGD with momentum and in practice works slightly better, while also benefiting from stronger theoretical convergence guarantees [48]. The essence of this approach is to evaluate the gradient at position, where term μv_{t-1} would take us, instead of evaluating it at the current position. Figure 2.6 shows the difference between standard SGD with momentum and NAG. Nesterov's formula can be written as:

$$\begin{aligned} \boldsymbol{v}_t &= \mu \boldsymbol{v}_{t-1} - \alpha \nabla f(\boldsymbol{\theta}_{t-1} + \mu \boldsymbol{v}_{t-1}) \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} + \boldsymbol{v}_t \end{aligned} \tag{2.4}$$

where for simplicity $\nabla f(\boldsymbol{\theta}_{t-1} + \mu \boldsymbol{v}_{t-1})$ refers to gradients of loss function on batch of data w.r.t model parameters at the look-ahead step, $\boldsymbol{\theta}_{t-1} + \mu \boldsymbol{v}_{t-1}$. However, it is more desirable to express the gradient w.r.t. previous momentum update, which can be achieved using variable transform, $\boldsymbol{\phi}_{t-1} = \boldsymbol{\theta}_{t-1} + \mu \boldsymbol{v}_{t-1}$. The modified NAG formula is the following:

$$\begin{aligned} \boldsymbol{v}_t &= \mu \boldsymbol{v}_{t-1} - \alpha \nabla f(\boldsymbol{\phi}_{t-1}) \\ \boldsymbol{\phi}_t &= \boldsymbol{\phi}_{t-1} - \mu \boldsymbol{v}_{t-1} + (1+\mu) \boldsymbol{v}_t \end{aligned} \tag{2.5}$$

All previously described global methods used static learning rate α throughout the duration of the training phase. It is usually beneficial to **anneal the learning rate** over time. The physics interpretation of this technique is that initially the system contains too much kinetic energy and the particle (parameter vector) is unable to settle down into deep and narrow ravines of the loss function. Knowing how to decrease the learning rate can be a subtle task. If we reduce it too fast, then our system will cool down rapidly before reaching its best possible position. Furthermore, reducing the learning rate too slowly will cause the same problems as not reducing it at all. There are three main methods of decaying the learning rate:

- **step decay** reduces the learning rate by a constant factor every few epochs. It can be expressed with the following formula, where β is the decreasing factor, e is the number of epochs between consecutive decays and α_0 is the initial learning rate:

$$\alpha(t) = \alpha_0 \beta^{\lfloor \frac{t}{e} \rfloor} \tag{2.6}$$

- **reduce on plateau**, similarly to step decay, reduces the learning rate by a constant factor, but only when a monitored value (validation loss or other accuracy metric) stops improving.
- **exponential decay** is another form of learning rate decay, which can be described using the following formula:
$$\alpha(t) = \alpha_0 e^{-kt} \tag{2.7}$$

Thus far, we have shown only global methods for the update step. Those approaches act equally on every single parameter and require expensive finetuning of the initial learning rate and decay hyperparameters. **Adaptive methods** are a solution to this problem, where parameters are updated on per-parameter basis. Those methods usually do not require as much expensive finetuning as they are well-behaved across a wider range of hyperparameters. Some of the methods, which were empirically proven to improve training are:

- **AdaGrad (Adaptive Gradient)** was one of the first per-parameter adaptive methods originally developed in the convex optimization literature and ported over to neural networks [49]. The core idea is to adapt learning rate to each parameter, such that parameters receiving small gradients will have their effective learning rate increases and conversely parameters receiving high gradients will have their effective learning rate decreased. Equation 2.8 shows the formulation of AdaGrad, where c is the cache of accumulated gradients over time, $\hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}}$ is an element-wise multiplication and the division in second line is also performed element-wise. Lastly, the ϵ (usually $1e-8$) is added to each element of the square-root of cache in order to avoid dividing by zero.
$$\begin{aligned} \boldsymbol{c}_t &= \boldsymbol{c}_{t-1} + \hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}} \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} - \alpha \frac{\hat{\boldsymbol{g}}}{\sqrt{\boldsymbol{c}_t} + \epsilon} \end{aligned} \tag{2.8}$$
AdaGrad proved to be very successful [50] and robust for sparse data, such as GloVe word embeddings [51]. However, the problem with it is that the cache c will eventually get very large and the learning will stop as it acts as a decaying factor.
- **RMSProp** (short for Root Mean Square Propagation) tries to combat the problem of AdaGrad stopping to early, by making the cache variable decay using a running

exponential average. Equation 2.9 shows the update step formula, where γ is the decay rate (usually set to $0.9, 0.99$ or 0.999). The first line computes the moving average of the squared gradients, which prevents the learning rate from decreasing aggressively.

$$\begin{aligned} \boldsymbol{c}_t &= \gamma \boldsymbol{c}_{t-1} + (1-\gamma)\hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}} \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} - \alpha \frac{\hat{\boldsymbol{g}}}{\sqrt{\boldsymbol{c}_t} + \epsilon} \end{aligned} \tag{2.9}$$

- **Adam** (stands for Adaptive Momentum Estimation [52]) is yet another improvement of adaptive methods, which further builds upon RMSProp, by combining it with momentum. Equation 2.10 shows the formula of Adam with bias correction. The first line calculates the momentum update, expressed as an exponential moving average of gradients, while the second line is exactly the same as in RMSProp. The third and fourth lines perform bias correction, which is needed for the first few updates as $\boldsymbol{m}$ and $\boldsymbol{v}$ are both biased at $\mathbf{0}$ due to initialization. Finally, the last line performs the actual update by taking both momentum $\hat{\boldsymbol{m}}$, which helps stabilize the noisy gradient and scaling factor $\hat{\boldsymbol{v}}$, which makes sure that parameters are updated individually as described in AdaGrad. Adam is the standard choice in current literature as it enjoys the best results without the need of heavy finetuning. Hyperparameters β_1, β_2 and ϵ are commonly initialized to 0.9, 0.999 and $1e-8$ respectively, while learning rate α requires some finetuning and is generally problem specific.

$$\begin{aligned} \boldsymbol{m}_t &= \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\hat{\boldsymbol{g}} \\ \boldsymbol{v}_t &= \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}} \\ \hat{\boldsymbol{m}}_t &= \frac{\boldsymbol{m}_t}{1-\beta_1^t} \\ \hat{\boldsymbol{v}}_t &= \frac{\boldsymbol{v}_t}{1-\beta_2^t} \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} - \alpha \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon} \end{aligned} \tag{2.10}$$

All methods described thus far are first-order methods as they only require the first derivative. There exist a whole subfield of neural network optimization using **second-order methods**. Second-order methods lead to much faster convergence due to curvature calculation and lack the need of hyperparameter tuning, e.g learning rate.. The parameter update is a Newton's method (2.12) derived from second-order Taylor expansion (2.11):

$$\mathcal{L}(\boldsymbol{\theta}) \approx \mathcal{L}(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\intercal \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\intercal \boldsymbol{H} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\intercal \tag{2.11}$$

$$\boldsymbol{\theta}^* = \boldsymbol{\theta} - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_0) \tag{2.12}$$

The main drawback of second-order methods is the need to compute the inverse Hessian matrix $\boldsymbol{H}^{-1}$. It becomes intractable as the network can easily have more than a million parameters, since the complexity to compute it is $O(n^3)$. The most popular method is called BFGS [53], which is a quasi-newton method and only approximates the inverse Hessian with time complexity $O(n^2)$. However, BFGS still suffers from high space complexity as its often impossible to even store the Hessian matrix in memory. L-BFGS [54] counters this by not forming and storing the full inverse Hessian. Another problem of second-order methods is the fact that they work very well in full batch, where each update has to scan through an entire training set, but they fail to work in mini-batch setting. Currently, applying L-BFGS in stochastic mini-batch setting is an active area of research. Lastly, L-BFGS has found its application in style transfer, which combines content of an image with a style of another image using CNNs [55, 56].

2.2.7 Weight Initialization

Proper parameter (weight) initialization is a crucial aspect of training neural networks. For instance, if the network was initialized with zeros only, then all neurons in each layer would output the same value and gradient, leading to the same parameter updates and lack of asymmetry between neurons. Therefore, it is important to initialize parameters according to some random distribution, thus making the parameters unique at the beginning of training. One way of achieving this is through drawing parameters from standard uniform distribution scaled by some small factor. In case of deep neural networks, this approach can lead to vanishing gradient problem, where gradient signal falls to 0.

Main problem with randomly initializing weights is that the variance of the output at a given layer grows proportionally with the number of inputs. We can force the neurons to have output with unit variance by scaling the standard uniform distribution by $\frac{1}{\sqrt{n}}$. The derivation is the following; if we consider output of a single neuron y before activation function: $y = \sum_{i=1}^{n} w_i x_i$, then we can write its variance as:

$$\begin{aligned}\mathrm{Var}[y] &= \mathrm{Var}[\sum_{i=1}^{n} w_i x_i] = \sum_{i=1}^{n} \mathrm{Var}[w_i x_i] \\ &= \sum_{i=1}^{n} [\mathbf{E}[w_i]]^2 \mathrm{Var}[x_i] + \sum_{i=1}^{n} [\mathbf{E}[x_i]]^2 \mathrm{Var}[w_i] + \mathrm{Var}[x_i]\mathrm{Var}[w_i] \\ &= \sum_{i=1}^{n} \mathrm{Var}[w_i]\mathrm{Var}[x_i] = (n\mathrm{Var}[w])\mathrm{Var}[x]\end{aligned} \tag{2.13}$$

The first two steps of derivation 2.13 make use of variance properties. The third step assumes that inputs and weights are zero-centered, which will be the case if activations are linear or tanh, but ReLU units would have positive mean. Finally, the forth step assumes that all w_i and x_i are identically distributed. We can see that in order to make $\mathrm{Var}[y] = \mathrm{Var}[x]$, we have to have $\mathrm{Var}[w] = \frac{1}{n}$. Therefore, if we sample w from unit Gaussian distribution and scale it by $\frac{1}{\sqrt{n}}$, we get the desired variance, since $\mathrm{Var}[aX] = a^2\mathrm{Var}[X]$.

A more detailed analysis carried out by Glorot et al. in [57] led to common initialization scheme called **Xavier Initialization**. It shows that instead of scaling by $\frac{1}{\sqrt{n}}$, we should scale by $\sqrt{\frac{2}{n_{in}+n_{out}}}$, where n_{in} and n_{out} are number of inputs and outputs respectively of a given layer. However, for layers with ReLU activations there exists another initialization scheme called **He Initialization**, where we scale by $\sqrt{\frac{2}{n}}$, which gives better results and is currently recommended [44].

Batch Normalization is technique that alleviates most of the problems that come with bad weight initialization [58]. The idea is to force activation layers receive a unit gaussian throughout the network. It is possible, because normalization is a differentiable operation. Equation 2.14 shows the formulas for calculating output of the batch normalization layer. $\hat{x}_i$ is the normalized batch and y_i is the actual output of the batch norm layer, where γ and β are learnable parameters. The output y_i is scaled and shifted, because the activation functions might prefer the input to moved and scaled. It is important to notice, that the network has the capacity of learning the identity function and thus omit the batch normalization if it learns $\gamma = \sqrt{Var[x]}$ and $\beta = \mathbf{E}[x]$. Batch normalization not only is robust to weight initialization, but also allows for higher learning rates and improves the gradient flow throughout the network leading to improved results on common problems [59].

$$
\begin{aligned}
\mu_{\mathrm{B}} &= \frac{1}{m}\sum_{i=1}^{m} x_i \\
\sigma_{\mathrm{B}}^2 &= \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathrm{B}})^2 \\
\hat{x}_i &= \frac{x_i - \mu_{\mathrm{B}}}{\sqrt{\sigma_{\mathrm{B}}^2 + \epsilon}} \\
y_i &= \gamma \hat{x}_i + \beta
\end{aligned}
\tag{2.14}
$$

2.3 Convolutional Neural Networks

Convolutional Neural Networks have been originally developed by Yann LeCun at al. [60] in 1998. However, they remained relatively untouched until 2012, when Alex Krizhevsky at al. famously won the 2012 ILSVRC (ImageNet challenge [61]) using a convolutional neural network called *AlexNet* [20]. Ever since, CNNs have be successfully applied to a variety of classic computer vision problems completely disrupting the field [62–67].

The Multilayer perceptrons (MLPs) described thus far operate on N dimensional input vectors. However, for data which has a known grid-like topology, such as images or time-series data with samples taken at regular intervals, these types of network do not yield satisfying performance. Regular networks, which only have fully-connected layers (section 2.2.2) do not scale well to images. The reason behind it is that even a small image of size $32 \times 32 \times 3$ (such as those from CIFAR-10 dataset [68]) has effectively 3072 dimensions. Certainly, we would like to use images with higher resolution, but fully-connected layers with such input would quickly become intractable. This is the case, because the number of parameters in a fully-connected layer is $n_{in} * n_{out}$, where n_{in} and n_{out} are the input and output dimensions respectively. For instance, if an MLP with one hidden layer that has the number of units equal to the dimensionality of the input and outputs a single value was used to process a 200×200 RGB image, then it would require $(200 \times 200 \times 3)^2 + (200 \times 200 \times 3) = 1.44 * 10^{10}$ parameters, which translates to 57.6GB of required memory. Additionally, pixel spatiality is extremely important in images, for instance; pixels in a neighborhood will not differ much from each other. However, regular networks do not take advantage of the topological structure of images, because they have to be flattened before processing, thus losing their topological structure.

This section focuses on giving a brief overview of convolutional neural networks (2.3.1), current state-of-the-art architectures (2.3.2) and visualization and understanding of CNNs (2.3.3).

2.3.1 Overview

Regular neural networks process information encoded as vectors. On the other hand, convolutional neural networks (CNNs) take advantage of the grid-like structure of images and the information flowing through the network is represented using tensors, which are generalization of multi-dimensional arrays. A scalar value is a tensor with rank 0, while vectors and matrices are tensors with ranks 1 and 2 respectively. Many real-world objects can be represented as tensors, for example: RGB images with resolution $Height \times Weight \times 3$ are tensors with rank 3, videos composed of N frames have rank 4, where the tensor dimensions are: $N \times Height \times Weight \times 3$. Likewise, waveforms can be represented with rank 2 tensors of $N \times K$ dimensionality, where N is the number of samples and K is the dimensions of feature vector describing a single sample.

The name, convolutional neural networks suggests that those networks employ mathematical operation of convolution. This operation is linear and also fully differentiable, therefore can be used when training neural networks using backpropagation (2.2.3). Convolution of a 2D array such as a gray-scale image I and a 2D filter F is defined as following:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.15}$$

Convolution is applied at every location (i,j) of the input array and produces an output with smaller size, but padding can be applied in order to maintain the same size. Motivation behind applying convolutions in neural networks is, as previously argued, that individual pixels are not independent from the pixels surrounding it. Furthermore, features present in an image, such as edges and corners are independent from its location. Convolution measures the amount of known signal (filter F) present in an image I. We now describe the building blocks of convolutional neural networks.

Convolution Layers, sometimes referred to as Conv Layers, are the essence of every CNN. They operate on input tensors of dimensionality $x \times y \times d$. Each Conv Layer consists of n filters, where each filter has size $x_f \times y_f \times d$, such that $x \geq x_f$ and $y \geq y_f$. Filters are convolved over the input at different spatial locations. The distance between these locations is defined by two hyperparameters s_x and s_y called stride. If $s_x = s_y = 1$, then filters are applied at every single spatial location. Equation 2.15 describes the convolution operation for rank 2 tensors, however CNNs operate on tensors with an additional dimension d. Therefore, we need to expand the convolution to:

$$S(i,j,k) = (K * I)(i,j,k) = \sum_{m=1}^{x_f} \sum_{n=1}^{y_f} \sum_{l=1}^{d} I(i-m, j-n, k-l)K(m,n,l) \tag{2.16}$$

Equation 2.16 shows that convolution is essentially computing a dot product between the sliding filter and the input tensor at those selected (according to stride) spatial locations. We can notice, that the all filters have the depth dimension d same as the input tensor, but this not the case for spatial dimensions x and y. Therefore, the output of convolving the input with a single filter will have size $x_o \times y_o \times 1$, where $x_o < x$ and $y_o < y$. It is sometimes desirable not to reduce the spatial dimensions x and y. This can be achieved by padding the input, usually with zeros, such that $x_o = x$ and $y_o = y$.

$$x_o = \frac{x - x_f + 2p_x}{s_x} + 1 \tag{2.17}$$

$$y_o = \frac{y - y_f + 2p_y}{s_y} + 1 \tag{2.18}$$

Equations 2.17 and 2.18 show how calculate the output's spatial dimensions x_o, y_o, where p_x and p_y are the sizes of padding. It is important to notice, that we might select a stride s_x or s_y producing an output dimension that is not an integer. In this case, we consider such setting as invalid and do not allow it. Furthermore, if we solve the equations for p_x and p_y, when $x = x_o$ and $y = y_o$, we obtain:

$$p_x = \frac{x(s_x - 1) - s_x + x_f}{2} \tag{2.19}$$

$$p_y = \frac{y(s_y - 1) - s_y + y_f}{2} \tag{2.20}$$

From equations 2.19 and 2.20 we get the amount of padding required for maintaining the spatial dimensions. Typically, CNNs use either; *valid* padding, where no padding is applied or *same* padding, where enough padding is applied to maintain the spatial dimensions. Lastly, convolutional layers use n filters, where each filter is used independently producing $x_o \times y_o \times 1$ and all outputs are concatenated along the depth dimension forming final output of size $x_o \times y_o \times n$. Figure 2.7 depicts an example of a convolution

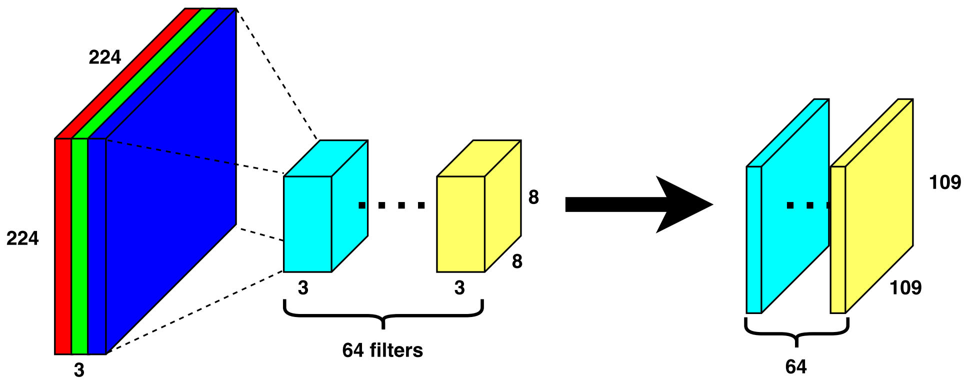


FIGURE 2.7: Example of a Convolution Layer applied on a RGB image

layer with sixty-four $8 \times 8 \times 3$ filters, strides $(s_x, s_y) = (2, 2)$ and *valid* padding applied over $224 \times 224 \times 3$ input, thus producing $109 \times 109 \times 64$ output tensor. The number of parameters in a convolution layer is $(x_f * y_f * d + 1) * n$, where 1 is the bias term added to every filter. Therefore a layer with configuration from figure 2.7 would have $64 * (8 * 8 * 3 + 1) = 12352$ parameters, which is a very small number compared to what a fully-connected layer would need in order to produce output of the same size.

Pooling Layers are another building blocks of Convolutional Neural Networks. Their goal is to reduce the spatial dimensionality of the tensor flowing through that layer. This is achieved by applying a reduction operation in spatial neighborhood. Similarly to Conv Layers, we slide a windows of size (w_x, w_y) with strides (s_x, s_y) through the input tensor and perform a function that returns a single value. Usually, it is max or average operation, naming the layers *MaxPool* and *AvgPool* respectively. All slices of the input tensor along the depth dimension are treated independently from each other. Figure 2.8 illustrates an example of Pooling Layer applied to a $4 \times 4 \times d$ tensor with a 2×2 pooling size and $(2, 2)$ strides producing a $2 \times 2 \times d$ tensor. More generally, the output of the pooling layer is calculated using equations 2.21:

$$x_o = \frac{x - x_w}{s_x} + 1, \; y_o = \frac{y - y_w}{s_y} + 1 \tag{2.21}$$

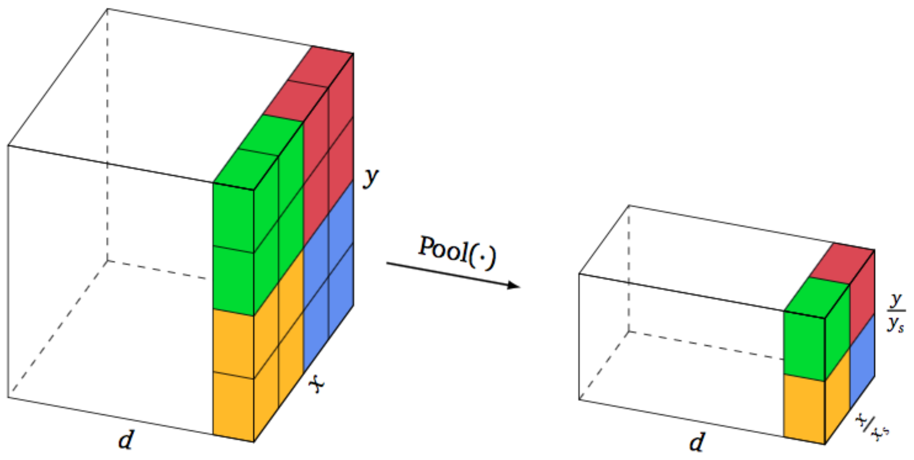


FIGURE 2.8: Example of a Pooling Layer [8, Figure 2.6]

2.3.2 Architectures

This section aims to explain common structure of Convolutional Neural Networks as well shows current state-of-the-art architectures. Figure 2.9 illustrates a basic structure of a CNN. Convolution layers are always followed immediately by ReLU activation. This is due to the fact that convolution is a linear operation, therefore stacking 2 convolution layers would be equal of having just one convolution layer. Those convolution layers most often employ *same* padding in order not to decrease the spatial dimensions too quickly. Multiple Convolution Layers with ReLU activation are stacked and followed by a Pooling layer, which reduces the spatial dimensions. These blocks of Conv Layers with Pooling are further stacked and form the core of a CNN, also known as a feature extractor. The understanding of what CNNs are computing is explained in detail in section 2.3.3. Finally, the core is followed by a stack of fully-connected layers with ReLU activations, while the last fully-connected layer commonly has linear activation. This architecture was originally developed for classification problems such as ImageNet challenge [61], but has since been proven to transfer to a variety of other problems [4, 69, 70]

A common example of architecture that employs the linear structure illustrated in 2.9 is **VGG16**. It was the runner-up in 2014 ImageNet challenge and showed the importance of the depth of the network [71]. The network has a highly homogeneous structure as it only uses Convolution Layers with filters of size 3×3 and Pooling Layers with 2×2 window size. One of the downsides of this model is the computational cost

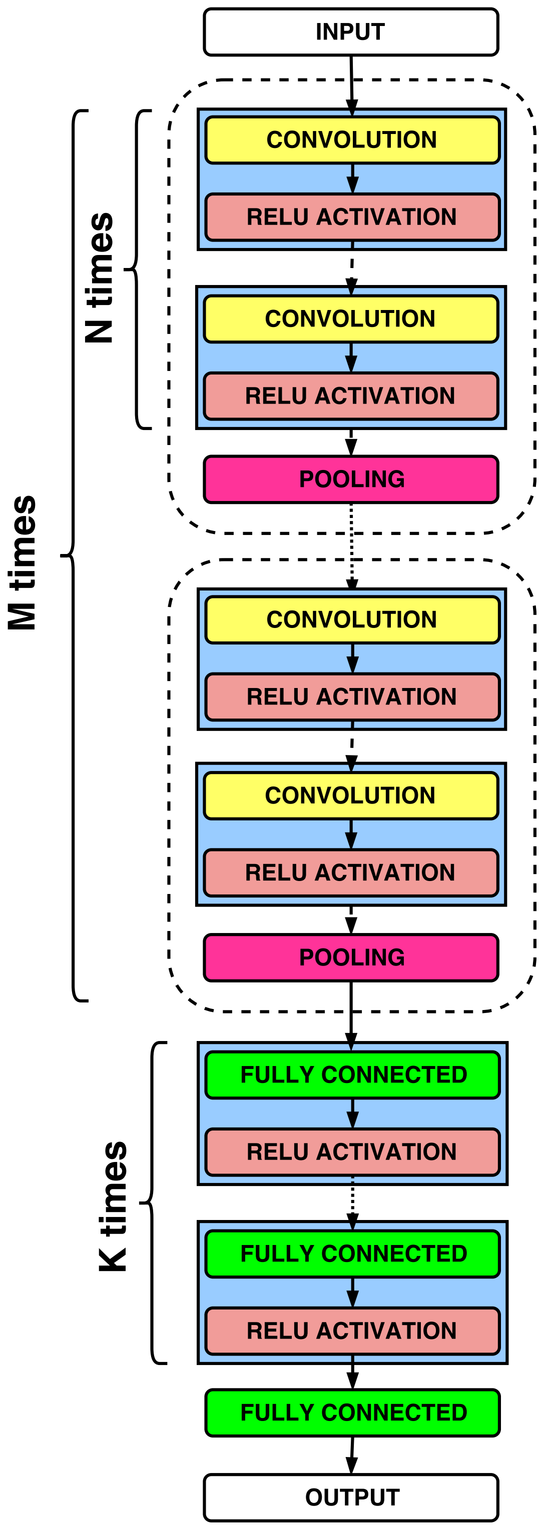


FIGURE 2.9: Common architecture of a Convolutional Neural Network

it comes with. It requires approximately 93MB of memory for forward pass (test-time) and twice that amount for backward pass (training-time). The model itself contains 138 million parameters, which translates to 552MB of storage required, thus rendering it unusable for mobile devices. Figure 2.10 illustrates the architecture of VGG and shows how the spatial and depth dimensions change as the image flows through the network.

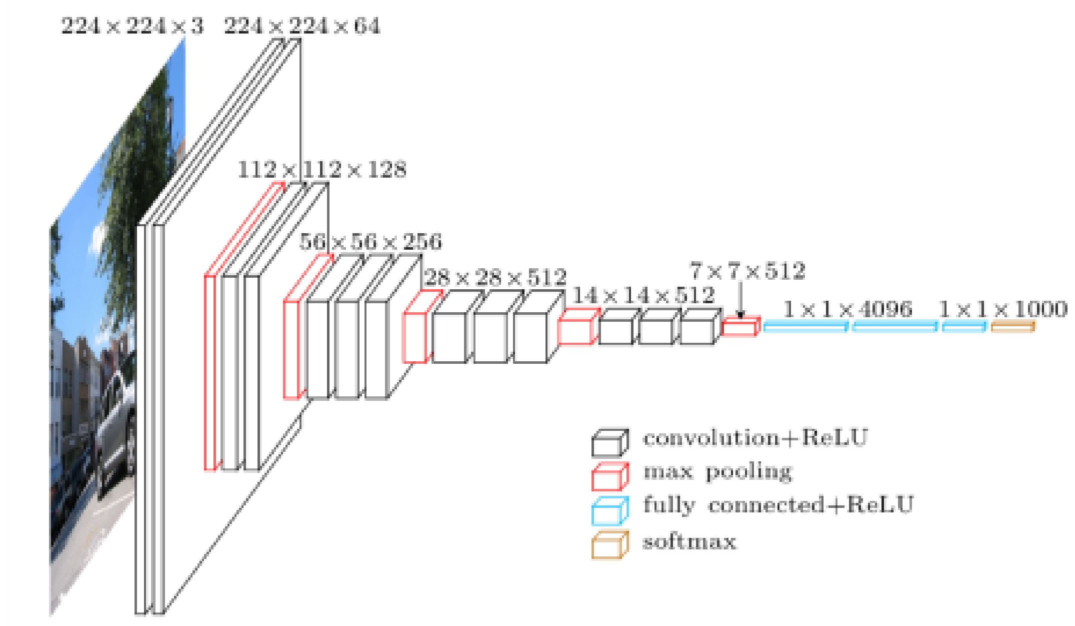


FIGURE 2.10: Architecture of the VGG16 network [72, Slide 18]

GoogLeNet, also known as *InceptionV1* is another interesting network architecture developed by Szegedy et al. from Google in their famous paper *Going deeper with the convolutions* [73] and further developed in [74, 75]. It has won the 2014 ILSVRC challenge and introduced the so called, *Inception Module*, which is responsible for heavily reducing the number of parameters. The model has only $\sim$ 6M parameters compared to VGG16's 138 million parameters but yields better accuracy. Figure 2.11 depicts the Inception Module used in GoogLeNet, which does not follow the linear architecture as shown in figure 2.3.2. Instead the input of the inception module branches out, where each branch has a copy of the input tensor. We can notice the use of convolution layers with filters of size 1×1, which were first investigated in [76]. In inception modules, they are applied to reduce the depth dimension and add additional non-linearity, before expensive 3×3 and 5×5 convolutions. Lastly, the branches are merged together along the depth dimension. Figure 2.12 illustrates the full architecture of GoogLeNet. We can notice, that it has three outputs (also sometimes called heads) in order allow better gradient flow through the network and combat the vanishing gradient problem. Moreover, the model uses **Local Response Normalization (LRN)** layers, which were introduced in [20] and have been found to improve generalization of the network. The

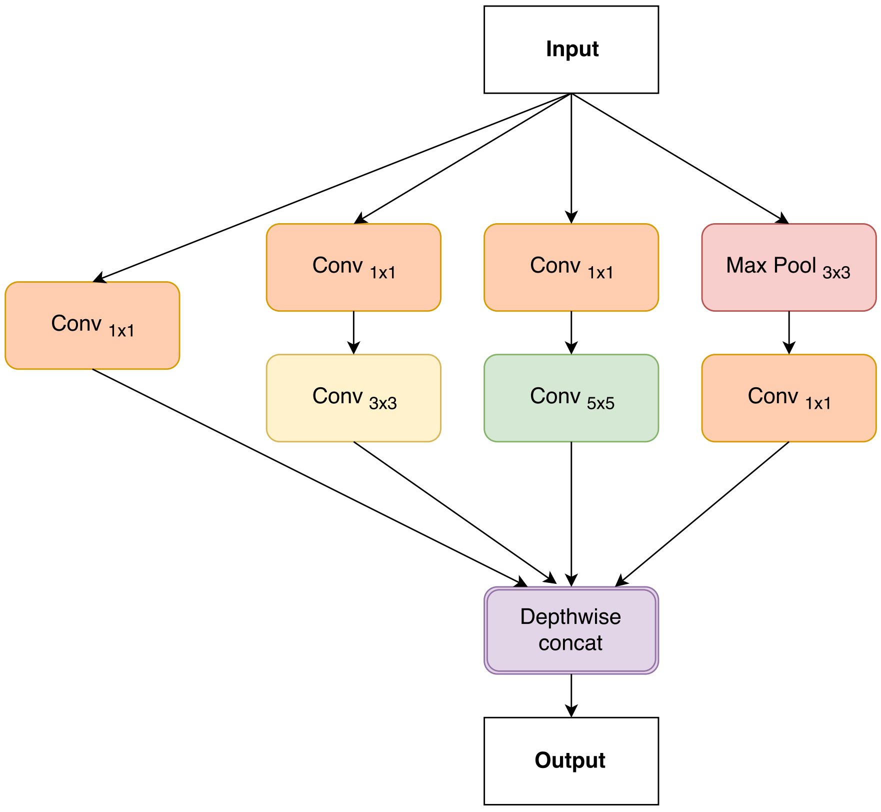


FIGURE 2.11: Inception Module used in GoogLeNet

layer is defined as follows; given an input tensor $\boldsymbol{a}$ with some spatial dimensions and depth dimension d, each value $a_{x,y,i}$ at spatial location (x, y) and depth i is normalized using the following formula:

$$a_{x,y,i} \leftarrow a_{x,y,i} \left(k + \alpha \sum_{j=max(0,i-\frac{n}{2})}^{min(d-1,i+\frac{n}{2})} (a_{x,y,j})^2 \right)^{-\beta} \tag{2.22}$$

where k, α, n and β are hyperparameters. GoogLeNet uses $k = 1, \alpha = 2 * 10^{-5}, n = 4$ and $\beta = 0.75$.

ResNets, known as Residual Networks were developed by Kaiming He at al. in [59] and recently further developed in [75, 77, 78]. They notably won the 2015 ImagetNet challenge surpassing human-level accuracy as well as all main tracks of the COCO 2015 challenges involving object detection, localization and segmentation. The core idea behind ResNets is that increasing the number of layers in a naive way does not necessary increase the performance. ResNets are built using residual blocks, which are sequences of convolutions that are bypassed with *skip connections*. This causes the model to learn residual values in the convolution layers. Figure 2.14 illustrates an

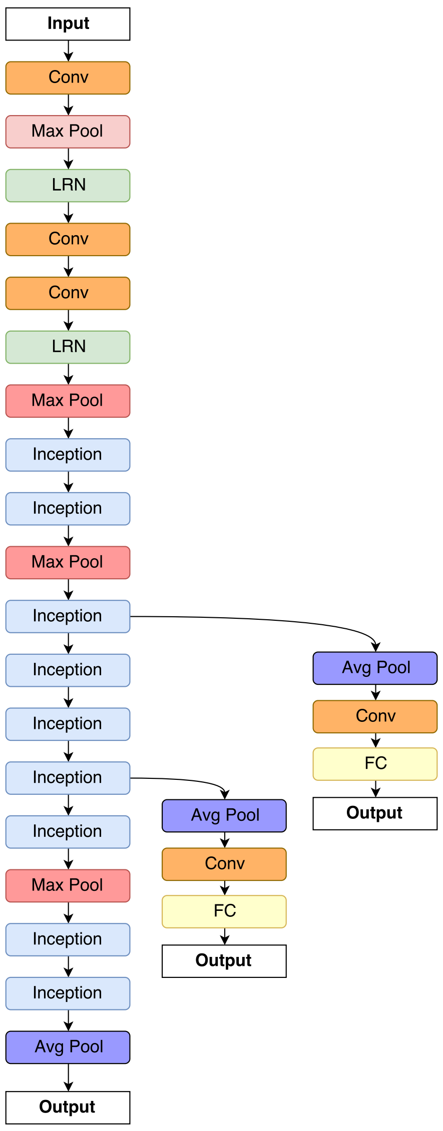


FIGURE 2.12: GoogLeNet aka InceptionV1 with LRN layers

example of a residual block. We can notice that during backpropagation the gradient flowing through the addition will be equally distributed to the convolution layer stack and the skip connection. This is a very desirable property, because it allows the gradient to skip even until the beginning of the network and solve the vanishing gradient problem.

Inception ResNets combine the residual blocks with Inception Modules to further improve the performance [75]. Figure 2.14 depicts the architecture of Inception ResNet V2 from Google, which achieves 3.07% top-5 error on the ImageNet dataset.

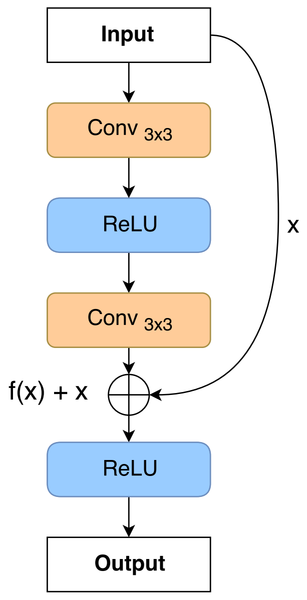


FIGURE 2.13: Basic residual block

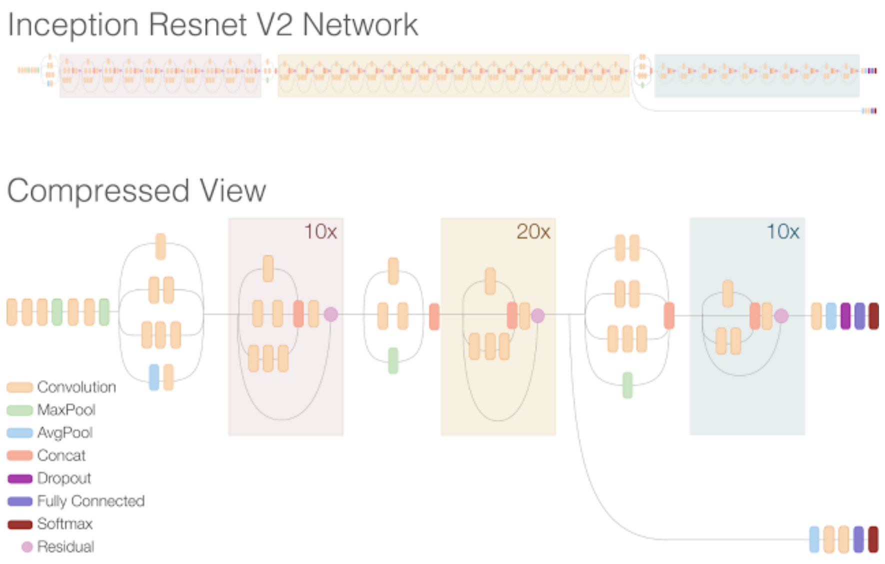


FIGURE 2.14: Inception ResNet V2 [79, Figure 2]

2.3.3 Understanding and Visualizing CNNs

The most common criticism of Convolutional Neural Networks is that they lack interpretability due to the their black-box nature. As a result, a variety of methods have been developed over the years, leading to their better understanding and empirically proving that they can learn a deep representation of the input space. In this section, we discuss some of these approaches [80–88].

- **First conv layers** - The easiest way to get insight is to focus on the first Conv

Layer of a CNN. If the CNN was designed to process RGB images, then the depth dimension of filters in the first layer will be 3. Therefore, we can simply visualize those filters as images. The intuition is that convolution performs inner products at each spatial location between the input and the filter, thus the output will be the highest, when the filter perfectly matches with the area it covers, which is equivalent to template matching. As a result, those filters will visualize what the CNN looks for in a given image. Figure 2.15 illustrates the 96 filters from the first Conv Layer present in AlexNet [20] architecture. We can see that the network has effectively learned how to recognize edges of different size and angle, color blobs and small texture patches. These filters are often called gabor-like filters.

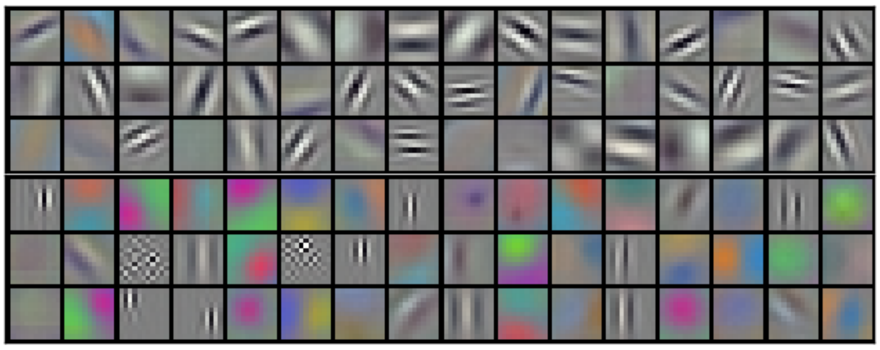


FIGURE 2.15: 96 convolutional filters from AlexNet [20, Figure 3]

- **Intermediate conv layers** - Visualizing filters from intermediate layers is more tricky, as those filters have now depth dimensions higher than 3. One solution is to look at different slices along the depth dimension and visualize them as gray-scale images. However, it usually does not give satisfying results as those filters are applied to the output of the first conv layer. Another way is to visualize activations of those intermediate filters as suggested by Yosinski et al. in [81]. In this case, we can see how the activations change as different images are run through the network. For instance, authors showed that a CNN trained on ImageNet dataset had a neuron that lighted up when a face was present in an image, even though the dataset did not contain any images explicitly labeled as face.

- **Last layers** - Similarity to first and intermediate layers of the CNN, last layers can give us additional insight. For instance, in image classification task, last layers are usually large fully-connected layers with a typical output dimensionality $1024, 2048$ or 4096. It is difficult to understand those highly dimensional spaces, but as it turns out, images from the same class are actually nearest-neighbors in that space

[20]. Moreover, we can reduce the dimensionality to a 2D or 3D space using **t-SNE** algorithm [83]. Figure 2.16 depicts such embedding of the VGG16's penultimate layer output on 400 images from the ImageNet dataset.

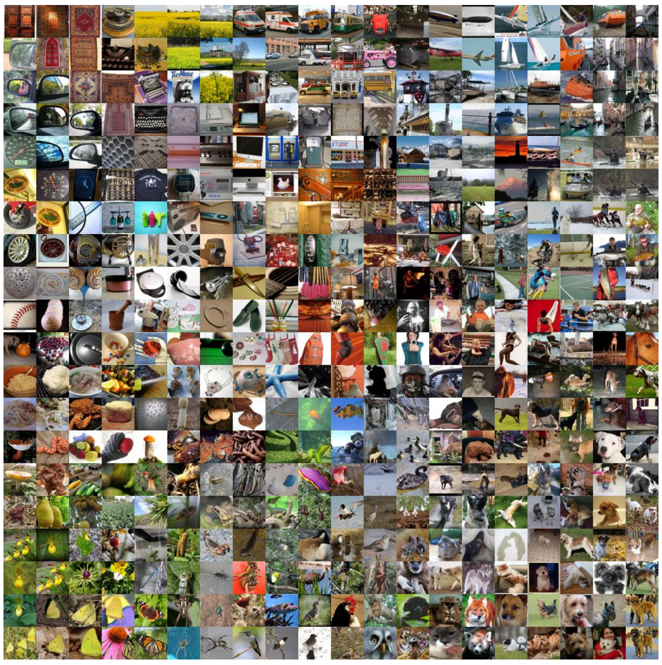


FIGURE 2.16: VGG16's penultimate layer visualization using t-SNE on 400 ImageNet images [89, Figure 4]

- **Occlusion experiment** - Another way of probing the network trained for classification task was the experiment conducted by Zeiler and Fergus in [82]. Authors showed what was particularly important to the network in correctly predicting class scores by occluding parts of images and looking at the network output.

- **Saliency maps** show which part of the input image is the most important for the network during prediction. The core idea is to compute gradient of network's output with respect to the input image: $\nabla_I f(I; \boldsymbol{\theta})$, where f is the network with parameters $\boldsymbol{\theta}$ represented as a function and I is the input image. This formulation is known as *vanilla gradient* was first introduced in [80] and further improved in subsequent work [86–88]. Those methods are illustrated in figure 2.17 and described below:

  - GUIDED BACKPROPAGATION [86] improves *vanilla gradient* producing more crisp images. The idea comes from the assumption that neurons are detectors of particular image features and therefore, we are only interested in image features detected by the neuron as opposed to not detected. This is implemented by suppressing all negative gradients, meaning that all activation functions have to be modified in the network, thus making it sometimes difficult to implement.

– Integrated Gradients [87] is another improvement to *vanilla gradient* method, that does not require any modification of the original network and has strong theoretical justification. The integrated gradients method satisfies multiple desired properties, such as *sensitivity* as opposed to guided back-propagation. The formula for computing the integrated gradients applied to images is the following:

$$\texttt{IntegratedGrads}(I) ::= I \int_{\alpha=0}^{1} \nabla_I f\Big(\alpha I; \boldsymbol{\theta}\Big)\, d\alpha \tag{2.23}$$

where I is the input image, α is the scaling factor, and $f(I, \boldsymbol{\theta})$ is the network with parameters $\boldsymbol{\theta}$ represented as function. The method scales the original input image by an average of multiple gradients of scaled images.

– SmoothGrad [88] helps further reduce noise in saliency maps by smoothing the gradients with a Gaussian filter. This helps because gradients w.r.t image may fluctuate sharply at small scales, which is partly true due to the use of ReLUs as they are not continuously differentiable. We can compute the smoothed saliency map with the following formula:

$$\hat{M}(I) = \frac{1}{n} \sum^{n} M(I + \mathcal{N}(0, \sigma^2)) \tag{2.24}$$

where $M(I)$ is the saliency map for an image I and n is the number of samples. The authors suggest that $n = 50$ and $\sigma = 0.2 * (max(I) - min(I))$. Furthermore, we can see that this method can be combined with other saliency map methods.

2.4 Recurrent Neural Networks

One major shortcoming of traditional neural networks (section 2.3 and 2.2) is their lack of persistence. We as humans use past information to reason about present. For instance, it would be impossible to say what is happening in a movie just by looking at a

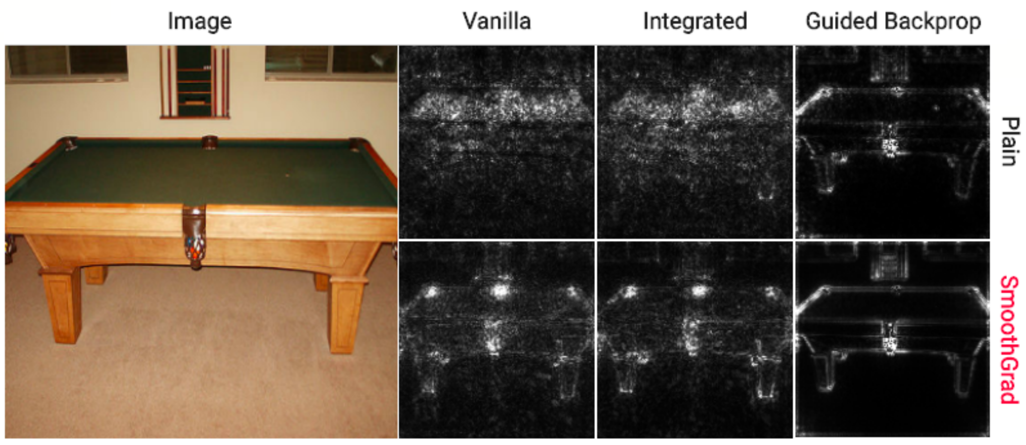


FIGURE 2.17: Comparison of saliency map methods (modified figure based on [90])

single frame. Recurrent neural networks (RNNs) overcome this limitation by introducing loops, which carry information between time-steps. Figure 2.18 illustrates a simple RNN, where block A takes a fixed-sized input at a current time $\boldsymbol{x}_t$, fixed-sized output from the previous time-step $\boldsymbol{h}_{t-1}$ and produces a new fixed-sized output at a current time-step $\boldsymbol{h}_t$. The right-hand side of the figure shows the same network, but with an unrolled loop by copying the network t times, which allows us to treat it as a traditional neural network with a varying input and output sizes.

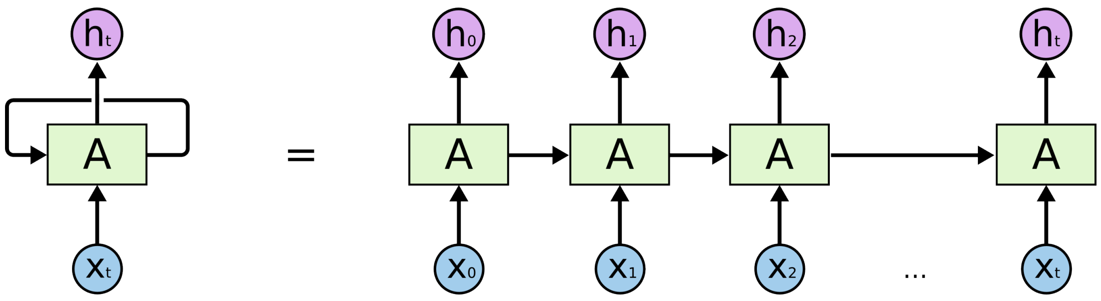


FIGURE 2.18: RNN and the same RNN with loops unrolled [91, Figure 2])

The ability to process information where input and output is not fixed is very important. For instance, image captioning [92] is a task, where given an input image we are asked to generate sequence of words (a sentence) describing that image, which is an example of a ONE-TO-MANY model. Sentiment analysis [93], on the other hand, is a MANY-TO-ONE model, because its task is to predict whether a given sentence has positive or negative sentiment. Machine Translation [94, 95] is a task of translating a sentence from one language to another and therefore can be expressed as a MANY-TO-MANY model. Finally, frame-by-frame image-based localization [5] is also a MANY-TO-MANY model. However, it differs from Machine Translation, because the model is *"synced"*, meaning that output at time-step t is directly influenced (without a delay) by the input at the same time-step[3].

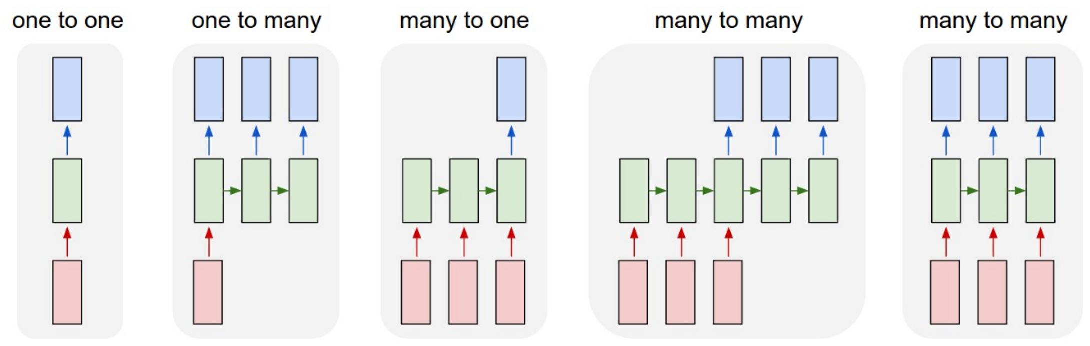


FIGURE 2.19: Unrolled RNN architectures for different model types [96, Figure 1])

Figure 2.19 illustrates the unrolled RNN architectures depending on the model type. Red boxes indicate fixed-sized inputs, green boxes are hidden layers and blue boxes show fixed-sized outputs. The rightward direction marks the time dimension and shows the causality between elements of the sequence. RNNs are trained using a method called Backpropagation Through Time (BPTT), where every element from the sequence is forwarded through the network and consequently the loss is computed and backpropagated for every output time-step. It is important to notice, that all copies of the network share the same set of parameters. However, BPTT can sometimes be infeasible if the sequence is very long, in this case Truncated BPTT (TBPTT) is employed. It divides the long sequence into shorter sequences and runs BPTT on these short sequences, passing the final state from the last time-step to the initial state of the first time-step.

The RNNs can be viewed as a fixed computer programs with some inputs and internal variables. In fact, it was proven that RNNs are Turing-Complete [97], meaning that for every computable function there exist a finite RNN. Recurrent neural networks are also useful for processing information that does not seem to be sequential at a first glance, such as images. For example, steering attention around an image to look for important features can be viewed as a sequential task [98].

2.4.1 Vanilla RNN

The first RNNs had a very simple structure for combining the input at a current time-step $\boldsymbol{x}_t$ with the output of the previous time-step $\boldsymbol{h}_{t-1}$. The formulation of the

[3]time-step commonly refers to a position in sequence, e.g a word in a sequence or a frame in a video

so-called vanilla RNN is the following:

$$\boldsymbol{h}_t = tanh(\boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{b}) \tag{2.25}$$

where $\boldsymbol{W}_{hh}$ and $\boldsymbol{W}_{xh}$ are parameter matrices and $\boldsymbol{b}$ is the bias vector.

Learning dependencies is the main objective of RNNs. However, dependencies can be long- or short-term. For short-term dependencies we only have to look into the near past in order to correctly solve the problem. Long-term dependencies demand more context. For example, predicting the last word from sentence: *"I grew up in France . . . I speak fluent French* requires remembering facts that appear at the beginning of the sentence. RNNs are, in theory, capable of dealing with arbitrarily long dependencies. However, in practice vanilla RNNs struggle, when trained with gradient descent methods due to the vanishing and exploding gradient problems. Exploding gradient can be mitigated by clipping the gradient. However, the vanishing gradient issue cannot be solved easily. It was was studied in depth by Hochreiter [99] and Bengio et al. [100], who showed fundamental reasons why that might be the case.

2.4.2 Long-short Term Memory

Long-short Term Memory (LSTM) is a type of RNN, which addresses the problem of learning long-term dependencies. It was first introduced by Hochreiter in [101] and designed to avoid the vanishing gradient problem, thus vastly improving the capability to learn long-term dependencies.

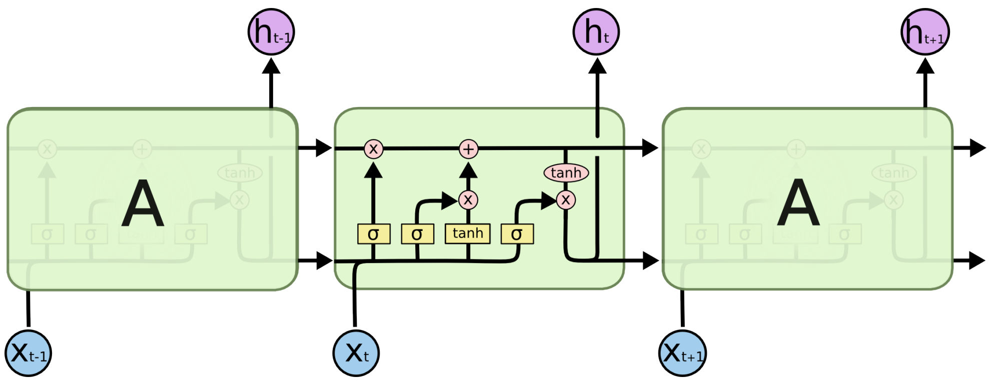


FIGURE 2.20: Inner-workings of an LSTM block [91, Figure 6])

Similarly to a vanilla RNN architecture (figure 2.18), LSTMs have a block applied at every time-step. This difference is that the block (usually called an LSTM cell) now takes two states from the previous time-step: hidden state h_{t-1} and cell state c_{t-1}. Furthermore, the actual output from this LSTM block is also the hidden state this is passed to the next block in the sequence. Figure 2.20 illustrates the inner-workings of an LSTM block, where operations shown in circles are applied element-wise and yellow rectangles represent fully-connected layers with sigmoid or tanh activations. Splitting arrows correspond to a copy operation and connecting arrows depict concatenation. The top horizontal line represents the flow of the cell state. We can notice, that this state is only modified by element-wise addition and multiplication, which means that the gradient will flow uninterruptedly throughout the entire sequence in a similar fashion to residual networks. The cell state acts as a memory for a given LSTM block and is manipulated by a set of gates. *Forget gate* ($\boldsymbol{f}$) decides whether we erase information from the cell, *input gate* ($\boldsymbol{i}$) determines whether we write any information to the cell state and finally *output gate* ($\boldsymbol{o}$) tells us how much of the cell state is revealed. Mathematically, a set of equations (2.26) describes the behavior of a single LSTM block at time t. Sigmoid activation functions are used, because they squash input into $[0, 1]$ range, which can then effectively block or allow information. On the other hand, hyperbolic tangent (tanh) activation squashes the input into $[-1, 1]$ range allowing both positive and negative values to be added to the cell state.

$$\begin{aligned}
\boldsymbol{f}_t &= \sigma(\boldsymbol{W}_f x_t + \boldsymbol{U}_f \boldsymbol{h}_{t-1} + \boldsymbol{b}_f) \\
\boldsymbol{i}_t &= \sigma(\boldsymbol{W}_i x_t + \boldsymbol{U}_i \boldsymbol{h}_{t-1} + \boldsymbol{b}_i) \\
\boldsymbol{o}_t &= \sigma(\boldsymbol{W}_o x_t + \boldsymbol{U}_o \boldsymbol{h}_{t-1} + \boldsymbol{b}_o) \\
\boldsymbol{c}_t &= \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tanh(\boldsymbol{W}_c x_t + \boldsymbol{U}_c \boldsymbol{h}_{t-1} + \boldsymbol{b}_c) \\
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t)
\end{aligned} \tag{2.26}$$

2.4.3 Variants on Long-short Term Memory

LSTMs have been introduced quite a while ago, but only recently started getting traction. This has led researchers to investigate what is so special about the LSTM structure leading to creation of multiple variants. A popular variant introduced by Gers and Schmidhuber in [102] is called LSTM with "peephole connections", which essentially allows all internal fully-connected layers to access the cell state.

Gated Recurrent Unit (GRU) [94] is another more dramatic variation of an LSTM. It combines the forget get with the input gate into a single *update gate* ($\boldsymbol{z}$) and merges the cell state with the hidden state. Figure 2.21 illustrates the inner-workings of a GRU block. GRUs have less parameters, leading to faster training and usually require less data. However, all trade-offs between LSTMs and GRUs have not been fully explored [103]. Finally, Jozefowicz et al. explored the space of various LSTM architectures and found a specific structure that outperformed both LSTMs and GRUs, but only on specific problems. They also showed that adding a bias of 1 to the forget gate of LSTM closes the gap between LSTMs and GRUs.

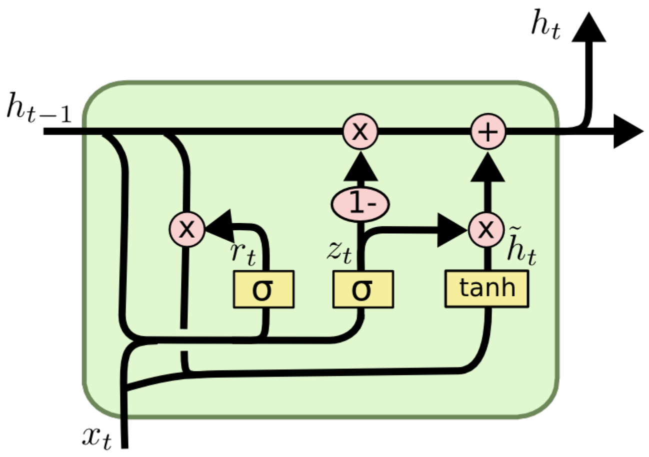


FIGURE 2.21: Inner structure of a GRU block [91, Figure 16])

2.5 Transfer Learning

Training CNNs from scratch takes a great amount of time and requires extremely large datasets. For example, training network architectures described in section 2.3.2 can take 2-3 weeks on multiple high-end GPUs. Therefore, in practice only a few people train CNNs from scratch. However, CNNs have proven to be great at extracting features, which can then transfer across a variety of different computer vision tasks [67, 69, 70]. Transfer learning refers to the use of pretrained CNNs: (1) as a fixed feature extractor or (2) for finetuning (some) weights of the model.

In order to use CNNs as fixed feature extractors, we take a pretrained model such as GoogLeNet trained on some dataset and discard the last fully-connected layers responsible for image classification. We can now take a new small dataset, perform forward pass and store the output to disk. For VGG16, this will give us a 512-D vector for each image that can be viewed as a compact representation of the that image. We can use those CNN features as a base for solving different computer vision tasks, such as image captioning, video classification or human-pose estimation. It is crucial to notice that a choice of the dataset used for pretraining the model matters and can greatly improve or hinder the performance of the final model. If our small dataset differs a lot from the initial dataset used for pretraining, then we might expect the performance to be mediocre. Therefore, we should always strive for using models pretrained on datasets as close as possible to our dataset.

It is often better to finetune the pretrained model than to simply use it as a feature extractor. Finetuning consists of selecting some convolutional layers from the pretrained model and freezing all the other layers. The frozen layers will not be updated during training and the selected layers must be initialized to values from the pretrained model. If the new dataset is large, then we might opt for finetuning larger portion of the CNN. However, the first conv layers extract basic features such as edges or corners and those transfer well between datasets, therefore finetuning the whole network is usually not necessary. Moreover, if we decide to add any layers to the network, then it is generally best to first freeze all layers except the new ones, train the network and then finetune some conv layers. The reason is that newly added layers will be randomly initialized and the gradient might completely wreck the pretrained conv layers if they are not frozen.

Chapter 3

Camera Relocalization using Deep Learning

In this chapter, we formulate the problem of camera relocalization using deep learning (section 3.1) and discuss the related work (section 3.2). Afterwards, we describe different loss functions used in our models (section 3.3) and introduce a novel loss function based on quaternion algebra. Finally in section 3.4, we examine three neural network architectures, that allow us to train the whole system in an end-to-end fashion.

3.1 Problem Statement

Camera relocalization task, also known as image-based localization task is defined as the task of determining the location of a given image in an arbitrary coordinate frame. In the simplest form, each RGB image $\boldsymbol{x}$ is processed independently in order to predict a location $\boldsymbol{y} = f(\boldsymbol{x})$. In our case, the location is expressed as a 6-DOF pose, consisting of position $\boldsymbol{y}_{pos} \in \mathbb{R}^3$ and orientation $\boldsymbol{y}_{rot} \in SE(3)$. We also extend this task to a sequences of n RGB images $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(n)}$ taken at constant rate, where all images from the sequence are localized jointly, $\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, \ldots, \boldsymbol{y}^{(n)} = f(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(n)})$.

In this thesis, we consider an online localization system for applications in robotics, mainly unmanned aerial vehicles (UAVs) and unmanned ground vehicles (UGVs). However, the system can also be used for smart-phone localization, as present smart-phones have excellent cameras and enough computational power for on-board processing. We focus on indoor environments, where localization is usually difficult due to the lack of GPS signal, but our system can be easily extended to outdoor environments. Lastly, localization takes place in a well-defined search area, such as a university campus.

There are three main approaches to localization using supervised learning. In this setting, we are given a training dataset X_{train} that contains pairs of images and their corresponding locations in a given coordinate frame $X_{train} = \{(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}), \dots, (\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})\}$. We use this dataset to train a model denoted as function $\hat{\boldsymbol{y}}^{(i)} = f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})$, where $\hat{\boldsymbol{y}}^{(i)}$ is the prediction for image $\boldsymbol{x}^{(i)}$ and $\boldsymbol{\theta}$ are the model parameters.

3.1.1 Relocalization using Content-Based Image Retrieval system

A common approach to camera relocalization task is through a Content-Based Image Retrieval (CBIR) system. This method uses a model $\boldsymbol{z} = f(\boldsymbol{x})$ to transform an input image $\boldsymbol{x}$ into a representation $\boldsymbol{z} \in \mathbb{R}^n$, called feature vector. The feature space is often significantly less dimensional than the original input space and the feature vector is usually obtained using SIFT [1], SURF [2] or ORB [3] feature extraction algorithms. However, these algorithms use hand-crafted features, based on multiple assumptions and therefore might not necessary perform well on images with large texture-less areas such as close-up images of walls. Therefore, a machine learning based approach, which learns the transformation might yield superior performance as the appropriate features are learned from the data during training.

Once the images are transformed into the feature space, a reference database composed of N pairs of feature vectors and their corresponding locations is required in form $\{(\boldsymbol{z}^{(1)}, \boldsymbol{y}^{(1)}), (\boldsymbol{z}^{(2)}, \boldsymbol{y}^{(2)}), \dots, (\boldsymbol{z}^{(N)}, \boldsymbol{y}^{(N)})\}$, where $\boldsymbol{y}^{(i)}$ are locations of images as described in section 3.1. Then given a query image $\boldsymbol{I}^{'}$ and its feature vector $\boldsymbol{z}^{'}$ we can predict its location using nearest neighbor method:

$$\boldsymbol{z}^{*} = \arg\min_{\boldsymbol{z}}(\|\boldsymbol{z} - \boldsymbol{z}^{'}\|) \tag{3.1}$$

where $\boldsymbol{z}^{*}$ is the closest image to the query image in the feature space. The actual prediction $\hat{\boldsymbol{y}}'$ can be then retrieved by association from the reference database.

3.1.2 Relocalization as Classification Problem

Positions and orientations are continuous in real-world. However, if we divide the search area into a set of N non-overlapping blocks, then we can convert the task into a classification problem with N different classes corresponding to different locations. The downside of this approach is that the maximum prediction accuracy will be limited to the size of the dividing block. Moreover, orientation and altitude are usually not considered in this approach as they increase the number of possible classes thus making it very difficult to train.

A common classification model $f : \boldsymbol{x} \mapsto \mathbb{R}^N$ takes an input image $\boldsymbol{x}$ and computes a class score vector $\boldsymbol{s} \in \mathbb{R}^N$, where N is the number of classes. The class scores are usually interpreted as unnormalized log probabilities, thus a softmax function can convert the class score vector into a discrete probability distribution over the classes, where the predicted class is the one with the highest probability: $\hat{y} = \arg\max_i(\boldsymbol{s}_i)$. The model is trained end-to-end using a cross-entropy loss function, defined as:

$$\mathcal{L}(\boldsymbol{\theta}) = -\log\left(\frac{e^{\boldsymbol{s}_y}}{\sum_j e^{\boldsymbol{s}_j}}\right) \tag{3.2}$$

where $\boldsymbol{s}_j$ is the j^{th} element of the class score vector, $\boldsymbol{s}_y$ is therefore the element of the class score that corresponds to the correct class for that image. The advantage of classification is that we automatically get the confidence value for each location. PlaNet [104] is an example of performing image-based localization using classification on a global scale, where authors divided earth's surface and used CNN to predict class scores. They also extended the problem to a sequence of images and improved the accuracy using LSTMs.

3.1.3 Relocalization as Regression Problem

As described in section 1.1, regression problems refer to problems where the prediction lies in continuous space. Localization is an example of such problem, where the goal is to predict a tuple $\boldsymbol{y} = (\boldsymbol{y}_{pos}, \boldsymbol{y}_{rot})$ composed of position $\boldsymbol{y}_{pos} \in \mathbb{R}^3$ and orientation $\boldsymbol{y}_{rot} \in SO(3)$, commonly referred to as 6DOF pose. It is important to predict position and orientation at the same time, because two images taken at the same position but facing different directions can vastly differ. Similarly to classification problem, the model can be trained in an end-to-end fashion, where locations are predicted directly from images without the need to hand-craft features. The choice of the loss function is more complicated than in the case of classification and section 3.3 describes different loss functions designed for this problem.

The main advantage of viewing localization as a regression over classification is that regression can produce arbitrarily fine grained predictions, while classification is bounded by the size of the block dividing the search space [4]. However, the downside is that regression in the simple form cannot provide uncertainty measures for predictions, which are crucial for many applications such as robotics. However, there exist models, which can predict multiple poses from a single image. The uncertainty measure can be then computed from the variance of these poses. The downside of this approach is the additional computational cost of performing forward pass multiple times [6].

3.2 Related work

There are multiple approaches to image-based localization aside from those described in the section above. For example, **map-matching** methods use a map of the environment in form of navigable paths or floor-plans consisting of traversable and non-traversable areas. They rely on strict data-association and use interoceptive and exteroceptive sensors in order to retrieve a global pose estimate usually through sequential Monte Carlo filtering [105] or Hidden Markov Models [106]. These methods also process sequential observations similar to one of our proposed methods, but often yield inferior performance compared to localization methods based on 3D maps of sparse features.

Localization methods based on 3D maps of sparse features, often called **sparse feature based localization**, require a 3D model of discriminative features usually obtained from a Structure-from-Motion (SfM) pipeline. Prediction of query images is then obtained using camera re-sectioning, which requires matching against very large 3D models. This process is very computationally expensive and needs a lot of memory for the map, which often prevents it from being implemented on resource-constrained platforms, such as aerial robots. Several methods have been proposed to combat this issue. For example, in [107] authors introduce an active search method to efficiently find reliable correspondences. In [108], Sattler et al. proposed a quantized vocabulary for direct 2D-to-3D matching, where the camera pose is searched using a combination of RANSAC and a PnP algorithm.

Another interesting approach to image-based localization, which is closer to our work was developed by Shotton et al. in [12]. The authors use **regression forests** to learn the mapping of RGB-D pixels to 3D points in the scene's world coordinate frame. The correspondences are then fed to a RANSAC algorithm in order to produce a pose prediction. The authors also extend the model and consider sequences of images with the purpose of exploiting the temporal regularity present in image sequences. However, the main problem with this approach is that it requires depth information, which is usually not available.

Motivated by the recent advent of deep learning PoseNet [4] first tried directly regressing poses from image pixels. Authors showed that while still far from competing against traditional methods, they have managed to produce good results with a naive loss function and without making any attempt at capturing the structure of the scene or exploiting the temporal smoothness in image sequences. The general idea was further explored in [5–7, 9–11] and is the main part of this thesis.

3.3 Loss functions for Relocalization as Regression

In this section, we describe various loss functions used for training regression models for the camera relocalization task. In section 3.3.1 we show a naive multi-task loss for regressing position and orientation simultaneously. Then in section 3.3.2, we demonstrate an improved approach for dealing with multi-task loss that does not require expensive hyperparameter tuning. Finally, in section 3.3.3 we introduce a novel orientation loss function based on quaternion algebra.

3.3.1 Weighted loss

Learning position and orientation at the same time is crucial as described in section 3.1.3. However, the set of possible positions is very different from the set of possible orientations, since positions are in $\mathbb{R}^3$, while orientations are in $SO(3)$. In order to train a model to simultaneously predict both orientation and position, the position loss has to be combined with the orientation loss. This is achieved with a weighted loss defined as:

$$\mathcal{L}(\boldsymbol{I}) = \mathcal{L}_{pos}(\boldsymbol{I}) + \beta \mathcal{L}_{rot}(\boldsymbol{I}) \tag{3.3}$$

where $\mathcal{L}(\boldsymbol{I})$ is the total loss of the position and orientation regression multi-task given an input image $\boldsymbol{I}$. $\mathcal{L}_{pos}(\boldsymbol{I})$ is the position loss function and $\mathcal{L}_{rot}(\boldsymbol{I})$ is the orientation loss function, while β is the weighting hyperparameter that balances these two loss functions. The total loss function can be further broken down into:

$$\mathcal{L}(\boldsymbol{I}, \boldsymbol{x}, \boldsymbol{q}; \boldsymbol{\theta}) = \|\boldsymbol{x} - \hat{\boldsymbol{x}}\|_p + \beta \left\| \boldsymbol{q} - \frac{\hat{\boldsymbol{q}}}{\|\hat{\boldsymbol{q}}\|} \right\|_p \tag{3.4}$$

where $\boldsymbol{I}$ is the input image with its corresponding ground-truth position $\boldsymbol{x}$ and orientation $\boldsymbol{q}$ as a quaternion. $\boldsymbol{\theta}$ are model parameters used to compute the position $\hat{\boldsymbol{x}}$ and orientation $\hat{\boldsymbol{q}}$ predictions for the input image. The position loss function takes form of a p-norm, where p is usually 1 or 2, thus making it an L1 or an L2 norm. On the other hand, orientation loss function is also a p-norm, but the predicted quaternion is first normalized to a unit quaternion, which represents a valid rotation in 3D space.

The main problem with this loss function is the need for expensive finetuning of the β hyperparameter. The models are extremely sensitive to its choice and can yield very good performance or are even unable to converge.

3.3.2 Homoscedastic uncertainty based loss

As discussed in section 3.3.1, the weighted multi-task loss function suffers from the existence of the β hyperparameter. Ideally, we would like to develop a loss function, which can learn the optimal choice of β from the data, thus removing the need for expensive hyperparameter search. This can be achieved using *homoscedastic uncertainty* also known as *task-dependant uncertainty*. It is an aleatoric uncertainty[1] that does not depend on the input data. Kendall and Cipolla showed in [7] how to use this uncertainty in order to combine losses from different tasks in a probabilistic manner.

$$\mathcal{L}_\sigma(\boldsymbol{I}) = \mathcal{L}_{pos}(\boldsymbol{I})\hat{\sigma}_{pos}^{-2} + \log \hat{\sigma}_{pos}^2 \; + \; \mathcal{L}_{rot}(\boldsymbol{I})\hat{\sigma}_{rot}^{-2} + \log \hat{\sigma}_{rot}^2 \tag{3.5}$$

[1]Aleatoric uncertainty describes the uncertainty with respect to information that cannot be explained by our data. It can be reduced by increasing the precision of all observable variables.

Equation 3.5 shows the position and orientation multi-task loss based on the objective of minimizing the homoscedastic uncertainties, where position and orientation losses $\mathcal{L}_{pos}$, $\mathcal{L}_{rot}$ are the same as in section 3.3.1. $\hat{\sigma}_{pos}$ and $\hat{\sigma}_{rot}$ are the position and orientation homoscedastic uncertainties. Moreover, they are trainable parameters (scalar values), which are updated during backpropagation in the same manner as other model parameters. The terms, log $\hat{\sigma}^2_{pos}$ and log $\hat{\sigma}^2_{rot}$ are regularization terms in order to prevent the network from learning infinite uncertainty and therefore zero loss. The homoscedastic uncertainty for rotation is expected to be smaller than the uncertainty for position, since quaternions are restricted to a unit manifold, whereas positions can have arbitrary large errors. Finally, in order to improve numerical stability, we transform the equation 3.5 to:

$$\mathcal{L}_{\sigma}(\boldsymbol{I}) = \mathcal{L}_{pos}(\boldsymbol{I})\ e^{-\hat{s}_{pos}} + \hat{s}_{pos}\ +\ \mathcal{L}_{rot}(\boldsymbol{I})\ e^{-\hat{s}_{rot}} + \hat{s}_{rot} \tag{3.6}$$

where $\hat{s} \leftarrow \log \hat{\sigma}^2$, which avoids a potential division by zero. Throughout the remainder of this thesis, we refer to this loss function as Naive-Homoscedastic or NH.

3.3.3 Quaternion error loss

The orientation loss function defined in equation 3.4 assumes that the difference between two quaternions can be computed using L1 or L2 loss. However, that is not the case, as quaternions have their own algebra. Quaternion is defined as a collection of 4 real numbers, such that $\boldsymbol{q} = [q_0, q_1, q_2, q_3]$, where q_0 is the real part of the quaternion and $\boldsymbol{q}_v = [q_1, q_2, q_3]$ is the vector part. A unit quaternion $\|\boldsymbol{q}\| = 1$ represents a rotation in 3D space. Moreover, quaternions are a double map of $SO(3)$, particularly $-\boldsymbol{q}$ and $\boldsymbol{q}$ represent the same orientation. A minus operation in $SO(3)$ is defined as $\ominus : SO(3) \times SO(3) \mapsto \mathbb{R}^3$. It returns the vectorial difference $\boldsymbol{\theta} \in \mathbb{R}^3$ between two orientations in 3D space and is expressed in the vector space tangent to the reference element [109]. If the rotations chosen for calculating the difference are quaternions $\hat{\boldsymbol{q}}$ and $\boldsymbol{q}$, then the difference is defined as:

$$\boldsymbol{\theta} = \log\left(\boldsymbol{q}^{-1} \otimes \hat{\boldsymbol{q}}\right) \tag{3.7}$$

where $\otimes$ is the quaternion product and $\boldsymbol{q}^{-1}$ is the inverse quaternion. A quaternion product is defined as:

$$\boldsymbol{p} = \boldsymbol{p} \otimes \boldsymbol{q} = \begin{bmatrix} p_0q_0 - p_1q_1 - p_2q_2 - p_3q_3 \\ p_0q_1 + p_1q_0 + p_2q_3 - p_3q_2 \\ p_0q_2 - p_1q_3 + p_2q_0 + p_3q_1 \\ p_0q_3 + p_1q_2 - p_2q_1 + p_3q_0 \end{bmatrix} \tag{3.8}$$

and an inverse quaternion is equal to quaternion conjugate for unit quaternions:

$$\boldsymbol{q}^{-1} = \frac{\boldsymbol{q}^*}{\|\boldsymbol{q}\|^2} = \boldsymbol{q}^* = [q_0, -q_1, -q_2, -q_3] = [-q_0, q_1, q_2, q_3] \tag{3.9}$$

As stated previously, quaternions represent the same rotations as their negative counterparts, therefore it is more beneficial to only negate the real part q_0 of a quaternion in order to speed up computation. Finally, the logarithm of a quaternion is defined as:

$$\log \boldsymbol{q} = \boldsymbol{u}\theta = \boldsymbol{q}_v \frac{\arctan(\|\boldsymbol{q}_v\|, q_w)}{\|\boldsymbol{q}_v\|} \approx \frac{\boldsymbol{q}_v}{q_w}\left(1 - \frac{\|\boldsymbol{q}_v\|^2}{3q_w^2}\right) \approx \boldsymbol{q}_v \xmapsto[\theta \mapsto 0]{} \mathbf{0} \tag{3.10}$$

where $\boldsymbol{u}$ and θ are the axis-angle representation of $\boldsymbol{q}$. We use the last approximation to speed up the computation and avoid division by zero. Subsequently, we can now define our orientation loss function based on quaternion algebra as:

$$\mathcal{L}_{rot}(\boldsymbol{I}) = \| \log \left(\boldsymbol{q}^{-1} \otimes \hat{\boldsymbol{q}}\right) \|_p \tag{3.11}$$

We combine this loss function with the previously defined homoscedastic loss function and for the remainder of the thesis, we refer to it as Quaternion-Error-Homoscedastic or QEH.

3.4 Proposed Methods

In this section, we describe our proposed regression models for the camera relocalization task. In section 3.4.1, we show a simple regression model built on top of a pretrained CNN. We then improve this model in section 3.4.2 by adding spatial LSTM module as showed in [11]. Lastly, in section 3.4.3, we extend the camera relocalization problem to sequence of images and exploit the temporal smoothness constraint to further improve the performance. All proposed models utilize transfer learning, as described in section 2.5, where the base models are: (1) GoogLeNet pretrained on ImageNet [73], (2) GoogLeNet pretrained on Places365 [110], (3) Inception ResNet V2 pretrained on ImageNet [75] and (4) VGG16 pretrained on Hybrid1365 [110]. We explore the importance of CNN architectures and the datasets used for training them as the base for camera relocalization task.

3.4.1 Regressor

The Regressor model is based on PoseNet [4]. The output of the model is a 6-DOF pose prediction expressed as a tuple $\hat{\boldsymbol{y}} = (\hat{\boldsymbol{x}}, \hat{\boldsymbol{q}})$, where $\hat{\boldsymbol{x}} \in \mathbb{R}^3$ and $\hat{\boldsymbol{q}} \in \mathbb{R}^4$. As previously stated, we use transfer learning applied to 4 different CNN models. We apply the following changes to these models:

- **GoogLeNet** - For the two models based on GoogLeNet architecture (ImageNet and Places365) we discard the two auxiliary output branches as well as the fully connected layers from the main branch. We also freeze all layers except the last three inception modules. Given a $224 \times 224 \times 3$ input image, the model now produces a 1024D feature vector.
- **Inception ResNet V2** - We remove the auxiliary branch as well as the full connected layer from the main branch. Furthermore, we freeze all layer except the last 10 inception resnet blocks as seen in figure 2.11. The new model now takes a $299 \times 299 \times 3$ input image and produces a 1536D feature vector.
- **VGG16** - We remove all fully connected layers, freeze all layers except the final 3 convolutional layers and add a Global Average Pooling layer [76] after the last convolutional layer. The resulting model takes a $224 \times 224 \times 3$ input image and produces a 512D feature vector.

As state above, we freeze a considerable portion of each CNN model in order to significantly reduce computational resources needed to train these models. This is a different approach to those currently present in the literature [4, 5, 7, 11], which do not freeze any conv layers and finetune the whole network. Moreover, freezing the layers means that instead of just blocking the parameter updates, we perform the forward pass of the entire training and testing datasets until the last frozen layer and store this output to disk. Afterwards, during actual training, we only instantiate the non-frozen part of the model and feed it with the stored features instead of actual images. This allows for much faster convergence and training on GPUs that do not have enough VRAM to fit decent batch sizes.

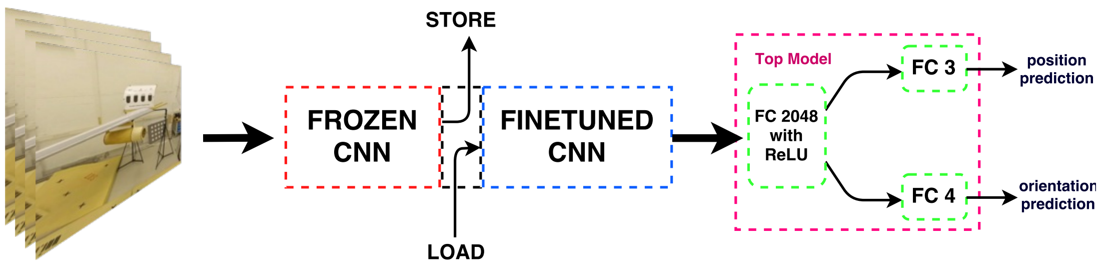


FIGURE 3.1: General structure of the Camera Relocalization model - showing Regressor model

Figure 3.1 illustrates a general structure of the Camera Relocalization model, where CNN refers to one of the models described above. Images are forwarded through the frozen part of the CNN and the resulting features are stored to disk. Subsequently, those features are loaded during training and only the remaining part of the CNN is finetuned. The figure illustrates an example with a regressor model as the top model attached to the CNN output. The regressor model consists of a fully-connected (FC) layer with 2048 neurons and a ReLU activation function, followed by a fork that creates two branches. The first branch contains a fully-connected layer with 3 neurons and a linear activation predicting position, while the second branch has a fully-connected layer with 4 neurons and a linear activation and outputs a quaternion that corresponds to the orientation prediction.

3.4.2 Spatial LSTM

Despite LSTMs being mostly used for processing temporal sequences, recent work started showing that the memory capability of the LSTM can be advantageous for encoding contextual information. For example, Visin et al. in [111] replaced convolutional layers with RNNs sweeping across spatial dimensions and showed that this type of architecture is a viable alternative to CNNs, but further investigation is required. In

[112–114] authors successfully applied spatial LSTM for person re-identification, scene labeling and semantic object object parsing.

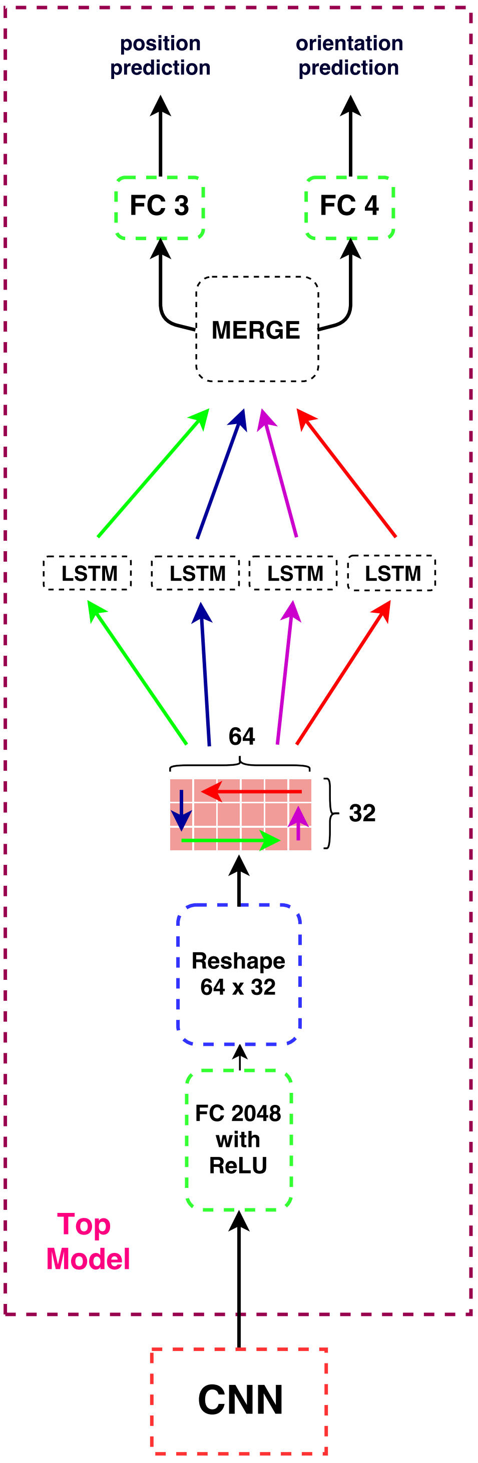


FIGURE 3.2: Spatial-LSTM model

The proposed Spatial-LSTM model is a modification to the original PoseNet model proposed by Walch et al. in [11]. Instead of directly regressing position and orientation from the output of the first fully-connected layer (FC 2048), we add an additional layer, which reshapes this output into a 64×32 matrix. This matrix can be then processed by an LSTM along different directions. For example, if we treat rows from this matrix as sequences, we can then apply a top-down and bottom-up LSTMs. Furthermore, treating the columns as sequences allows us to employ left-right and right-left LSTMs. Figure 3.2 depicts architecture of the proposed Spatial-LSTM model with four-way LSTM processing. For simplicity, the figure illustrates CNN as a single block, but is treated in the same way as in the Regressor model, where only a small part of it is finetuned.

3.4.3 Temporal GRU

The methods described in sections 3.4.2 and 3.4.1 process a single image at a time. However, one can easily notice that it should be beneficial to process images originating from videos and exploit the temporal smoothness between consecutive frames. For example, if two images taken from different locations look very similar due to perceptual aliasing, then it might be impossible to correctly predict the location without a broader context. This context can be provided by recurrent neural networks, which have proven to work exceedingly well for these types of problems, such as video classification [115] or visual question answering [116]. More closely to our task of localization, PlaNet [104] used LSTMs to improve the joint prediction of multiple images from a photo album, while VidLoc [5] used short video clips in order to exploit even more sever temporal smoothness, which is present in videos. Our work follows closely the VidLoc model.

We base our model on GRUs (described in 2.4.3), as they have empirically proven us to perform better than LSTMs in this problem. Figure 3.3 depicts the architecture of our Temporal-GRU model. It depicts the unrolled model, which employs bidirectional structure, meaning that at a given point in time, the sequence is scanned backwards and forwards. The output of GRU units from both directions is then concatenated along the last dimension and passed to the second layer of GRUs. In the last step, the network branches out and predicts position and orientation in the same way as in the previously described models. All trainable parameters across all time-steps are shared.

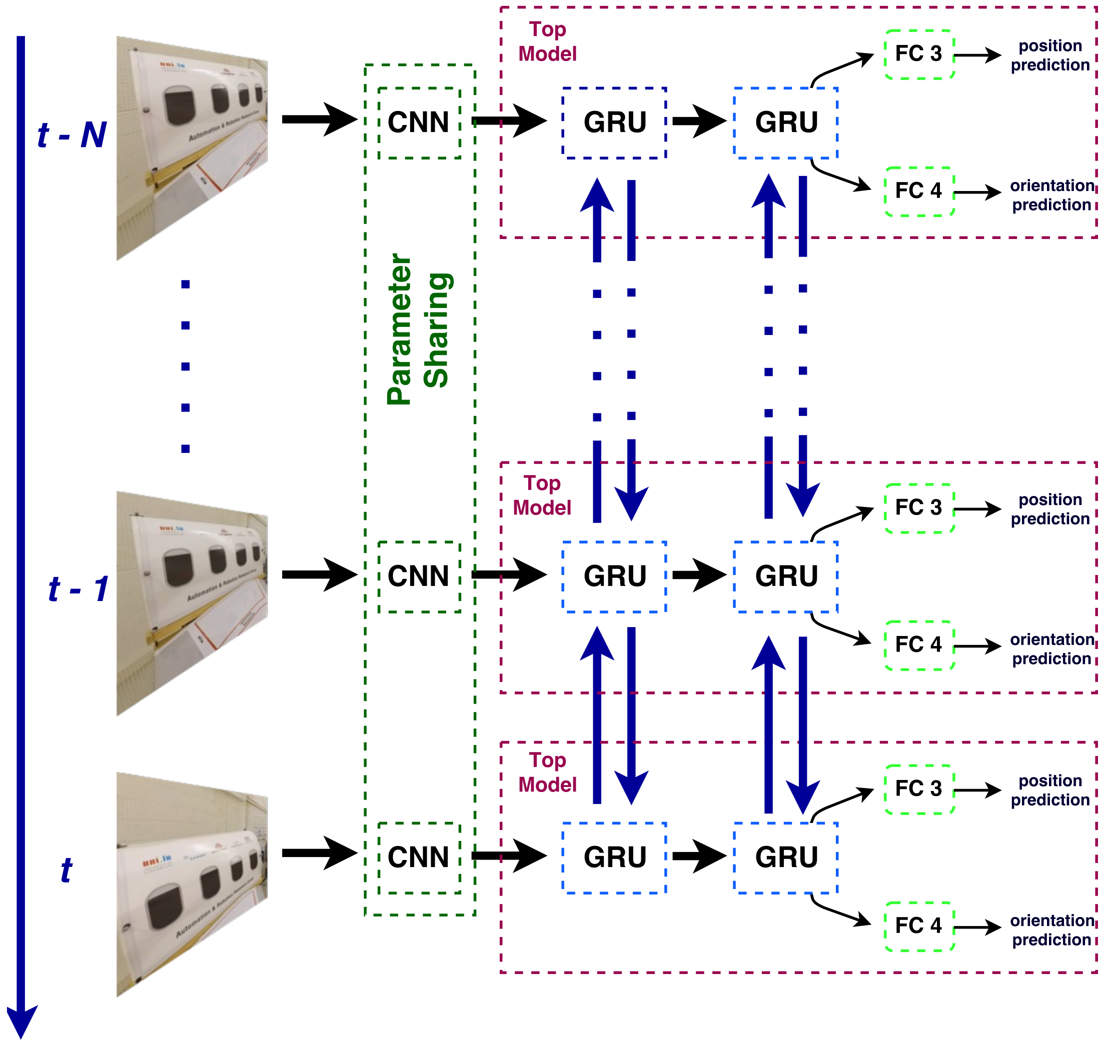


Figure 3.3: Unrolled Temporal-GRU model

Chapter 4

Experiments

In this chapter, we carry out an experimental evaluation of the methods introduced in 3. In section 4.1, we first introduce the datasets, which are the subject of this evaluation. In section 4.3, we show the results of the proposed models on the 7*Scenes* dataset, while in section 4.4 we show the results on the novel *Airframe* dataset. Throughout this section Spatial-LSTM refers to the model defined in section 3.4.2, while Regressor refers to the model defined in section 3.4.1. QEH and NH denote different loss function used in conjunction with the model architecture, where the first one corresponds to Quaternion-Error-Homoscedastic (see 3.3.3) and the second one to Naive-Homoscedastic (see 3.3.2).

4.1 Datasets

This section presents two datasets used in the experiments. 7*Scenes* is an indoor dataset published by Microsoft and introduced in [12] together with the SCoRe Forest algorithm. We also propose a new challenging dataset called *Airframe*, where the goal is to perform localization with respect to a dynamic airframe model.

4.1.1 7Scenes

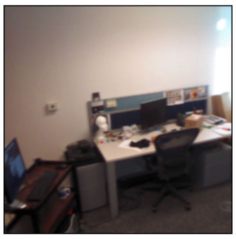
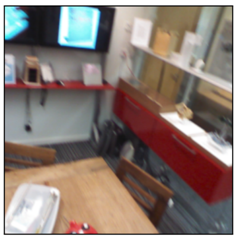
7*Scenes* dataset [12] comprises of seven distinct indoor datasets of RGB-D images and their corresponding 6-DOF poses. The poses are recorded in form of $SE(3)$

Table 4.1: 7Scenes dataset summary

	# images		Spatial	Volume
Scenes	Train	Test	Extent [m]	[m^3]
Chess	4000	2000	$3 \times 2 \times 1$	6
Fire	2000	2000	$2.5 \times 0.5 \times 1$	1.25
Heads	1000	1000	$2 \times 0.5 \times 1$	1
Office	6000	4000	$2.5 \times 2 \times 1.5$	7.5
Pumpkin	4000	2000	$2.5 \times 2 \times 1$	5
Red Kitchen	7000	5000	$4 \times 3 \times 1.5$	18
Stairs	2000	1000	$2.5 \times 2 \times 1.5$	7.5

transformations. Each dataset consists of multiple training and testing sequences with each sequence amounting to 1000 images. Table 4.1 depicts the summary of the 7*Scenes* dataset, showing the spatial extent, volume and number of training and testing images for each scene. Figure 4.2 presents some sample images from the dataset. The dataset was gather by several people using a hand-held Kinect RGB-D camera producing images with 640x480 resolution. Ground-truth data was obtained using KinectFusion system. The images exhibit specularities, motion-blur, flat-surfaces, varying lighting conditions and ambiguities, such as repeated steps in Stairs scene. The dataset does not provide any calibration files.

Table 4.2: Example images from 7Scenes dataset

Chess	**Fire**	**Heads**	**Office**
			
			
Pumpkin	**Red kitchen**		**Stairs**

4.1.2 Airframe

We introduce a new indoor dataset called *Airframe* depicting an airframe model built in the Automation & Robotics Research Group's [117] lab at University of Luxembourg. The dataset is composed of 6 subsets, each containing RGB images of the same airframe model, but appearing in a different position and orientation. The images map to camera poses with respect to the airframe model and are recorded as tuples $\boldsymbol{p} = (\boldsymbol{x}, \boldsymbol{q})$, where $\boldsymbol{x}$ is the translation and $\boldsymbol{q}$ is the quaternion.

The images were captured using a DJI M100 [118] aerial robot equipped with a gyro-stabilized Zenmuse X3 camera flying around the airframe model. We recorded RGB images at 1280×720 resolution using Robot Operating System (ROS) [119], while the ground-truth data was gathered using high-precision motion-capture OptiTrack system [120]. In order to collect the data and synchronize images with their corresponding poses, we have developed a collection of ROS packages:

- **extract-and-sync** is a tool that allows to sync image messages from a rosbag with a desired **tf** transformation [121]. The package is publicly available at: `https://github.com/asiron/extract-and-sync`.
- **dji_sdk_utils** is a collection of ROS packages, which adds additional functionality to DJI aerial robots, such as adding **tf** transformation tree of the camera gimbal, converting GPS messages into a local Euclidean coordiante frame using WGS-84 standard. The package is publicly available from `https://github.com/snt-robotics/dji_sdk_utils/tree/master/dji_rtk_tools`

Figure 4.1 illustrates the aerial robot used for gathering the dataset and figure 4.3 depicts position and orientation of the airframe model in all subsets of the *Airframe* dataset. We name these subsets **Position 1**, **Position 2**, ... Finally, we create two datasets from the original set and call them **airframe-mixed** and **airframe-ind** respectively. The first one contains training and testing images from all 6 positions, while the second one is significantly more difficult as positions 1 through 5 are designated for training only, leaving the last sixth position as the only testing sequence. This means, that if the model performs well on **airframe-ind**[1] then it has truly learn how to localize

[1]**airframe-ind**, where "ind" stands for independent due to last sequence being entirely independent from all other training sequences.

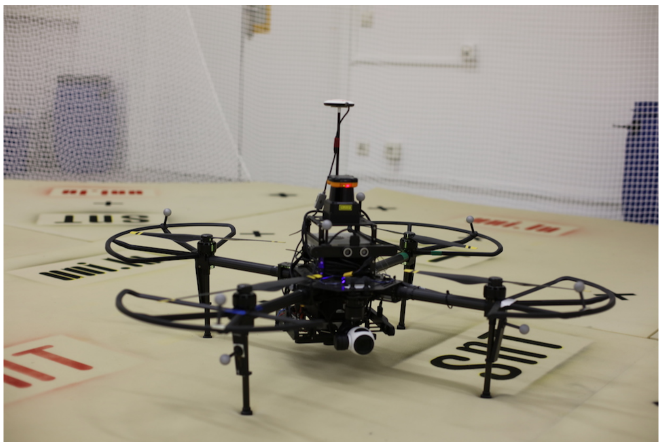
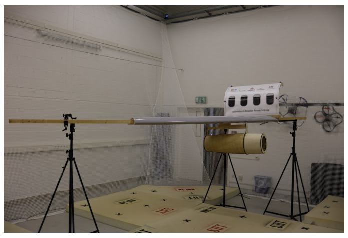
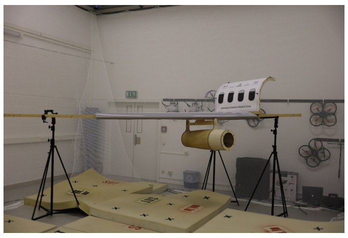
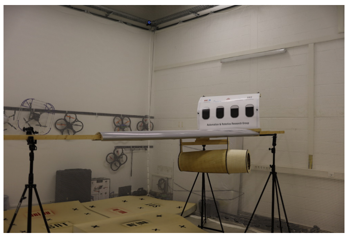
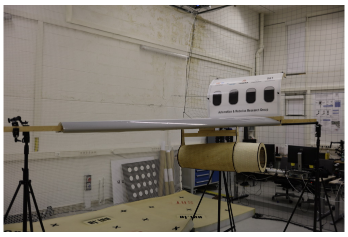
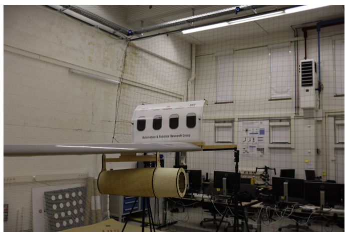
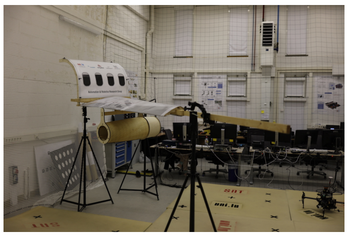


FIGURE 4.1: M100 with Manifold and Zenmuse X3 camera

TABLE 4.3: 6 airframe positions in Airframe dataset

Position 1	**Position 2**	**Position 3**
$\boldsymbol{p} = (-0.405, 2.355, 1.621)$ $\boldsymbol{q} = (-0.010, 0.014, 0.870, -0.493)$	$\boldsymbol{p} = (0.616, 2.980, 1.623)$ $\boldsymbol{q} = (0.013, -0.012, -0.707, 0.707)$	$\boldsymbol{p} = (1.799, 1.165, 1.625)$ $\boldsymbol{q} = (-0.007, 0.022, 0.983, -0.182)$
		
		
Position 4	**Position 5**	**Position 6**
$\boldsymbol{p} = (0.801, -1.214, 1.625)$ $\boldsymbol{q} = (0.005, 0.017, 0.769, 0.639)$	$\boldsymbol{p} = (0.798, -1.796, 1.628)$ $\boldsymbol{q} = (0.003, 0.018, 0.788, 0.615)$	$\boldsymbol{p} = (2.262, -0.985, 1.629)$ $\boldsymbol{q} = (-0.005, 0.011, 0.972, 0.234)$

itself with respect to that object without relying on any visual clues present in the environment. It means that it had learned the object detection task and ignores all other objects. Table 4.4 shows the summary of training and testing splits for both **airframe-mixed** and **airframe-ind**. Lastly, the spatial extent of the *Airframe* dataset is $6.7 \times 5.1 \times 5 \approx 170\ [m^3]$.

TABLE 4.4: Airframe dataset summary

	airframe-mixed		**airframe-ind**	
	# images		# images	
	train	test	train	test
Position 1	3301	918	4219	0
Position 2	1451	1087	2538	0
Position 3	1733	809	2542	0
Position 4	1120	385	1505	0
Position 5	2002	670	2672	0
Position 6	3792	1628	0	5420
Sum	13399	5497	13476	5420
Total	18896			

4.2 Training methodology

The proposed models have been implemented in Keras [42] (a high-level API of TensorFlow [39]) and are publicly available at `https://github.com/snt-robotics/camera_relocalization`. The repository contains Python scripts for:

- extracting and storing CNN features used during training (as described in section 3.4.1)
- computing necessary mean files
- creation of image sequences for training Temporal-GRU models
- preprocessing of 7Scenes data
- computation of saliency maps
- training and evaluating models

When training all our models, we used **Adam** optimizer (see 2.2.6) with default hyperparameters: $\beta_1 = 0.9, \beta_2 = 0.999$ and$\epsilon = 1 \times 10^{-8}$. We split the training set into training and validation sets in order to perform random hyperparameter search. We found out that dropout and L2 regularization actually reduced the performance. Therefore, we set dropout to $p = 0$ and L2 regularization strength to $\lambda = 0$. Lastly, we found the optimal learning rate to be $\alpha = 2 \times 10^{-4}$. We trained the models using NVIDIA GeForce GTX 950 GPU with 2GB of VRAM and a 12-core Intel Xeon E5645 CPU with 12 GB of RAM. Due to the low amount of accessible VRAM, we were not able to test our models to their full potential.

Table 4.5: SpatialLSTM and Regressor results on 7Scenes dataset using GoogLeNet pretrained on ImageNet

Data set	**Method**	**median**		**mae**		**max**		**std**	
		pos	orien	pos	orien	pos	orien	pos	orien
Chess	Spatial-LSTM,QEH	**0.157**	**6.95**	0.192	**8.29**	0.633	**27.13**	0.122	**5.36**
	Spatial-LSTM,NH	0.159	9.85	**0.188**	21.40	**0.618**	93.69	**0.121**	23.93
	Regressor,QEH	0.196	7.21	0.244	8.69	0.828	29.49	0.159	5.58
	Regressor,NH	0.166	9.73	0.200	21.93	0.664	94.11	0.131	24.40
Fire	Spatial-LSTM,QEH	0.325	**12.72**	**0.364**	**18.09**	**1.044**	**99.41**	**0.232**	**17.78**
	Spatial-LSTM,NH	0.342	15.49	0.390	24.50	1.055	179.83	0.234	27.38
	Regressor,QEH	0.321	12.92	0.379	19.08	1.286	106.79	0.234	19.28
	Regressor,NH	**0.319**	38.05	0.370	46.03	1.131	128.75	0.237	36.89
Heads	Spatial-LSTM,QEH	**0.164**	14.79	**0.191**	**16.06**	0.565	**50.00**	**0.103**	**9.09**
	Spatial-LSTM,NH	0.172	**13.91**	0.203	19.12	**0.552**	82.82	0.112	16.62
	Regressor,QEH	0.178	15.10	0.211	16.77	0.616	52.15	0.113	9.48
	Regressor,NH	0.178	19.89	0.202	28.33	0.572	169.02	0.100	22.05
Office	Spatial-LSTM,QEH	**0.233**	8.67	**0.300**	11.73	**1.221**	90.33	0.225	10.56
	Spatial-LSTM,NH	0.243	8.58	0.301	11.49	1.324	178.19	**0.217**	12.49
	Regressor,QEH	0.271	**8.31**	0.348	**11.07**	1.444	**81.44**	0.257	**9.88**
	Regressor,NH	0.327	9.04	0.393	15.75	1.503	179.86	0.250	22.961
Pumpkin	Spatial-LSTM,QEH	**0.332**	**20.90**	**0.383**	**52.06**	**1.209**	179.97	**0.234**	**60.43**
	Spatial-LSTM,NH	0.339	32.28	0.437	63.71	1.512	179.88	0.304	61.91
	Regressor,QEH	0.334	48.96	0.438	70.26	1.466	179.94	0.304	62.06
	Regressor,NH	0.423	39.85	0.475	68.81	1.352	179.94	0.271	63.19
Red kitchen	Spatial-LSTM,QEH	**0.322**	**8.17**	**0.430**	**13.10**	2.776	**160.51**	**0.377**	**18.50**
	Spatial-LSTM,NH	0.374	9.15	0.503	15.48	3.09	179.35	0.447	27.50
	Regressor,QEH	0.373	8.40	0.493	13.86	2.215	161.10	0.381	19.67
	Regressor,NH	0.429	9.10	0.555	16.18	**2.198**	179.52	0.387	24.58
Stairs	Spatial-LSTM,QEH	0.365	12.99	**0.359**	**13.51**	1.033	**29.67**	0.158	**6.41**
	Spatial-LSTM,NH	**0.363**	43.06	0.363	44.29	**0.96**	112.15	**0.151**	20.75
	Regressor,QEH	0.424	14.80	0.493	14.65	1.748	37.32	0.287	7.74
	Regressor,NH	0.434	**12.07**	0.505	13.83	1.799	117.97	0.299	8.29

4.3 Results on 7Scenes dataset

In this section, we present results on 7*Scenes* dataset using our proposed models.

Table 4.5 depicts absolute errors of proposed models on 7*Scenes* dataset using GoogLeNet pretrained on ImageNet network as a base feature extractor. We can notice, that QEH loss function almost always performs better than NH loss function, especially when it comes orientation error, which is expected. Moreover, QEH also decreases position error drastically, for example in Red Kitchen scene, it improves the median position error by approximately 5cm for both Spatial-LSTM and Regressor model types.

Places365 is a dataset for scene recognition task. Therefore, as shown in section 2.5, we should expect to get better results, when the CNN was pretrained on this dataset, because localization task is "closer" to scene recognition than image classification. However, when we compare results from ImageNet (table 4.5) to Places365 (table 4.6), we see that except for Red Kitchen median position error is lower for ImageNet

TABLE 4.6: SpatialLSTM and Regressor results on 7Scenes dataset using GoogLeNet pretrained on Places365

Data set	**Method**	**median**		**mae**		**max**		**std**	
		pos	orien	pos	orien	pos	orien	pos	orien
Chess	Spatial-LSTM,QEH	**0.177**	**6.41**	**0.219**	8.06	**0.628**	30.15	**0.135**	5.80
	Spatial-LSTM,NH	0.243	8.21	0.284	13.74	0.879	176.34	0.161	18.20
	Regressor,QEH	0.203	6.53	0.242	**7.87**	0.734	**28.10**	0.155	**5.43**
	Regressor,NH	0.183	9.45	0.222	21.27	0.775	93.32	0.142	23.98
Fire	Spatial-LSTM,QEH	**0.331**	**13.14**	**0.366**	**16.70**	1.276	111.20	**0.209**	**16.40**
	Spatial-LSTM,NH	0.346	38.08	0.383	46.24	1.406	136.64	0.236	37.74
	Regressor,QEH	0.362	15.01	0.405	19.34	1.369	**108.19**	0.231	17.62
	Regressor,NH	0.379	35.96	0.417	43.75	**1.265**	129.78	0.239	36.53
Heads	Spatial-LSTM,QEH	0.180	14.40	0.208	**17.23**	0.769	52.02	0.112	9.22
	Spatial-LSTM,NH	**0.167**	**13.77**	**0.189**	18.87	**0.592**	81.05	**0.089**	17.09
	Regressor,QEH	0.183	14.24	0.222	16.78	0.693	**43.45**	0.120	**8.32**
	Regressor,NH	0.180	34.52	0.212	42.85	0.694	122.04	0.115	28.94
Office	Spatial-LSTM,QEH	**0.253**	**7.97**	**0.310**	**10.53**	**1.135**	142.59	**0.197**	11.79
	Spatial-LSTM,NH	0.267	8.12	0.336	12.65	1.604	179.47	0.219	18.95
	Regressor,QEH	0.289	8.03	0.356	11.02	1.230	**102.30**	0.231	**10.64**
	Regressor,NH	0.307	8.25	0.374	14.40	1.309	175.96	0.238	21.32
Pumpkin	Spatial-LSTM,QEH	**0.328**	17.84	**0.398**	45.90	1.611	180.00	0.292	56.24
	Spatial-LSTM,NH	0.362	24.21	0.437	56.89	**1.273**	179.88	**0.271**	62.50
	Regressor,QEH	0.360	**10.52**	0.434	**15.38**	1.607	**73.39**	0.281	**13.05**
	Regressor,NH	0.361	26.41	0.460	65.20	1.475	179.99	0.316	64.99
Red kitchen	Spatial-LSTM,QEH	**0.299**	**7.44**	**0.401**	11.84	2.054	159.87	0.303	14.47
	Spatial-LSTM,NH	0.323	8.63	0.414	**11.10**	2.252	178.52	**0.297**	**11.56**
	Regressor,QEH	0.370	7.99	0.498	13.64	**1.818**	**116.55**	0.372	16.74
	Regressor,NH	0.348	8.60	0.486	13.07	1.833	177.91	0.364	18.55
Stairs	Spatial-LSTM,QEH	0.392	13.57	0.448	**13.73**	1.399	53.27	0.199	**6.89**
	Spatial-LSTM,NH	**0.390**	**12.15**	0.462	19.95	**1.341**	94.89	0.218	19.70
	Regressor,QEH	0.406	14.11	**0.447**	14.20	1.427	**50.76**	**0.192**	7.22
	Regressor,NH	0.419	40.68	0.458	43.65	1.436	110.95	0.192	19.53

pretrained model. This could be caused by the fact that we are finetuning the last 3 inception blocks of GoogLeNet, at which point all classification specific features may have disappeared.

Table 4.13 illustrates results of our proposed models when trained using Inception ResNet V2 pretrained on ImageNet as the base feature extractor. As shown, in the previous examples the difference in median angular error between QEH and NH is clearly visible. For example, for Fire scene QEH achieves approximately $\sim 15^\circ$ median error for both Spatial-LSTM and Regressor models , while NH barely gets below $\sim 40^\circ$. This phenomena is visible across the results. Even if NH loss function coincidently outperforms QEH it is never by a huge margin.

Table 4.8 displays results from VGG16 pretrained on Hybrid1365 dataset. This dataset is very interesting, because it combines the ImageNet dataset with the Places365

TABLE 4.7: SpatialLSTM and Regressor results on 7Scenes datasets using Inception ResNet V2 pretrained on ImageNet

Data set	**Method**	**median**		**mae**		**max**		**std**	
		pos	orien	pos	orien	pos	orien	pos	orien
Chess	Spatial-LSTM,QEH	0.164	**7.36**	0.209	**8.67**	**0.838**	**42.41**	0.149	**5.95**
	Spatial-LSTM,NH	**0.162**	72.07	**0.205**	72.36	0.869	179.88	**0.142**	45.06
	Regressor,QEH	0.197	7.78	0.253	9.59	1.129	46.45	0.181	6.61
	Regressor,NH	0.209	13.32	0.255	23.68	1.015	92.07	0.167	23.07
Fire	Spatial-LSTM,QEH	0.344	**15.28**	0.401	**20.10**	1.137	101.21	0.231	**17.08**
	Spatial-LSTM,NH	**0.333**	37.49	**0.369**	47.85	1.214	143.94	0.208	38.25
	Regressor,QEH	0.354	16.03	0.384	20.66	**0.986**	**87.25**	0.203	17.10
	Regressor,NH	0.365	40.16	0.398	48.67	1.053	147.81	**0.202**	38.29
Heads	Spatial-LSTM,QEH	0.358	**16.57**	0.400	**18.57**	0.992	53.47	0.212	9.55
	Spatial-LSTM,NH	0.398	21.18	0.484	23.99	1.040	61.04	0.229	12.49
	Regressor,QEH	0.262	17.05	0.284	18.95	0.874	**52.13**	0.140	**9.39**
	Regressor,NH	**0.201**	37.56	**0.227**	42.94	**0.739**	119.32	**0.128**	28.02
Office	Spatial-LSTM,QEH	**0.245**	**8.03**	**0.308**	**11.19**	1.351	105.26	**0.215**	**10.50**
	Spatial-LSTM,NH	0.253	9.10	0.315	15.24	1.458	178.47	0.220	22.35
	Regressor,QEH	0.282	8.54	0.346	11.49	1.351	**74.88**	0.231	10.49
	Regressor,NH	0.291	8.97	0.355	17.18	**1.335**	179.77	0.234	26.24
Pumpkin	Spatial-LSTM,QEH	0.446	50.47	0.473	72.16	**1.205**	179.99	**0.250**	59.96
	Spatial-LSTM,NH	0.448	51.05	0.500	70.02	1.497	179.91	0.286	56.44
	Regressor,QEH	**0.322**	**19.49**	**0.408**	**36.30**	1.463	179.89	0.290	**42.36**
	Regressor,NH	0.378	51.22	0.458	65.79	1.418	179.89	0.286	52.13
Red kitchen	Spatial-LSTM,QEH	**0.358**	9.19	**0.476**	14.70	2.179	143.70	0.372	19.88
	Spatial-LSTM,NH	0.366	**8.56**	0.483	**11.29**	**1.728**	174.74	**0.333**	**11.68**
	Regressor,QEH	0.353	8.93	0.482	14.32	2.468	**119.91**	0.378	16.36
	Regressor,NH	0.366	15.44	0.491	38.14	2.440	179.98	0.380	44.68
Stairs	Spatial-LSTM,QEH	**0.345**	14.69	0.559	16.16	1.934	50.51	0.441	**8.92**
	Spatial-LSTM,NH	0.350	11.92	0.571	19.40	1.928	86.96	0.450	19.38
	Regressor,QEH	0.346	15.05	0.429	**15.94**	1.813	**43.93**	0.297	9.59
	Regressor,NH	0.361	**11.70**	**0.422**	19.56	**1.717**	90.77	**0.251**	20.41

dataset [2]. It was also shown by Zhou et al. in [110] that a VGG16 pretrained on this dataset achieves great results on other smaller datasets. In fact, the VGG16-Hybrid1365 model combined with a Spatial-LSTM and QEH loss function gives us the best results for 5 out of 7 scenes compared to the previously shown architectures.

4.3.1 Summary

In this section, we present a summary to the experimental evaluation of the proposed models on 7*Scenes* dataset. We compare our results with the very recent state-of-the-art techniques. Table 4.9 illustrates a comparison between PoseNet [4], spatial LSTM introduced by Walch et al. in [11] and an improved PoseNet [7], that learns the optimal β by means of homoscedastic uncertainty loss function.

[2]ImageNet contains 1000 classes, which added to 365 classes from Places365 dataset form a hybrid dataset with 1365 classes

Table 4.8: SpatialLSTM and Regressor results on 7Scenes dataset using VGG16 pretrained on Hybrid1365

Data set	**Method**	*median*		*mae*		*max*		*std*	
		pos	orien	pos	orien	pos	orien	pos	orien
Chess	Spatial-LSTM,QEH	0.148	**5.261**	0.179	7.033	**0.674**	**24.63**	**0.121**	4.972
	Spatial-LSTM,NH	**0.137**	7.868	**0.172**	19.71	0.749	95.00	0.121	24.73
	Regressor,QEH	0.188	5.805	0.225	**6.621**	0.678	28.70	0.143	**4.293**
	Regressor,NH	0.197	8.157	0.246	19.68	0.715	97.09	0.171	24.27
Fire	Spatial-LSTM,QEH	**0.272**	**10.62**	**0.301**	**14.49**	**0.826**	**72.90**	0.186	**12.99**
	Spatial-LSTM,NH	0.281	35.42	0.318	45.86	0.904	135.03	**0.180**	39.99
	Regressor,QEH	0.432	12.884	0.497	17.08	1.327	86.68	0.275	15.58
	Regressor,NH	0.459	15.95	0.546	20.90	1.277	179.52	0.292	20.02
Heads	Spatial-LSTM,QEH	**0.177**	14.53	**0.200**	**14.83**	**0.503**	**36.26**	**0.103**	**7.08**
	Spatial-LSTM,NH	0.209	14.10	0.236	18.92	0.712	80.77	0.131	16.54
	Regressor,QEH	0.215	14.54	0.250	15.03	0.692	38.46	0.132	7.40
	Regressor,NH	0.227	**14.05**	0.250	18.83	0.728	78.16	0.126	16.43
Office	Spatial-LSTM,QEH	**0.212**	7.83	0.268	10.18	1.122	**81.02**	0.186	**9.35**
	Spatial-LSTM,NH	0.215	**6.89**	**0.267**	**9.40**	**0.968**	166.38	**0.185**	10.70
	Regressor,QEH	0.271	6.76	0.343	9.35	1.407	80.40	0.243	8.90
	Regressor,NH	0.317	7.03	0.371	9.45	1.422	178.05	0.241	12.52
Pumpkin	Spatial-LSTM,QEH	**0.264**	18.33	**0.353**	43.43	1.183	179.93	**0.270**	51.52
	Spatial-LSTM,NH	0.401	21.93	0.442	51.79	1.343	179.99	0.277	57.07
	Regressor,QEH	0.340	**10.66**	0.464	**15.15**	1.483	**63.60**	0.327	**11.98**
	Regressor,NH	0.403	25.90	0.491	53.63	**1.345**	180.00	0.294	55.62
Red kitchen	Spatial-LSTM,QEH	**0.291**	7.04	**0.403**	**11.83**	2.607	150.60	**0.360**	**18.89**
	Spatial-LSTM,NH	0.295	11.55	0.424	34.27	2.339	167.53	0.376	41.05
	Regressor,QEH	0.371	**6.93**	0.536	12.88	**2.310**	**154.93**	0.430	21.41
	Regressor,NH	0.422	8.55	0.589	15.61	2.52	179.85	0.456	29.66
Stairs	Spatial-LSTM,QEH	0.336	**11.79**	**0.336**	**11.87**	0.883	**26.66**	**0.126**	**5.29**
	Spatial-LSTM,NH	**0.330**	44.32	0.354	45.09	**0.667**	111.57	0.139	21.34
	Regressor,QEH	0.388	13.12	0.434	13.03	1.581	27.84	0.223	5.85
	Regressor,NH	0.461	13.28	0.529	14.17	1.802	160.24	0.306	11.52

Table 4.9: 7Scenes summary results using VGG16 pretrained on Hybrid1365

7Scenes	PoseNet[4] (β weight)	PoseNet [11] spatial LSTM	PoseNet [7] Learn σ^2 Weight	This work Spatial-LSTM
Chess	0.32m, 6.60°	0.24m, 5.77°	0.14m, **4.50°**	**0.137m**, 7.868° *
Fire	0.47m, 14.0°	0.34m, 11.9°	**0.27m**, 11.8°	0.272m, **10.62°** †
Heads	0.30m, 12.2°	0.21m, 13.7°	0.18m, **12.1°**	**0.164m**‡ 14.79° †
Office	0.48m, 7.24°	0.30m, 8.08°	**0.20m**, **5.77°**	0.212m, 7.83° †
Pumpkin	0.49m, 8.12°	0.33m, 7.00°	**0.25m**, **4.82°**	0.264m, 18.33° †
Red Kitchen	0.58m, 8.34°	0.37m, 8.83°	**0.24m**, **5.52°**	0.291m, 7.04° †
Stairs	0.48m, 13.1°	0.40m, 13.7°	0.37m, **10.6°**	**0.336m**, 11.79° †

* Naive-Homoscedastic or NH

† Quaternion-Error-Homoscedastic or QEH

‡ CNN is GoogLeNet-ImageNet

We can see that even though, out of 16 convolutional layers in VGG16-Hybrid1365 model, we only finetune the last 3 layers, we are able to achieve the same or even better results. This is due to the fact, that we combined the Spatial-LSTM architecture with a homoscedastic loss function and also modified the orientation loss function to be a proper loss function based on quaternion algebra. Therefore, we are able to beat the improved PoseNet [7] by up to 2cm using a fraction of computational power required to train it.

The empirical cumulative distribution functions are shown below. They prove the importance of QEH loss function over NH loss function, by showing that the angular error is almost always better and sometimes, as it is in the case of Stairs scene, leads to a much better performance.

Table 4.10 displays saliency maps (described in 2.3.3) of the trained Spatial-LSTM model which uses QEH loss function and was finetuned on a VGG16-Hybrid1365. We can notice that the network has learned to recognize distinct objects in the scene and uses them estimate the camera pose.

4.4 Results on Airframe dataset

Tables 4.11, 4.12, 4.13 and 4.14 show the results of the proposed models: Spatial-LSTM and Regressor, using two loss functions: Quaternion-Error-Homoscedastic (QEH) and Naive-Homoscedastic (NH) on 4 different CNN base feature extractors same as in 7*Scenes* example. We can immediately see that VGG16 outperforms all other architectures. In fact, we are mostly interested in **airframe-ind** dataset as it is more demanding than **airframe-mixed**. For this more difficult dataset, VGG16-Hybrid1365 achieves highest score with Inception ResNet V2 right behind it. This shows, that even if an architecture performs worse on one dataset such as 7*Scenes*, it does not necessarily mean that it will perform the same across all datasets used for image-based localization task.

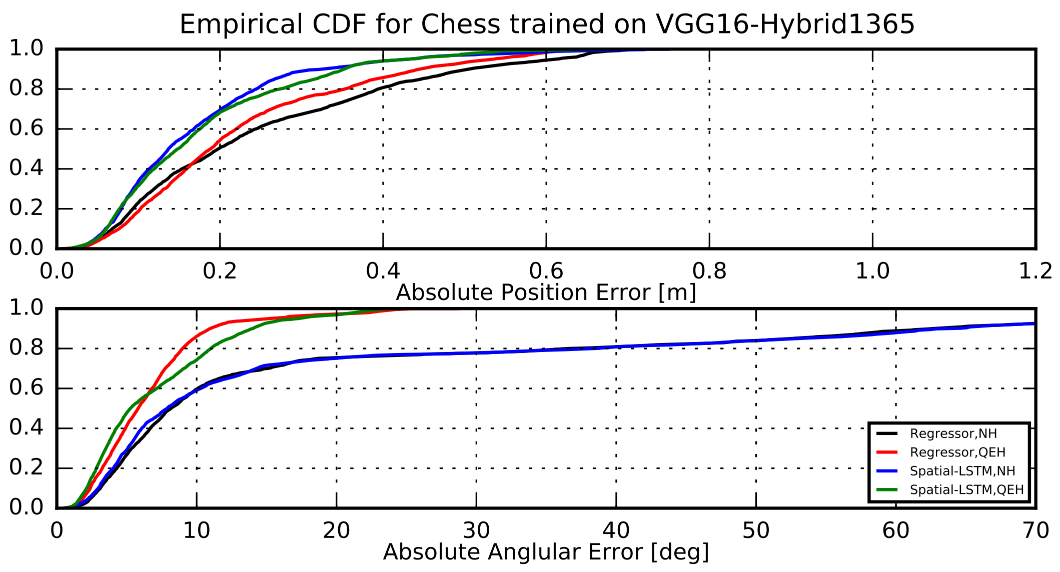
Figures 4.3 and 4.2 depict cumulative error histograms for the **airframe-mixed** and **airframe-ind**, where we see the same behavior as in 7*Scenes* dataset, meaning that QEH loss function outperforms NH loss function and Spatial-LSTM outperforms Regressor model.

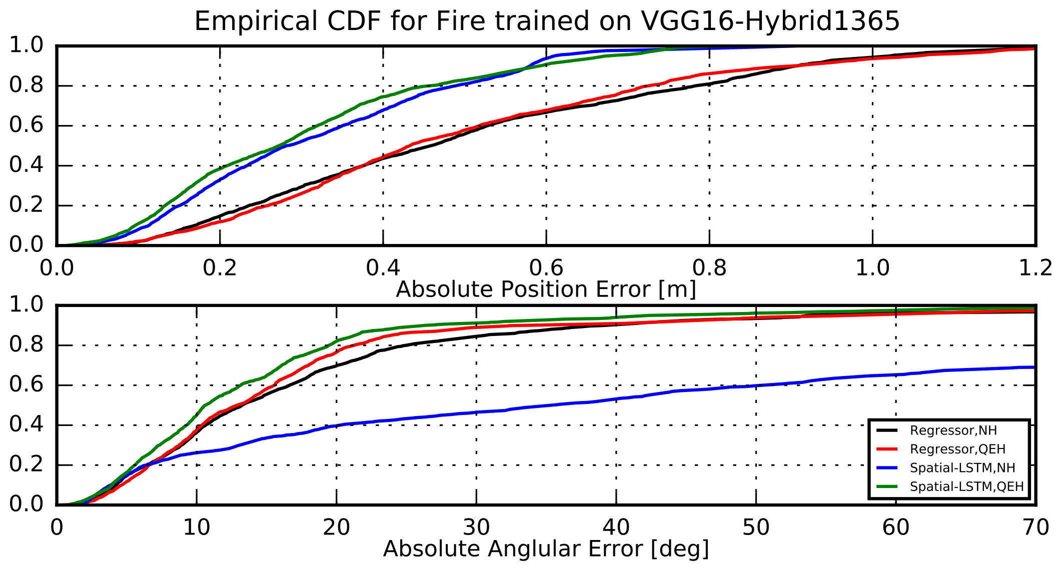
Figure 4.15 displays 6 selected trajectories for the **airframe-ind** dataset. The ground-truth trajectory is represented in green color and the prediction in red color. On the left-hand side, we can notice trajectories, where the model accurately predicted poses. However on the right-hand side, we see examples of predictions, which are far away from ground-truth or are very noisy.

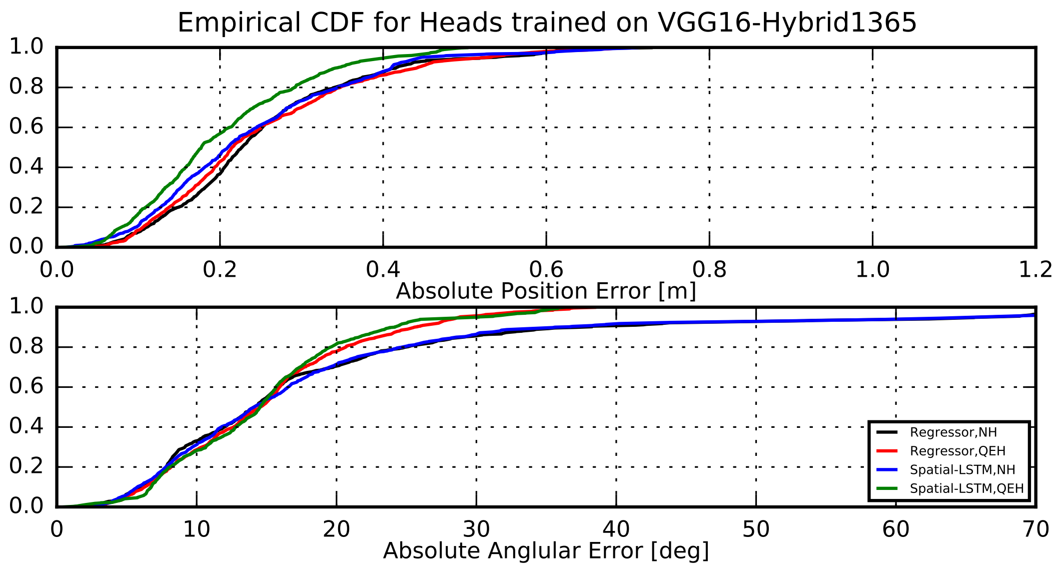
Figure 4.16 illustrates saliency maps for two selected images from the **airframe-ind** dataset. We should expect that in this case, the model would ignore the environment and only focus on the airframe model. However, that is not exactly the case, since some parts visible in the background are important to the network.

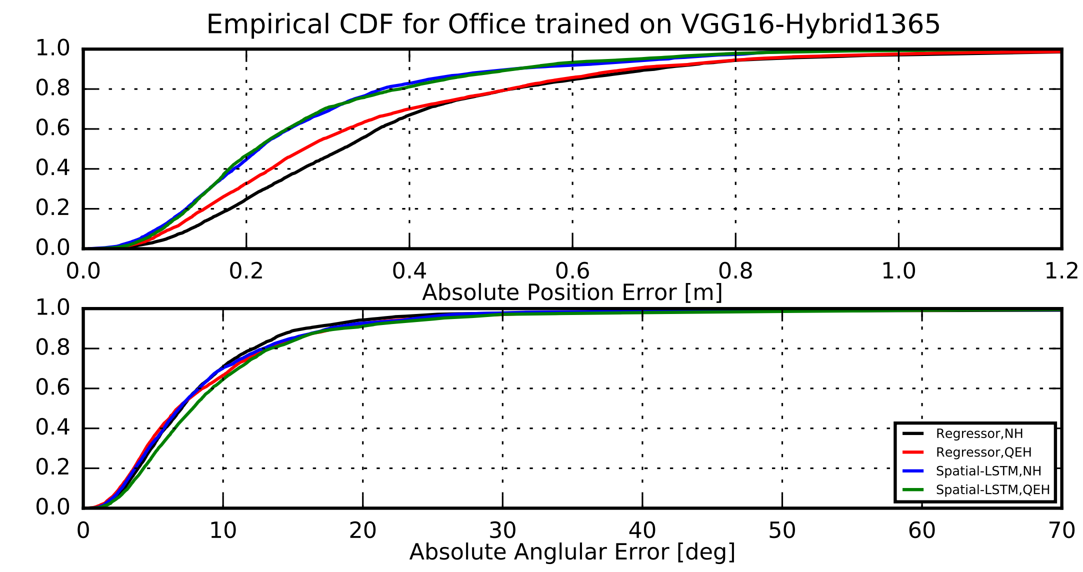
4.4.1 Temporal-GRU comparison

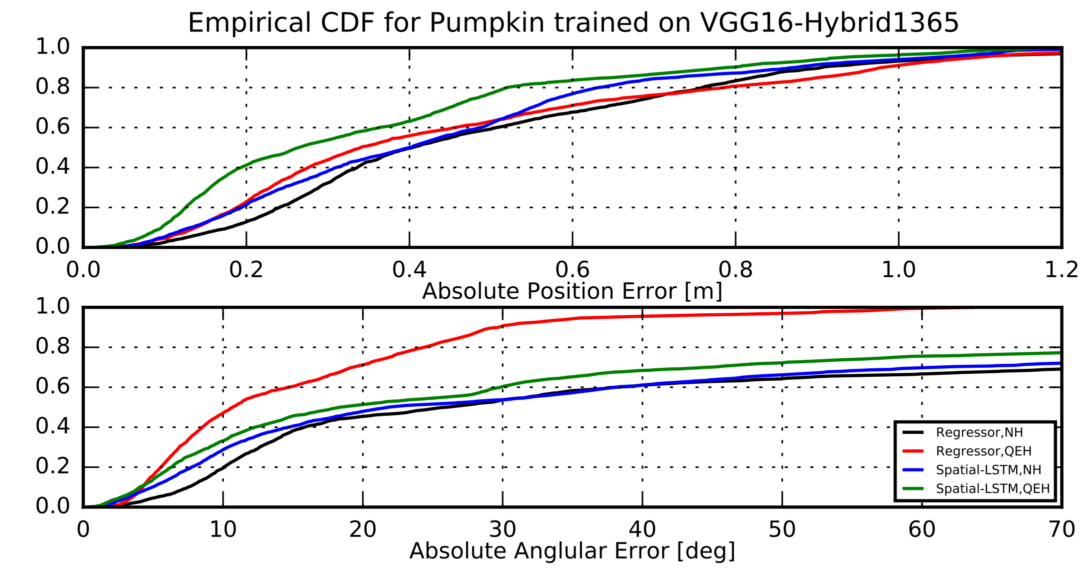
Lastly, we show a Temporal-GRU model, which works on sequences of images. However, due to resource constraints, we had to limit ourselves to sequences of 2 images, as larger sequences were not possible to train, due to not enough VRAM in the GPU. Figure 4.17 illustrates results for 3 network architectures: GoogLeNet-ImageNet, GoogLeNet-Places365 and VGG16-Hybrid1365. We use two loss functions as in previous examples QEH and NH. We found out that it is very difficult to train these temporal models. However, we were able to beat the previous best score of Spatial-LSTM with QEH trained on VGG16-Hybrid1365 model.

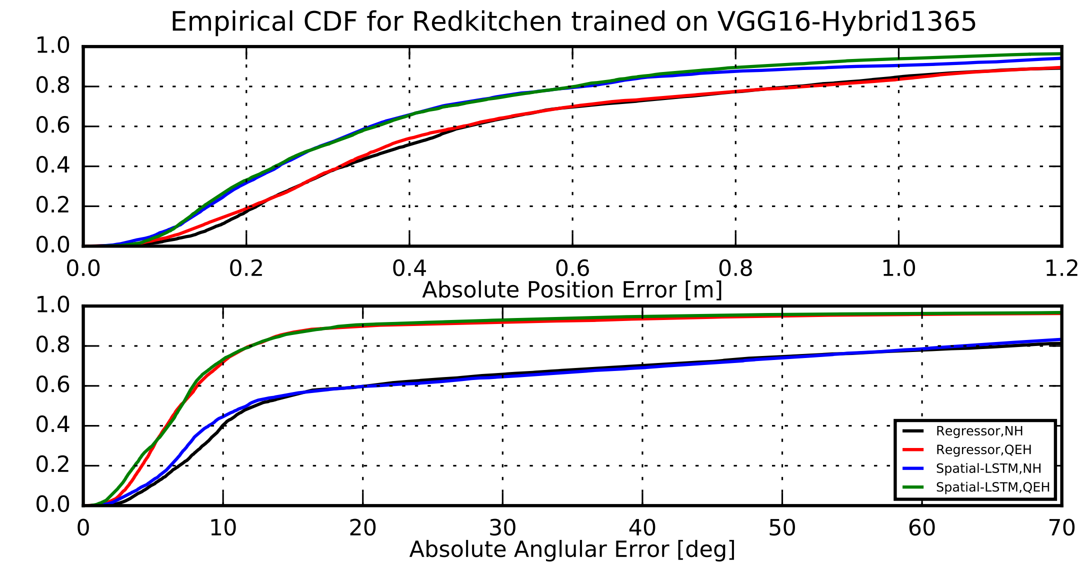

Empirical CDF for Chess trained on VGG16-Hybrid1365
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH


Empirical CDF for Fire trained on VGG16-Hybrid1365
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH


Empirical CDF for Heads trained on VGG16-Hybrid1365
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH


Empirical CDF for Office trained on VGG16-Hybrid1365
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH


Empirical CDF for Pumpkin trained on VGG16-Hybrid1365
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH


Empirical CDF for Redkitchen trained on VGG16-Hybrid1365
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

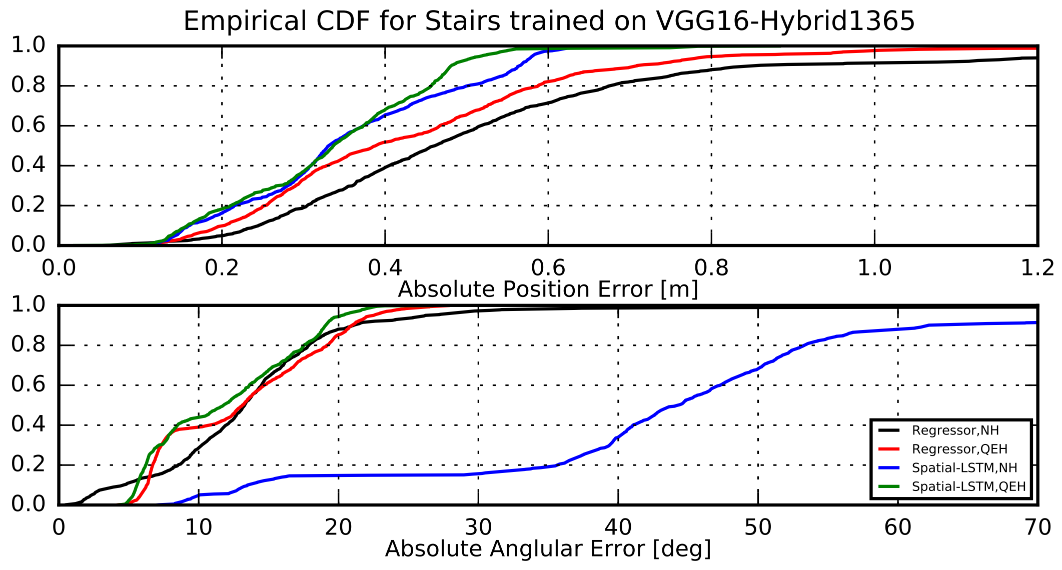

Empirical CDF for Stairs trained on VGG16-Hybrid1365
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1.2
Absolute Position Error [m]
0
10
20
30
40
50
60
70
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

Table 4.10: Saliency maps for 7Scenes dataset trained on Spatial-LSTM,QEH using VGG16-Hybrid1365

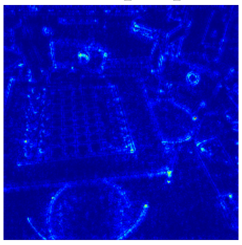
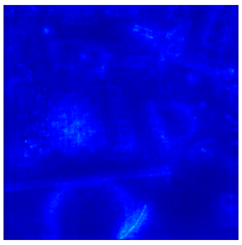
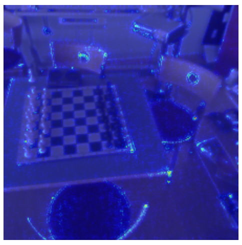
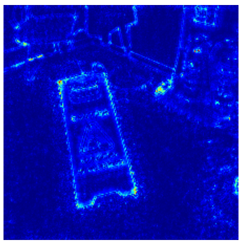
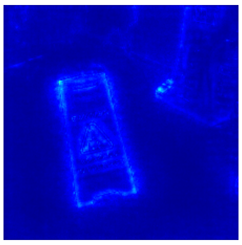
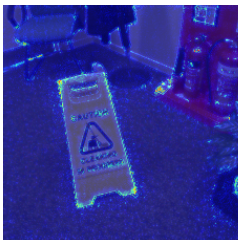
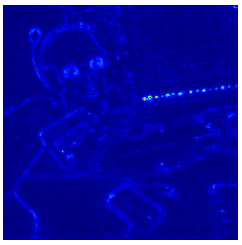
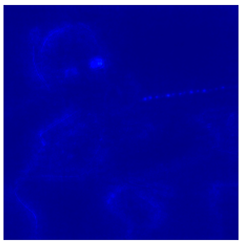
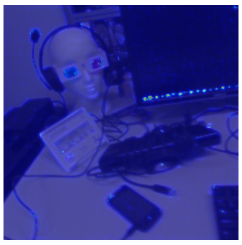
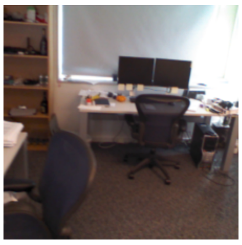
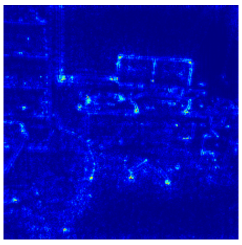
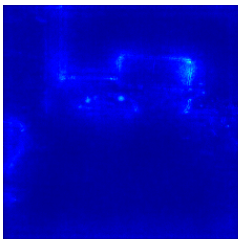
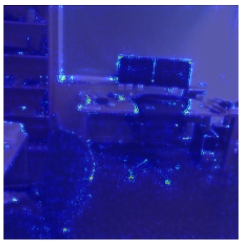
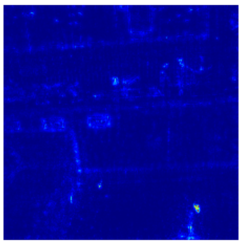
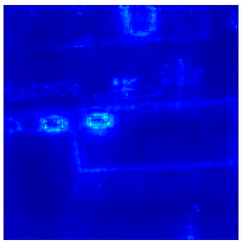
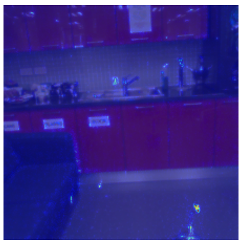
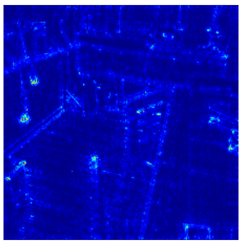
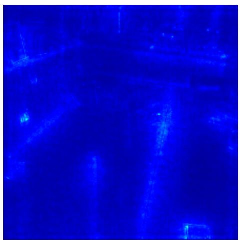
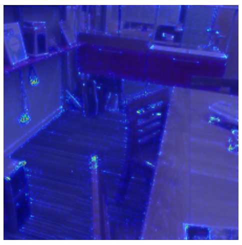
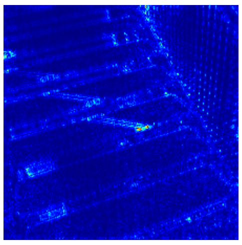
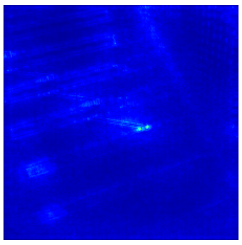
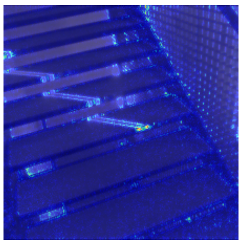
Original	**Guided Backprop**	**SmoothGrad**	**Overlaid**
			
			
			
			
			
			
			

Table 4.11: SpatialLSTM and Regressor results on Airframe datasets using GoogLeNet pretrained on ImageNet

Data set	**Method**	**median**		**mae**		**max**		**std**	
		pos	orien	pos	orien	pos	orien	pos	orien
airframe-mixed	Spatial-LSTM,QEH	**0.193**	**3.85**	**0.294**	**5.39**	**2.906**	165.31	**0.370**	**6.24**
	Spatial-LSTM,NH	0.259	5.69	0.355	9.88	2.921	177.70	0.387	16.37
	Regressor,QEH	0.264	3.91	0.407	5.82	3.379	164.41	0.454	7.02
	Regressor,NH	0.350	6.75	0.490	12.40	3.855	178.13	0.519	20.45
airframe-ind	Spatial-LSTM,QEH	**0.328**	**7.39**	**0.514**	**8.71**	**2.630**	**33.11**	**0.464**	**5.56**
	Spatial-LSTM,NH	0.418	13.44	0.594	15.76	2.712	90.73	0.467	10.65
	Regressor,QEH	0.444	7.95	0.694	9.22	3.191	30.78	0.643	5.60
	Regressor,NH	0.708	11.43	0.889	13.74	3.134	177.06	0.641	11.83

Table 4.12: SpatialLSTM and Regressor results on Airframe datasets using GoogLeNet pretrained on Places365

Data set	**Method**	**median**		**mae**		**max**		**std**	
		pos	orien	pos	orien	pos	orien	pos	orien
airframe-mixed	Spatial-LSTM,QEH	**0.279**	**4.75**	**0.421**	**6.91**	3.087	164.35	0.480	**7.88**
	Spatial-LSTM,NH	0.292	7.50	0.430	16.72	**2.802**	167.42	**0.454**	22.30
	Regressor,QEH	0.350	5.00	0.500	7.80	3.109	**163.70**	0.495	8.71
	Regressor,NH	0.399	6.30	0.583	11.95	3.226	179.61	0.552	16.82
airframe-ind	Spatial-LSTM,QEH	**0.545**	**10.41**	**0.655**	**11.72**	2.750	48.19	**0.408**	6.97
	Spatial-LSTM,NH	0.627	14.97	0.730	19.47	**2.477**	176.96	0.434	17.34
	Regressor,QEH	0.673	10.78	0.807	12.02	2.598	**47.14**	0.494	**6.75**
	Regressor,NH	0.906	17.09	0.978	21.25	2.686	173.46	0.541	15.88

Table 4.13: SpatialLSTM and Regressor results on Airframe datasets using Inception ResNet V2 pretrained on ImageNet

Data set	**Method**	**median**		**mae**		**max**		**std**	
		pos	orien	pos	orien	pos	orien	pos	orien
airframe-mixed	Spatial-LSTM,QEH	0.196	3.76	0.279	4.74	3.087	164.39	0.325	**4.65**
	Spatial-LSTM,NH	0.273	5.13	0.377	8.64	2.952	179.52	0.393	16.28
	Regressor,QEH	**0.194**	**3.37**	**0.287**	**4.69**	**2.086**	164.68	**0.284**	5.36
	Regressor,NH	0.265	5.86	0.435	11.10	2.898	179.55	0.474	20.48
airframe-ind	Spatial-LSTM,QEH	**0.282**	5.28	**0.417**	6.64	2.359	**32.98**	0.354	4.86
	Spatial-LSTM,NH	0.366	9.76	0.537	13.00	2.877	178.97	0.454	13.58
	Regressor,QEH	0.287	**4.63**	0.431	**5.88**	2.475	34.19	0.386	**4.74**
	Regressor,NH	0.391	6.78	0.500	11.95	**2.211**	179.27	**0.328**	17.71

Table 4.14: SpatialLSTM and Regressor results on Airframe datasets using VGG16 pretrained on Hybrid1365

Data set	**Method**	**median**		**mae**		max		std	
		pos	orien	pos	orien	pos	orien	pos	orien
airframe-mixed	Spatial-LSTM,QEH	**0.184**	4.22	**0.293**	5.69	2.600	164.51	0.340	**5.35**
	Spatial-LSTM,NH	0.229	5.66	0.32	8.75	**2.269**	179.44	**0.289**	12.59
	Regressor,QEH	0.229	**3.97**	0.358	**5.58**	2.733	163.81	0.370	6.03
	Regressor,NH	0.286	7.06	0.429	14.71	3.125	**163.22**	0.437	19.47
airframe-ind	Spatial-LSTM,QEH	**0.268**	**5.16**	**0.4206**	**6.53**	2.367	**28.86**	0.376	**4.59**
	Spatial-LSTM,NH	0.356	10.29	0.487	13.66	**1.747**	177.08	**0.337**	15.56
	Regressor,QEH	0.415	5.92	0.628	7.50	2.372	36.15	0.558	5.32
	Regressor,NH	0.439	50.12	0.588	57.23	2.301	179.81	0.473	35.12

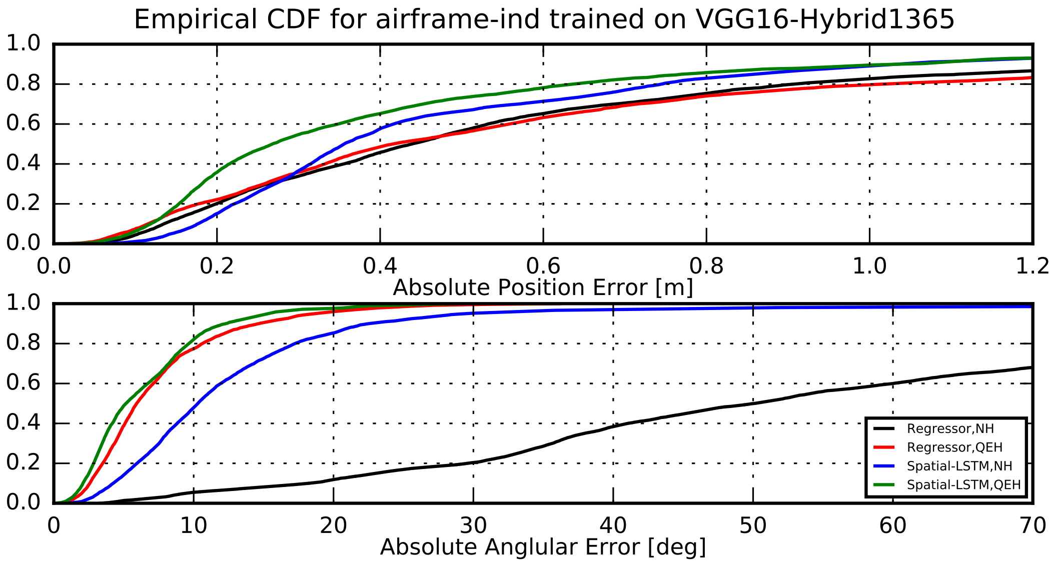


FIGURE 4.2: Error histogram for airframe-ind dataset trained on VGG16-Hybrid1365

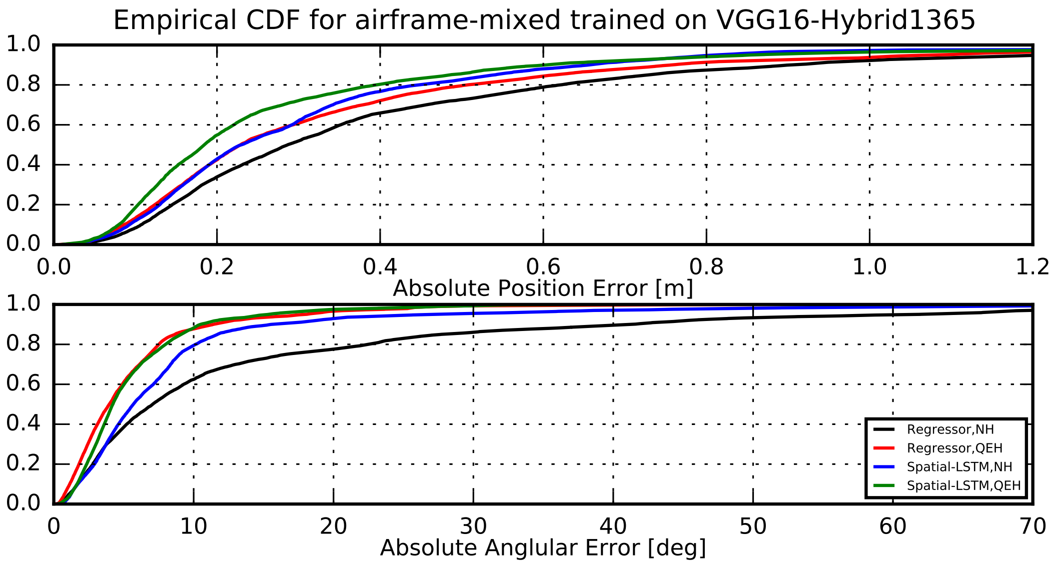


FIGURE 4.3: Error histogram for airframe-mixed dataset trained on VGG16-Hybrid1365

TABLE 4.15: Top-down view of predicted trajectories on airframe-ind dataset

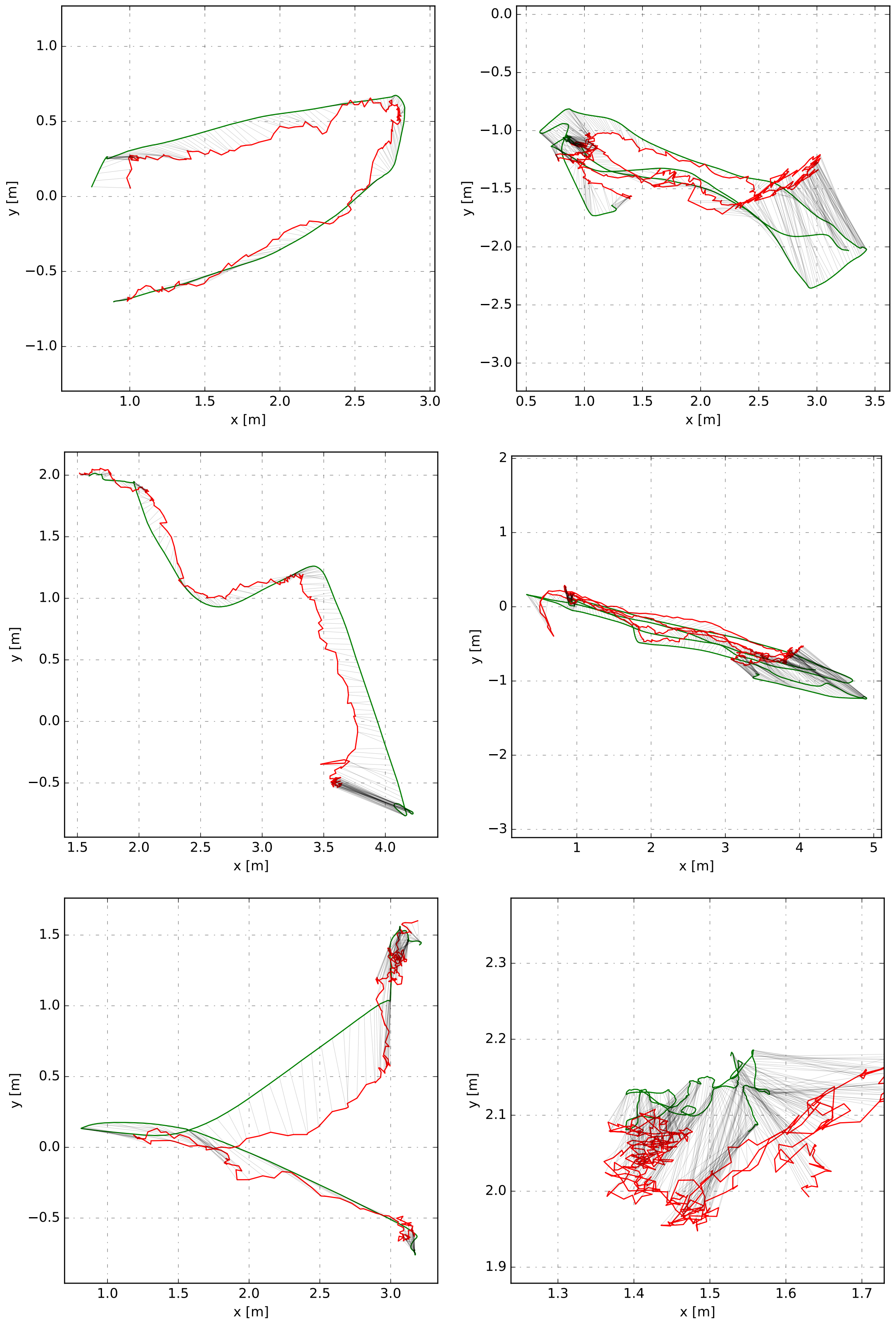


Table 4.16: Saliency maps for airframe-ind dataset trained on Spatial-LSTM,QEH using VGG16-Hybrid1365

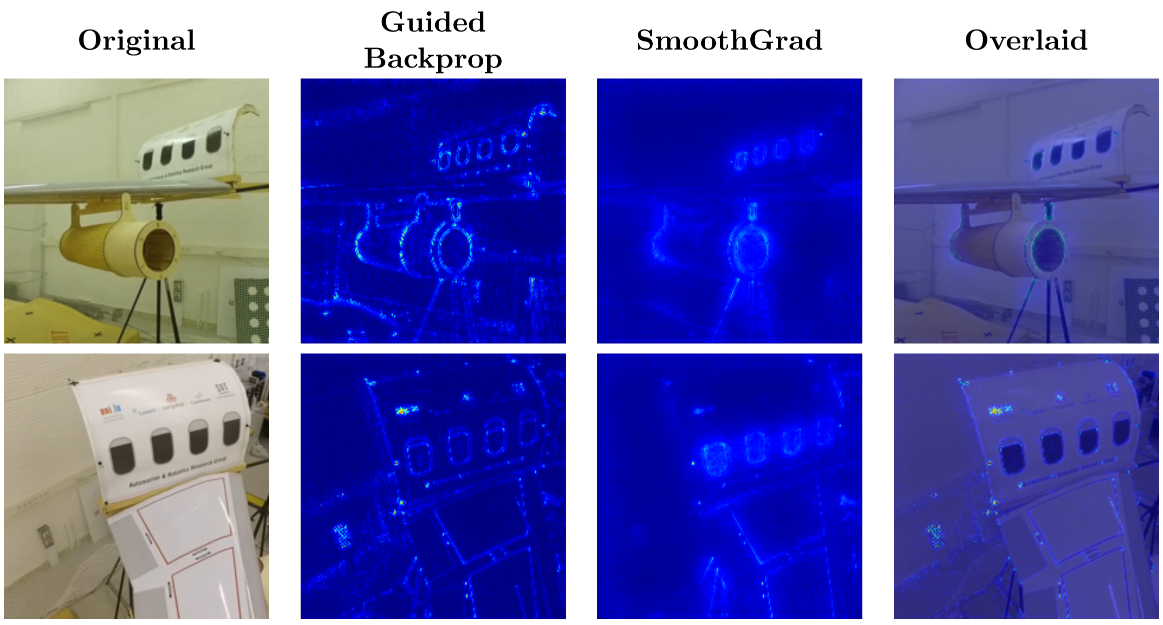


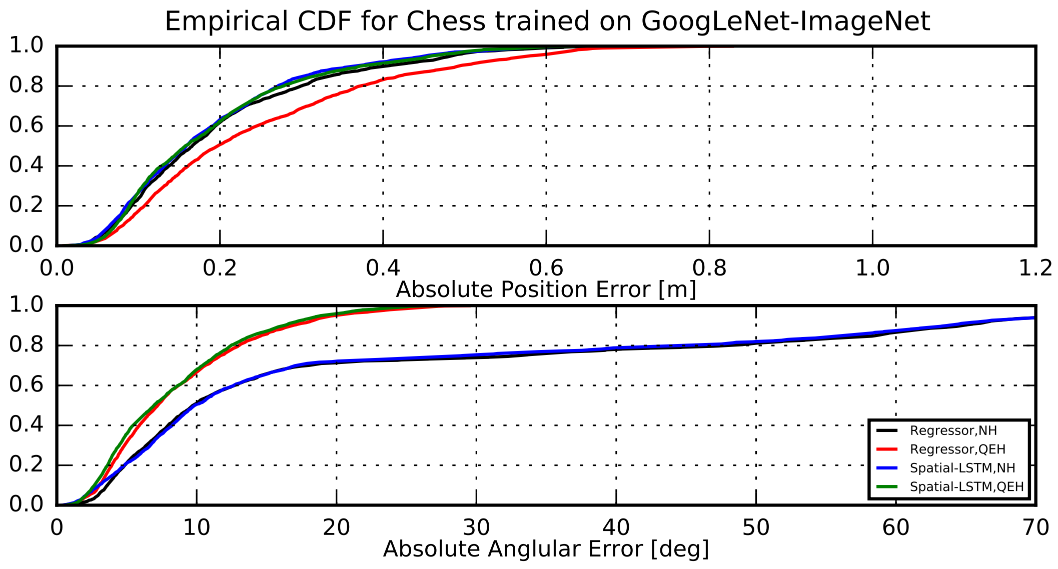
Table 4.17: Comparison of Temporal-GRU results on different CNN architectures

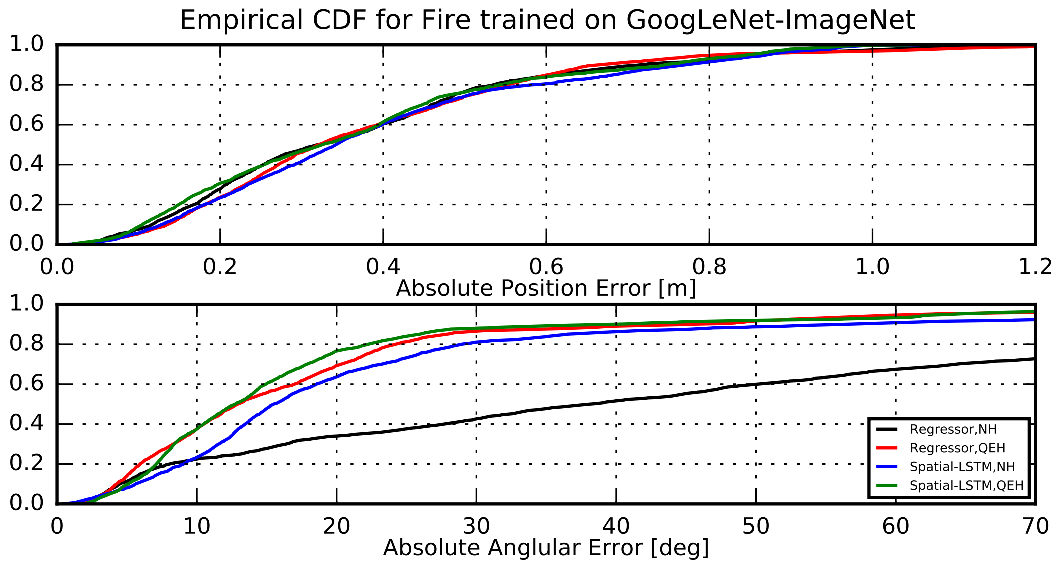
Data set	**Method**	**median**		**mae**		**max**		**std**	
		pos	orien	pos	orien	pos	orien	pos	orien
GoogLeNet ImageNet	Temporal-GRU,QEH	0.340	**7.059**	0.485	8.512	2.698	33.96	0.412	5.382
	Temporal-GRU,NH	0.437	8.68	0.565	11.79	3.426	174.1	0.434	11.90
GoogLeNet Places365	Temporal-GRU,QEH	0.476	9.67	0.567	11.90	2.100	49.00	0.345	7.76
	Temporal-GRU,NH	0.691	12.07	0.805	18.98	2.763	178	0.451	21.37
VGG16 Hybrid1365	Temporal-GRU,QEH	**0.247**	7.67	**0.378**	**8.48**	**1.592**	**32.40**	0.310	**5.00**
	Temporal-GRU,NH	0.367	53.55	0.443	62.86	1.701	179.8	**0.271**	37.28

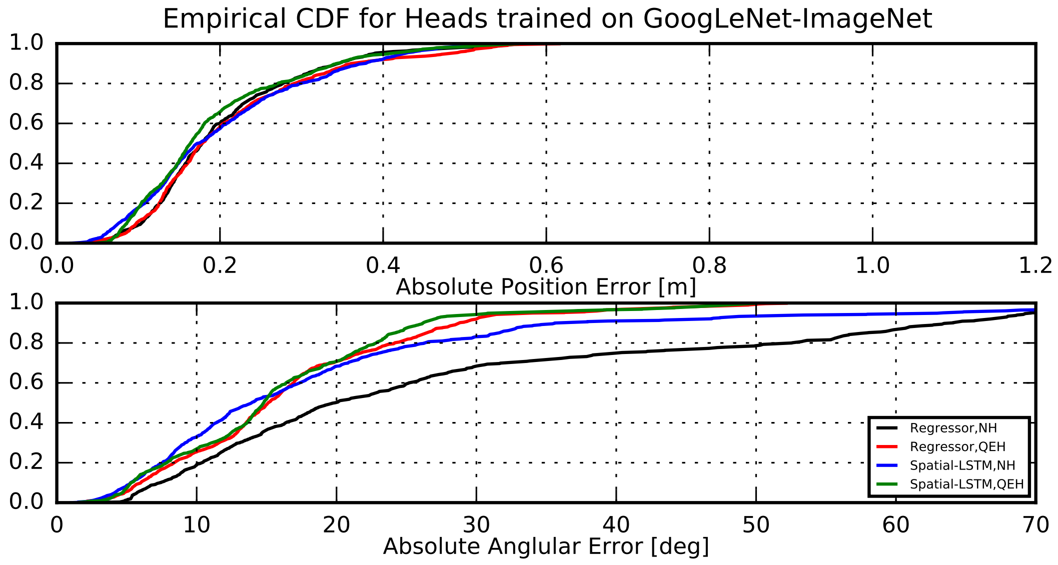
Conclusions

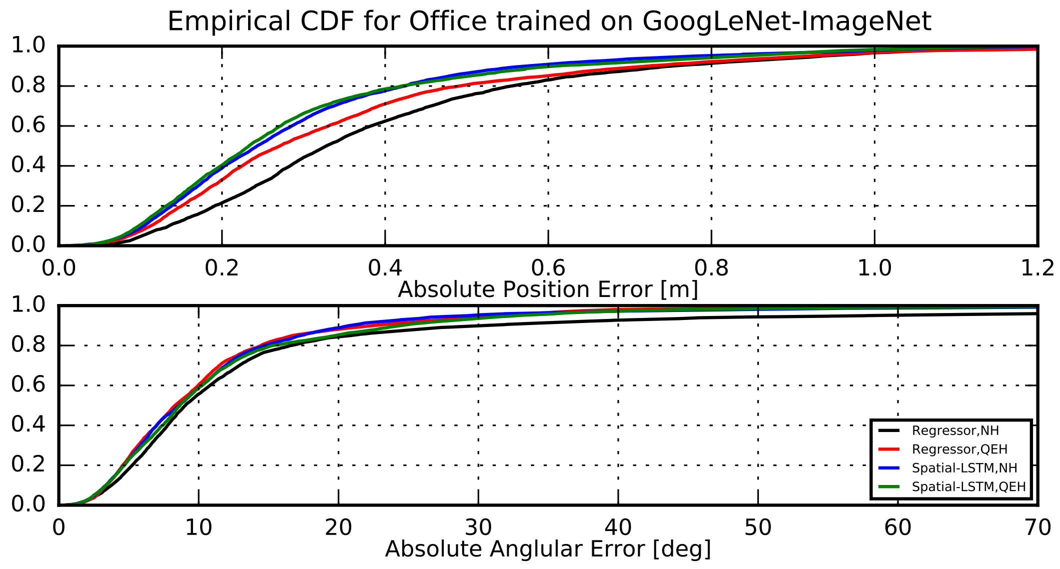
Image-based localization from RGB images remains a challenging task for deep learning. In this work, we have shown three end-to-end approaches to this problem. Firstly, we improved the naive orientation loss function by introducing proper quaternion algebra for proper computation of the loss. Secondly, we combined the Spatial-LSTM architecture recently introduced by [11] with another very recent method, which combats the need of heavy finetuning of weights in a multi-task loss function [7]. This has proven to be very significant allowing us to achieve competing and sometimes superior results on a widely recognized 7*Scenes* dataset. Moreover, we have also investigated temporal models for processing sequences of images, as suggested by [5]. This has been the most challenging phase of this thesis and we concluded that more investigation is required as the models are very promising. Lastly, we also introduced a novel dataset for image-based localization, comprising of 6 subsets each containing labeled RGB images of an airframe model situated at different positions and orientations.

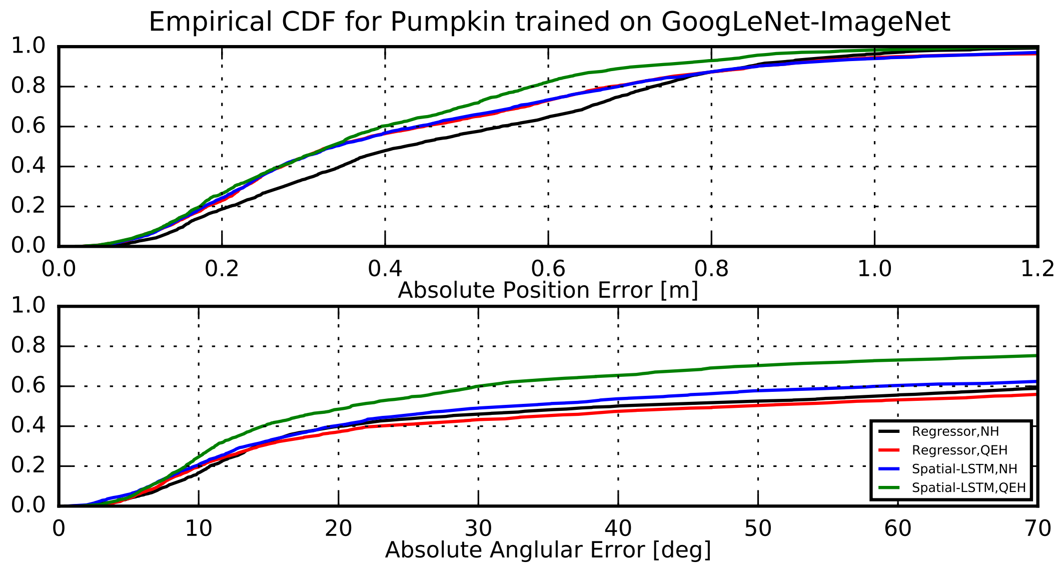
A. Error histograms

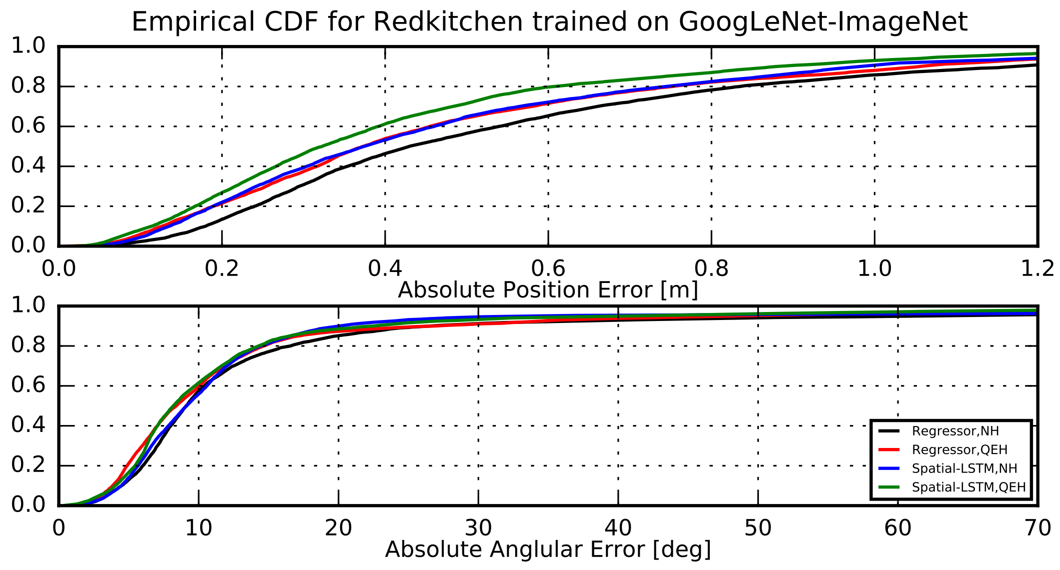

Empirical CDF for Chess trained on GoogLeNet-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

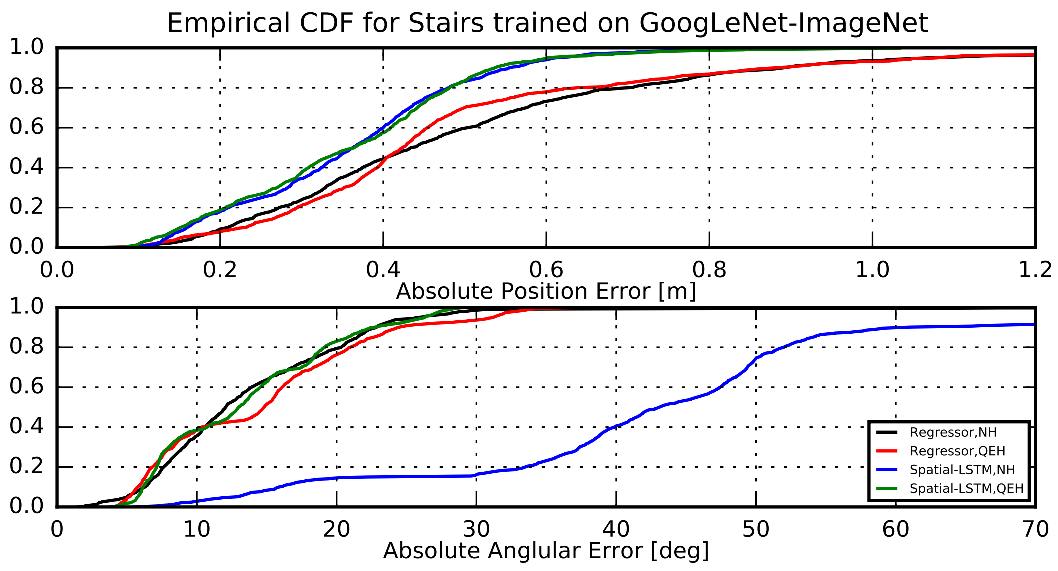

Empirical CDF for Fire trained on GoogLeNet-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

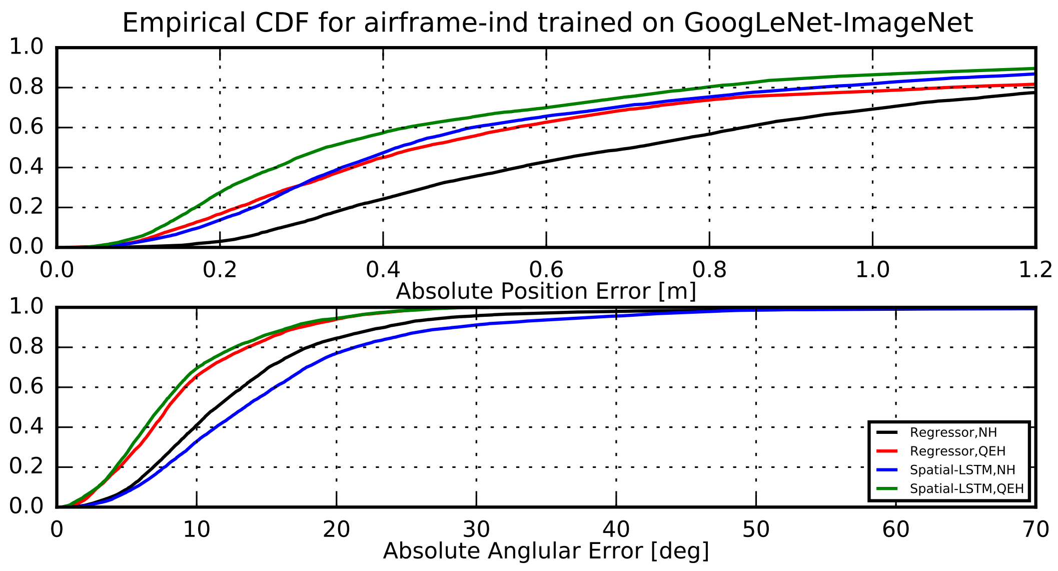

Empirical CDF for Heads trained on GoogLeNet-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

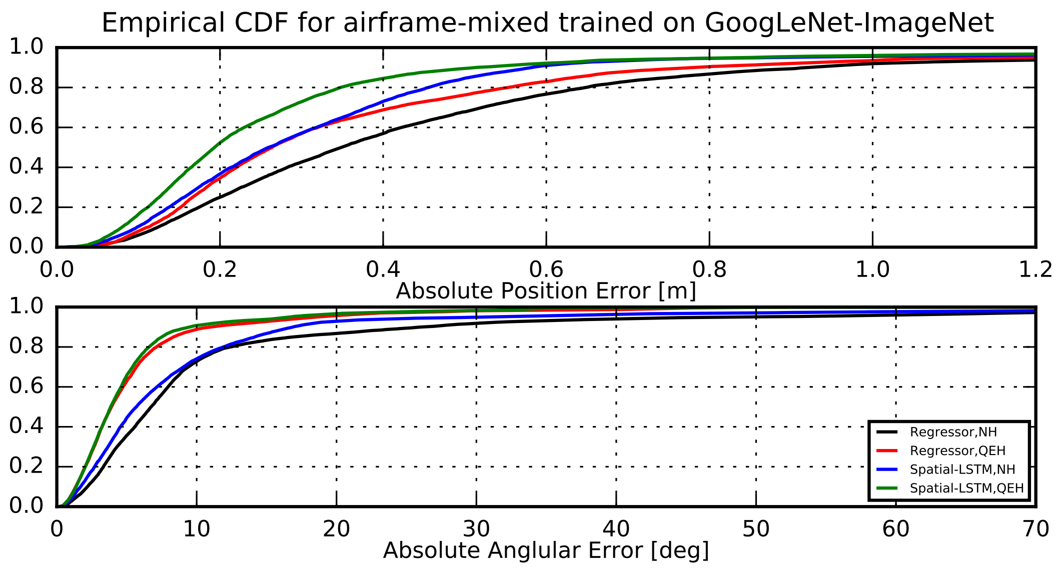

Empirical CDF for Office trained on GoogLeNet-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

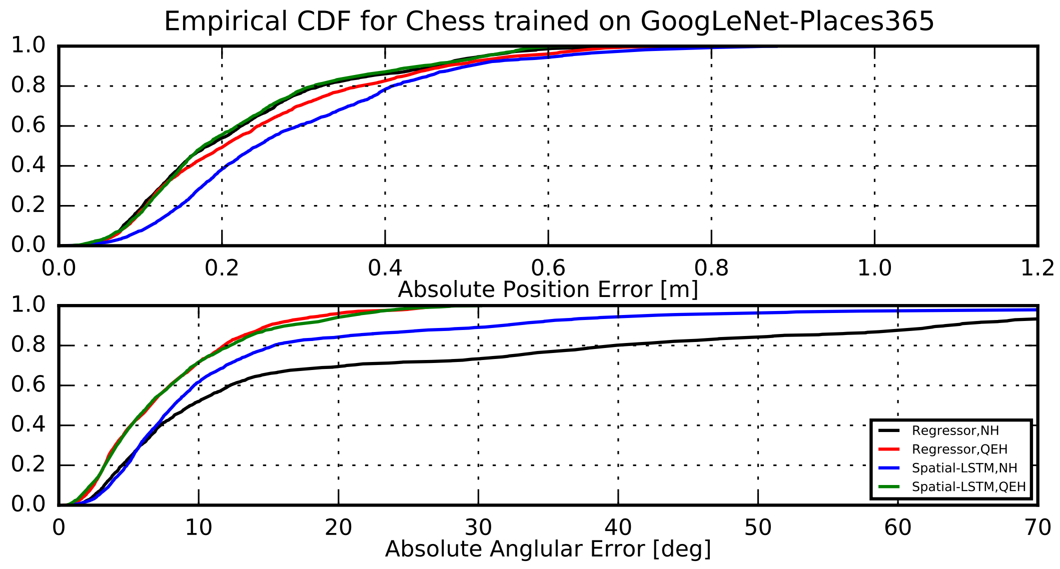

Empirical CDF for Pumpkin trained on GoogLeNet-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

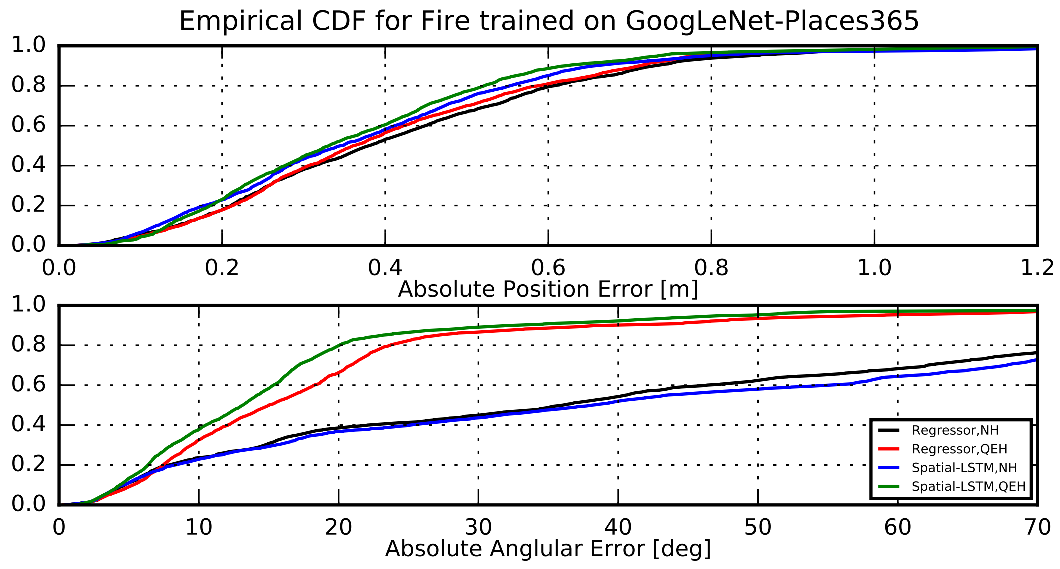

Empirical CDF for Redkitchen trained on GoogLeNet-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

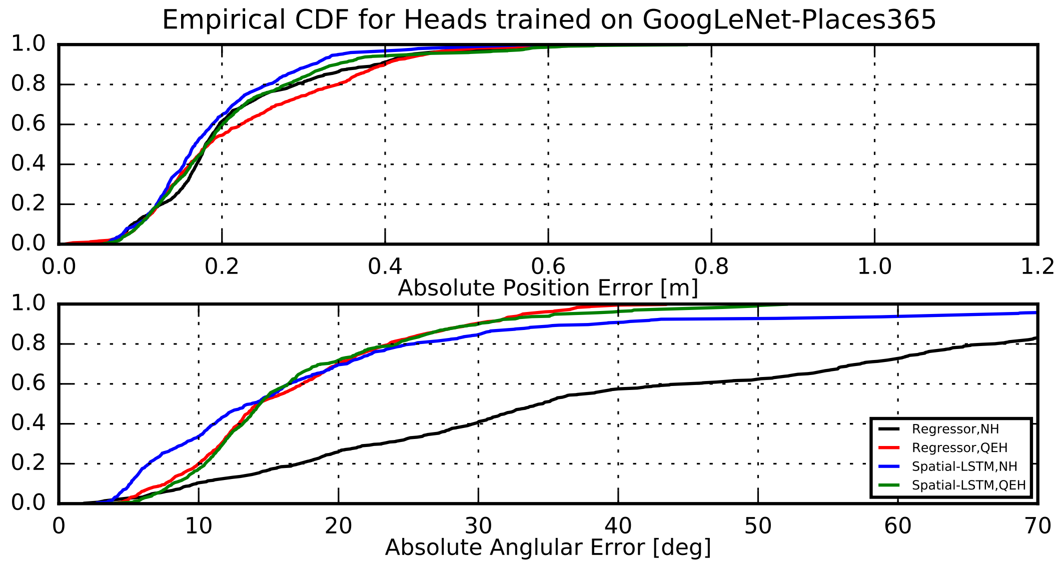

Empirical CDF for Stairs trained on GoogLeNet-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

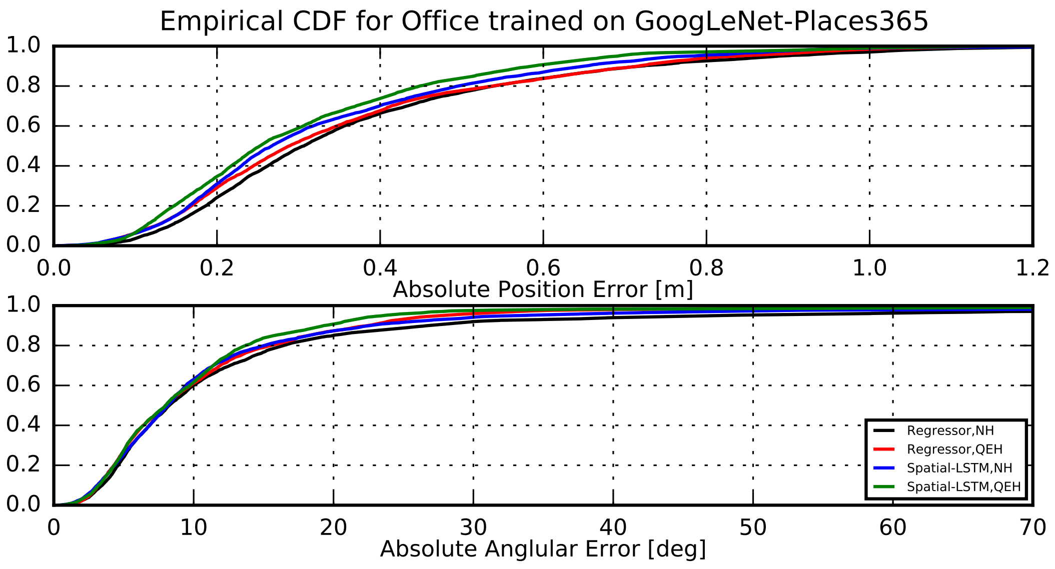

Empirical CDF for airframe-ind trained on GoogLeNet-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

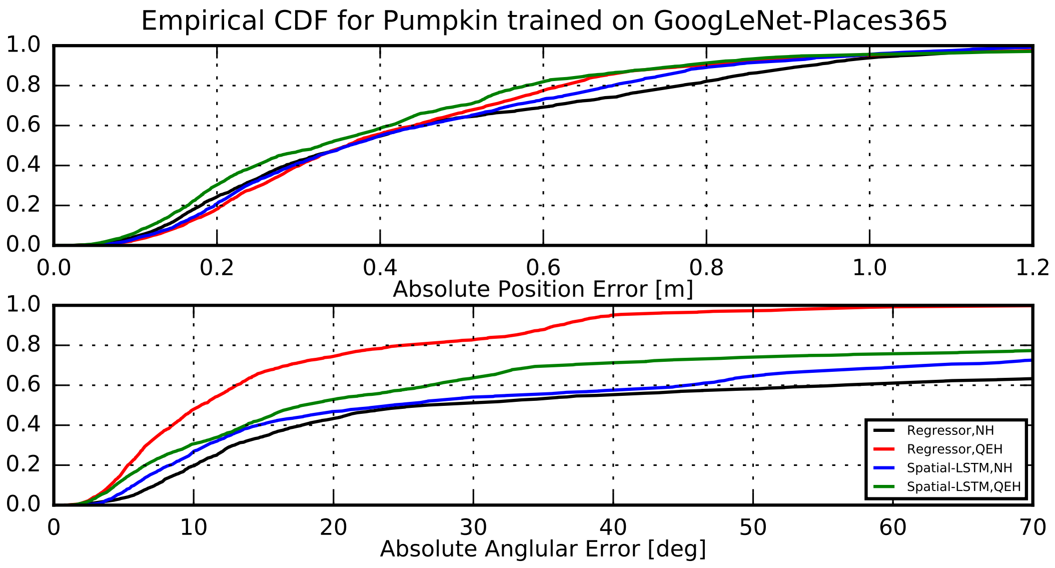

Empirical CDF for airframe-mixed trained on GoogLeNet-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

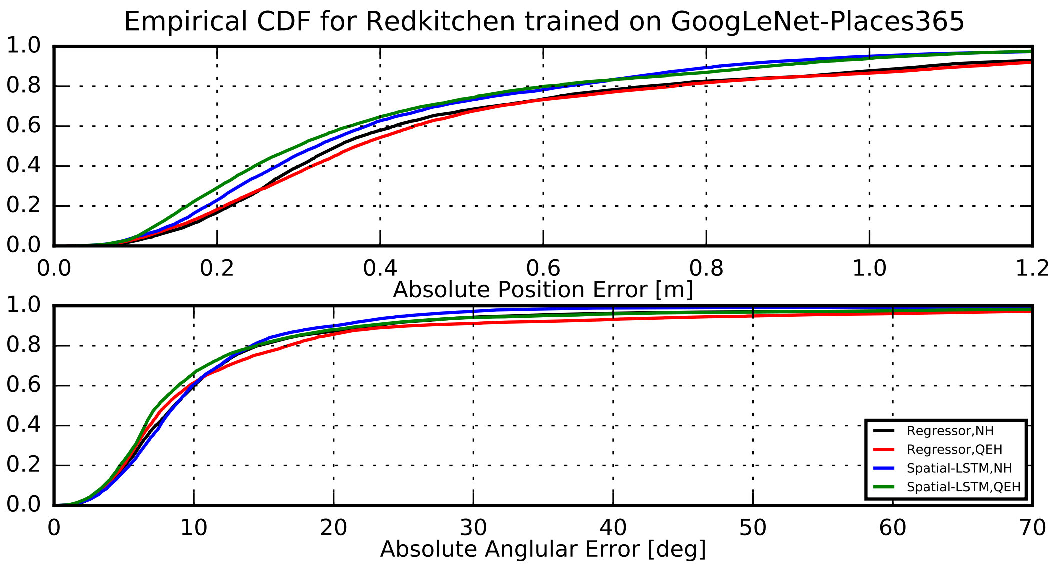

Empirical CDF for Chess trained on GoogLeNet-Places365
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

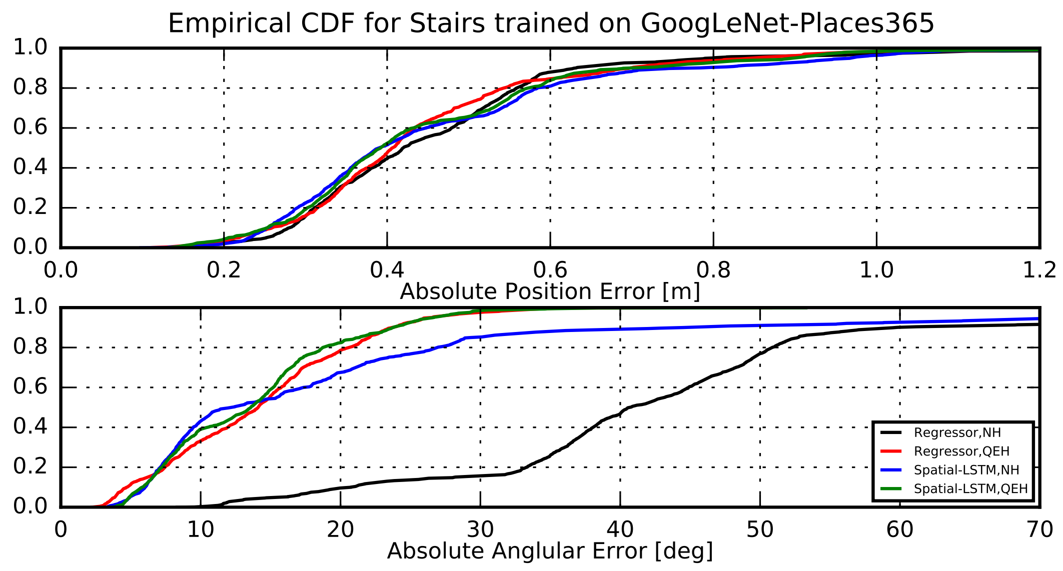

Empirical CDF for Fire trained on GoogLeNet-Places365
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

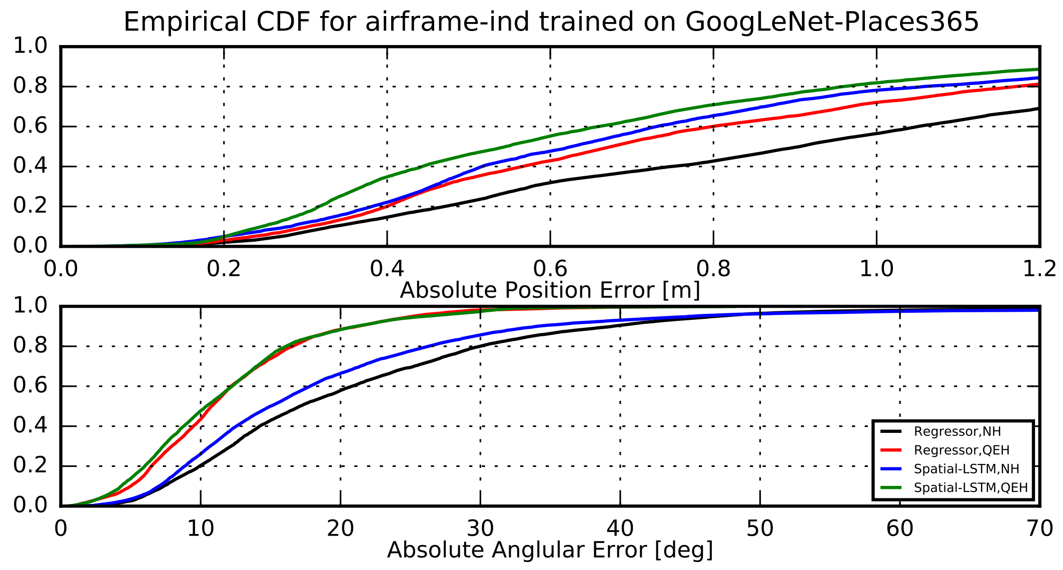

Empirical CDF for Heads trained on GoogLeNet-Places365
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

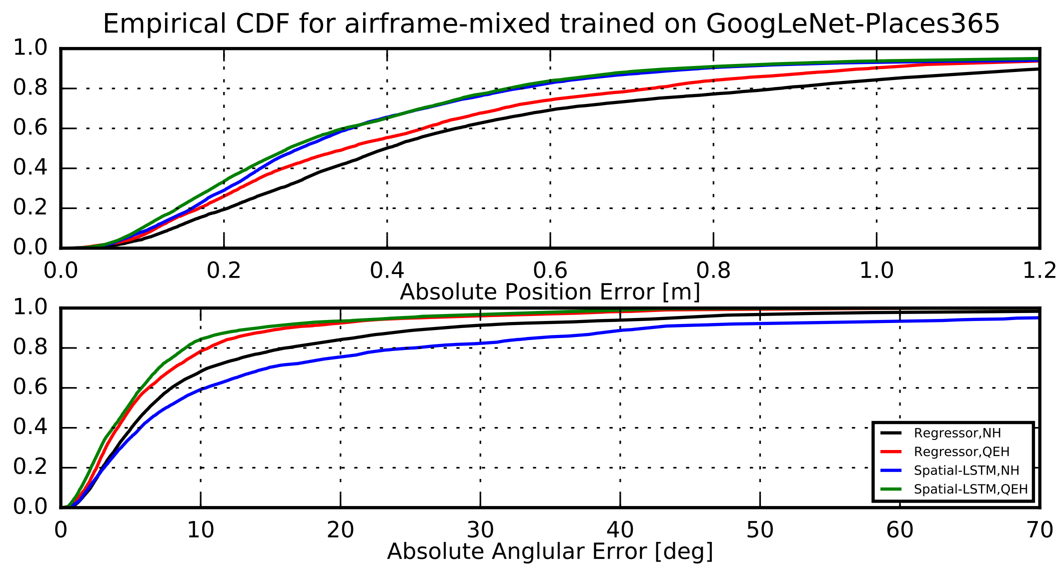

Empirical CDF for Office trained on GoogLeNet-Places365
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

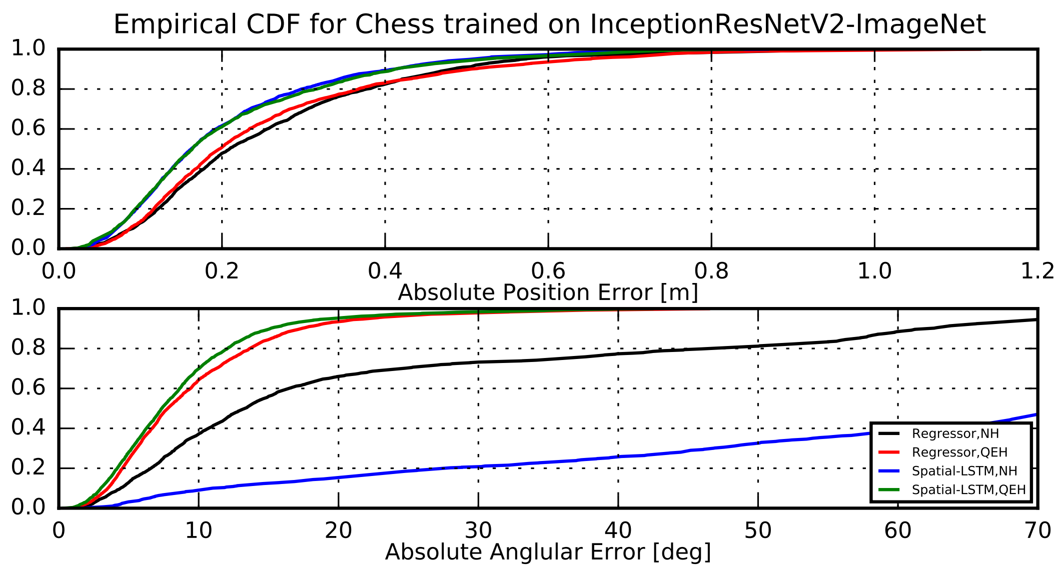

Empirical CDF for Pumpkin trained on GoogLeNet-Places365
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

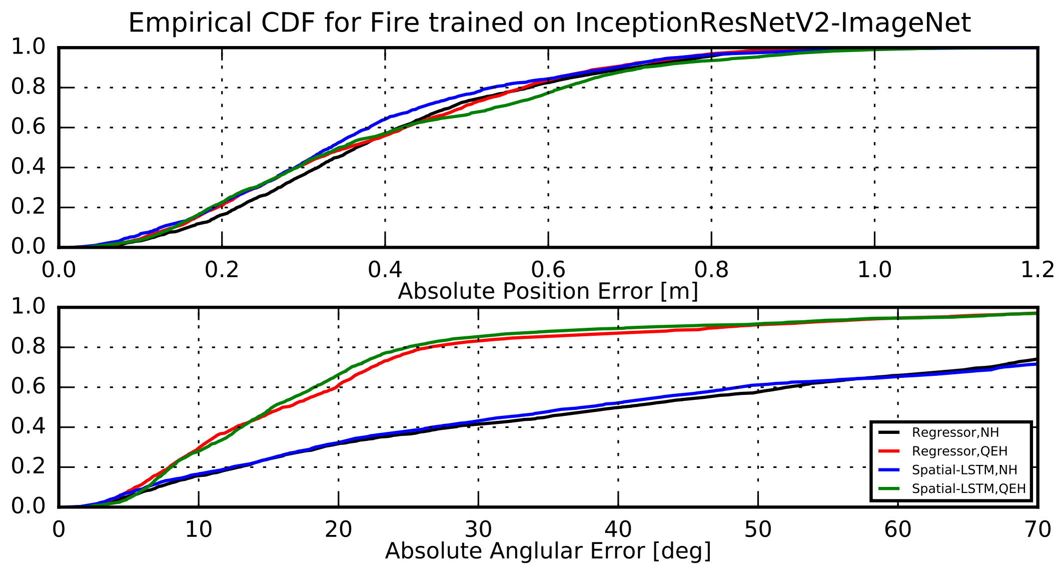

Empirical CDF for Redkitchen trained on GoogLeNet-Places365
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

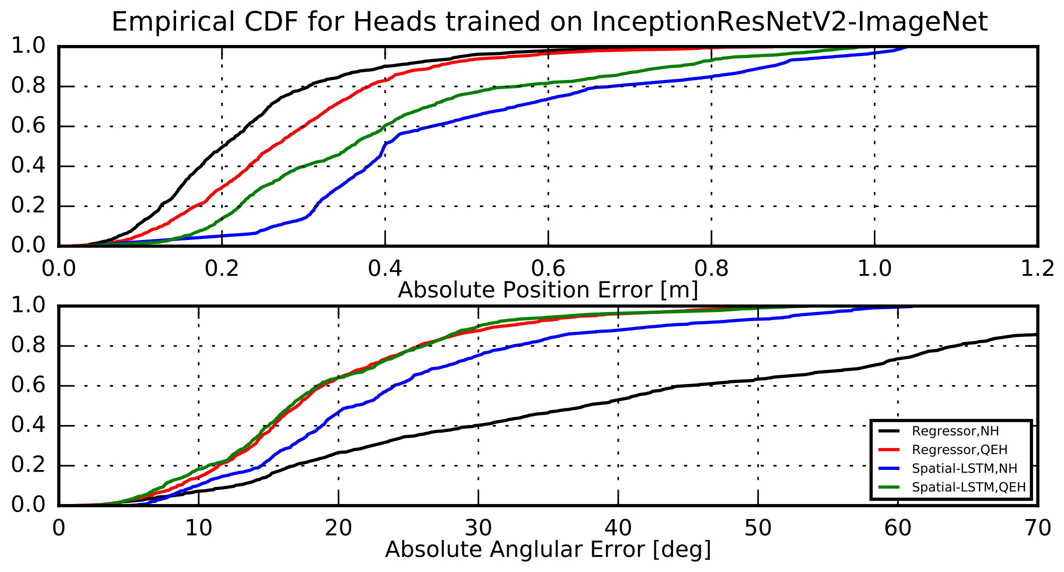

Empirical CDF for Stairs trained on GoogLeNet-Places365
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

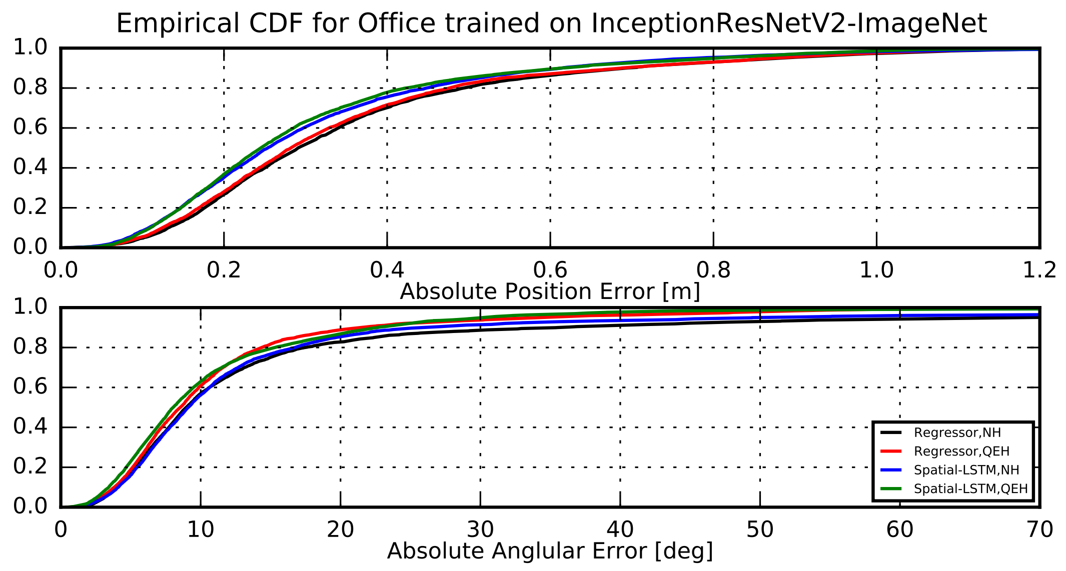

Empirical CDF for airframe-ind trained on GoogLeNet-Places365
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

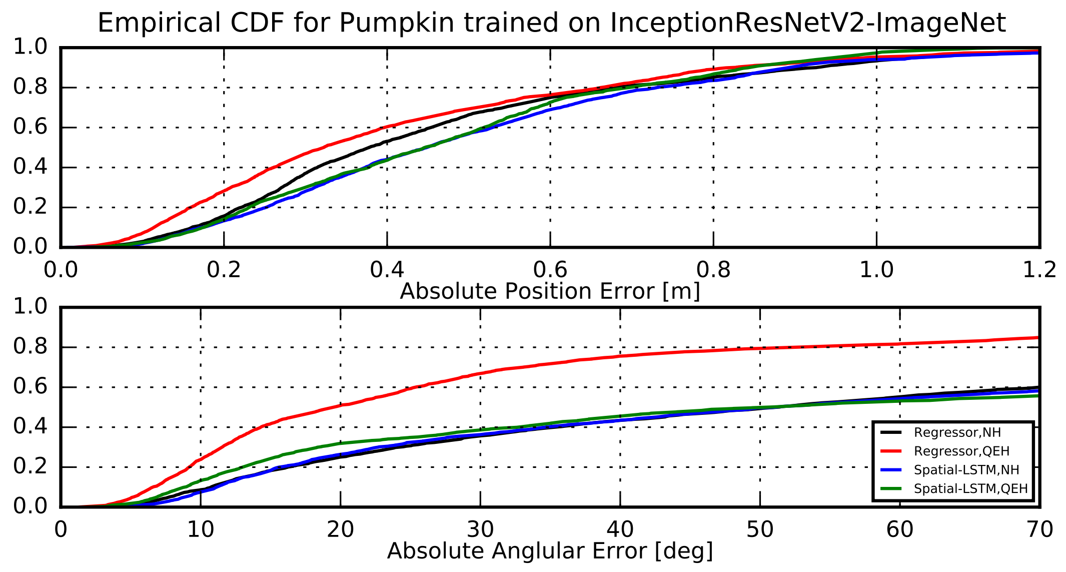

Empirical CDF for airframe-mixed trained on GoogLeNet-Places365
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

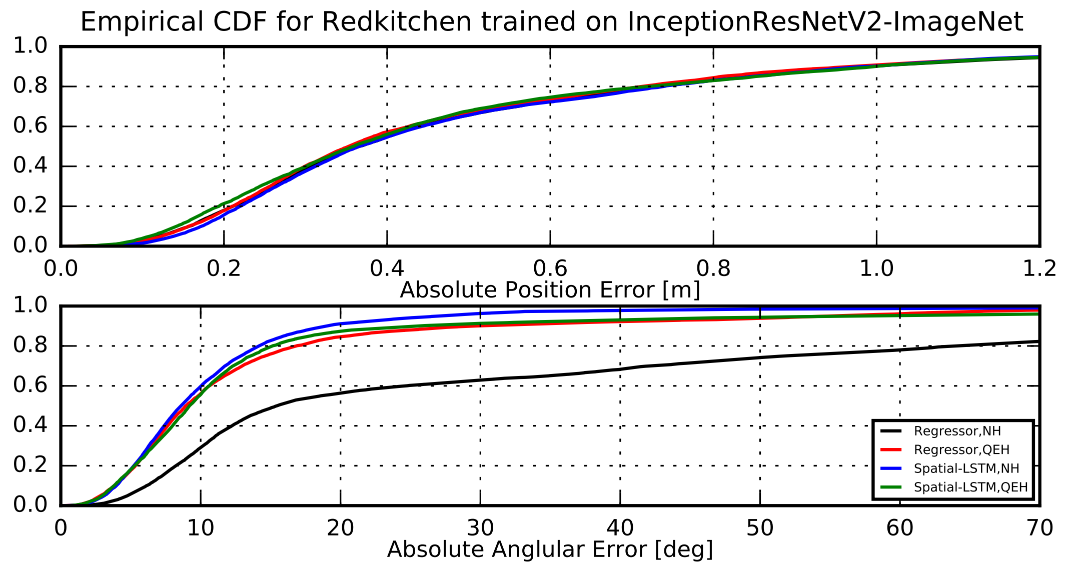

Empirical CDF for Chess trained on InceptionResNetV2-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

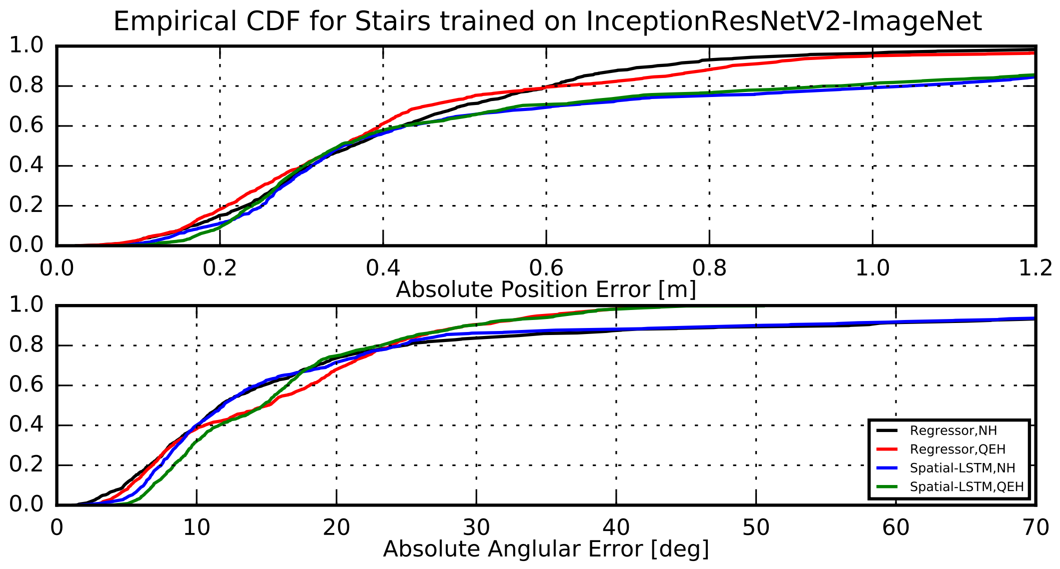

Empirical CDF for Fire trained on InceptionResNetV2-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

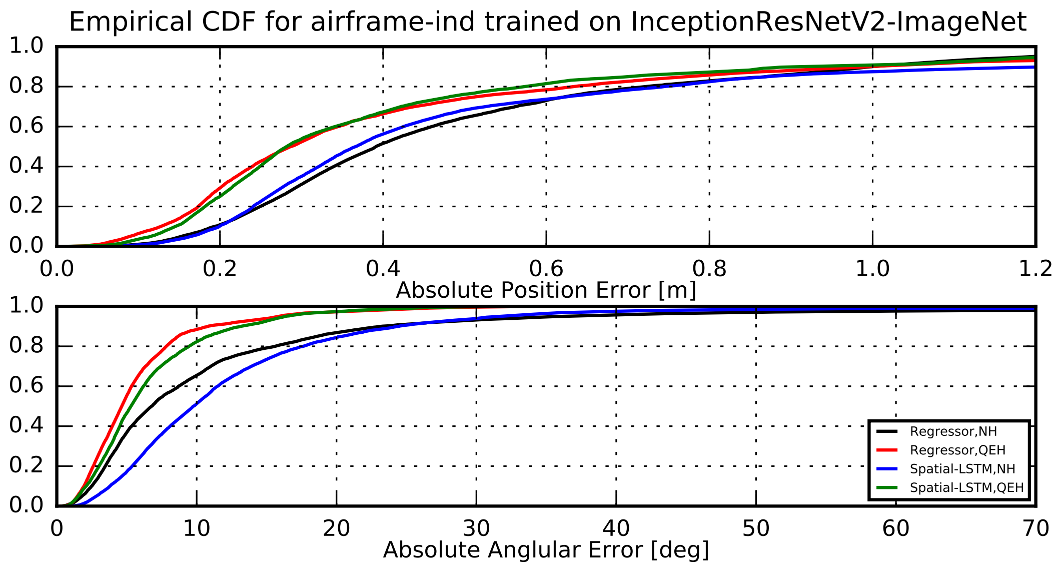

Empirical CDF for Heads trained on InceptionResNetV2-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

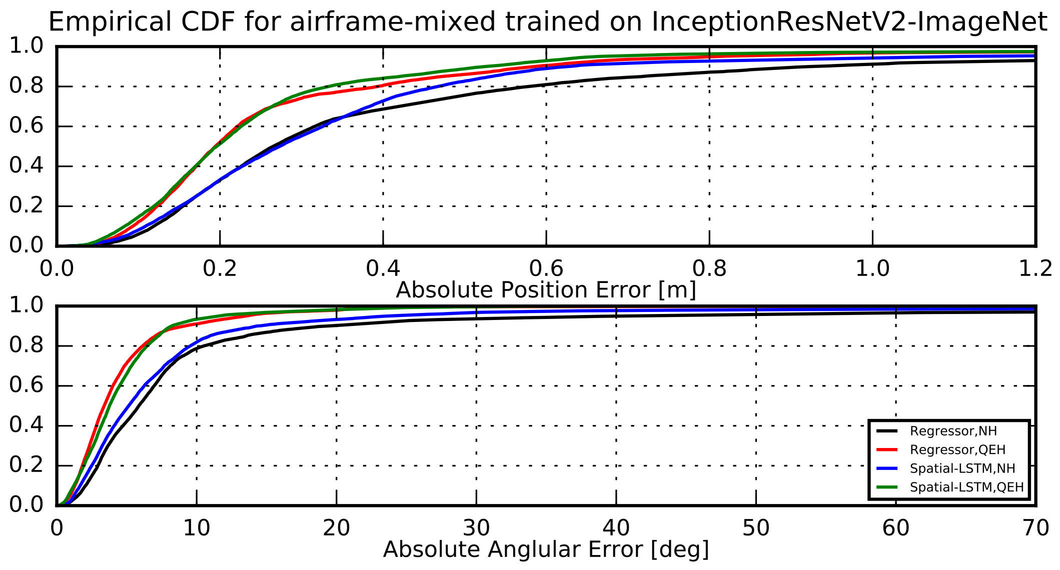

Empirical CDF for Office trained on InceptionResNetV2-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH


Empirical CDF for Pumpkin trained on InceptionResNetV2-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH


Empirical CDF for Redkitchen trained on InceptionResNetV2-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH


Empirical CDF for Stairs trained on InceptionResNetV2-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH


Empirical CDF for airframe-ind trained on InceptionResNetV2-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH


Empirical CDF for airframe-mixed trained on InceptionResNetV2-ImageNet
Absolute Position Error [m]
Absolute Anglular Error [deg]
Regressor,NH
Regressor,QEH
Spatial-LSTM,NH
Spatial-LSTM,QEH

B. Detailed VGG16 architecture

Table 18: VGG16 Architecture summary

Layer	Filters	Size	Strides	Input size	Memory	Weights
Input	-	-	-	$224 \times 224 \times 3$	$224 * 224 * 3 = 150$K	0
Conv	64	(3,3)	(1,1)	$224 \times 224 \times 64$	$224 * 224 * 64 = 3.2$M	$(3 * 3 * 3) * 64 = 1,728$
Conv	64	(3,3)	(1,1)	$224 \times 224 \times 64$	$224 * 224 * 64 = 3.2$M	$(3 * 3 * 64) * 64 = 36,864$
Pool	-	(2,2)	(2,2)	$112 \times 112 \times 64$	$112 * 112 * 64 = 800$K	0
Conv	128	(3,3)	(1,1)	$112 \times 112 \times 128$	$112 * 112 * 128 = 1.6$M	$(3 * 3 * 64) * 128 = 73,728$
Conv	128	(3,3)	(1,1)	$112 \times 112 \times 128$	$112 * 112 * 128 = 1.6$M	$(3 * 3 * 128) * 128 = 147,456$
Pool	-	(2,2)	(2,2)	$56 \times 56 \times 128$	$56 * 56 * 128 = 400$K	0
Conv	256	(3,3)	(1,1)	$56 \times 56 \times 256$	$56 * 56 * 256 = 800$K	$(3 * 3 * 128) * 256 = 294,912$
Conv	256	(3,3)	(1,1)	$56 \times 56 \times 256$	$56 * 56 * 256 = 800$K	$(3 * 3 * 256) * 256 = 589,824$
Conv	256	(3,3)	(1,1)	$56 \times 56 \times 256$	$56 * 56 * 256 = 800$K	$(3 * 3 * 256) * 256 = 589,824$
Pool	-	(2,2)	(2,2)	$28 \times 28 \times 256$	$28 * 28 * 256 = 200$K	0
Conv	512	(3,3)	(1,1)	$28 \times 28 \times 512$	$28 * 28 * 512 = 400$K	$(3 * 3 * 256) * 512 = 1,179,648$
Conv	512	(3,3)	(1,1)	$28 \times 28 \times 512$	$28 * 28 * 512 = 400$K	$(3 * 3 * 512) * 512 = 2,359,296$
Conv	512	(3,3)	(1,1)	$28 \times 28 \times 512$	$28 * 28 * 512 = 400$K	$(3 * 3 * 512) * 512 = 2,359,296$
Pool	-	(2,2)	(2,2)	$14 \times 14 \times 512$	$14 * 14 * 512 = 100$K	0
Conv	512	(3,3)	(1,1)	$14 \times 14 \times 512$	$14 * 14 * 512 = 100$K	$(3 * 3 * 512) * 512 = 2,359,296$
Conv	512	(3,3)	(1,1)	$14 \times 14 \times 512$	$14 * 14 * 512 = 100$K	$(3 * 3 * 512) * 512 = 2,359,296$
Conv	512	(3,3)	(1,1)	$14 \times 14 \times 512$	$14 * 14 * 512 = 100$K	$(3 * 3 * 512) * 512 = 2,359,296$
Pool	-	(2,2)	(2,2)	$7 \times 7 \times 512$	$7 * 7 * 512 = 25$K	0
FC	4000	-	-	$1 \times 1 \times 4096$	4096	$7 * 7 * 512 * 4096 = 102,760,448$
FC	4000	-	-	$1 \times 1 \times 4096$	4096	$4096 * 4096 = 16,777,216$
FC	4000	-	-	$1 \times 1 \times 1000$	1000	$4096 * 1000 = 4,096,000$
Total for forward pass per image:					24M *4 bytes $\approx$ 93MB	138M parameters
Total for backward pass per image:					24M * bytes * 2 $\approx$ 186 MB	

List of Figures

List of Tables

Bibliography

[1] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[2] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. *Computer vision–ECCV 2006*, pages 404–417, 2006.

[3] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *Computer Vision (ICCV), 2011 IEEE international conference on*, pages 2564–2571. IEEE, 2011.

[4] Alex Kendall, Matthew Grimes, and Roberto Cipolla. Posenet: A convolutional network for real-time 6-dof camera relocalization. In *Proceedings of the IEEE international conference on computer vision*, pages 2938–2946, 2015.

[5] Ronald Clark, Sen Wang, Andrew Markham, Niki Trigoni, and Hongkai Wen. Vidloc: 6-dof video-clip relocalization. *arXiv preprint arXiv:1702.06521*, 2017.

[6] Alex Kendall and Roberto Cipolla. Modelling uncertainty in deep learning for camera relocalization. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 4762–4769. IEEE, 2016.

[7] Alex Kendall and Roberto Cipolla. Geometric loss functions for camera pose regression with deep learning. *arXiv preprint arXiv:1704.00390*, 2017.

[8] Florian Walch. Deep Learning for Image-Based Localization. Master's thesis, TECHNISCHE UNIVERSITAT MUNCHEN, Munich, Germany, 2016.

[9] Daoyuan Jia, Yongchi Su, and Chunping Li. Deep convolutional neural network for 6-dof image localization. *arXiv preprint arXiv:1611.02776*, 2016.

[10] Iaroslav Melekhov, Juho Kannala, and Esa Rahtu. Relative camera pose estimation using convolutional neural networks. *arXiv preprint arXiv:1702.01381*, 2017.

[11] Florian Walch, Caner Hazirbas, Laura Leal-Taixé, Torsten Sattler, Sebastian Hilsenbeck, and Daniel Cremers. Image-based localization with spatial lstms. *arXiv preprint arXiv:1611.07890*, 2016.

[12] Jamie Shotton, Ben Glocker, Christopher Zach, Shahram Izadi, Antonio Criminisi, and Andrew Fitzgibbon. Scene coordinate regression forests for camera relocalization in rgb-d images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2930–2937, 2013.

[13] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072.

[14] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[15] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[16] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[17] Jesús Bobadilla, Fernando Ortega, Antonio Hernando, and Abraham Gutiérrez. Recommender systems survey. *Knowledge-based systems*, 46:109–132, 2013.

[18] Yehuda Koren. The bellkor solution to the netflix grand prize. *Netflix prize documentation*, 81:1–10, 2009.

[19] Gulshan V, Peng L, Coram M, and et al. Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. *JAMA*, 316(22):2402–2410, 2016. doi: 10.1001/jama.2016.17216. URL +http://dx.doi.org/10.1001/jama.2016.17216.

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

[21] Justin Johnson, Andrej Karpathy, and Li Fei-Fei. Densecap: Fully convolutional localization networks for dense captioning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4565–4574, 2016.

[22] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[23] Ryan Turner. A model explanation system. In *Machine Learning for Signal Processing (MLSP), 2016 IEEE 26th International Workshop on*, pages 1–6. IEEE, 2016.

[24] Shi-ho Cheng. Unboxing the Random Forest Classifier: The Threshold Distributions. http://nerds.airbnb.com/unboxing-the-random-forest-classifier, 2015. Online, accessed 6 September 2017.

[25] David H Wolpert. The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1341–1390, 1996.

[26] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[27] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[28] Andrej Karpathy. CS231n: Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io, 2016. Online, accessed 10 September 2017.

[29] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[30] Jasper Snoek, Kevin Swersky, Rich Zemel, and Ryan Adams. Input warping for bayesian optimization of non-stationary functions. In *International Conference on Machine Learning*, pages 1674–1682, 2014.

[31] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. of LION-5*, page 507–523, 2011.

[32] Hal Daumé III. Hyperparameter search, Bayesian optimization and related topics. https://nlpers.blogspot.lu/2014/10/hyperparameter-search-bayesian.html, 2015. Online, accessed 11 September 2017.

[33] John Elder. *Handbook of statistical analysis and data mining applications*. Academic Press, 2009.

[34] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(4):15, 2012.

[35] Kaggle. Data Leakage. https://www.kaggle.com/wiki/Leakage. Online, accessed 13 September 2017.

[36] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[37] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[38] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[39] 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, Yangqing 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, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[40] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[41] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[42] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[43] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[45] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

[46] Léon Bottou. Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, 91(8), 1991.

[47] Gabriel Goh. Why momentum really works. *Distill*, 2017. doi: 10.23915/distill.00006. URL http://distill.pub/2017/momentum.

[48] Yurii Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

[49] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[50] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[51] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[52] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[53] Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

[54] Ciyou Zhu, Richard H Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software (TOMS)*, 23(4):550–560, 1997.

[55] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576*, 2015.

[56] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Texture synthesis and the controlled generation of natural stimuli using convolutional neural networks. *arXiv preprint arXiv:1505.07376*, 12, 2015.

[57] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[58] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[59] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

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

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

[62] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015.

[63] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[64] Haoxiang Li, Zhe Lin, Xiaohui Shen, Jonathan Brandt, and Gang Hua. A convolutional neural network cascade for face detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5325–5334, 2015.

[65] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[66] Massimo Bertozzi, Alberto Broggi, Mike Del Rose, Mirko Felisa, Alain Rakotomamonjy, and Frédéric Suard. A pedestrian detector using histograms of oriented gradients and a support vector machine classifier. In *Intelligent Transportation Systems Conference, 2007. ITSC 2007. IEEE*, pages 143–148. IEEE, 2007.

[67] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813, 2014.

[68] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[69] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *International conference on machine learning*, pages 647–655, 2014.

[70] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

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

[72] Thibaut Durand and Nicolas Thome. Deep learning and weak supervision for image. http://webia.lip6.fr/~cord/pdfs/news/TalkDeepCordI3S.pdf. Online, accessed 20 September 2017.

[73] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[74] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[75] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, pages 4278–4284, 2017.

[76] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[77] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[78] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pages 630–645. Springer, 2016.

[79] Alex Alemi. Improving Inception and Image Classification in TensorFlow. https://research.googleblog.com/2016/08/improving-inception-and-image.html, 2016. Online, accessed 14 September 2017.

[80] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

[81] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*, 2015.

[82] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[83] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[84] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[85] Andrej Karpathy. What I learned from competing against a ConvNet on ImageNet. http://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/, 2014. Online, accessed 20 September 2017.

[86] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[87] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. *arXiv preprint arXiv:1703.01365*, 2017.

[88] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg. Smoothgrad: removing noise by adding noise. *arXiv preprint arXiv:1706.03825*, 2017.

[89] Andrej Karpathy. t-SNE visualization of CNN codes. http://cs.stanford.edu/people/karpathy/cnnembed/, . Online, accessed 21 September 2017.

[90] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda Viégas, Martin Wattenberg. SmoothGrad - PAIR Code project page. https://pair-code.github.io/saliency/, 2014. Online, accessed 25 September 2017.

[91] Christopher Olah. Understanding LSTM Networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015. Online, accessed 14 October 2017.

[92] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, pages 2048–2057, 2015.

[93] Sara Rosenthal, Noura Farra, and Preslav Nakov. Semeval-2017 task 4: Sentiment analysis in twitter. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pages 502–518, 2017.

[94] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[95] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[96] Andrej Karpathy. The Unreasonable Effectiveness of Recurrent Neural Networks. http://karpathy.github.io/2015/05/21/rnn-effectiveness/, . Online, accessed 28 September 2017.

[97] Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. *Journal of computer and system sciences*, 50(1):132–150, 1995.

[98] Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu. Multiple object recognition with visual attention. *arXiv preprint arXiv:1412.7755*, 2014.

[99] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91, 1991.

[100] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[101] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[102] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[103] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[104] Tobias Weyand, Ilya Kostrikov, and James Philbin. Planet-photo geolocation with convolutional neural networks. In *European Conference on Computer Vision*, pages 37–55. Springer, 2016.

[105] Fredrik Gustafsson, Fredrik Gunnarsson, Niclas Bergman, Urban Forssell, Jonas Jansson, Rickard Karlsson, and P-J Nordlund. Particle filters for positioning, navigation, and tracking. *IEEE Transactions on signal processing*, 50(2):425–437, 2002.

[106] Paul Newson and John Krumm. Hidden markov map matching through noise and sparseness. In *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 336–343. ACM, 2009.

[107] Torsten Sattler, Bastian Leibe, and Leif Kobbelt. Improving image-based localization by active correspondence search. In *European conference on computer vision*, pages 752–765. Springer, 2012.

[108] Torsten Sattler, Bastian Leibe, and Leif Kobbelt. Fast image-based localization using direct 2d-to-3d matching. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 667–674. IEEE, 2011.

[109] Joan Sola. Quaternion kinematics for the error-state kalman filter. *arXiv preprint arXiv:1711.02508*, 2017.

[110] Bolei Zhou, Agata Lapedriza, Aditya Khosla, Aude Oliva, and Antonio Torralba. Places: A 10 million image database for scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[111] Francesco Visin, Kyle Kastner, Kyunghyun Cho, Matteo Matteucci, Aaron Courville, and Yoshua Bengio. Renet: A recurrent neural network based alternative to convolutional networks. *arXiv preprint arXiv:1505.00393*, 2015.

[112] Rahul Rama Varior, Bing Shuai, Jiwen Lu, Dong Xu, and Gang Wang. A siamese long short-term memory architecture for human re-identification. In *European Conference on Computer Vision*, pages 135–153. Springer, 2016.

[113] Wonmin Byeon, Thomas M Breuel, Federico Raue, and Marcus Liwicki. Scene labeling with lstm recurrent neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3547–3555, 2015.

[114] Xiaodan Liang, Xiaohui Shen, Donglai Xiang, Jiashi Feng, Liang Lin, and Shuicheng Yan. Semantic object parsing with local-global long short-term memory. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3185–3193, 2016.

[115] Joe Yue-Hei Ng, Matthew Hausknecht, Sudheendra Vijayanarasimhan, Oriol Vinyals, Rajat Monga, and George Toderici. Beyond short snippets: Deep networks for video classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4694–4702, 2015.

[116] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C Lawrence Zitnick, and Devi Parikh. Vqa: Visual question answering. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2425–2433, 2015.

[117] Automation & Robotics Research Group. SnT - A&RG. https://wwwen.uni.lu/snt/research/automation_robotics_research_group, 2016. Online, accessed 15 September 2017.

[118] SZ DJI Technology Co. DJI - Matrice 100. https://www.dji.com/matrice100, 2016. Online, accessed 10 September 2017.

[119] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[120] Natural Point, Inc. OptiTrack - Motion Capture system. http://optitrack.com/, 2016. Online, accessed 10 September 2017.

[121] Tully Foote. tf: The transform library. In *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, Open-Source Software workshop, pages 1–6, April 2013. doi: 10.1109/TePRA.2013.6556373.