TECHNICAL UNIVERSITY OF MUNICH

TUM Data Innovation Lab

Data-Science from Whiteboard to Production: End-to-End Image Captioning

Authors	Oliver Borchert, Nika Dogonadze, Mürüvvet Hasanbasoglu, Murtaza Raza, Anastasia Stamatouli
Mentor(s)	Dr. Robert Pesch, Sebastian Blank, Julia Kronburger inovex GmbH
Co-Mentor	Michael Rauchensteiner
Project Lead	Dr. Ricardo Acevedo Cabra (Department of Mathematics)
Supervisor	Prof. Dr. Massimo Fornasier (Department of Mathematics)

Feb 2020

Abstract

Automatically generating captions for images is an exciting area of research, combining the challenges of modeling human vision and natural language understanding. Typically, this challenge is tackled with modern deep learning concepts, resulting in complex neural network models. Deploying these models to a production environment requires careful engineering to ensure high availability.
This report summarizes our project aiming to develop an image captioning model that is deployed on the cloud, embedded in a robust production environment. The development of our model is based on latest research, using convolutional neural networks to extract image features and LSTMs to generate text of arbitrary length. Visual attention and algorithmic post-processing steps enabled us to build a model with state-of-the-art performance on famous open-source datasets.
We embedded our model into a service-oriented architecture (SOA) that enables multiple desirable properties of a production environment beyond high availability. This includes elasticity, i.e. responding to changes in demand on the service in near real-time, continuously deploying new versions of individual services without disrupting the system, and monitoring failures with a high level of granularity. Eventually, we leveraged the power of the Google Cloud Platform to focus on application development rather than provisioning infrastructure. With the usage of GPUs to perform model inference, our system easily scales to hunderds of concurrent requests while keeping response times below one second despite the apparent complexity. The service is also publicly reachable at https://dilab.inovex.de.

Contents

1 Introduction

Over the last decade, machine learning has taken the world by storm, exhibiting impressive performance on numerous tasks ranging from playing games over understanding natural language to replicating aspects of human vision. One task, in particular, that has been studied to great extent in the scientific community is the problem of automatically generating captions for images. This is a challenging task considering that a machine learning model has to have both the ability to perceive images as well as to generate text — the latter still being a mostly open research question due to the discrete nature of natural language. Nonetheless, researchers have been able to develop powerful models that are able to accurately summarize the contents of images [1, 16, 17, 18].
With our project, we want to enable end users to leverage these advances in image captioning. This allows not only the general public to get a feeling for the current state of image captioning research but also enables developers to generate captions for a large number of images. This, in turn, allows for more advanced information retrieval pipelines merely based on images.
However, building a scalable image captioning service requires skills not only in statistical modeling but also in data and software engineering. Deploying a machine learning model into production and enabling hundreds of simultaneous users goes far beyond the area of expertise of a data scientist. Data preparation, model development and training constitute only a tiny fraction of the tasks required to build a robust system as already discussed in depth by Sculley et al. in 2015 [14].
The complexity of deploying machine learning models with state-of-the-art techniques is only amplified by recent developments in application management. Over the last couple years, advances in low-overhead virtualization, cluster management software, microservice architectures, and cloud computing have changed the way companies approach the design of large-scale software systems.
Our system was developed over the duration of four months in partnership with inovex GmbH and under the guidance of Dr. Robert Pesch, Sebastian Blank and Julia Kronburger.
In this document, we want to outline the approach we have taken to build our image captioning service. In Section 2, we will first dive into the design of our model driving our system's predictions. Building upon that, Section 3 will then outline techniques to enable continuously evaluating new iterations of model architectures. Lastly, in Section 4, we will focus on our system's architecture and how we ensure high availability under the assumptions of operating under high load, the existence of adversaries, and unreliable infrastructure. To round things off, Section 5 will give an overview about the application programming interface exposed by our system as well as the user interface visible to the end-user along with the model and load testing results.

1.1 Development Cycle

Agile methodology is recently commonly adapted in many areas as a more interactive approach within the development cycle. The following briefly explains how we applied agile methodology during our software development cycle.
The project was kicked off by a brainstorming session where team members and mentors

collaboratively decided on the project's points of focus. Based on a variety of ideas, a common understanding of the project's progress emerged and the backlog was filled by the mentors. Scrum was chosen as a project management process framework as illustrated in Figure 1 in order to enable a high degree of flexibility over the short duration of the project. We worked in sprints of two weeks, always introducing a new set of features within that time-frame. In order to keep track of current and upcoming tasks, we made use of Jira. Furthermore, Confluence was leveraged to document our progress and share valuable information, Slack was used for asynchronous communication, and GitLab for code collaboration as well as continuous integration and deployment as further described in Section 4.3. Additionally, each sprint was concluded with a Sprint Retrospective where we individually provided positive and negative feed-backs for the completed sprint, and based on the feed-backs, took actions that needs improvement within the following Sprint.

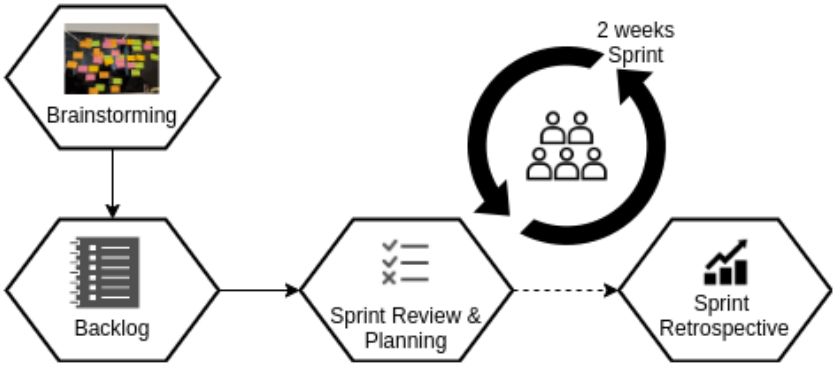

Figure 1: Scrum Framework

2 Model Development

In the following, we want to outline the architectures of the models that we have developed to perform image captioning. To justify our choices, we will provide some general background on artificial and convolutional neural networks as well as LSTMs first.

2.1 Background

Artificial neural networks have originally been inspired by biological neural networks. Mathematically, a simple two-layer fully connected neural network can be described as follows:

$$\boldsymbol{s} = \boldsymbol{W}_2 \max(\boldsymbol{0}, \boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$
$$(\boldsymbol{W}_1 \in \mathbb{R}^{N \times D}, \boldsymbol{x} \in \mathbb{R}^D, \boldsymbol{b_1} \in \mathbb{R}^N, \boldsymbol{W_2} \in \mathbb{R}^{Q \times N}, \boldsymbol{b_2} \in \mathbb{R}^Q$$

Here, $\boldsymbol{W}_{1,2}$ are weights and $\boldsymbol{b}_{1,2}$ are biases that can be optimized by some procedure such as gradient descent. One of the most important aspects is the non-linearity (*activation function*) that allows for arbitrarily complex, highly non-linear, functions when stacking many layers.

2.1.1 Convolutional Neural Networks (CNN)

Convolutional neural networks are a category of neural networks that are particularly applicable when working with images. They combine three architectural ideas to ensure some degree of shift, scale, and distortion invariance: local receptive fields, shared weights and spatial or temporal sub-sampling [6]. Using these concepts, they generally require much fewer parameters and provide better performance.
The convolutional layers can be described via three dimensions — width, height and depth. The depth describes the number of *filters* with learnable parameters. Intuitively, their purpose is to extract "shapes" or "visual features" by being slid across the input.
In addition to the convolutional layers, pooling layers are often used to decrease the width and height dimension and decrease computational complexity in subsequent layers.

2.1.2 Long Short Term Memory Cells (LSTMs)

Long Short Term Memory cells were introduced by Hochreiter and Schmidhuber and belong to the family of recurrent neural networks (RNNs) [5]. They are most commonly used when working with sequential data such as text. Conceptually, they can encode variable length inputs into a fixed-size vector and, likewise, generate a sequence from a a fixed-size encoding. RNNs take sequence of vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ and an initial state vector $\boldsymbol{h}_0$ as an input and return corresponding output vectors $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n$ and hidden states $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N$. $\boldsymbol{y}_i$ depends on all the inputs and hidden states with index $j < i$. So $\boldsymbol{y}_n$ can potentially capture information about the whole input sequence.
RNNs are fully differentiable and can be trained via gradient descent by performing *unrolling*. However, when sequences are long it becomes more challenging due to vanishing

gradients [12]. LSTMs are a specific type of RNNs which attempt to solve this problem by introducing *gates* that enable gradients to "flow" better and to detect long-term dependencies.

2.2 Vanilla Encoder-Decoder Model

The encoder-decoder image captioning method works in a simple end-to-end manner and consists of the following steps:

1. A convolutional neural network is used to extract features from the image.
2. The output of step 1 is linearly transformed to have the same dimension as the LSTM network which processes it into a sequence of words. The next words are generated based on the current time step and the previous hidden state. This process continues until and "end-of-sequence" token is generated.

For the development of our own encoder-decoder model we used resnet-152 [4], pre-trained on ImageNet. The decoder was chosen to be an LSTM network. The arhcitecture is illustrated in Figure 2.
In the training phase, for the decoder part source and target texts are predefined. For example, if the image description is **"A dog is lying on the grass"**, the source sequence is a list containing **['<start>', 'a', 'dog', 'is', 'lying', 'on', 'the', 'grass']** and the target sequence is a list containing **['a', 'dog', 'is', 'lying', 'on', 'the', 'grass', '<end>']**. Using the source and target sequences as well as the feature vector, the LSTM decoder is trained as a language model conditioned on the feature vector.
In the test phase, the encoder works in the same way as in the training phase. However, the LSTM decoder can not see the image description. Therefore it feeds back the previously generated word to the next input (*auto-regressive approach*).

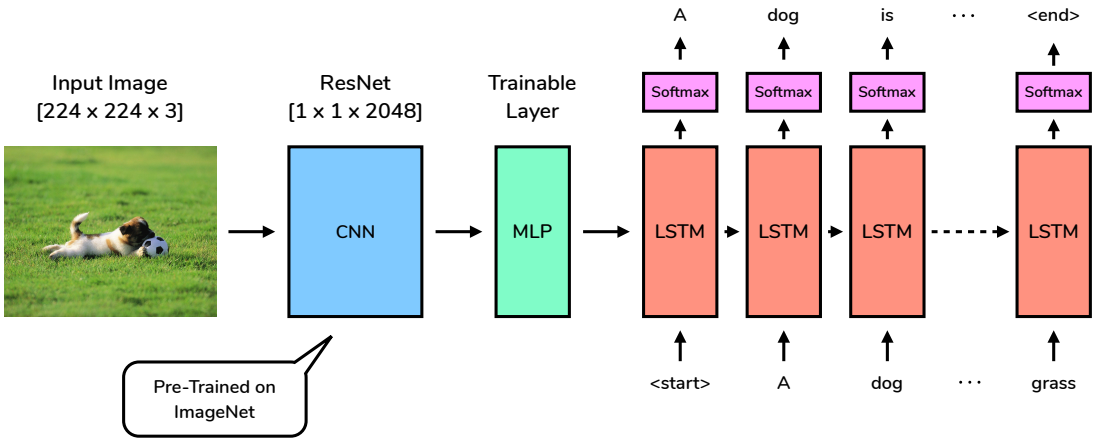

Figure 2: Vanilla encoder decoder

2.3 Attentive Encoder Decoder

A common issue with encoder-decoder architectures is the *bottleneck problem*. The encoder has to encode the input into a fixed-size vector containing all the required information for

sentence generation. As this is a non-trivial task, attention mechanisms are introduced to improve the performance.

2.3.1 Attention in Neural Networks

The main idea of attention is to break down the input into smaller parts and allow the decoder to prune unnecessary parts. This process is conditioned on the current state of the decoder, allowing different sub-parts of an input to be considered at different decoding steps, hence mitigating the bottleneck problem.
Suppose $\boldsymbol{x} = \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ are the parts of an input. At the decoding step, let $\boldsymbol{h}_{t-1}$ and $\boldsymbol{o}_{t-1}$ be previous hidden state and previous output, respectively. Then, the attention value at decoding time step t is generally defined as $\boldsymbol{a}_{\text{att}}^{(t)} = f_{\text{attention}}(\boldsymbol{h}_{t-1}, \boldsymbol{x})$. The input to the current decoding step becomes a concatenation: $[\boldsymbol{o}_{t-1}; \boldsymbol{a}_{\text{att}}^{(t)}]$. Here, $f_{\text{attention}}$ is often taken as a trainable single layer neural network which outputs a distribution over the inputs and then simply takes their weighted sum, yielding a value for $\boldsymbol{a}_{\text{att}}$. This requires the input to be naturally composed of smaller parts. In case of machine translation, these can be source sentence words. In our case of image captioning the small parts of the input decoder can "attend to" are individual pixels.

2.3.2 Visual Attention for Image Captioning

Visual attention for image captioning was proposed by Xu et al. [18]. The general architecture is shown in Figure 3. This model is a significant improvement over the basic encoder-decoder described above, although it still follows the same structure.
The first difference comes in the encoder part, where the encoding of an image is no longer a single 2048-dimensional vector, but a feature map of size $14 \times 14 \times 2048$. The image is reduced to a lower-resolution feature map where each pixel corresponds to some region of smaller pixels in the original image and has a 2048-dimensional encoding. We will use this encoding for calculating attention values in the decoder and get the distribution of attention over original image pixels.

The first hidden state $\boldsymbol{h}_0 = [\boldsymbol{c}_0, \boldsymbol{s}_0]$ is calculated by taking the mean of the image encoding and passing it through separate trainable multi-layer perceptrons (MLP). The first input $\boldsymbol{x}_0$ is just a random embedding for a special token word $\langle\text{start}\rangle$ concatenated with initial attention $\boldsymbol{a}_0$. $\boldsymbol{w}_t$ denotes the output word with highest probability at time step t.

$$\boldsymbol{E}_{\text{img}} = \text{Encoder}_{\text{CNN}}(\text{image}) \in \mathbb{R}^{14\times14\times2048}$$
$$\bar{\boldsymbol{E}} = \text{mean}(\boldsymbol{E}_{\text{img}}) \in \mathbb{R}^{2048}$$
$$\boldsymbol{c}_0 = f_{\text{init},c}(\bar{\boldsymbol{E}}) \in \mathbb{R}^{512}$$
$$\boldsymbol{s}_0 = f_{\text{init},s}(\bar{\boldsymbol{E}}) \in \mathbb{R}^{512}$$
$$\boldsymbol{E}_{\text{att}}^t = f_{\text{attention}}(\boldsymbol{E}_{\text{img}}, \boldsymbol{h}_{t-1}) \in \mathbb{R}^{196}$$
$$\boldsymbol{x}_t = [\boldsymbol{w}_{t-1}; \boldsymbol{E}_{\text{att}}^t] \in \mathbb{R}^{256+196}$$
$$\boldsymbol{h}_t = \text{LSTM}_R(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t, \boldsymbol{a}_t) \in \mathbb{R}^{1024}$$

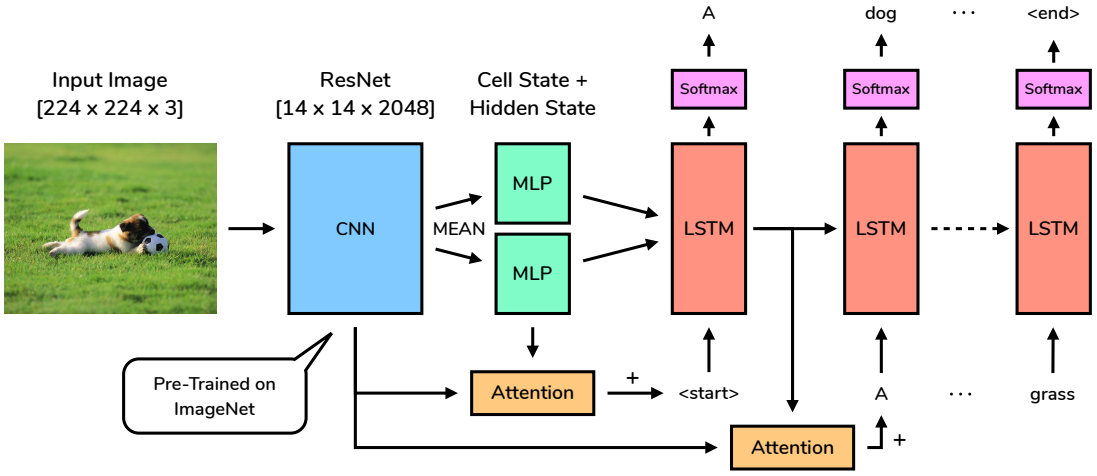

Figure 3: Encoder-decoder model with visual attention

The attention function $f_{attention}$ can be defined in various ways. Xu et al. describes a variant called hard attention [18] which randomly samples only a single pixel out of $14 \cdot 14 = 196$. This unfortunately makes the model no longer differentiable, so simple gradient descent can not be used for optimization anymore. Since the performance improvement is not significant, we chose to use soft attention defined below.

$$f_{\text{attention}}(\boldsymbol{E}_{\text{img},\boldsymbol{h}_{t-1}}) = \boldsymbol{E}_{\text{img}} \odot \boldsymbol{\alpha}$$

$$\boldsymbol{\alpha} = \text{softmax}(\boldsymbol{A}\boldsymbol{W}^{\text{att}})$$
$$\boldsymbol{A} = \text{ReLU}(\boldsymbol{E}_{\text{img}}\boldsymbol{W}^{\text{img}} + \boldsymbol{W}^{h}\boldsymbol{h}_{t-1}) \quad , with$$

$$\boldsymbol{W}^{\text{img}} \in \mathbb{R}^{2048\times1024}, \boldsymbol{W}^{h} \in \mathbb{R}^{1024\times1024}, \boldsymbol{W}^{\text{att}} \in \mathbb{R}^{1024\times1}, \boldsymbol{A} \in \mathbb{R}^{14\times14\times1024}.$$

Because $\boldsymbol{\alpha}$ is a distribution and a weighting factor of the input image, it represents which parts of the input image the decoder gets to have as an input. Using this knowledge, we can visualize this distribution over the original input image and see which part of the image the decoder uses when predicting a particular word. Examples of such visualizations from our attention model are given in Figure 4.

2.3.3 Additional Benefits of Attention

Having an attention module in neural networks often improves the model performance. This has been shown numerous times in the field of machine translation [9]. Attention also helps with the infamous gradient vanishing problem of recurrent neural networks by providing additional pathways for gradients to flow through during backpropagation. But perhaps the most important benefit is interpretability which is especially helpful when exploring model errors. Such examples are shown in Figure 5.

(a) a person is skiing down a snowy slope

(b) a herd of cattle grazing on a lush green hillside

(c) an elephant is standing in the field with trees

Figure 4: Examples of visual attention for underlined words

(a) a bird that is standing on a tree

(b) a group of elephants swimming in the water

Figure 5: Attention visualization for words that the model got wrong

2.4 Beam Search

So far, only the greedy decoding approach has been described as taking the most probable word at each decoding step as shown in Figure 6.
This is intuitively sub-optimal, because there is no way to undo mistakes at earlier steps. A less probable beginning in a sentence could have resulted in better continuation and overall a better caption. One efficient solution of this problem is called beam search and an example for decoding a sentence using beam search is shown in Figure 7.
Instead of taking a single most likely word at each time step, beam search takes k most probable words. Then, for each of those k previous words another k possible continuations will be generated. To keep number of tracked sequences from increasing exponentially, the algorithm only keeps the top k of possible sequences after each decoding steps. This does not lead to an optimal solution to the problem, but in practice, it provides significant

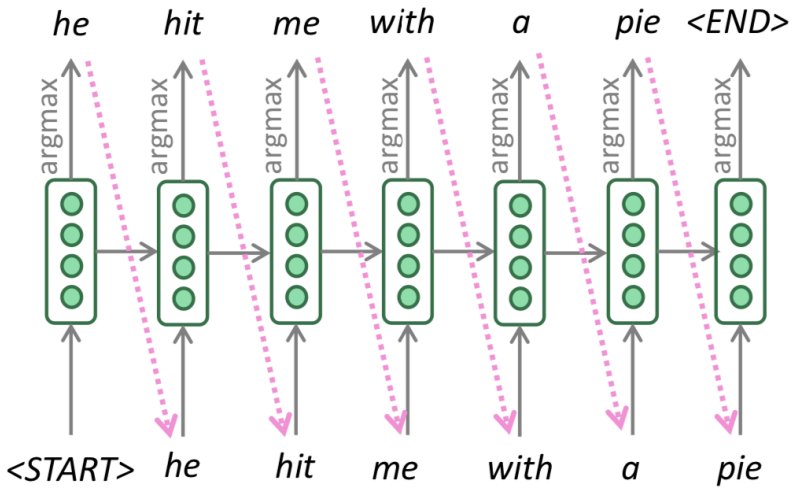

Figure 6: Greedy decoding of a sentence. Image from cs224-github.

improvements over greedy decoding.
k is often referred to as **beam width**. Using beam search decoding also allows to have multiple different sentences as a decoding result, ranked by their estimated probability.
To gain a better understanding of why greedy decoding is a sub-optimal approximation, we can resort to a probabilistic perspective. The goal of the decoding step is to find the most probable caption c^*, i.e. the most probable sequence of words $w_1, \ldots, w_n$ given the image $\boldsymbol{I}$. While beam search approximately finds the mode of the joint distribution p, greedy search only considers the factorized distribution p defined by $p_1, \ldots, p_n$, formalized as follows:

$$c^*_{\text{beam}} \approx \arg\max_{w_1,\ldots,w_n} p(w_1, \ldots, w_n | \boldsymbol{I})$$

$$c^*_{\text{greedy}} = \arg\max_{w_1,\ldots,w_n} \prod_{i=1}^{n} p_i(w_i | \boldsymbol{I}).$$

Hence, in order for c^*_{greedy} to be optimal, words in a sentence must be independent. Intuitively, this independence assumption does not hold true, supported by the evidence that uni-gram language models are well known to perform poorly.

2.5 Data Sets

Image captioning has a lot of widely available public data sets. We chose MSCOCO [8], Flickr30k [13] and Flickr8k since they are most commonly used data sets to report evaluation scores on. The statistics of these data sets can be found below in Table 1.

2.6 Evaluation Metrics

The comparison of the generated text with the ground truth is often addressed challenging in machine translation and image captioning tasks. In our project, the MSCOCO evalu-

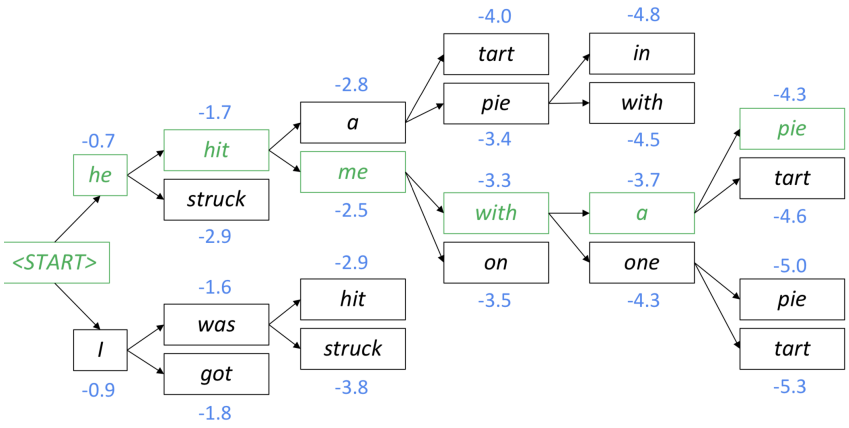

Figure 7: Decoding a sentence using beam search. Numbers given are log probabilities of words. Image from cs224-github.

Dataset	Dataset Type	# of Images	Images with # of Captions			
			4	5	6	7
MSCOCO	train	82783		82586	196	1
MSCOCO	val	40504		40373	128	3
Flick30k	train	31783	31782			
Flickr8k	train	6000	1	5993		
Flickr8k	val	952		950		
Flickr8k	test	953		952		

Table 1: Image Captioning Data Set Statistics

ation server [2] that is used for the MSCOCO Captions dataset is adjusted to be used in our system for the evaluation of the captions generated by our model. For the evaluation of our statistical model BLEU, Meteor, ROUGE-l and CIDEr scores are implemented in order to assess the quality of the generated caption. The score for a batch of images is obtained by taking the average over the batch.

2.6.1 BLEU

BLEU score is the most commonly used text evaluation metric. It analyzes the precision-based exact n-gram matches between the candidate and the references. An n-gram is a set of one or more ordered words [11]. The BLEU score per n-gram is calculated as follows:

$$B_n(C, R) = \frac{\text{count}(C_n \cap R_n)}{\text{count}(C_n)},$$

where C_n is the set of n-grams of candidate and R_n is the set of n-grams of all references per image. The matches between candidate and references are considered position independent among the references. To favor the short candidates and penalize the long candidates, a brevity penalty is used as stated in the following formula:

$$b(C,R) = \begin{cases} 1, & \text{if } l_C > l_R \\ e^{1-l_R/l_C}, & \text{if } l_C \leq l_R \end{cases},$$

where l_C is the total length of candidate and l_R is the total length of the closest reference. The overall BLEU score is then computed by taking the average of individual n-grams including the brevity penalty.

$$\text{BLEU}_N(C,R) = b(C,R) \exp\left(\sum_{n=1}^{N} w_n log(B_N(C,R))\right),$$

where $N = 1,2,3,4$ and $w_n = \frac{1}{N}$ to consider the average of all n-grams.

To sum up: even though BLEU score is easy and fast to calculate it does not correlate well with human judgement.

2.6.2 Meteor

Meteor considers sentence-level word alignments between candidate and references individually [3]. Unlike BLEU, Meteor does not only evaluate exact token matches but also considers word stems, synonyms and paraphrases.

Considering all the matches between the best scoring reference and the candidate, the precision P_M and recall R_M are computed as follows where m is denoted by the set of any match, l_C is the length of the candidate, l_R the length of the reference, and the number of chunks[1] ch. α, β and γ are free parameters such that the metric fits human judgement better.

$$P_M = \frac{|m|}{l_C} \quad R_M = \frac{|m|}{l_R} \quad F_{\text{mean}} = \frac{P_M R_m}{\alpha P_M + (1-\alpha) R_M}$$

$$\text{Pen} = \gamma.(\frac{\text{ch}}{|m|})^{\beta}$$

$$\text{Meteor} = (1 - \text{Pen}) F_{\text{mean}}$$

2.6.3 Rouge

Rouge has a set of metrics designed differently such as the measurement of the sentences based on n-gram recall, a set of shared words and skip bi-grams [7]. We adapted Rouge-l which uses a measure based on a set of shared words that exists both in the candidate and the reference, named the *longest common subsequence* (LCS). Rouge-l is calculated on a sentence-level which means it considers the score of the best matching reference per candidate. Unlike n-grams approaches, Rouge-l allows in-sequence matches instead of consecutive matches, and there is no need to predefine n-gram length as it already covers the longest possible match.

Rouge-l is referred as the F-measure calculated from the recall R_{lcs} and the precision P_{lcs} between the sequence of words of candidate and reference.

[1] A chunk is defined as the series of contiguous and identically ordered matches in both candidate and reference

$$R_{\text{lcs}} = \frac{\text{LCS}(C,R)}{l_R} \quad , \quad P_{\text{lcs}} = \frac{\text{LCS}(C,R)}{l_C}$$
$$F_{\text{lcs}} = \frac{(1+\beta^2)R_{\text{lcs}}P_{\text{lcs}}}{R_{\text{lcs}} + \beta^2 P_{\text{lcs}}},$$

Here, C is denoted as the set of words of the candidate and R similarly of the reference. $\text{LCS}(C,R)$ refers to the longest common subsequence of C and R, and $\beta = P_{\text{lcs}}/R_{\text{lcs}}$.

2.6.4 CIDEr

CIDEr evaluates how good a candidate matches the consensus of references by considering the TF-IDF weights of each n-grams [15]. By TF-IDF, the n-grams that commonly occur across all images are weighted lower as they are likely to be less informative and vise versa. A sentence will be first represented as a set of its n-grams considering one to four words. Then, a TF-IDF weighting of each n-gram is calculated with the following formula:

$$g_n(S) = \frac{S_n}{\sum_{l \in V} S_l} \log \left(\frac{|I|}{\sum_{I_p \in I} min(1, \sum_q S_q)} \right).$$

The first part holds for TF where S_n is the number of times an n-gram occurs in the sentence S, V is the vocabulary of all the n-grams. I is the set of all images and the second part holds for IDF value.
The CIDEr score for n-grams of length n is calculated by the average cosine similarity between the candidate and references as following

$$\text{CIDEr}_n(C,R) = \frac{1}{m} \sum_j \frac{g^n(C).g^n(R)}{||g^n(C)||.||g^n(R)||},$$

where $g^n(C)$ is a vector formed by $g_n(C)$ of all n-grams of length n. The varying length of n-grams is then combined by taking the average of all $CIDER_n$ scores where $n = 1, 2, 3, 4$.

$$\text{CIDEr}(C,R) = \sum_{n=1}^{N} \frac{1}{N} \text{CIDEr}_n(C,R).$$

3 Experiment and Model Management

Model development code is comprised of a very small part of any production ready machine learning system. Sculley et al. discuss the current hidden debts in general machine learning systems in details which include debts in data dependencies, reproducibility, process management, configuration and abstraction [14].
In order to tackle these technical debts we have introduced a model evaluation framework by leveraging the technologies mentioned in Figure 8 as well as a complex service architecture which we will discuss later in Section 4. This architecture allows us to develop models as part of an infrastructure which reduces the need for such dependencies.

Figure 8: Model Evaluation Framework

3.1 Data Loading Interface

As discussed in above sections, there are a variety of publicly available data sets. In order to reduce data set dependency in model development, we introduced a data loading interface as visualized in Figure 9. This interface not only allows us to abstract data dependencies away from model development, but also enables the development of models with reproducible results, as the desired data sets can easily be configured.

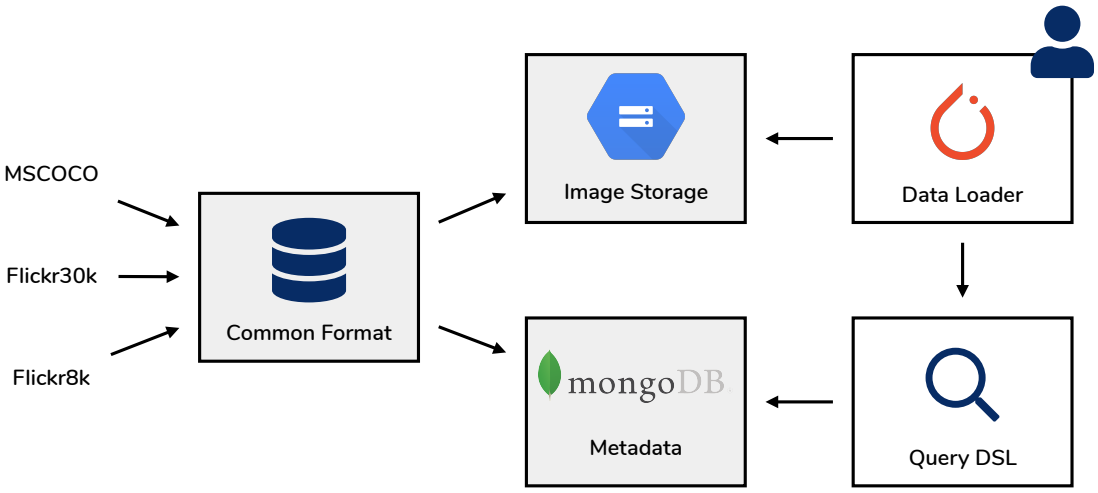

Figure 9: Data Loading Interface

All data sets are converted into a common format where the images are stored in a Google Cloud Storage Bucket. The images' URLs, along with other metadata information (such as dataset of origin, image height and width) as well as all available reference captions are then stored in MongoDB, a NoSQL database. The database is then queried by a query DSL and a PyTorch data loader provides an interface for model development, combining metadata and images.
One of the key components of the data loading interface is the Query DSL that we introduced. It allows for a very easy configuration of the data set that is used to run an experiment. The custom regular expression is comprised of the following four parts:

1. Dataset (MSCOCO, Flickr30k, Flickr8k)
2. Dataset split (Train, Validation, Test)
3. Number of items
4. Seed for sorting items

<dataset>[<datasetSplit>]-<numitems>***s***<seed>

mscoco[train]-1000***s***42

3.2 Training Interface

Image captioning models are developed either locally or in the cloud. Model training environments can differ depending on which types of resources are available. Model development should be free from any process management or environment dependencies. We introduced abstractions, which allowed models to be trained on CPUs and GPUs depending on the resources which are available.
One of the key components of any model training process are its hyper-parameters which require tuning over different experiments. Such parameters are important to be configurable for ease of training as well as be logged to ensure the results are always reproducible. The model training repository contains a Python evaluation script which can be run by providing the name of the chosen model directory where the model is defined along with its set configurations. The script automatically trains and later, evaluates the model *on a fixed pre-defined data set* based on the evaluation metrics discussed in Section 2.6.

3.3 Experiment Tracking

The models trained and evaluated across different environments can get lost until all of them are logged and stored in one place. A MLflow server acts as logging server which enables us to track all experiments, log metrics evaluation results and store models as artifact in a model repository. The desired models are then registered in the MLflow server and can later be loaded as elaborated in the next section.

4 Service Architecture

Modern production systems are expected to fulfill a broad variety of requirements. One important aspect is high availability, ensuring that the service is available at all times. In this regard, measures to achieve fault tolerance need to be taken, assuming that the physical underlying infrastructure is unreliable. At the same time, response times have to be kept low even if lots of users are concurrently accessing the service. This, in turn, requires a way to automatically increase computational resources under a high load. These aspects will be focused on in Sections 4.1 and 4.2.
Additionally, new iterations of the service ought to be deployed without taking the service down at any point and failures should be monitored continuously. These two aspects will be discussed in Sections 4.3 and 4.4, respectively.

4.1 Cloud Computing and Virtualization

Since the introduction of Amazon Web Services in 2006, cloud computing has transformed the way many companies provide services to their users. Instead of managing their own on-premises data centers, many companies have moved their workload to different cloud computing platforms. Generally, cloud computing is characterized by providing companies with an on-demand, seemingly infinite, pool of computing resources [10]. While cloud computing platforms provide many forms of services, our project was focused around their "Infrastructure as a Service" (IaaS) offering.
This offering enabled us to easily use fundamental compute resources where we could run arbitrary software. Importantly, we could also leverage cloud computing platforms' property of "rapid elasticity", enabling to "scale rapidly outward and inward commensurate with demand" [10].

4.1.1 Google Cloud Platform

The Google Cloud Platform (GCP) is the cloud computing service offered by Google. Out of the many offerings, we made use of three components in particular:

- Cloud Storage enables storing large amounts of (binary) data cheaply. As already described in the previous section, we primarily used this service to store the data used for training our models.
- Compute Engine allows for creating virtual machines (VMs) with arbitrary configuration regarding number of CPUs, available RAM as well as GPUs. In order to train our models in a reasonable time, we created VMs of appropriate size.
- Kubernetes Engine provides a fully managed version of Kubernetes, a cluster management system that we will describe in more detail in the following. The Kubernetes cluster represents the core of our production-grade architecture.

inovex provided us with a sufficient budget to make use of all these components to a reasonable extent.

4.1.2 Docker and Kubernetes

At the same time as cloud computing emerged, virtualization techniques[2] gained ever more traction — the incentive being that multiple heterogeneous applications should be deployable easily into a production environment although development happens locally.
One of the most prominent tools for ensuring a consistent environment for applications is Docker. Here, a so-called *Dockerfile* is used to specify a base operating system, dependencies of the application and install the application itself, resulting in an *image*. That image can then be used to run the application *fully isolated from all other processes* by leveraging low-overhead Linux virtualization concepts. The running image is then referred to as *container*.
Kubernetes then builds upon Docker. As a *container orchestration system*, it is responsible for distributing containers among the nodes of a (large) compute cluster depending on the containers' resource requirements. For an application to be run via Kubernetes, so-called *Kubernetes manifests* are used to specify higher-level properties of the application. While a *pod* is a collection of multiple Docker images, *deployments* manage pods to ensure a certain replica count. That means, whenever a pod goes down for some reason, Kubernetes ensures that it is restarted again, distributing traffic automatically among the running pods — a first step towards high availability. Likewise, whenever an entire node goes down, pods are re-scheduled on healthy nodes. Further features of Kubernetes and how we make use of them will be elaborated in the following sections.

4.2 System Design

The design of our system revolves around the idea of a *cloud-native application*. One of the most prominent properties of such an application is the use of a *service-oriented architecture* (SOA). Instead of building a monolithic system, responsibilities are partitioned into independent (preferably stateless) *microservices* with a single, well-defined task.
Such an SOA brings a multitude of benefits. First, potential errors are isolated and the system remains functional in the case of occasional interruptions. Second, components can be developed independently while using different programming languages — only the interface between the components has to be defined. Lastly, microservices can easily be scaled *horizontally*[3] and traffic can be load-balanced among them.
In our case, we isolate these microservices using Docker containers and orchestrate them across a cluster of nodes using Kubernetes. The entire infrastructure is eventually provisioned by the Google Cloud Platform.
Figure 10 gives a high-level overview of our architecture. All requests that are made against our service go through an edge-level load balancer provisioned via the Google Cloud Platform as well as an "internal" load balancer, a so-called *ingress controller*. This concept will be explained in more detail in Section 4.2.1.
Depending on the request, the traffic is eventually routed to the correct backend service. On the one hand, we established a frontend service serving the static files[4] needed to

[2]Here and in the following, we are referring to *type II (OS-level)* instead of *type I virtualization*.
[3]Horizontal scaling refers to scaling by adding more nodes and using more replicas of an application. This is in contrast to vertical scaling where more powerful nodes are used.
[4]In our case, this comprises HTML, CSS and JavaScript files.

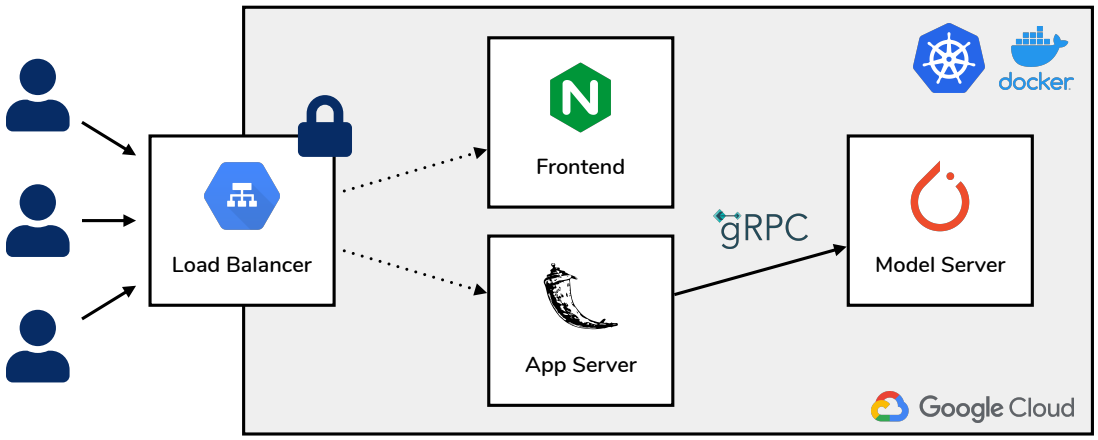

Figure 10: High-Level System Design

display a graphical user interface in the browser.
On the other hand, we use an application server managing the "business logic" of our application. Its responsibilities will be explained in more detail in the following but its main purpose is to provide a "gateway" to the model server. The model server is only reachable from inside the cluster and performs the inference for the currently deployed image captioning model. Communication is performed via gRPC, an efficient cross-language communication protocol. We will discuss the reasoning behind this choice in the following.

4.2.1 Load Balancing

Load balancing is a very important concept of cloud-native applications as it allows to mitigate the effect of isolated failures as well as to distribute a high load. When balancing between a set of resources, it periodically checks these resources' health and distributes traffic among the healthy ones. Generally, we use load balancing on different levels:

- An edge-level load balancer provisioned by the Google Cloud is responsible for distributing incoming traffic among all of the cluster's nodes. Traffic is distributed in a round-robin fashion upon every incoming TCP connection (*L4 load balancing*). Additionally, the load balancer exposes a static IPv4 address to which we point a DNS A record.

- Once the traffic hits the cluster, it is redirected automatically to one replica (of possibly multiple) *Nginx ingress controllers*. First, these controllers provide load balancing based on the content of the HTTP request (*L7 load balancing*). All requests against a route starting with "/api" are routed to the application server while all other requests are routed to the frontend. Second, they provide TLS termination. Hence, all traffic is secured via HTTPS once it leaves the cluster's network.

- Lastly, we perform load balancing when routing traffic from the application server to the model server. While Kubernetes provides L4 load balancing within the cluster

by default, the usage of gRPC requires L7 load balancing[5]. For this, we make use of Linkerd which we will discuss in more detail in Section 4.4.1. Its load balancing capabilities are very advanced — for instance, it distributes requests inversely proportional to an exponentially decayed average of response times.

4.2.2 Frontend

The frontend service is responsible for serving the static files that are used by the browser to display the graphical user interface. We use Vue.js as a JavaScript framework that enables reactive web applications. Such a web application does not require any reloads but re-renders the user interface once any data that should be displayed is changed. While using Vue.js, we opted for using TypeScript instead of JavaScript — by introducing types, many issues can be caught prior to compilation.
Using webpack, the Vue.js code is then "compiled" into HTML, CSS, and JavaScript files[6]. These files are further minimized — e.g. variable names are replaced by as few letters as possible. The reduced size eventually brings the benefit for a decreased page load time.
The static files are finally included in a Docker container running Nginx. As a web server, Nginx is optimized to serve these files as quickly as possible while providing additional features. For example, files are "gzipped" to further decrease their size.

4.2.3 Application Server

The application server provides the interface for our application programming interface (API) and acts as a "gateway" to the model server. Whenever a captioning request comes in, the image is sent to the model server and its response is returned back to the client. In the meantime, the image is uploaded to Google Cloud Storage. The link to the image is combined with the captions outputted by the model and are logged to a MongoDB instance.
The entire request is further associated with an ID. That ID can then be used to provide feedback for the model's predictions. As soon as feedback is provided via the API, it is associated with the log that was previously written to MongoDB. Eventually, this information can be leveraged to improve model performance.

4.2.4 Model Server

The model server is possibly the most central component of our architecture as it performs the most compute-heavy task. Given some image as input, it infers the captions based on a previously trained model. That model is implicitly defined simply by a name and a version number. When starting up, the model server loads the specified model from the MLflow model repository and uses it to perform inference. Based on availability and required response times, the model can either run on a GPU or on a CPU. Notably, the entire code for the model does *not need to be included in the model server*. We use PyTorch's just-in-time compilation feature to compile the model into a binary file after training. We

[5]gRPC uses HTTP/2 under the hood which reuses TCP connections. As a result, load balancing of TCP connections does not distribute load evenly and request-based load balancing is required.

[6]The resulting JavaScript code is ES5-compliant, i.e. it runs on older browsers as well.

can then use this compiled model to run inference with a "black-box-component", resulting in a set of possible captions and (non-normalized) probability scores.
As far as communication with the model server is concerned, we already described above, that the server is only reachable from within the cluster. For such intra-cluster communication, gRPC is a common choice as it enables very fast cross-language communication, decoupling components from each other. Under the hood, gRPC spawns a new thread for every request coming in. Further, it uses Google's Protobuf which is optimized for fast (de-)serialization[7] to minimize latency.

4.2.5 Autoscaling

Autoscaling describes a system's ability to automatically increase or decrease computational resources commensurate with demand. Generally, there are two kinds of autoscaling that we use in our image captioning service:

- Service-level autoscaling adjusts the number of replicas of a given service (e.g. the model server) once its resource requirements hit a certain limit. For example, we allow a model server replica to use two CPUs. We then spawn as many model servers as required to keep average CPU usage below 50% to guarantee fast response times.
- Cluster-level autoscaling, on the other hand, adjusts the computational resources that are available for all services combined. Based on the requirements of all services, entire nodes (with pre-configured size) are automatically added/removed from the cluster. Kubernetes makes sure that services are then distributed among all nodes.

4.3 Continuous Integration and Deployment

Continuous integration (CI) and continuous deployment (CD) are two concepts that are often employed to ensure a high-level of quality of production systems despite frequent code changes. The idea behind CI is to run a certain set of operations each time a code change is pushed to a git repository. In our case, these operations include automated tests, building container images and uploading these images to a so-called *container registry* where they can later be referenced. CD, in turn, refers to the deployment of new iterations of a service to the production environment without disrupting the service.
For all of our services, we adapted the GitHub Flow: the code that runs in production is always on the master branch. Whenever a new feature is developed, a new branch is opened and can be merged back into master whenever all automated tests succeed and code review has been performed. Additionally, whenever deploying into production, we issue a git tag following SemVer2. Docker images are then build for each new tag and can later be referenced again. This means, it is always possible to go back to earlier iterations of a service.

4.3.1 GitLab Runner

In order to execute the operations to be done in the CI/CD pipeline, we used GitLab runner as it integrates well with our repository management system GitLab. Every repos-

[7] Describes transforming binary data into native objects and vice versa.

itory with such a pipeline includes a ".gitlab-ci.yml" file that defines multiple steps to take whenever a specific operation happens. A common sequence of events is the following: once master is tagged, run automated tests, build the Docker image, tag it with the git tag, and deploy it to production. If any of the steps fails, the entire pipeline is stopped. Hence, the production system is not impacted until the very last step and theoretically continues to function properly.
A common issue when running such pipelines is that credentials are required, e.g. to connect to databases. In order to include these credentials as "configuration parameters" into the repository, but not leave them readable for everyone, we make use of sops. It uses Google Cloud KMS, a fully-managed cryptographic key management service, to store encrypted credentials in the repository.
Lastly, it is mentionable that the GitLab runner runs in its own Kubernetes cluster. This way, the non-continuous workload by the CI/CD pipelines do not interfere with our production system.

4.3.2 Kubernetes Helm

Helm is often referred to as package manager for Kubernetes. In our case, we mainly use it to bundle a set of Kubernetes manifests (as shortly introduced in Section 4.1.2) for each of our microservices (independently). Whenever something is changed within these manifests, Helm detects these changes and modifies Kubernetes resources accordingly with as little changes as possible. Apart from that, we considered the usage of Helm to be beneficial due to the following reasons:

- Helm makes it possible to use templates for (the rather complex) Kubernetes manifests and uses simple *value files* for configuration, reducing configuration effort to a minimum.
- Helm provides rollbacks. Whenever a deployment fails for some reason, one can very easily go back to an earlier version of the application that was running without failures.

Lastly, we use Helm to deploy some third-party components such as the MongoDB database, the MLflow server, and the GitLab runner instance. In these cases, we only need to provide the value files to configure template Kubernetes manifests.

4.4 Monitoring

Continuously monitoring workloads in a production environment is very important to identify failures and anomalies, and resolve them quickly. A particular challenge with service-oriented architectures is the distributed nature of the entire system. Failures often propagate through the system and identifying the root cause is non-trivial. In our system, we used two different kinds of monitoring.

4.4.1 Linkerd Service Mesh

A service mesh is a very recent technique towards monitoring service-oriented architectures. As microservices communicate frequently with each other to fulfill their tasks,

monitoring intra-cluster communication is very relevant. For this purpose, we use Linkerd which also happens to provide load balancing capabilities as described in Section 4.2.1.
Linkerd injects a so-called *sidecar proxy*, an additional Docker container into every service that is running. All ingress and egress traffic to and from the service goes through that proxy. This enables logging all requests and identifying anomalies such as a high failure rate.

4.4.2 Prometheus and Grafana

Prometheus and Grafana are very common tools for cluster-wide monitoring. While Prometheus scrapes and stores metrics such as the CPU usage over time (both of the cluster in total and individual services), Grafana can be used as a dashboard, visualizing these time series.

5 Results

The evaluation of our system is separated into two main functional parts which are evaluated independently. The first one regards the statistical model that actually generates the captions. Here, we evaluate the quality of the captions by comparing them to the ground truth given in the respective training data set.
The second one evaluates the performance of our system as a whole entity. In order to evaluate the system we create an artificial load testing method — manually generating system traffic while continuously monitoring software performance metrics: **response time** and **throughput** (number of requests processed per second).
Last but not least, a description of the graphical user interface and the application programming interface are given.

5.1 Model Results

We report all the aforementioned image captioning metrics for both the baseline and visual attention models. Tests were conducted on the random 10% test split of the respective datasets. We also provide the results from the reference paper [18] that our visual attention implementation is based on. We used beam width of size 10. Unfortunately, the reference paper does not state the exact beam width parameter they used. All the results are summarized in the Table 2.

Dataset	Model	BLEU1	BLEU4	ROUGE	METEOR	CIDER
Flickr8k	Baseline	35.8	4.5	32.0	12.1	6.6
	Attention	62.6	18.2	48.3	18.4	38.5
	Attention + Beam	**67.6**	**24.6**	**52.4**	**21.4**	**60.1**
	Reference	67	21.3	-	20.3	-
Flickr30k	Baseline	36.8	5.3	33.3	13.4	7.4
	Attention	61.2	16.6	48.3	16.8	25.6
	Attention + Beam	**68.4**	**23.4**	**50.0**	**19.3**	**44.3**
	Reference	66.7	19.1	-	18.4	-
MSCOCO	Baseline	41.4	11.0	43.7	19.6	35.3
	Attention	70.8	24.0	52.9	22.7	48.3
	Attention + Beam	**72.7**	**30.3**	**54.4**	**24.1**	**87.6**
	Reference	70.7	24.3	-	23.9	-

Table 2: Evaluation results of our and reference models on different metrics and datasets. The reference model evaluation numbers are taken from the "Show Attend and Tell" [18]. On every data set and metric the attention model with beam search performs the best.

5.2 Load Testing

For testing the capacity of our cluster to serve users and scale automatically we did synthetic load testing. Namely, we generated a load of 10 users simultaneously and repeatedly requesting captions for random images. Load testing results are described in

detail on Figure 11 demonstrating the ability of the system to adapt. Scaling horizontally ensures that there is no limit on how many users the system could serve provided that the number of machines in the cluster scales commensurate with the request rate.

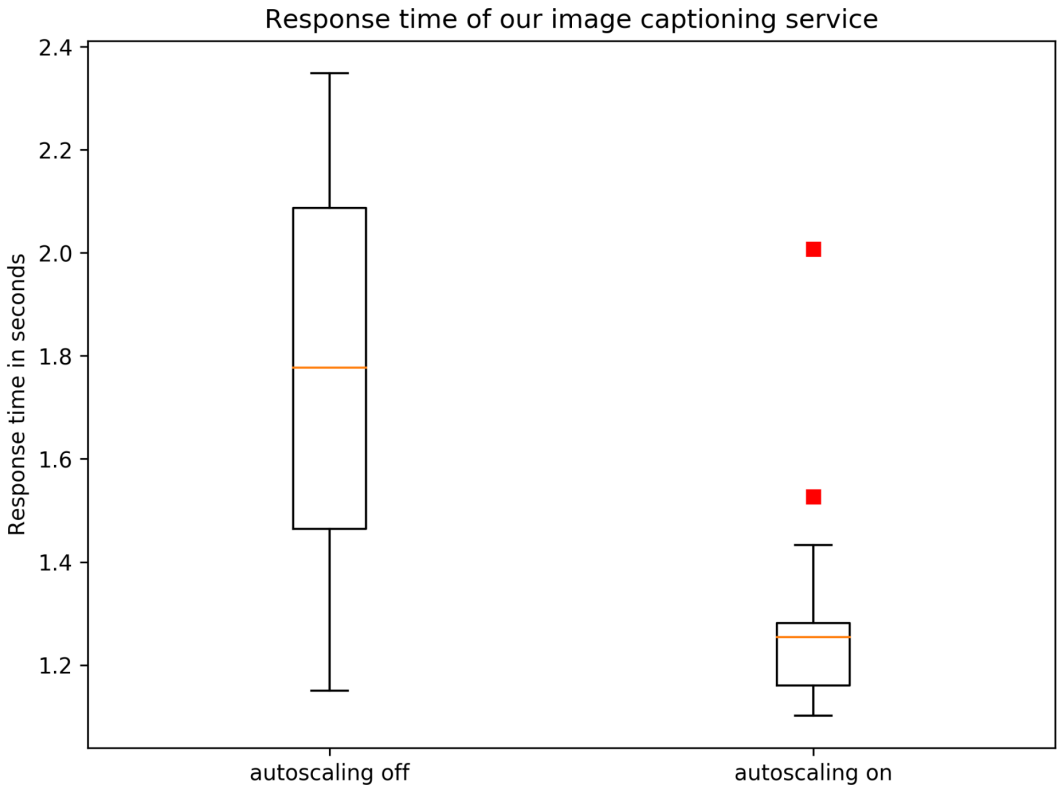

Figure 11: The effect of auto-scaling on response time of image captioning service under high load. Minimum response time stays about the same, but mean and maximum time improves significantly.

5.3 Graphical User Interface

Graphical user interface(GUI) refers to the web-based interface where the end-users can interact with the Image captioning system. It is essential to have an easy to use and responsive UI design.

In Figure 12 the following possible user interactions are visualized:

1. User visits the website at https://dilab.inovex.de/ and uploads an image by clicking on the upload icon button.
2. User clicks on "Caption Image" button.
3. User receives the listed captions for the image.
4. User visualizes the attention of the word "airplane" of the first caption by either hovering or clicking on the word.
5. User visualizes the attention of the word "people" of the last caption.
6. User either clicks on the like button to leave a feedback for the best interpreted caption or goes back to step 1.

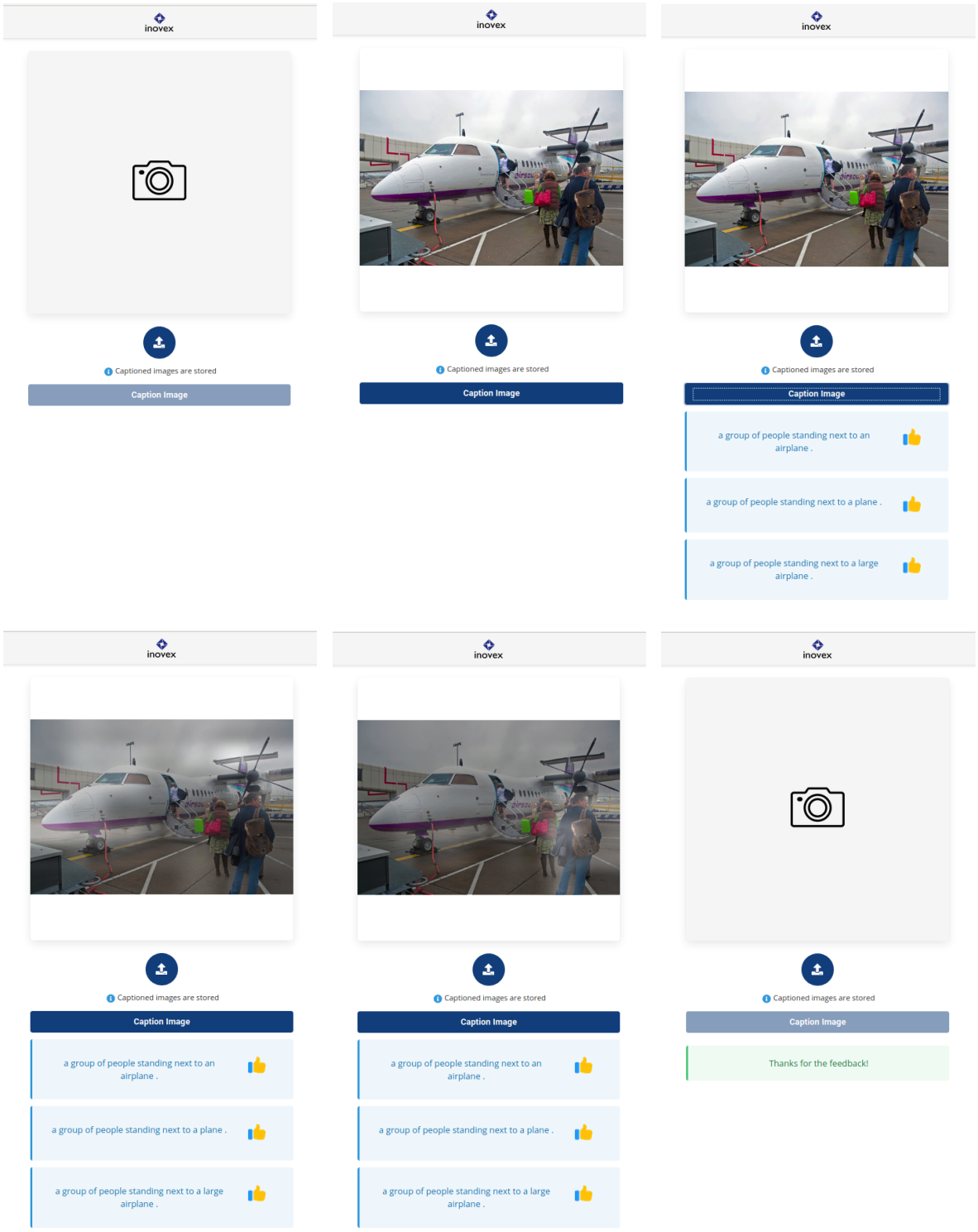

Figure 12: GUI with user interactions

5.4 Application Programming Interface

The Image captioning System is reachable through an application programming interface (API) as well. Our image captioning API is a way of letting developers reach our system

and integrate it with their own applications by sending HTTP requests and receiving HTTP responses. There are two endpoints provided by the image captioning API as listed below:

- A user sends an HTTP POST request at https://dilab.inovex.de/api/caption including the image as a file in a JSON data format with the 'image' key. A potential response is shown as an example below.

```
{
    "logId": string,
    "captions": [
        {
            "attentions": list of attention images,
            "probability": integer,
            "text": string
        },
        {
            "attentions": list of attention images,
            "probability": integer,
            "text": string
        },
        {
            "attentions": list of attention images,
            "probability": integer,
            "text": string
        }
    ]
}
```

- A user sends an HTTP POST request at https://dilab.inovex.de/api/feedback as a JSON form data including the logId of the image with 'logId' key and the best caption text as a feedback with 'feedbackCaption' key. As a response, a success status code is returned.

6 Conclusion

Eventually, our service is driven by a state-of-the-art deep learning model for image captioning. Starting from a simple encoder-decoder architecture consisting of a CNN followed by an LSTM responsible for text generation, we later included visual attention, resulting in improved performance. State-of-the-art was then achieved by moving from greedy search to beam search to heuristically extract the most likely sentence from the model's output distribution instead of the sequence of most likely words.
During model development, we made use of an experiment management system and a dedicated evaluation framework. This way, we could track all experiments and easily ensure that they are evaluated on the same (unknown) data such that the best performing model can easily be identified.
The most advanced model was then deployed within a service-oriented architecture (SOA) that ensures high availability under the assumption of unreliable infrastructure. At the same time, we leveraged the benefits of an SOA to ensure elasticity to respond to varying demand and continuous deployment of individual services to facilitate development.
In order to provide ordinary users as well as developers to use our service conveniently, we introduced both a graphical user interface hosted at https://dilab.inovex.de and an HTTP API. For the latter, we even provided easy-to-use SDKs in Python and TypeScript.

6.1 Future Work

As far as the development of the model is concerned, plenty of future steps can be taken. On the one hand, it might be beneficial to include some kind of uncertainty measure in the output of the image captioning model. This way, the model can abstain from generating meaningless captions if the image does not provide enough information (e.g. a completely black picture) or the image is well outside the distribution of the images that the model was trained on (e.g. diagrams). In order to prevent hand-picking a probability threshold for the model output to classify an image as "non-captionable", one would have to include such images into the training procedure and adapt the architecture of the model.
On the other hand, external knowledge sources can be leveraged to generate more meaningful captures as presented by Wu et al. in 2017 [17]. External knowledge sources can identify better connections between the entities present in the image and thus enable more meaningful captions.
Furthermore, a potential model improvement is the incorporation of user feedback which we currently log into the database. The model can be trained continuously to improve its performance according to the user feedback. In particular, the model will be able to improve its performance on images that are outside of the distribution of the data set that it was trained on. Also, the model might be trained to improve its performance on a particular set of entities.
From an engineering perspective, a possible improvement is the rewrite of the model server in C++ instead of Python. While the inference time itself is expected to decrease only by a small margin as the compute-heavy steps are executed on the GPU anyway, one can expect a significant drop in latency as well as memory consumption. Eventually, this enables a higher throughput and reduction in costs, following a decreased demand for computational resources.

References

[1] Peter Anderson et al. "Bottom-Up and Top-Down Attention for Image Captioning and Visual Question Answering". In: *CVPR*. 2018, pp. 6077–6086.

[2] Xinlei Chen et al. "Microsoft coco captions: Data collection and evaluation server". In: *arXiv preprint arXiv:1504.00325* (2015).

[3] Michael Denkowski and Alon Lavie. "Meteor universal: Language specific translation evaluation for any target language". In: *Proceedings of the ninth workshop on statistical machine translation*. 2014, pp. 376–380.

[4] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[5] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[6] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[7] Chin-Yew Lin. "ROUGE: A Package for Automatic Evaluation of Summaries". In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. URL: https://www.aclweb.org/anthology/W04-1013.

[8] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. 2014. arXiv: 1405.0312 [cs.CV].

[9] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. "Effective Approaches to Attention-based Neural Machine Translation". In: *CoRR* abs/1508.04025 (2015). arXiv: 1508.04025. URL: http://arxiv.org/abs/1508.04025.

[10] Peter Mell and Tim Grance. "The NIST Definition of Cloud Computing". In: *Special Publication 800-145* (2011).

[11] Kishore Papineni et al. "BLEU: a method for automatic evaluation of machine translation". In: *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics. 2002, pp. 311–318.

[12] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "Understanding the exploding gradient problem". In: *CoRR* abs/1211.5063 (2012). arXiv: 1211.5063. URL: http://arxiv.org/abs/1211.5063.

[13] Bryan A. Plummer et al. *Flickr30k Entities: Collecting Region-to-Phrase Correspondences for Richer Image-to-Sentence Models*. 2015. arXiv: 1505.04870 [cs.CV].

[14] David Sculley et al. "Hidden Technical Debt in Machine Learning Systems". In: *NeurIPS*. 2015, pp. 2503–2511.

[15] Ramakrishna Vedantam, C. Lawrence Zitnick, and Devi Parikh. "CIDEr: Consensus-based Image Description Evaluation". In: *CoRR* abs/1411.5726 (2014). arXiv: 1411.5726. URL: http://arxiv.org/abs/1411.5726.

[16] Oriol Vinyals et al. "Show and Tell: A Neural Image Caption Generator". In: *CVPR*. 2015, pp. 3156–3164.

[17] Qi Wu et al. “Image Captioning and Visual Question Answering Based on Attributes and External Knowledge”. In: *TPAMI* 40.6 (2017), pp. 1367–1381.

[18] Kelvin Xu et al. “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention”. In: *ICML*. 2015, pp. 2048–2057.