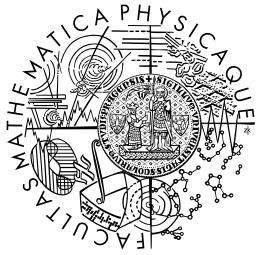


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Samuel Michalik

Deep Learning and Visualization of Models for Image Captioning and Multimodal Translation

Institute of Formal and Applied Linguistics

Supervisor of the bachelor thesis:	Mgr. Jindřich Helcl
Study programme:	Computer Science
Study branch:	IOI

Prague 2019

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to thank my supervisor Jindřich Helcl for his guidance, patience and kind and positive attitude. I would like to thank my parents for their unconditional support. I would like to thank Ivan Adam, who knows the meaning of sacrificing for one's goals, for being an inspiration to me and supporting me when I needed it the most. I would like to thank the head of the faculty for postponing the thesis submission deadline in response to the coronavirus pandemic, without which I would probably miss the date. I would like to thank the whole staff of the faculty for their dedication to spreading knowledge, particularly Tomáš Holan, Pavel Ježek and Milan Straka. Finally, I would like to thank MetaCentrum for providing the computational resources for the entire experimental part of this thesis.

Title: Deep Learning and Visualization of Models for Image Captioning and Multimodal Translation

Author: Samuel Michalik

Institute: Institute of Formal and Applied Linguistics

Supervisor: Mgr. Jindřich Helcl, Institute of Formal and Applied Linguistics

Abstract: In recent years, the machine learning paradigm known as deep learning has proven to be well suited for the exploitation of modern parallel hardware and large datasets, helping to advance the frontier of research in many fields of artificial intelligence and finding succesfull commercial applications. Deep learning allows end-to-end trainable systems to tackle difficult tasks by building complex hierarchical representations. However, these internal representations often avoid easy interpretation. We explore the possibilities of interpretable visualizations of attention components and beam search decoding at the task of image captioning and multimodal translation and build an application – Macaque, that can be run as an online service, to meet this end. Furthermore, we propose a novel attention function formulation, called scaled general attention. We experimentally evaluate scaled general attention along common attention functions on four different model architectures based on the encoder-decoder framework at the tasks of image captioning and multimodal machine translation. We utilise Macaque during qualitative analysis.

Keywords: deep learning vizualisation image captioning multimodal translation

Contents

1. Introduction

1.1 The Problem

The Big Dream. We will begin by briefly describing a problem, which can be seen as the context of the subject of the following pages. After that, we will proceed to give a formal specification of the problems which this thesis is directly concerned and upon which it aims to build its contribution.

The idea of creating intelligent machines has appealed to people for a long time. Besides curiosity, engagement in this endeavour can be justified by the potential prospects of such an invention. Recently, methods which make sense of the unprecedented volumes of data that is being collected in all imaginable domains are in high demand. The evergrowing complexity and specialization inside scientific fields such as physics, mathematics or others increase the time it takes for newcoming researchers to gain expertise on par with the current level of understanding and begin to contribute to the field. In the distant future, artificial agents could systematize and aid the process of education of prospective scientists or even make advances in the field on their own.

The tasks of recreating and understanding intelligence are closely linked. By sharpening our notions of intelligence and how the mind works, we advance our understanding of ourselves.

Towards Artificial Intelligence. A reasonable objection is that a systematic approach towards these goals requires one to first make it clear what is meant by terms as "intelligence" or "thinking". For the purposes of this thesis, we will sidestep this issue by assuming a position based on the Turing test[1](Turing, 1950); an intelligent machine is one that passes the Turing test. What steps can we then take towards building such a machine?

In the first half of the history of artificial intelligence, the dominant paradigm was the *knowledge-based* approach. The behavior of programs was determined from hard-coded knowledge in the form of basic facts from which conclusions were drawn by the use of inference rules. This approach has however been unsuccessful to solve the tasks which people solve most intuitively. Another approach, *machine learning*, has achieved greater success in recent times. Machine learning is a paradigm in which programs learn to solve a task by discovering patterns in data. Central to this approach are *models*. A common type of machine learning models are mathematically parametrized functions, we will be dealing with these in this thesis. The task which the model is given to solve is encoded in the form of the function's inputs and outputs. For example, in machine translation the model could be a mapping from sentences in language A to sentences in language B. The parameters of the model are then gradually tuned by a general optimization algorithm, such as stochastic gradient descent, towards values that approach optimal performance at the task. This process is referred to as *training*.

[1]The Turing test tests a machine's ability to exhibit intelligent behaviour based on the evaluation of a human judge. The judge communicates with the machine and another human being trough a computer screen and keyboard. The machine has passed the test if the evaluator is not possible to reliably tell it apart from the human.

During training the model commonly encounters many instances of the problem it is designed to solve. On nontrivial tasks, it usually takes many examples for the model to learn the desired behavior. For that reason, machine learning is a data-hungry approach.

Historically, the input to these algorithms have been carefully crafted features, which described the relevant aspects of the task at hand. A model could aid a bank's decision whether to give a loan to a candidate based on features such as monthly salary and level of acquired education. Manually crafting feature representations can however be very time-consuming. In addition, there are many problems in which the desired features are hard to specify. A classic example is object detection. How does one prescribe rules to identify a West Terrier from a picture, but not to mistake it with a Wheaten Terrier? It is often hard to describe formally what we recognize very intuitively. Machine learning algorithms can be used to come up with representations of data on their own. This scenario is referred to as *representation learning*.

Machine learning models can be mathematical functions and often these functions are expressible as a composition of simpler functions. This will become evident once we look at exact mathematical definitions for some of the models. Under such a view, models can be seen as solving the given task by learning a series of representations of the data, where each function is a transformation that yields a new representation. The initial representations learn to express basic concepts extracted from the input data. Subsequent representations describe increasingly higher-level concepts. Machine learning which deals with this type of models, where the hierarchy of concepts grows in complexity, is called *deep learning*.

Machine learning and deep learning in particular has in the last few years came to the forefront of attention thanks to successes in numerous subfields of computer science where these methods helped to advance the state of the art. The current wave of interest was ignited by the success of deep convolutional neural networks in object recognition on the ImageNet dataset (Krizhevsky et al., 2012). In tasks with more structured label spaces such as language modelling, machine translation and speech recognition, complex multistep piplenes were replaced by end-to-end trainable neural alternatives based on recurrent neural networks (Bengio et al., 2003; Sutskever et al., 2014; Graves et al., 2013), and are now the standard in commercial systems. More recently, in speech synthesis, deep learning models have been able to produce nearly human speech with as little as 24 hours of training data (Shen et al., 2017).

Even though these breakhrough successes have come in recent times, many of the key ideas behind deep learning, such as artificial neural networks and gradient descent are much older. Besides theoretical advances, the remarkable achievements of the last years were made possible by giant improvements in computer hardware and large-scale labeled datasets. Even if no progress should be made on the theoretic side, new datasets and hardware will continue to drive the performance improvement of current systems.

An objection againts the main current of machine learning is that it is "merely" pattern recognition. Models are driven to predict inside the boundaries of the task which they are given to solve, but lack the characteristics which make human cognition so remarkable; the ability to build mental models that allow broader

generalization, learning to understand the problem by supporting explanation as opposed to learning to simply solve the task. It is possible to increse the generalization ability by scaling up the amount of training data (Radford et al., 2018) However, increasingly larger amounts of data will be needed to sustain constant generalization imporvement. In contrast, people can often acquire the understanding of a new concept after seeing only one example.

An Algorithmic Link Between Vision and Language. Under the flag of building increasingly intelligent machines, there are many directions to pursue. One of them is the problem of algorithmically relating the visual domain and natural language. This is also the concern of this thesis.

An intelligent machine will necessarily need exposure to large amounts of data, on which it will build its knowledge of the world. The two primary sources of information are the physical domain and the Internet. The primary interface to the former is raw visual data, while in the case of the latter it is language. Therefore, advancing methods that learn to understand information coming from the interfaces to these two domains are crucial, as well as means to relate the two modalities.

In addition to endowing our artificial agents with knowledge, language seems as the ideal communication channel between future machines and people. As the machines become more complex, ways to interact with them meaningfully will become more important.

What are the concrete steps towards this goal? In order to make progress, we need to explicitly specify what we expect from such systems. What will be the input and the desired output. How will we measure the performance of these systems?

Possible answers to these questions come in the form of two tasks that have recently been gaining growing recognition, *image captioning* and *multimodal translation*.

Image Captioning and Multimodal Translation. Image captioning is a task that aims to model the relation between vision and language. In short, the goal of the task is to produce natural language descriptions of the content of images. The description, or caption, should contain information about the most salient aspects of the image.

On input, we receive an image and the goal is to produce a sentence in natural language that describes the contents of the image. During training, each image is accompanied by one or more reference captions. After the model generates its caption conditioned only on the input image, the measure of success of the model at the task is given by the degree of agreement between the model's output and the reference captions. The agreement can be measured in multiple ways. The most reliable is human judgment, however due to being expensive and lengthy, automatic evaluation metrics are generally used to evaluate the performance of a model.

A related task to image captioning is multimodal translation. On input, in this case, the image is accompanied by a caption in a source language. The task is to produce a caption describing the image in a target language. As in the case of machine translation, only one reference caption is used to compute the model

performance. Multimodal translation can be seen as standing between image captioning and machine translation.

We will dive deeper into image captioning and multimodal translation in Chapter 2, but for now it suffices to know that the state-of-the-art approaches to these tasks are all based on the deep learning paradigm. Deep neural networks based on the encoder-decoder architecture in which a convolutional network module or a transformer module is used as the encoder to produce a representation of the image which is then decoded by a neural language model, implemented as a recurrent neural network or a transformer decoder. In the case of convolutional encoders with recurrent decoders, an attention module is usually added to boost performance even further.

The performance of state-of-the-art models is remarkable given the difficulty of the task, however in comparison to humans the results are substantialy inferior.

Model Interpretability. Large deep neural networks are sophisticated systems that carry out complex decisions. Methods that shed light into the decision process are crucial. Take for example the case of self-driving cars; is a model that flawlessly passes all tests, even though the process by which it makes decisions is not properly understood, ready to be deployed into traffic? In contrast to classic programming, the correctness of these algorithms cannot be formally proved. Interpretability is the degree to which we can understand the "reasoning" by which the model arrives at its decisions.

The curiosity to understand the inner workings of deep learning models has motivated plenty of research. There have been investigations into the semantic meaning of individual neurons by searching for examples that maximize activations. Adversarial examples created from examples in the training set by small guided perturbations, undistinguishable by humans, but causing neural networks to misclassificate, have refuted certain conjectures about the way in which neural networks generalize (Szegedy et al., 2013).

Some components naturally allow a degree of interpretation, for example the attention mechanism. When generating a word in the target caption, parts of the image are associated with scores which describe the importance of that region in generating the current word. We can visualize "where the model looks" based on these scores.

A sibling issue to interpretability in model understanding is clear presentation of variables that already have a meaning. An example of this is the beam search decoder which manages a tree of candidate hypotheses when generating an output sequence. Usually, we only take the best hypothesis, however looking at the tree of intermediate candidate hypotheses can provide additional insight into the model's inference process.

Visualization Tools. In addition to possessing theoretical methods for interpreting models, the next step is implementing these into practical tools that can be used to enhance research.

In the ideal case, the researcher would have at his disposal easy to use tools that could be attached to his models at runtime and upon extracting the desired variables from the model, would provide readable, intuitive visualizations. The

researcher's code should be completely independent of the tool, and the tool should be integrated as easily as possible.

At present, there is a general lack of such tools. It may be explained by the fact that there are many software frameworks for deep learning research, existing tools are tied to only one of them, for example Tensorboard. Additionaly, truly generally applicable interpretability methods are sparse, if any. Most are tied to a specific component of the model's architecture, such as the beam search decoder. This futher reduces the potential for general tools, as most features would be limited to only specific architectures on specific tasks.

Another problem has to do with the implementation aspect. Given the specific framework would allow the extraction of the variables in demand for interpretation, it is still necessary to establish an interface between the model and the tool.

1.2 The Goals

In the previous section we have given a picture of the general context in which this thesis is situated. Now it is time to formulate the specific goals which it hopes to accomplish and the ways by which it intendes to do so.

The goal of this thesis

Explore the possibilites for interpretability and intuition building related to the inference process of deep learning models at the tasks of image captioning and multimodal translation on both the instance-level and the dataset-level. Provide a mean to do so systematically, easily, independentnly of the evaluation metric, dataset or model architecture, with minimal demands on the side of the user, with future extendability in mind.

The main contribution which this thesis aims to present is *Macaque*, an application designed to help researchers and machine learning practitioners build intuitions about the workings of their models.

Macaque attempts to be an aid at research, while preserving independence from the research code. It focuses on the methods of interpretability of model inference at the tasks of image captioning and multimodial translation, additionaly seeking to organize and present information that is available in the common setting of these tasks in a clear way. It attempts to provide the most relevant possible features at minimal demands on the user's part.

Macaque is designed to function with models in the Neural Monkey framework. However, upon satisfying a minimal interface, any model written in Python can be supported trough a plugin system.

To test Macaque, we perform our own experiments, on four different model architectures and three different attention functions and analyzing our results using the application. The experiments are the subject of the second chapter.

The following table lists the main features contained in the tool.

A list of Macaque's features

- Visualization of attention distribution.
- Visualization of candidate hypotheses during beam search decoding.
- Natural support for Neural Monkey models.
- Support for any Python model trough a plugin system.
- Support for both image captiong and multimodal machine translation models.
- Built-in image preprocessing functionality.
- Exploring different model setups by combining preprocessors, encoders and decoders.
- Automatization trough configuration files.

1.3 The Outline

Chapter 2 gives the theoretical background on deep learning necessary for the rest of the thesis.
Chapter 3 contains the experimental part of the thesis. We explain the framework under which the experiments are performed and propose a new attention function formulation, called *scaled general attention*. We experimentally evaluate it at the tasks of image captioning and multimodal translation.
Chapter 4 presents Macaque, a tool for providing insight into deep learning models. The first part of the chapter is the user's guide. In the second part, we present the software design from the developer's point of view.
The thesis is closed by the **Conclusions** chapter in which we look back at what has been accomplished and lay out potential future directions.

2. Neural Networks

In this chapter we give a brief introduction to the theoretical background behind the models we deploy in our experiments and for which Macaque is designed. All of these models are artificial neural networks (ANNs)[1]. The first section defines an artificial neural network at a general level. Subsequent sections introduce specialized types of ANNs, and involved concepts relevant to this thesis. For an introductory textbook on neural networks and more generally deep learning, we refer the reader to Goodfellow et al. (2016).

2.1 Neural Networks in Supervised Learning

Neural networks are mathematical functions defining a mapping $\mathbf{y} = f(\mathbf{x}; \theta)$ where $\mathbf{x}$ is a real-valued vector of the network inputs and θ is the set of the network's parameters. The parameters θ are learned through an optimization algorithm so that the network approximates some function f^*.

The degree to which the network is successful at approximating f^* on a set of inputs $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ is measured by a *loss function*. An example loss function is the *mean squared error*:

$$MSE(\theta) = \sum_{i=1}^{n} (f(\mathbf{x}_i; \theta) - f^*(\mathbf{x}_i))^2.$$

The neural network's ability to approximate a desired function on a set of instances is guided by methods that systematically adapt θ. An example of such an optimization algorithm that minimizes the loss function of a network is *gradient descent*. Given a loss function $L(\theta)$, the gradient descent algorithm updates the values of θ in the opposite direction to the gradient of L with respect to θ. Formally, in the gradient descent algorithm the following rule is applied, until a stopping condition is met:

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$$

where α is the *learning rate*, a hyperparameter that controls the magnitude of the steps.

In practice, more modern and effective alternatives of gradient descent are used, such as *stochastic gradient descent* (SGD), or the Adam algorithm (Kingma and Ba, 2014).

An algorithm called *backpropagation* (Rumelhart et al., 1988) is used to compute the partial derivatives of the network's parameters with respect to the loss function. It is based on the chain rule of calculus and dynamic programming.

2.2 Densely-connected Neural Networks

Densely-connected neural networks, also called fully-connected neural networks are the most basic type of NNs. They compute a geometric transformation of the

[1] We are going to use the term "neural networks" interchangeably with "artificial neural netowks" throughout this thesis, always referring to ANNs

input by repeated application of affine transformations and nonlinearities. The output of each dense layer is computed using from the activation values of the neurons in the previous layer. Formally, given a real valued input vector $\mathbf{x} \in \mathbb{R}^d$, which can be the network input, or the output of a previous layer, the output of a neuron in a dense layer is computed as:

$$h = \phi(\mathbf{w}^\top \mathbf{x} + b)$$

where $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$ are the weight and bias terms belonging to the network's set of parameters θ which are learned durning training. ϕ is a nonlinearity, called an *activation function*. The output of the whole layer can be written down using matrix multiplication:

$$\mathbf{h} = \phi(W^\top \mathbf{x} + \mathbf{b}).$$

In this case $\mathbf{h} \in \mathbb{R}^h$ and $W \in \mathbb{R}^{d \times h}$, where h is the number of neurons in the dense layer, referred to as the layer's *size*. The bias term $\mathbf{b} \in \mathbb{R}^h$ completes the affine transformation.

2.3 Convolutional Neural Networks

Initially developed for image classification (LeCun et al., 1998), Convolutional Neural Networks (CNNs) contain hard-coded assumptions about the topology of the input data representation. These assumptions make them more suitable than fully-connected NNs for data types where such topology is present, such as images, video, text or sound, in addition to being more time and memory efficient.

In densely conneceted NNs, each neuron has connections to all units in the previous layer. If we were trying to build a neural network to classify RGB images of 224×224 pixels using a fully connected layer as the next layer, a neuron in this layer would be connected to all $224 \times 224 \times 3 = 150,528$ units of the input, meaning we would have to store $150,528$ weight parameters (typically 16 or 32 bit floating point numbers). As each neuron in the hidden layer is connected to all of the input, scaling the values with unique weights, the total number of parameters in this single layer would quickly grow into millions.

The assumptions on which the design of CNNs is based are that complex features emerge from the interconnection of simpler ones, boiling down to the most basic ones which are local. Moreover, these simplest features can occur anywhere in the input. As an example, consider the task of recognizing whether a cat is present in an image. At the highest level, the cat's presence could be determined by detecting its head which could be detected by recognizing its eyes, ears and whiskers, which in turn could be determined by recognizing certain textures, colors or curves or combinations of these. The presence of textures or edges can be established based on information only from a small region of the input.

Standard CNNs are built from *convolutional layers* and *pooling layers*, the network's architecture often ending with fully-connected layers. Additionally, the nonlinear activation functions which are part of the description of fully-connected

layers are in the context of CNNs most often considered independently as a layer of their own.

A basic convolutional architecture would consist of a sequence of alternating convolutional layers and nonlinear transformation layers, followed by a pooling layer. This pattern could be repeated multiple times and the network would finish with a few fully-connected layers.

Convolutional layers. Convolutional layers are based on the convolution operation. Formally, convolution combines two functions, the *input* and the *kernel* and its discrete one-dimensional version is given by

$$\text{conv}(i) = \sum_{m=-\infty}^{\infty} \text{input}(i)\text{kernel}(i-m).$$

Intuitively, the kernel slides along the input and at each position i the dot product between the input and the kernel is computed. To demonstrate how this idea translates to the context of neural networks, let's consider an image represented by a $W_1 \times H_1 \times D_1$ array of floating point numbers, and refer to it as *img*. W_1 and H_1 represent its pixel width and height respectively and D stands for the number of channels used to represent the color at a particular pixel.

With fully-connected neural networks, to process this type of data, we would have to flatten the array to a vector, then we would multiply it by the weight matrix, add the bias vector and pass it trough an elementwise nonlinearity, which would yield us a new representation in the form of a vector of activations. Notice how the original input topology is irrelevant to the fully-connected network, as the neurons of the hidden layer are connected to all of the input units.

The convolutional layer operates on the original array *img* and its output (let's refer to it as *conv*) has the shape $W_2 \times H_2 \times D_2$. Each $W_2 \times H_2$ slice of the output along the D_2 dimension is called a *feature map* and is associated with a particular *kernel*, also called a *filter*. The kernel will be a 3D array of learnable parameters with shape $K \times K \times D_1$, the third dimension is equal to the third dimension of the previous layer (the input in this case). K is usually much smaller than either of H_1 or W_1. We refer to the kernel associated with the feature map $conv_{*,*,k}$ as ϕ^k. The value of a neuron in the convolutional layer is given by

$$\text{conv}_{i,j,k} = b_k + \sum_{l=1}^{D_1} \sum_{m=1}^{K} \sum_{n=1}^{K} \phi^k_{m,n,l} \text{img}_{i \cdot s+m, j \cdot s+n, l}.$$

The values of the neurons in the convolutional layer are computed by sliding the kernel along the image and computing the dot product at each location. The parameter s is the *stride* – the step size by which the kernel moves along the input. b_k is the bias term.

The particular input units which take part in the computation of a convolutional layer neuron's value are called the neuron's *receptive field* and encompass the idea of *local connectivity*.

Local connectivity. Local connectivity means the layer neuron's connections are restricted only to a continuous subset of the previous layer. With convolutional neural networks this subset is a continuous block that extends along the whole depth dimension. Since the receptive field is small (usually 3×3), this restriction on the input forces the kernels in the first layers to learn to extract features at this scale, such as edges, corners, textures and colors.

Consider two succesive convolutional layers with kernels of size 3×3 and a stride of 1. The first convolutional layer operates on the input image. A neuron in this layer has a corresponding 3×3 receptive field in the image. A neuron in the second hidden layer has the same receptive field over the first convolutional layer, which given a stride of 1, translates to a receptive field of 5×5 in the original image. With a stride of 2, this would be 7×7. This illustrates how convolutional layers in greater depth come to describe larger portions of the image, learning to extract more and more complex features such as the presence of a whole cat.

Weight sharing. As can be seen from the equation above, the parameters of the kernel and the bias are shared across the entire feature map. Besides the obvious advantage of reducing the number of trainable parameters compared to a fully connected layer, this leads to the feature map being a homogenous representation of the presence of a particular feature which the kernel has learned to detect. As such, the network learns to be translation invariant with regards to the input – once a filter is trained to detect cat tails, it will be able to detect them no matter where in the image they appear.

Recall that the convolutional layer consists of multiple feature maps. Each feature map is parametrized by a kernel and a bias. It is common that convolutional layers contain many feature maps (even hundreds). As kernels operate over all of the feature maps of the previous layer, they can combine this information to detect the presence of complex features.

Pooling layers. Pooling layers have two functions. First, they reduce the spatial dimensionality of the feature maps, helping to decrease the number of parameters of the network. This makes running the network less computationally demanding and reduces overfitting. Second, they allow the network to become more robust to transformations of the input. This can be understood once we see how they work.

There are several types of pooling layers, the most popular is the *max pooling layer*, also notable is the *average pooling layer*. Max pooling layers are parametrized by two hyperparameters – the size of the receptive field $P_1 \times P_2$ and the stride s. In contrast with convolutional layers, pooling layers always preserve the size of the depth dimension (the number of feature maps). The neurons of a max pooling layer simply contain the maximum value of the neurons in their receptive field.

If we followed a convolutional layer of size $W \times H \times D$ with a pooling layer with receptive fields 2×2 and stride 2, the pooling layer would be of size $W/2 \times H/2 \times D$ – the number of activations would be reduced to a quarter.

The pooling operation discards a lot of information from the intermediate representations. For that reason the number of convolutional layers is always larger than the number of pooling layers. However, pooling layers make the

network more robust in the cases when it does not matter precisely where a particular feature occured, just that it occured in some region. This way the network can learn to be invariant to small rotations a skewing. The max pooling layer can be understood as an existential quantifier, picking a single value from the receptive field, while the average pooling layers act as a universal qunatifier.

2.4 Recurrent Neural Networks

Many tasks involve sequences as input or output. For example, in machine translation, the input to a system is typically a sequence of words in one language and the output is a sequence of words in another language.

It is common that sequence elements are mutually dependent and that the sequence length is variable. Often the alignment of input and output symbols is not one-to-one.

Fully-conneceted and convolutional neural networks are not well suited for these requirements. They expect their inputs to have a fixed dimensionality. We could still process sequences one element at a time, but this approach would completely neglect the relationships between individual elements and would impose a one-to-one input-output correspondence.

Recurrent neural networks (henceforth RNNs) are a type of neural networks designed for processing sequences. Given a sequence of input elements $\{x^{(1)}, \ldots, x^{(n)}\}$, the general RNN formulation is as follows:

$$h^{(t)} = \phi(h^{(t-1)}, x^{(t)}).$$

The variable $h^{(t)}$ is the hidden state of the recurrent network at timestep t. It is a function of the current input symbol and the previous hidden state. In this way, it is dependent on all the input symbols up to $x^{(t)}$. The network's objective encourages the network to store in $h^{(t)}$ a summary of the previously encountered symbols which is relevant to the task at hand.

This formulation allows processing of variable length sequences. Moreover, the same function is applied at each timestep which makes it possible for the network to generalize to sequence lengths not encountered durning training.

Mapping of a sequence to another sequence whose length can differ is made possible due to the *encoder-decoder* framework, which is presented in a later section.

Context Conditioned RNNs. The basic RNN formulation given above can be extended to become conditioned on the context. By adding the context c as an additional input to the equation, it is possible for the model to learn to generate different outputs given identical inputs in differing contexts. The RNN equation now looks as follows:

$$h^{(t)} = \phi(h^{(t-1)}, x^{(t)}, c).$$

This is key for many applications, including those which are the subject of this work, as we will see later. In an image captioning system, the image for which a caption should be generated can be provided to a RNN language model in the form of the context. Additionally, the system can be extended by an attention

component, which will provide the language model with a fine-tuned representation of the input image with respect to the currently generated word.

2.4.1 Types of Recurrent Neural Networks

Vanilla Recurrent Neural Network. In the Vanilla RNN(Elman, 1990) the new hidden state is computed by adding linear transformations of the input and the previous hidden state and a bias and applying an elementwise nonlinearity. Formally, the hidden state at time t is computed as:

$$\mathbf{h}^{(t)} = \tanh(W_h\mathbf{h}^{(t-1)} + W_x\mathbf{x}^{(t)} + \mathbf{b}).$$

Assuming the input sequence elements belong to $\mathbb{R}^n$ and the hidden states to $\mathbb{R}^m$, $W_x \in \mathbb{R}^{m\times n}$ and $W_h \in \mathbb{R}^{m\times m}$ are the learnable matrices defining the linear transformations of the input and hidden state respectively and $\mathbf{b} \in \mathbb{R}^m$ is the bias term. The *tanh* function is the hyperbolic tangent which is defined as:

$$\tanh(\mathbf{x}) = \frac{e^{\mathbf{x}} - e^{-\mathbf{x}}}{e^{\mathbf{x}} + e^{-\mathbf{x}}}.$$

Exploding and vanishing gradients. When a recurrent neural network is executed by the computer the recurrence is unrolled. The depth of the resulting network thus depends on the length of the sequence being processed, which is often much larger than the depth of nontrivial convolutional of feedforward architectures.

The fact that RNNs work by repeatedly applying the same transformations has a negative effect on the flow of information durning training. The gradients tend to diminish over subsequent layers, known as *the vanishing gradient problem*. Sometimes the opposite happens, a phenomenon referred to as *exploding gradients*.

The diminishment of gradients is caused by the hyperbolic tangent function. The activation function has its range in the $[-1, 1]$ interval. In longer sequences, as a consquence of the chain rule, repeated multiplication by numbers with magnitude less than one drive the gradient towards zero.

The way to combat exploding gradients is by giving a threshold on the magnitude of the gradient's norm beyond which its value is clipped - a method known as *gradient clipping*.

To tackle vanishing gradients, sophisticated improvements to the Vanilla RNN were developed, which are now commonly being used. These architectures are referred to as Gated Recurrent Networks and include most notably the Long Short-Term Memory and the Gated Recurrent Unit.

Long Short-Term Memory. The Long Short-Term Memory (Hochreiter and Schmidhuber, 1997) or LSTM for short is a recurrent network architecture desgined to allow gradients to flow back durning backpropagation in a stable manner for even extremely long sequences. We will present the most common LSTM for-

mulation and then break it down, looking at the role of individual terms.

$$\begin{aligned}
\mathbf{h}^{(t)} &= tanh(\mathbf{c}^{(t)}) \odot \mathbf{o}^{(t)}, && \textit{block output} \\
\mathbf{c}^{(t)} &= \mathbf{z}^{(t)} \odot \mathbf{i}^{(t)} + \mathbf{c}^{(t-1)} \odot \mathbf{f}^{(t)}, && \textit{cell} \\
\mathbf{o}^{(t)} &= \sigma(W_o\mathbf{x}^{(t)} + V_o\mathbf{h}^{(t)} + \mathbf{b}_o), && \textit{output gate} \\
\mathbf{f}^{(t)} &= \sigma(W_f\mathbf{x}^{(t)} + V_f\mathbf{h}^{(t)} + \mathbf{b}_f), && \textit{forget gate} \\
\mathbf{i}^{(t)} &= \sigma(W_i\mathbf{x}^{(t)} + V_i\mathbf{h}^{(t)} + \mathbf{b}_i), && \textit{input gate} \\
\mathbf{z}^{(t)} &= tanh(W_z\mathbf{x}^{(t)} + V_z\mathbf{h}^{(t-1)} + \mathbf{b}_z). && \textit{block input}
\end{aligned}$$

As before, $\mathbf{h}^{(t)}$ and $\mathbf{x}^{(t)}$ is the hidden state and input, respectively, at timestep t. The main idea in LSTM networks is to include an inner recurrence in the form of a memory cell $\mathbf{c}^{(t)}$, which persists trought time. The memory cell is coupled with a system of gates which allow to control the measure by which old information is replaced by new information. The gating mechanism consists of a forget gate $\mathbf{f}^{(t)}$, an input gate $\mathbf{i}^{(t)}$ and an output gate $\mathbf{o}^{(t)}$.

In addition to allowing important features to preserve in the memory, the additive formulation of the transition from $t-1$ to t in the memory cell creates a "shortcut path" for the gradient to travel along durning backpropagation (Chung et al., 2014).

Gated Reccurent Unit. The Gated Recurrent Unit (GRU; Cho et al., 2014a) is based on the LSTM cell but aims to simplify it, at the same time trying to preserve the properties which make it effective. It goes about without the memory cell, instead computing the value of the hidden state as a linear interpolation of the previous hidden state and a candidate activation. The coefficient of the interpolation is given by a learnable gate, called the update gate. Formally,

$$\begin{aligned}
\mathbf{h}^{(t)} &= \mathbf{z}^{(t)} \odot \mathbf{h}^{(t-1)} + (1 - \mathbf{z}^{(t)}) \odot \tilde{\mathbf{h}}^{(t)}, && \textit{block output} \\
\mathbf{z}^{(t)} &= \sigma(W_z\mathbf{x}^{(t)} + V_z\mathbf{h}^{(t-1)} + \mathbf{b}_z), && \textit{update gate} \\
\tilde{\mathbf{h}}^{(t)} &= tanh(W_h\mathbf{x}^{(t)} + V_h(\mathbf{r}^{(t)} \odot \mathbf{h}^{(t)}) + \mathbf{b}_h), && \textit{candidate activation} \\
\mathbf{r}^{(t)} &= \sigma(W_r\mathbf{x}^{(t)} + V_r\mathbf{h}^{(t-1)} + \mathbf{b}_r). && \textit{reset gate}
\end{aligned}$$

The candidate activation is computed reminiscently of the simple RNN with the improvement of an added reset gate, which controls the forgetting of the previous hidden state values. The GRU block preserves the additive interaction present in LSTMs, which substantially mitigates the vanishing gradient problem. It also contains the gating mechanism allowing to selectively keep or forget information. In contrast to LSTMs, the amount of new information passed trough to the hidden state and the amount of information to forget are controled solely by the update gate. In LSTMs, separate gates (the forget gate and the input gate) control each quantity.

The architecture of Gated Recurrent Networks can seem ad-hoc, especially in case of LSTMs. A study by Jozefowicz (Jozefowicz et al., 2015) which tested over 10 000 Gated RNN variants by evolutionary architecture search however concluded that there does not appear to be much room for significant improvements by adjustments to the current architectures.

RNN Language Models. A language model captures a probability distribution over sentences. Typically, the probability of a sentence $\mathbf{w} = \{w_1, w_2, \ldots, w_T\}$ is computed using the chain rule of probability as:

$$p(\mathbf{w}) = \prod_{t=1}^{T} p(w_t | w_1, \ldots, w_{t-1}).$$

These conditional probabilities can be modeled using a recurrent neural network. One way how this could be done is by linearly transforming the hidden state $\mathbf{h}^{(t-1)}$ into a vector of word scores $\mathbf{s}^{(t)}$ over the vocabulary and further normalizing this vector into a valid probability distribution with a softmax function. As the hidden state is a function of $w_1, \ldots, w_{t-1}$, this corresponds to estimating the probability of a word occuring at the next position of the sentence given the previous words. Formally,

$$\begin{aligned} \mathbf{s}^{(t)} &= W\mathbf{h}^{(t-1)}, \\ p(w_t = w | w_1, \ldots, w_{t-1}) &= \text{softmax}(\mathbf{s}_w^{(t)}). \end{aligned}$$

This approach can be used to sample sentences from the model. When additionally conditioned by an input sentence in a different language, this scheme can be used for machine translation. Recurrent neural network LMs and neural network LMs in general have a broad applicability in NLP and are showing much more promise (Radford et al., 2018).

When at each decoding step the word with the highest probability is chosen as the output token for the given timestep, the decoding is referred to as *greedy*. Greedy decoding is a heuristic, the generated sentence does not necessarily have the maximum probabilty of all possible sentences. An improved approach to decoding is called *beam search decoding*.

2.4.2 Beam Search Decoding

We saw how the sequence of previously generated words influence the word being currently generated by the language model. By considering more of these sequences, we could find a hypothesis with a higher overall score, than the one discovered by greedy decoding. The idea of keeping multiple partial hypotheses during decoding is the foundation of beam search decoding.

In beam search decoding, instead of chosing the token with the highest conditional probability, given previously generated tokens, we choose b of them. The parameter b is called the *beam size*. The current hypothesis is extended into b hypotheses. In the next step, the decoder is fixed and run b times, once for each hypothesis. For each of the b decoder states, a number of most likely tokens (typically the whole vocabulary, or a multiple of b) are considered for extension. The b most likely hypotheses are chosen.

This way, the decoding process considers more hypotheses, with the final one as least as good as the output of greedy decoding.

2.5 Sequence to Sequence Learning

2.5.1 Encoder-Decoder Framework

The Encoder-Decoder framework (Cho et al., 2014b) is a high-level neural network architecture, which consits of an encoder and a decoder. The encoder maps the input into a new representation and the decoder then generates an output based on this representation. The overall system is trainable end to end.

The framework was first introduced in the context of machine translation (Cho et al., 2014b; Sutskever et al., 2014). Its importance relied on its ability to map sequences to sequences of potentially different lengths. The encoder would map an input sentence to a fixed-length vector – the context vector c. A decoder would then use c to produce the translated sentence. As the context vector c they used the last hidden state of a RNN. For the decoder, they used a RNN language model conditioned on c which would sequentially generate the translation.

The problem of mapping sequences to sequences encompasses other problems besides machine translation, such as speech or handwriting recognition. Additionally, there are other problems, which naturally lend themselves to the encoder-decoder conceptualization, such as image captioning. Therefore the encoder-decoder framework has broad applicability.

2.5.2 Attention in Neural Networks

In vanilla sequence-to-sequence learning, the encoder produces a fixed-sized vector representation of the input that the decoder uses to generate the output. This has the disadvantage that all of the necessary information from the input needs to be encoded to this limited representation.

The attention mechanism solves this problem by providing a way to dynamically compute the context vector for each step of the decoding.

The initial version of the attention mechanism was introduced by Bahdanau et al. (2014) in the context of machine translation. The input to the attention mechanism is a sequence of annotation vectors $\{\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_n\}$ [2] produced by the encoder. Next, at step i during decoding, the alignment model a computes an energy score e_{ij} for each annotation vector. Where $\mathbf{s}_{i-1}$ is the state of the decoder at step $i-1$.

$$e_{ij} = a(\mathbf{s}_{i-1}, \mathbf{h}_j).$$

The energy score e_{ij} measures the degree of relevance of the j-th annotation vector at the i-th decoding step. The energy scores for all annotation vectors are subsequently normalized by softmax, producing a weight vector α_i. Finally, the context vector c_i for the decoding step i is computed as an weighted average of the annotation vectors.

[2] In Bahdanau's implementation, the encoder consisted of a bidirectional recurrent neural network. A bidirectional recurrent neural network reads the input sequence in both directions. The annotation vectors were produced by concatenating the forward and backward passes' hidden states.

$$\alpha_{ij} = \text{softmax}(e_{ij}), j \in \{1, \ldots, n\},$$
$$\mathbf{c}_i = \sum_{j=1}^{n} \alpha_{ij} \mathbf{h}_j.$$

The alignment model a in the original implementation is defined as a single-layer feedforward neural network. The decoder is a recurrent neural network conditioned on the context vector, the previously generated word and the previous hidden state.

A General Attention Function Formulation. The above given attention model can be formulated in more general terms. By giving a more high-level definition of attention, it is easier to see how other attention models build upon the same idea, and how the framework can be utilised to design new attention functions.

The attention function can be seen as operating a lookup function on a set of key-value pairs given a query (Vaswani et al., 2017). The query, keys and values are all vectors. A compatibilty function computes a weight that measures the relevancy of each key with respect to the query. The output of the attention function is then computed as the weighted sum of the values.

$$e_i = a(\mathbf{q}, \mathbf{v}_i),$$
$$\alpha_i = \text{softmax}(e_i), i \in \{1, \ldots, l\},$$
$$\mathbf{c} = \sum_{i=1}^{l} \alpha_i \mathbf{v}_i.$$

In the above equations, $\mathbf{q}$ is the query, $\mathbf{k}_i$ is the i-th key and $\mathbf{v}_i$ is the i-th value. The total number of key-value pairs is given by l. Finally, a is the compatibility function and $\mathbf{c}$ is the attention function output.

Many different attention functions can be derived from this formulation. Specifically, they can differ in the choice of the compatibility function and the assignment of query, keys and values inside the model.

Additive Attention. The attention function used by Bahdanau et al. (2014) is referred to as *additive attention*. In additive attention, the energy score of input annotation j at decoding timestep i is computed as:

$$e_{ij} = \mathbf{u}^\top tanh(W\mathbf{q}_i + U\mathbf{k}_j).$$

The additive attention effectively implements a fully-connected feedforward network with one hidden layer and single neuron as output. The input is the concatenated query vector $\mathbf{q}_i$ and key vector $\mathbf{k}_j$.

Dot Product Attention. Luong et al. (2015) proposes a less computationally expensive attention function which computes the energy score as the dot product between the query and key vectors:

$$e_{ij} = \mathbf{q}_i^\top \mathbf{k}_i.$$

The dot product attention requires the keys and query to be of the same dimensionality.

Scaled Dot Product. Vaswani et al. (2017) note that as the dimension of the keys grows larger, additive attention outperforms dot product attention, due to the growing magnitude of the dot product which subseqeuntly pushes the softmax function into regions where it has small gradients. They propose to add a scaling factor to the dot product:

$$e_{ij} = \frac{\mathbf{q}_i^\top \mathbf{k}_i}{\sqrt{d}}.$$

Where d is the dimension of the key vectors.

General Attention. The *general attention*, proposed by Luong et al. (2015) extends the dot product attention by linearly projecting the keys. This allows the attention module to learn a new representation of the key space, and offers a solution when the dimensionality of the keys and query is not the same, rendering the dot product unusable.

$$e_{ij} = \mathbf{q}_i^\top W \mathbf{k}_i.$$

3. Experiments

3.1 Overview

This chapter presents the experiments and an analysis of the results. First, we introduce the tasks of image captioning and multimodal translation, along with a brief genealogy of the approaches used to tackle these problems, from past to present. Subsequently, the experimental methodology is laid out, after which comes a presentation of the datasets used in experiments. Next, we touch on the issue of evaluating model performance on these tasks. At this point, all pieces are laid out and we focus on our experiments. We describe our aim – their purpose, the procedure and analyse results. We propose a new attention function formulation and empirically evaluate it. During analysis, we utilise Macaque to examine the results, demonstrating how it can be used in practice.

3.2 Image Captioning

Image captioning is the task of providing natural language sentences describing the contents of a given image. It is a difficult task that encompasses the ability to recognize objects present in a scene and expressing this information along with the relationships between these objects in natural language. As such it is a problem that falls under both computer vision and natural language processing and utilises knowledge from both of these fields.

Past and present approaches. Prior to the encoder-decoder framework, approaches to image captioning included retreival of captions from a database based on image similarity and secondly of using pre-generated caption templates.

In the caption retrieval case, Farhadi et al. (2010) describe a method for computing image sentence similarity, based on which it is possible to associate the most suiting caption from a set of captions to an image, and vice versa. In the second case, pre-generated caption templates are filled based on object recognition and attribute discovery procedures deployed on the image (Kulkarni et al., 2011).

The quality of image captioning systems improved substantially by introducing neural networks together with the encoder-decoder framework. Several works, notably Xu et al. (2015) and Karpathy and Fei-Fei (2014) use convolutional networks to obtain a vectorial representation of the image, which is then fed into a recurrent neural network language model to generate the output sentence. Recently, the transformer architecture (Vaswani et al., 2017), which abandons both convolutional and recurrent networks in favor of attention modules was shown to achieve superior results (Sharma et al., 2018).

3.3 Multimodal Translation

In the multimodal translation task the input image is accompanied by one description of that image in a source language and the desired output is a description

of the image in a target language, the target language differing from the source language.

At training, the target description is a direct translation of the source description. At test time, the evaluation is done against a single reference caption.

The task can be seen as extending the image captioning scenario by additional input information, and at the same time, as supplementing machine translation by an additional input modality.

Past and present approaches. The presence of multimodal translation on the computer science problem spectrum is relatively recent due to the previous unavailability of datasets for the task. Since the task is close to both machine translation and image captioning, approaches to multimodal translation utilise and adapt techniques that have proven themselves in the former scenarios. It has been shown to be difficult to take advantage of the additional data in a way that improves model performance (Helcl and Libovický, 2017). However, the task is subject to active research, Helcl et al. (2018) demonstrated a way of extending the Transformer architecture that yielded significant improvements at evaluation.

3.4 Supervised Training

This section provides a description of the theoretical framework along which we are going to perform our experiments.

Supervised Learning. We tackle both image captioning and multimodal translation in the *supervised learning* setting. At training time, we run our models on datasets that consist of example instances and *labels*. The label is the desired output for the given instance that we want to mimic with our model. The Flickr30k, Multi30k and MSCOCO datasets are examples of such labeled datasets.

Training. The degree of agreement between the label and the model's actual output on the instance is measured by the *loss function*. The loss function value summed over the entire dataset is minimized during training by an *optimization algorithm* such as stochastic gradient descent or more recently, Adam (Kingma and Ba, 2014). This way the model learns to approach desired behaviour on exemplar cases. However, the aim is generalization, not optimization on the training set. In other words, we want the model to learn to solve the task, equally well for unseen data, not present in the training set.

Validation. There are many methods that attempt to minimize the generalization error, such as regularization techniques. Specifically connected to the experimental methodology is *validation*. A smaller portion of the dataset is usually set aside, referred to as the *validation set*. This data is used to measure the models generalization error and guide the training. After some fixed interval, such as after every epoch of training, the model is run on the validation set. At some point the validation error (the model error on the validation set) ceases to reduce, while the training error continues to decrease. This phenomena is referred

to as *overfitting* – the model begins to shift from "understanding" of the task to "memoization" of the training set. Training is generally over, when the validation error and training error begin to diverge. The model with the lowest validation error is chosen.

Testing. In the experimental setting, another portion of the dataset is saved, referred to as the *test set*. After training is done, this set, mutually exclusive with both the training and validation sets is used to evaluate the performance of the model at solving the task, in the closest possible scenario to real usage. The results of evaluation on this set are reported in papers and on models are compared on their basis.

3.5 Datasets

The performance of a model is ultimately dependent on the data it is trained on. For example, a machine translation of a sentence will not contain words not present in the training vocabulary. Even though, the model can learn to copy unseen words from the source to the target, it cannot learn a notion that cannot be inferred from the dataset. Therefore, it is important for the success of our applications, that our training datasets are representative of the task we are trying to solve. It follows, that to estimate the limits of the accuracy of a model, it is essential to have a thorough understanding of the properties of the data being used.

In this section we discuss the benchmark datasets used in the tasks of image captioning and multimodal translation.

3.5.1 Flickr30k

The Flickr30k dataset (Young et al., 2014) consist of $31,783$ manually selected images from the Flickr social network, each paired with 5 reference captions. It was designed for image description, with the intent to address the shortcomings of previous attempts on different datasets. Specifically, the reference captions of existing datasets at the time were in many cases not descriptive of the contents of the images. In other cases, they were overly detailed of non-salient objects, or otherwise unfavorable. This is a problem, because the reference captions shape the learning objective of the model, which is to describe images based on information obtainable from them. Therefore, the captions were produced specially for this task, via crowdsourcing, with precise guidelines concerning their form and content.

3.5.2 MSCOCO

The MS COCO (Microsoft: Common Objects in Context; Lin et al., 2014) was designed as a computer vision multi-task dataset. It comprises of $123,000$ labeled images, with another over 120k unlabeled. In addition to 5+ reference annotations describing the contents of each image, instances of objects from 91 categories which are present in the image are labeled and segmented on the level of individual pixels. The object categories were chosen to be representative of

the set of all categories and to be practically relevant. The images were selected to contain instances of these objects in both iconic and non-iconic (whether the object is dominant in the scene) views in scenes, picturing them in relationship with other objects rather than in isolation.

The MS COCO dataset does not have a standardized split into training, validation and test sets. The splits of Karpathy, which we use in our experiments, have become the de facto norm.

3.5.3 Multi30k

The Multi30k dataset (Elliott et al., 2016) is an extension of the Flickr30k dataset for the task of multimodal translation. In this setting the input image is accompanied by a caption in a source language. Therefore, Multi30k consists of the exact same image data as Flickr30k but the English captions are paired with German ones. Translations of 30 thousand captions were provided by professional translators. Additional 150 thousand German descriptions were provided by crowdsourcing.

3.6 Evaluation Metrics

During training, the objective is to maximize the likelihood of reference captions. This is intuitive, given the supervised learning framework. However, difficulties arise when evaluating the outputs of the model on unseen data. A very reliable way of evaluating captions produced by the model is human judgment. Obvious disadvantages of this approach are cost and time needed to provide results. For this reason, research effort has been dedicated at the development of automatic evaluation metrics.

Automatic evaluation metrics measure the closeness of the model output to a set of reference outputs according to a numerical metric. Automatic evaluation metrics have their own disadvantages and limitations, but despite this, they are the standardized way of evaluating and comparing model performance in image captioning and multimodal translation.

3.6.1 BLEU

BLEU (bilingual evaluation understudy; Papineni et al., 2002) is based on the idea that correct translations share common words and phrases (sequences of words). This notion is captured in matching n-grams[1].

The BLEU score is calculated using *modified n-gram precision*. Given a candidate hypothesis c, a set of references and n, let $\{a_1, \ldots, a_k\}$ be the set of n-grams collected from the candidate hypothesis. Let C_i be the number of occurences of a_i in the candidate hypothesis for $i \in \{1, \ldots, k\}$ and M_i the maximum number of occurences of a_i in any of the references. Then the BLEU score for the candidate hypothesis and n is computed in the following way:

[1] An n-gram is a contiguous sequence of n words. In other contexts, sequences of characters, or any tokens could be considered instead.

$$bleu_n(c) = \frac{\sum_{i=1}^{k} min(C_i, M_i)}{k}.$$

BLEU is commonly extended by a *brevity penalty* factor that keeps the hypotheses lengths in check. The original paper proposes an advanced method of computing the score on a corpus-level, using a geometric mean over the sentence-level scores for different values of n.

3.7 Exeprimental Setting

Our experiments are aimed at exploring the effect of different model architectures based on the encoder-decoder framework and different attention function formulations at the tasks of image captioning and multimodal machine translation. Specifically, we focus on the ability of the attention mechanism to exploit the information in the input image and to focus on relevant parts of the image as the output sentence is generated. We judge the permormance of the attention mechanism quantitatively by evaluating the trained models on test sets, and qualitatively, by visualizing how the attention mechanism learns to focus on the image. For qualitative analysis, we employ Macaque. We start with a baseline model architecture, which we subsequently tweak. We implement our experiments in the Pytorch[2] machine learning framework.

3.7.1 Model Details

Baseline Model. Our baseline model (Model 0) for image captioning is based on the soft-alignment architecture proposed in Xu et al. (2015). The model consists of a pretrained convolutional encoder and a recurrent decoder, additionally conditioned by the output of the attention mechanism.

The image encoder is a VGG19 network pretrained on the ImageNet dataset (Deng et al., 2009). We use Keras' implementation of the network and the checkpoint it provides, which we do not adapt during training. We use features extracted from the last convolutional layer (the layer name is "conv5_block4"). The shape of these features is [14, 14, 512] per instance.

The baseline decoder is a signle-layer GRU with hidden state size 1024. The input to the decoder at each time step is the embedded previously generated output word, concatenated with the context vector from the attention mechanism. The output of the RNN is linearly projected and processed by softmax, resulting into next output word logits. We set the maximum input and output caption length to 20 tokens.

The attention component is an additive attention; the decoder's hidden state from the previous timestep is fed as the query, and the encoder outputs are passed as both keys and values (this is the way proposed in Xu et al., 2015). The size of the hidden layer in the attention component is 1024.

The size of the embedding is 512.

In the original paper, Xu et al. (2015) extend the architecture by two feed-forward networks that compute the initial hidden state and the initial context

[2] `https://pytorch.org/`

vector, and a deep output layer (Pascanu et al., 2013). We don't include these components, for they increase the complexity of the model and should have similar performance boosting effects on all of our architectures.

Model 1. The architecture of Model 1 is the same as the baseline model, with the exception that the input to the decoder is not a vector concatenation of the previously generated embedded word and the context vector. Instead, the concatenation vector is linearly projected and processed by a ReLu function. This adds a new layer of representation, combining the previously generated word and the current context vector. The method was used in Luong et al. (2015) in the context of neural machine translation.

Model 2. The architecture of Model 2 is similar to the baseline and Model 1, in the parts it uses, but we alter the computational path. Let y_{t-1} be the embedded output word generated at timestep $t-1$. The decoder RNN receives only y_{t-1} and its previous hidden state h_{t-1} as input and produces a new state h_t. The context vector c_t is computed by the attention component which receives h_t as query, and encoder outputs as keys and values. h_t and c_t are subsequently concatenated, linearly projected and passed trough the ReLu function. This combined representation, $\tilde{h}_t$ is then used to generate the next output word probability distribution by linearly projecting into a vector of the size of the output vocabulary and computing softmax over it. This sequence of steps was proposed by Luong et al. (2015) in the context of neural machine translation.

Model 3. Model 3 is the same as Model 2 with the single difference, that at each timestep, the input to the decoder is not only y_{t-1} but $(y_{t-1}, \tilde{h}_{t-1})$. This provides the decoder information about where the model focused its attention in the previous steps. Luong et al. (2015) conjecture that the decoder can learn to use this information to form constraints about how to pay attention, for example to focus on all parts of the input during the generating process.

Multimodal Machine Translation Models. For our multimodal machine translation experiments we use the same four architectures with the extension of a source caption encoder and a second attention module. The caption encoder is implemented as a single-layer bidirectional GRU with hidden state of size 512. The source caption embedding size is 512. The attention module operates on the caption encoder's outputs. In experiments using additive attention, we set the hidden layer size of both attention modules to 512 neurons.

Attention Functions. For each of the above architectures, we experiment with different attention functions. The attention functions we use are additive attention (2.5.2), general attention (2.5.2) and a combination of general and scaled dot product attention (2.5.2), which we call *scaled general attention*.

Scaled General Attention. Dot product attention tends to increase fast with the growing dimension of the keys and the query. The large magnitude of the dot product then pushes the softmax function into regions where the gradient

vanishes and training hinders (Vaswani et al., 2017). Scaling the dot product by a factor of the keys dimension helps mitigate this problem. The same issue can be observed with general attention – the linear projection projects the keys into the dimension of the query and then their dot product is computed. Therefore, in scaled general attention, we scale the dot product after the linear transformation by the square root of the query dimension d:

$$e_{ij} = \frac{\mathbf{q}_i^\top W \mathbf{k}_j}{\sqrt{d}}.$$

In terms of computational efficiency, scaled general attention has the advantage over additive attention that it does not involve the computation of a non-linear function.

3.7.2 Training Configuration

Objective. The training objective consists of minimizing across the whole dataset the negative log likelihood of the generated caption given the input image, or the input image and the source caption in the image captioning case, and in the multimodal translation case, respectively.

$$\arg\min_\theta \sum_{i=1}^{N} -\log p(y_i|x_i;\theta) \qquad \textit{image captioning objective}$$

$$\arg\min_\theta \sum_{i=1}^{N} -\log p(y_i|x_i,v_i;\theta). \qquad \textit{multimodal translation objective}$$

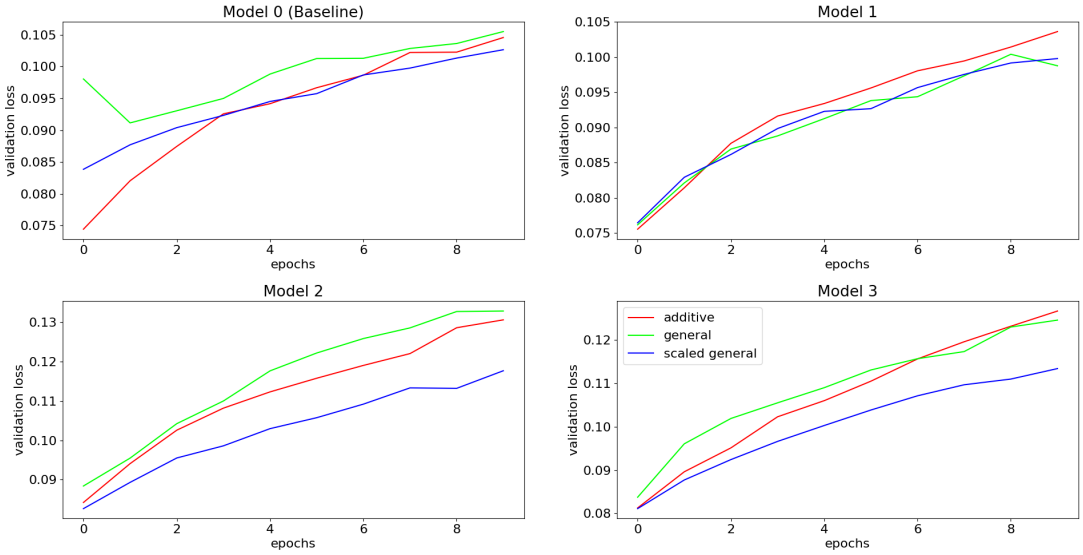
N is the number of training instances, θ is the set of the model's trainable parameters, y_i, x_i and v_i are the i-th model generated output caption, input image and source caption, respectively. $p(y_i|x_i;\theta)$ and $p(y_i|x_i,v_i;\theta)$ are the models' estimates of the probability of y_i given x_i, v_i and θ in the unknown underlying data generating distribution.

Hyperparameters. The training procedure consists of running our Flickr30k and Multi30k models for 20 epochs and MSCOCO models for 10 epochs on the training datasets. We perform validation after each epoch; during validation we compute the loss on the validation dataset and evaluate model outputs using BLEU. We report results on the test set using a checkpoint of the model's parameters that produced the highest BLEU score during the training. We set the batch size to 64. We use the Adam optimization algorithm (Kingma and Ba, 2014) with initial learning rate $\alpha = 0.0001$ and coefficients $\beta_1 = 0.9$ and $\beta_2 = 0.999$. We set the maximum magnitude of the gradient's norm to 1 and clip gradients that exceed it to prevent exploding gradients. We use a teacher forcing ratio of 1.0.

All of our experiments are performed on GPUs trough MetaCentrum[3]. Due to much longer waiting time in job queues for specific clusters, we do not fix the machine architecture troughout our experiments.

[3] `https://metavo.metacentrum.cz/`

Figure 3.1: Validation losses on MSCOCO.



3.7.3 Results

Quantitative Analysis. Our experiments did not show that any of our proposed architectures would outperform the baseline. However, scaled general attention outperformed general attention in BLEU score on test sets in 6 of 8 cases, and performed comparably well to additive attention.

We observed, that in most of our experiments validation loss began to increase after only a few epochs of training. This indicates that the model started overfitting early. Due to time constraints, we could not experiment more with model hyperparameters, which could lead to substantially remedying this issue.

In contrast to diverging validation loss, BLEU score on the validation set continued to improve, as training proceeded, in most experiments. This lack of correlation between BLEU and validation loss is not uncommon (Xu et al., 2015).

For BLEU evaluation we left out the brevity penalty, as is commonly done in image captioning. We report scores obtained from using the TensorFlow's Python implementation [4]. This implementation is not comparable with scores found in literature[5].

Qualitative Analysis. In our experiments, we observed that the interpretability of additive attention surpassed scaled general attention. Our models with additive attention learned to attend to the most salient objects in the scene, while generating the corresponding word in the caption (see Fig. 3.5). Similarly, when generating a less tangible word, such as "to" the attention distribution was more uniformly spread across the image. In contrast, the attention distributions learned by scaled general attention were overally less interpretable and appeared

[4] `https://github.com/tensorflow/nmt/blob/master/nmt/scripts/bleu.py`

[5] However, using a Python implementation allowed us to easily evaluate BLEU durin training and we did not want the test set results to differ in the method by which they were obtained from the validation set results. Secondly, results published in literature are achieved with nontrivial finetunning, and it was not our aim to be competitive with the state-of-the-art.

Figure 3.2: Validation BLEU-4 scores on MSCOCO.

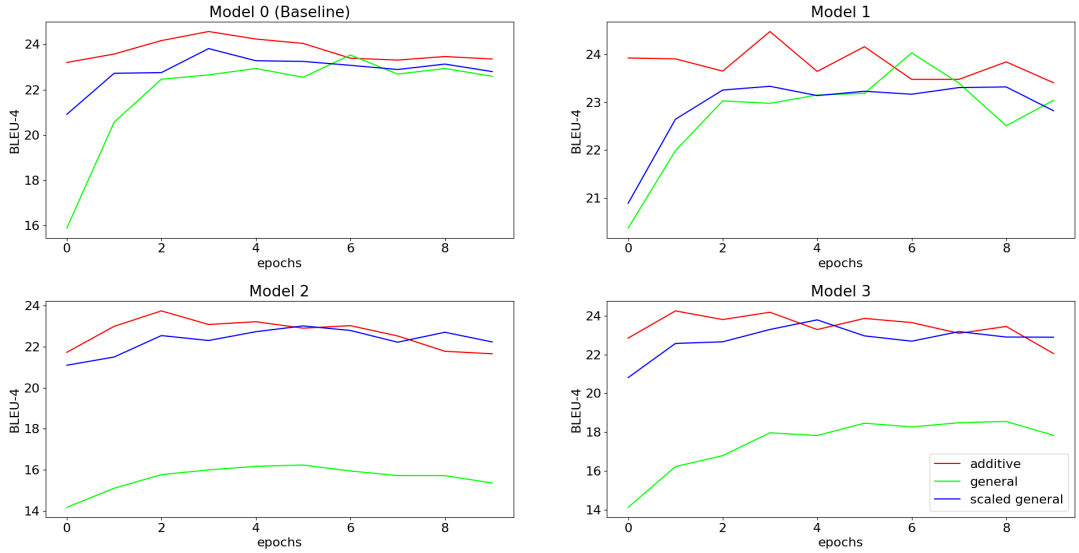


Figure 3.3: Validation losses on Multi30k.

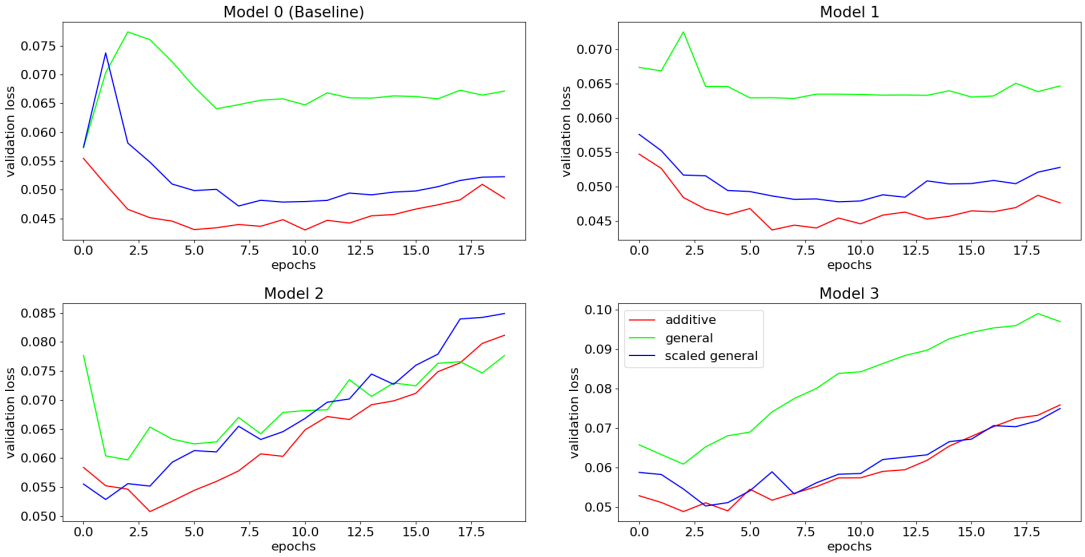


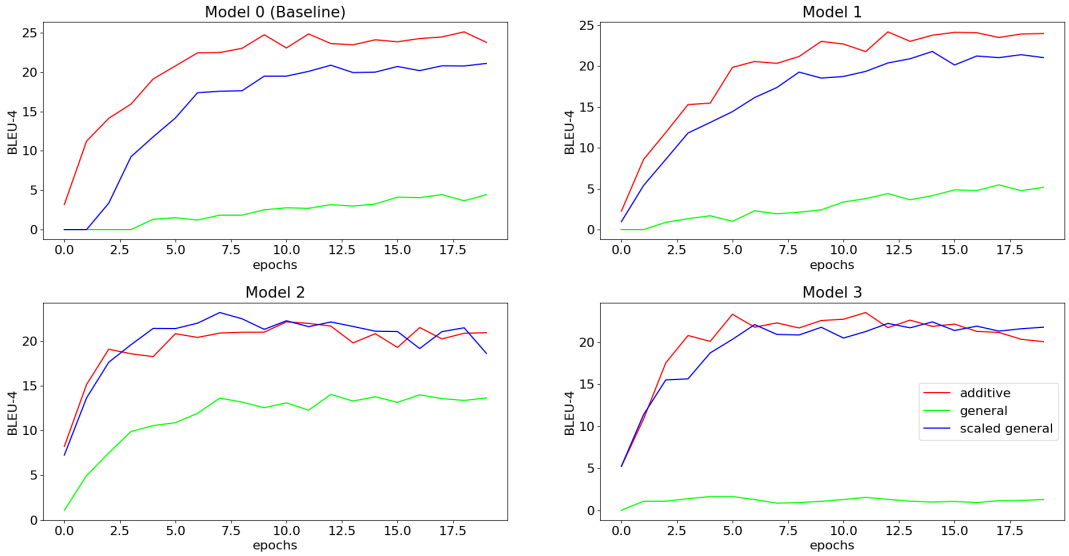
Table 3.1: Test scores on MSCOCO at image captioning.

Model	Attention	BLEU-1	BLEU-2	BLEU-3	BLEU-4
model 0	additive	66.6	48.3	34.0	23.9
model 0	general	65.5	47.4	33.3	23.3
model 0	scaled general	65.4	47.1	33.1	23.3
model 1	additive	66.8	48.6	34.4	24.4
model 1	general	65.8	47.4	33.3	23.5
model 1	scaled general	65.7	47.5	33.3	23.4
model 2	additive	65.5	47.2	33.2	23.6
model 2	general	57.2	37.6	24.0	15.8
model 2	scaled general	65.2	47.2	33.2	23.5
model 3	additive	65.8	47.8	33.8	24.1
model 3	general	60.2	41.1	27.4	18.6
model 3	scaled general	65.7	47.5	33.6	23.9

Table 3.2: Test scores on Multi30k at mutlimodal machine translation.

Model	Attention	BLEU-1	BLEU-2	BLEU-3	BLEU-4
model 0	additive	50.1	36.8	27.4	20.8
model 0	general	26.8	13.9	7.2	3.8
model 0	scaled general	45.4	32.1	23.1	16.8
model 1	additive	49.7	36.8	27.7	21.1
model 1	general	25.9	14.4	8.1	4.6
model 1	scaled general	46.6	33.8	34.4	24.4
model 2	additive	44.8	31.9	23.1	16.9
model 2	general	41.4	27.6	18.5	12.6
model 2	scaled general	47.1	33.6	24.4	18.1
model 3	additive	46.3	33.3	24.5	18.1
model 3	general	15.3	5.6	2.0	0.9
model 3	scaled general	47.2	33.9	24.6	17.9

Figure 3.4: Validation BLEU-4 scores on Multi30k.



to be random in a number of cases.

We were able to improve on the interpretability of scaled general attention by using a loss penalty factor (Xu et al., 2015). The penalty added to the loss function constraints the model to pay equal attention to each part of the image.

Figure 3.5: The attention distribution over an image of the baseline model with additive attention. Notice how the girls in the picture are under focus when generating the word “woman”.

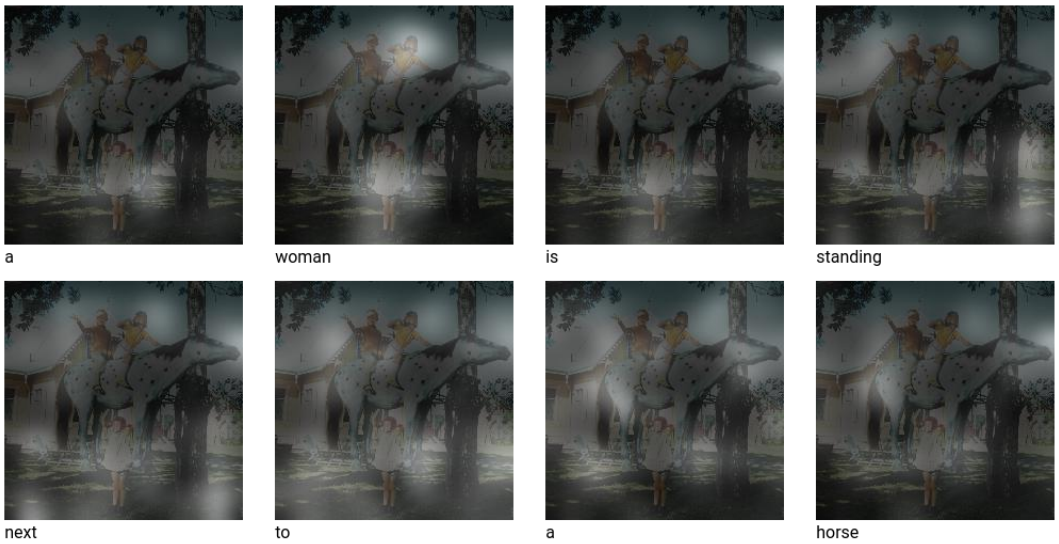


Figure 3.6: The attention distribution over an image of Model 2 with scaled general attention. The attention distribution is inferior to the one of additive attention in terms of interpretability.

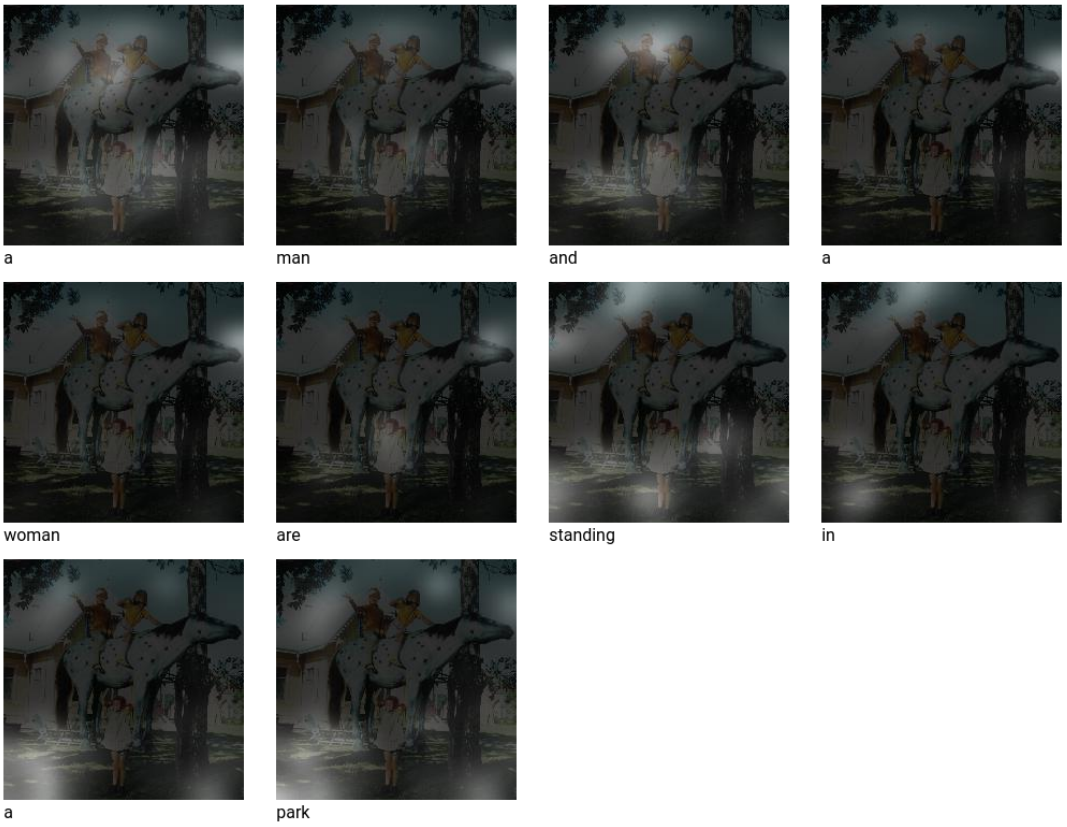


Figure 3.7: The attention distribution over an image of Model 0 with scaled general attention. The attention distribution does not obviously relate to the generated caption.

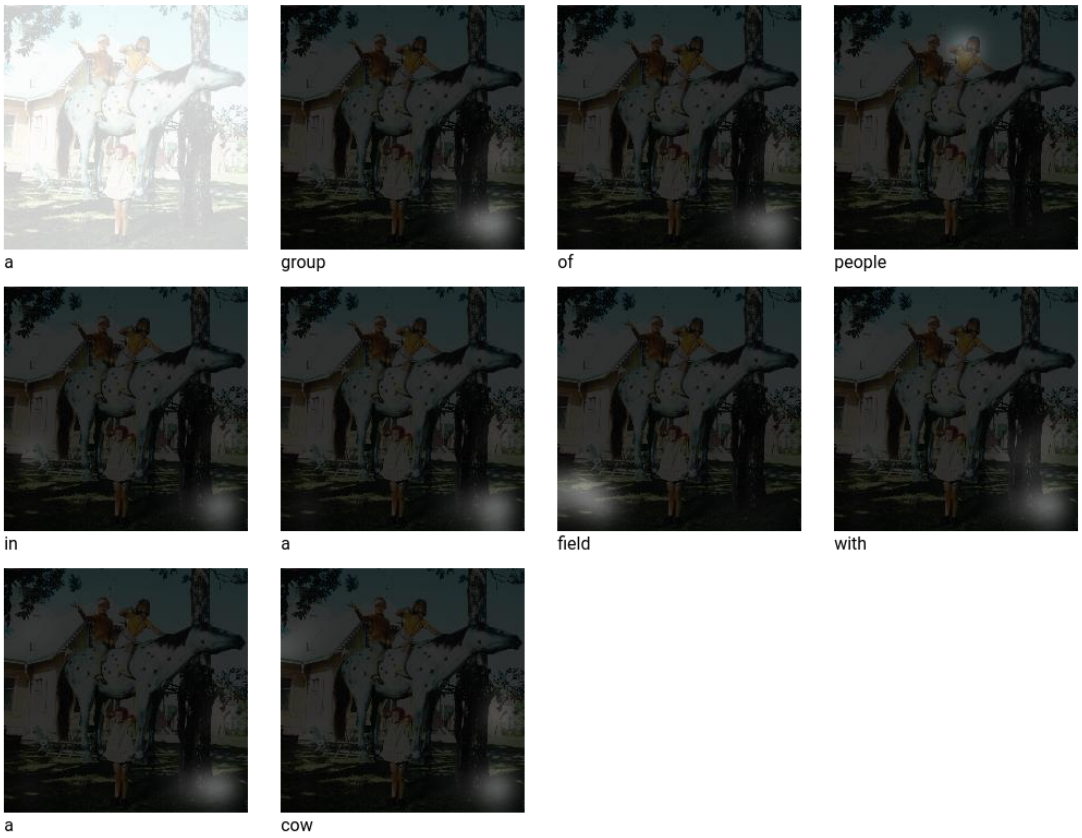


Figure 3.8: The attention distribution over an image of Model 2 with scaled general attention and a loss penalty pushing the model to pay equal attention to all parts of the image.

4. Macaque

This chapter presents Macaque, the main contribution of this thesis. Macaque is a software application that allows users to explore the outputs of deep learning models for image captioning and multimodal translation.

The application is oriented towards models written using the Neural Monkey framework, but any Python model can be used upon the user's implementation of a plugin that satisfies a simple interface (section 4.1.10).

Even though Macaque executes Python models, the presentation of results takes place in the web browser and is accomodated through JavaScript, allowing to take advantage of the combination of built-in browser support for complex interactive webpages created with HTML, CSS, JavaScript and the rich repertoire of packages accessible via node's npm. A complete list of Macaque's functionality can be found in section 4.1.3.

An instance of Macaque is running at `http://195.113.21.149:5000/`.

4.1 User's Guide

4.1.1 Installation

Requirements. Macaque has been tested on Ubuntu 18.04 LTS but any UNIX system should be compatible.

The following list summarizes the requierments on the underlying system.

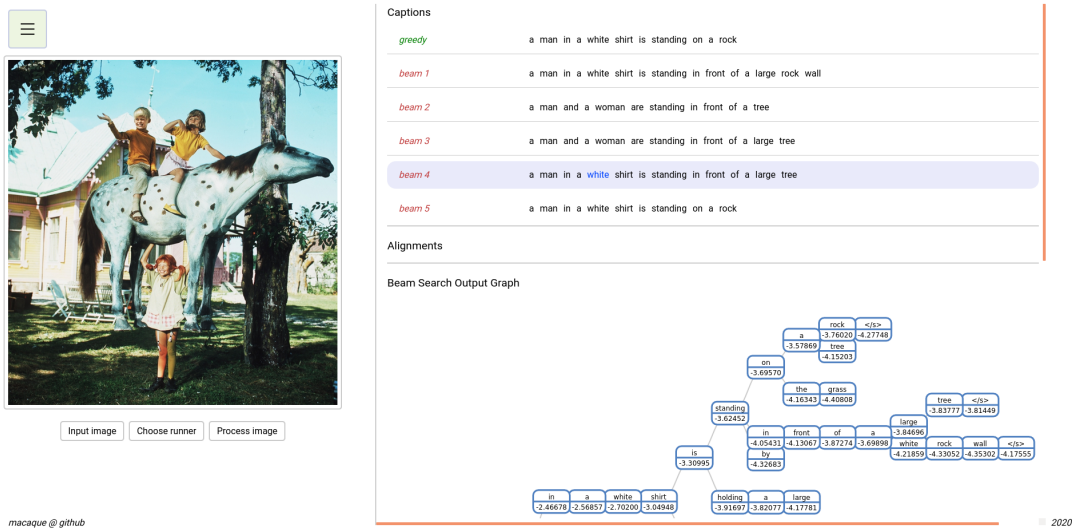
- A UNIX system or a UNIX subsystem accessible through a shell.
- Python 3.6
- pip3
- npm
- a web browser

Note, that any mentions of Python throughout this chapter will refer to Python3, unless stated otherwise.

Installation Instructions. The following instructions require an internet connection.

Before installing, you might want to consider creating a Python virtual environment. The process is straightforward and it is a common practice that makes it possible to manage different versions of a Python dependency between different projects. To create a virtual environment, open a shell, navigate to a directory, where you want to place the environment's file hierarchy and run `python3 -m venv venv-name`. This executes Python's `venv` module, resulting in creating the virtual environment `venv-name`. Activate the environment by entering `source venv-name/bin/activate`. To exit the environment, simply type `deactivate` into the shell.

Figure 4.1: A screenshot from Macaque's user interface.



To install Macaque, open a shell and navigate to the root directory of the project. Enter `./install.sh` inside the terminal. The installation script downloads all necessary Python and JavaScript packages and bundles the JavaScript source files into a single file using webpack – this step might take a few minutes.

4.1.2 Running

Run `./macaque` in the root directory of the project from inside your shell. After that, open a window in your prefered internet browser such as Mozilla Firefox and navigate to localhost at port 5000 at 127.0.0.1:5000.

Specifying Port. It is possible to specify the port on which the application runs by providing the `--port <port-number>` argument with a port number.

Running on Public IP Addresses. By default, Macaque listens on localhost. To launch Macaque on all public IP addresses of the system, add the `--public` flag.

Configuration Files Directory. Macaque supports its own configuration files which automate instantiating models and datasets. Configuration files are by default read from the `./models` directory located in the root directory of the project. It is possible to override this location by supplementing the `--config_dir <config_dir_path>` argument.

Figure 4.2: An example of launching Macaque.

```
# Installs Macaque.
./install.sh


# Launches Macaque with default settings - on localhost, port 5000.
./macaque.sh


# Launches Macaque on port 4321.
./macaque.sh --port 4321


# Launches Macaque on all public IP addresses of the system.
./macaue.sh --public


# Launches Macaque, specifying the location of its configuration files.
./macaque.sh --config_dir ./configs
```

4.1.3 Usage

Overview of Functionality

A list of Macaque's features

- Out-of-the-box inference with visualizations on images by a pretrained model.
- Support for custom image-based datasets.
- Running models on datasets and displaying results with visualizations.
- Dataset-level image preprocessing.
- Separate feature extraction by using a pretrained Keras or TensorFlow Slim encoder, or by a custom module.
- Support for Neural Monkey models.
- Support for custom models in Python upon implementing an interface.
- Visualizations of attention alignments.
- Visualizations of beam search decoding.

4.1.4 Home Tab

After launching Macaque and navigating to localhost's port 5000 in your web browser, you will be welcomed by the Home tab. The Home tab showcases most of Macaque's flashy features in a ready-to-use manner.

It allows users to choose images from their file system, process them by the models currently loaded and examine the results produced.

Click on the "Input image" button in the center of the page and pick a JPEG image. Upon selecting an image, it should be displayed in the center of the win-

dow. Additional buttons should appear below the image; "Choose runner" allows to choose the model which will process the image. *Runner* in Macaque refers to a specific inference-time configuration consisting from the preprocessor (optional), feature extractor (optional) and the model. Input images are preprocessed, then fed to the feature extractor, which encodes them into a new representation that the decoder-model is better at translating into the output. The "Process image" button on the right side triggers the inference procedure. After it finishes, the results are displayed.

The image should be on the left side of the window, the right side showcasing the results. The captions the model has produced are at the top right. Greedy captions are labeled "greedy" and captions produced by beam search are labeled "beam" accompanied by the index of the beam.

Hover over the captions with your mouse, if the model has an attention component, individual tokens will be highlighted when the mouse finds them, changing the cursor shape to a pointer. By clicking on a token you will display the attention alignment for the given token. The alignment is displayed on top of the original image on the left.

Lighter regions on the attention map reflect parts of the image, that the model found more important when generating the corresponding word. This feature allows you to examine whether the attention component of the model has been well trained, and how it shifted focus during the generation process.

Below the captions are the attention alignments. This section provides a unified view of all the attention alignments generated by the model.

If provided by the model, the beam search output graph is displayed in the lower half of the page. It visualizes the tree of hypotheses the model produces during beam search decoding. On the left side is the start token, the root of the tree from which the hypotheses emerge. The finished hypotheses end with an ending token. The nodes of the tree display, besides the word, the negative log probability associated with the hypothesis up to that point. After clicking on a node, the corresponding attention alignment is displayed on top of the image at the top of the page.

4.1.5 Configuration Interface

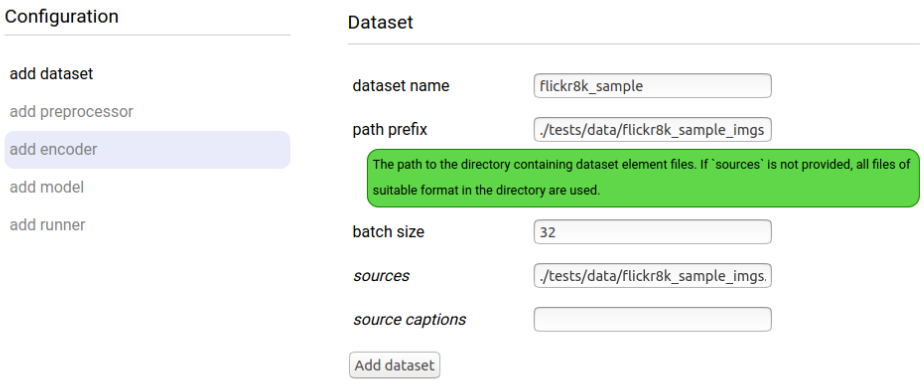
To create datasets of images or add new models to Macaque, navigate to the application menu in the upper left corner of the page and select "Configure". This switches the view to the Configuration tab.

There are five types of objects that the interface allows to instantiate: a dataset, a preprocessor, an encoder, a model and a runner.

Each object has its associated configuration form. We are not going to list each parameter required for each one of dataset, preprocessor, etc. – every input field in the forms will display a hint upon clicking on it. However, most of them are self-explaining.

Adding a Dataset. By utilising datasets you can run a model on multiple images at once. The associated form requires specifying the directory where the images are located and optionally a source file that has listed on lines the actual file names from said directory that make up the dataset. If such a file is

Figure 4.3: A screenshot of Macaque's Configure tab. Note the displayed hint at the path prefix field.



not provided, all files from the directory are used. You can also specify a file containing source captions for a multimodal translation model.

Adding datasets and models can be automatized by using configuration files. Configuration files are read after starting the application and automatically instantiate the entities in question, so the user can skip the manual part of filling up forms and jump into exploring the model outputs.

Adding a Preprocessor. Preprocessors preprocess the input images into formats that the model expects. That can include the shape of the image, number of channels, and the interval of values that pixels take. We implement 5 most basic types of preprocessing methods.

- **keep aspect ratio and crop** Rescales the image, preserving the aspect ratio, so that one of the sides matches the desired size and the other either matches, or exceeds it. If it exceeds the target size, the middle of the image is cropped so that it has the desired dimensions.
- **keep aspect ratio and pad** Rescales the image, preserving the aspect ratio, so that one of the sides matches, and the other either matches or falls short of the desired size. If it falls short, the image is zero-padded so that it has the desired dimensions.
- **rescale width rescale height** Rescales the image in both dimensions to match the target width and height.
- **rescale width and crop/pad height** Rescales the image so that its width matches the target width and crops or pads the image in the other direction.
- **rescale height and crop/pad width** Rescales the image so that its height matches the target height and crops or pads the image in the other direction.

Adding an Encoder. Macaque has built-in support for Keras encoders, utilising the `keras.applications` module and Tensorflow Slim encoder available via Neural Monkey's wrapper functions. Each interface provides a way to instantiate some of the vanilla image classification networks – VGG16, VGG19, ResNet and others.

Two types of information are required from the user for each of these networks. The path to the serialized model variables, and an identifier of the layer output of which will serve as the encoder output.

In the case of Keras – if no model checkpoint is provided, one is automatically downloaded and placed into `.keras/` in your home directory. This convenience is part of Keras' implementation and unfortunately is not available for the TensorFlow Slim case. If you wish to use TensorFlow Slim as the encoder interface, but do not have a network checkpoint, you can find a list of publiclly available checkpoints at `https://github.com/tensorflow/models/tree/master/research/slim#Pretrained`.

Obtaining layer names is slightly cumbersome in both Keras and TensorFlow Slim. With Keras, instantiating the network is necessary. The network object has a `layers` attribute, which is a list of layers. Each layer has a `name` attribute which stores the value. If using TensorFlow Slim, instantiatiating the model is performed by a function, not a constructor. This function, in the case of each network, returns a tuple - the network object and its endpoints, which is a list of layer names.

The third way of adding an encoder is trough a user plugin. This method allows to use arbitrary Python code as the feature extractor upon satisfying an API in the plugin. Plugins are explained in section 4.1.10.

Adding a Model. Macaque supports two types of model interfaces. The first is used exclusively for models written in the Neural Monkey framework and requires familiarity with the framework (4.1.9). The second is the plugin interface which supports employing arbitrary models written in Python using a common API. The plugin interface is explained in 4.1.10.

Adding a Runner. A runner puts all pieces of the processing pipeline together. Creating one consists of specifying a preprocessor (optional) a feature extractor (optional) and a model. After that, you can use this configuration to process images and datasets.

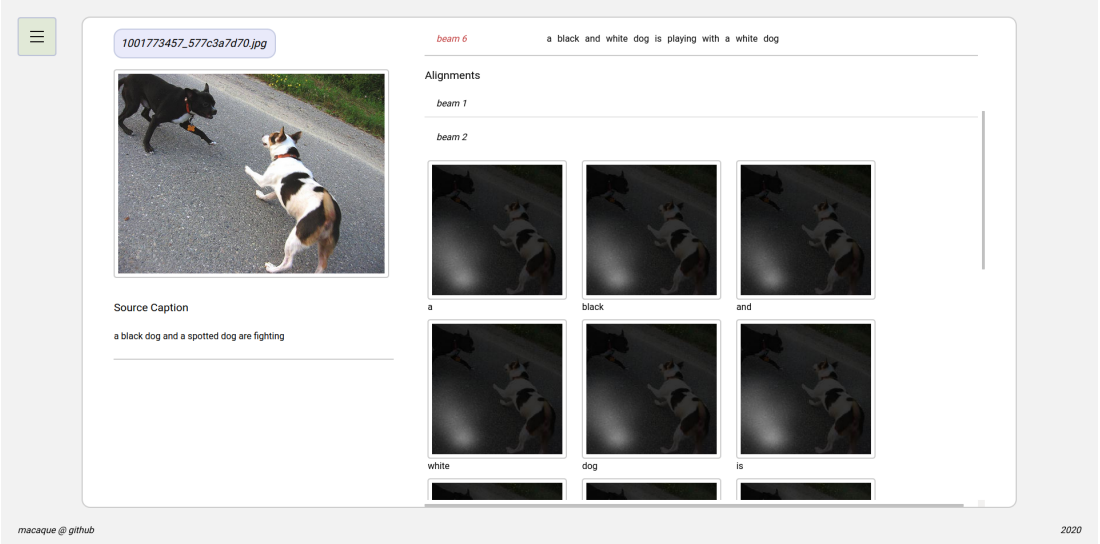
4.1.6 Datasets Tab

The Datasets tab available trough the "Datasets" option in the navigation menu offers a view of all registered datasets and the functionality to process them with runners, displaying the results for individual dataset elements.

After selecting a dataset in the menu on the left a list of dataset elements is presented in the middle of the page. Click on a row in the list to open the view of the instance. If no results are present only the image and optionally its source caption are displayed. After being processed by a runner, the results of model inference will be displayed. The features are the same as in the Home tab.

You can move between elements of the dataset by using arrow keys.

Figure 4.4: A screenshot of Macaque's Datasets tab showing attention alignments from a model whose attention component didn't properly learn to focus on objects in the image.



4.1.7 Configuration Files

The creation of datasets, preprocessors, encoders, models and runners can be automatized by configuration files. These files are read shortly after launching the application and make it possible to skip the step of manually filling the forms in the Configure tab at each run. The default location where Macaque looks for its configuration files is the `models/` directory in the root of the project. This setting can be overriden by supplementing the `--config_dir <dir>` parameter with the name of the alternative location when launching Macaque.

The files need to have the `.ini` extension to be recognized and need to follow the structure of standard INI files[1].

The file is split into sections. Each section starts with a header signaling the type of the object the section is defining, for example `[dataset]` or `[runner]`. Sections consist of key–value pairs separated by the = sign. Overall, there are only 5 types of sections (dataset, prepro, encoder, model, runner) and we provide exemplar configuration files covering each type along with a template, that the user can fill in for his purpose. These files can be found in the `models/templates` directory and in the appendix.

Each configuration file can contain every section only once (i.e. two `[runner]` sections can't be present in the same file). Cofiguration files are processed in alphabetical order of their names. The order in which sections of a file are processed is: datasets, preprocessors, encoders, models and finally runners. If Macaque finds a section with the `name` key value identical to an already existing section of the same type, the currently read section is skipped. This allows, for example, to define an encoder only in one configuration file, and to refer to it from others as well.

[1] `https://docs.python.org/3/library/configparser.html`

4.1.8 Exiting

To exit Macaque, navigate to your open shell where Macaque is running and simply press Ctrl-C.

4.1.9 Neural Monkey Crash Course

Macaque is primarily intended to be used alongside Neural Monkey models[2].

Neural Monkey (Helcl and Libovický, 2017) is a high-level sequence-to-sequence learning framework in Python 3 build on top of the TensorFlow machine learning library.

Neural Monkey provides a high-level API to control all the parts of a machine learning experiment. This includes specifications of the model, the datasets (training, validation, test), the training algorithm with its hyperparameters, the type of loss function, runners (that specify which information should be collected from the model at runtime) and evaluation metrics.

Models in Neural Monkey are not written as Python source code, but as configuration files. By using configuration files instead of programs, Neural Monkey abstracts plenty of work from the user, thus makes writing experiments and prototyping faster.

Core Concepts. Neural Monkey tackles the problem of sequence to sequence learning by introducing several concepts, which are utilised in practice through the configuration files.

In Neural Monkey, a dataset is a collection of *data series*. For example, an image captioning dataset would be represented by two data series, one for the images and another for the reference captions. This example also illustrates, that data series can be of different modalities. To handle this fact, *readers* are introduced. A reader in a data series configuration carries the information about the series data format, as well as providing a way to load the data.

The model is, in the most basic instance, an *encoder* and a *decoder*. Building on the image captioning example, the encoder could be a convolutional network and the decoder a recurrent neural network. The encoder recognizes the data, which it shall process, by the ID of the data series. Similarily, the decoder uses a data series ID to match its output with a reference, when computing the loss function.

A *trainer* specifies the learning algorithm to be used along with additional parametrizations and the loss function. A *runner* is used to produce outputs from the model. For example, to capture the greedy captions from the model, or the attention maps produced by the attention component.

Training a model in Neural Monkey requires exactly one configuration file. This file contains, besides other things, a description of the model and the training and validation datasets. At inference time, the same configuration is used to instantiate the model and another configuration file should be provided, which contains the description of the test dataset.

Neural Monkey provides the scripts *neuralmonkey-train* and *neuralmonkey-run* to train a model and perform inference.

[2]This, in fact, is the origin of its name.

Figure 4.5: An example of a Neural Monkey inference configuration file.

```
[vars]
data_prefix="/home/sam/thesis-code/macaque/tests/data/"

[main]
test_datasets=[<test_data>]

[test_data]
class=dataset.load
series=["feature_maps"]
data=[("{data_prefix}/flickr8k_sample_feats.txt", <numpy_reader>)]
outputs=[("greedy_caption", "output.txt")]

[numpy_reader]
class=readers.numpy_reader.from_file_list
prefix="{data_prefix}/flickr8k_sample_feats"
```

Configuration Files Syntax. We are going to explain the syntax and structure of Neural Monkey configuration files, but for simplification, we will restrain the exposition only to experiments in image captioning and multimodal translation. For more information about configuration files or generally about Neural Monkey, we redirect the reader to the original paper from Helcl and Libovický (2017) or to the documentation[3].

The configuration files follow the INI syntax; files are structured into sections, each section contains a header, followed by key-value pairs of parameters for the section. The section header contains the name of the section in square brackets. Each section, except the main section, describes a Python object, which is created at runtime. The key-value pairs then specify the name of the class and the constructor parameters, or the name of a creator function and its arguments.

The screenshot in figure 4.5 shows a Neural Monkey configuration file. This configuration file specifies a test dataset and could be used alongside a configuration file for the model to perform inference.

The section named `test_data` describes a Neural Monkey dataset. The dataset consists of one data series called "feature_maps". The actual data source along with a reader are specified by the `data` attribute.

The `numpy_reader` section describes the reader for the "feature_maps" data series. It signals that the data series elements are numpy arrays, and have to be specially loaded. It also lets to know the dataset constructor, that the file path in the first argument of the `data` attribute points to a file whose contents are names of files where these numpy arrays are serialized.

The `vars` and `main` sections are special. The first allows declaration of variables, which can be used in the INI file. The `main` section is an entrypoint for Neural Monkey. It puts together the datasets and the model, specifies which runners and evaluators to execute, properties of the training, such as the number of epochs, and validation periods.

[3] `https://neural-monkey.readthedocs.io`

Figure 4.6: An example of a naive implementation of a feature extractor plugin.

models > plugin_encoder.py > ...

```python
from my_feature_extractor import MyFeatureExtractor

class FeatureExtractorWrapper:

    def __init__(self):
        """Creates a custom feature extractor."""

        fe = MyFeatureExtractor()
        self.encoder = fe

    def extract_features(self, images):
        """Extracts the features from the images:

        Args:
            images: A Numpy array of RGB images of shape
                [batch, i_width, i_height, 3].
        Returns:
            A Numpy array of features of shape
            [batch, f_width, f_height, f_dim].
        """

        return self.encoder.run(images)

```

4.1.10 Plugin Interface

Macaque can be used even with other than Neural Monkey models. A plugin interface is defined to accomodate this. In this way, it is relatively straightforward for the user to employ Macaque with any image captioning or multimodal translation model written in Python.

To make this happen, the user is expected to provide a Python module implementing a certain class with a certain method.

Providing own components is allowed for the feature extractor and the decoder.

Feature Extractor Plugins. To use a custom feature extractor, the user has to provide a `.py` file containing a class named `FeatureExtractorWrapper` with a `extract_features` method. The method should accept one mandatory argument - the images, which should be converted into features. The images should be in the RGB format, represented as Numpy[4] arrays. The shape of the input array should be [batch, image width, image height, 3]. The output of this method should be the features, also in Numpy arrays of shape [batch, features width, features height, features dim]. An example of how this could look like in practice can be found in figure 4.6.

Model Plugins. To use a custom model (or decoder), the user has to provide a Python module with a class named `ModelWrapper` containing a method named `run`. When invoked, the method receives as its first argument Numpy arrays of either input images or extracted features, depending on the runner configuration. If the dataset contains source captions and the user-provided `run` function expects more than one argument, a list of strings is passed as the second argument. The list elements are the lines of the source caption file and preprocessing and

[4] `https://numpy.org/`

Figure 4.7: An explanation of the structure of output elements produced by custom models.

```
{
    'greedy': {
        'caption':      # A list of string tokens.
        'alignments':   # An iterable of Numpy arrays of shape [width, height].
    },
    'beam_search': {
        'captions':     # A list of lists of string tokens.
        'alignments':   # An iterable of iterables of Numpy arrays of shape [width, height].
        'graph':        # An instance of BeamSearchOutputGraph.
    }
}
```

tokenization is up to the user. The method should return a list of dictionaries. The length of the list has to match the number of the dataset elements, i.e. the 0-th dimension of the Numpy array provided as input. The dictionary has to adhere to a predefined structure for results to be acknowledged. The dictionary structure is given in figure 4.7. Keys can be missing, or associated values can be `None`, if they are not available from the model output.

BeamSearchOutputGraph Class. The `BeamSearchOutputGraph` class is responsible for holding the tree of the beam search hypothesis building process. It stores the hypothesis tokens, the alignments and the negative log probability associated with individual tokens. The class can be imported by using `from macaque.beam_search_output import BeamSearchOutputGraph`. Its constructor expects the following arguments, of which `tokens` and `parent_ids` are mandatory:

- **tokens** A list of lists of strings with structure $[maxtime, beamsize]$. All hypotheses should be prepended by a start token and finished hypotheses should be concluded by an end token.
- **alginments** A list of lists of Numpy arrays. As with *tokens*, the nested lists follow the structure $[maxtime, beamsize]$. The Numpy arrays are 2-dimensional and hold the alignments for individual tokens of the hypotheses.
- **scores** A Numpy array with shape $[maxtime, beamszie]$. Elements of the array are scores for intermediate hypotheses.
- **parent_ids** A Numpy array with shape $[maxtime, beamsize]$. Elements hold 0-based indexes of the parent hypotheses which the current token extends.

Hypotheses with length lower than *maxsize* should be padded by a padding token. Currently, custom values for the special tokens are not supported. Please follow Neural Monkey's convention in your implementation, utlising the strings "<s>" for the start token, "<\s>" for the end token, and "<pad>" for the padding token.

4.2 Developer's Documentation

Macaque can be divided into two parts – the backend and the frontend.

The frontend is written in JavaScript, building the architecture upon React's component model. The role of the frontend is to provide a user interface by which the user interacts with the application. Most data processing and computation happens on the backend. Using React components as building blocks results in a tree structure in the program's object composition. Components map to DOM elements – leaves of the tree are simple DOM elements that make up the appearacne of the application. React figures out which components to render on the screen. The root of the tree holds global application state. This structure contains most importantly the dataset, model and result representations. Some components perform http requests to the Python server, fetching data, which then may be stored in the global state. The frontend side of the application is explained in more detail in section 4.2.4.

The backend is written in Python. It consists primarily of http request handler methods which provide HTML or JSON-serialized objects (datasets, results, etc.) as responses, and wrappers, which facilitate running the user-provided models. Additionally, it contains classes responsible for the representation of datasets and individual dataset elements and code for image preprocessing and visualization of attention maps. Similarily to the frontend, it contains a global state - a dictionary which stores the representations of models, datasets, feature extractors, results and runner configurations. We will dive deeper into the backend in section 4.2.3.

4.2.1 The Structure of The Codebase

The structure of the codebase is intuitive and logical; the directories of interest found in the root directory of the project are `backend/` and `frontend/`. These directories contain all of the code that makes up Macaque. The root of the project alson holds the `tests/` directory, which contains application tests. The files found in the root directory include the scripts `install.sh`, `build.sh`, `macaque.sh`, `test.sh` for installing, building, running and testing respectively. The `build.sh` script bundles all the JavaScript source files into one using webpack and is necessary to perform, to apply changes made to the frontend. `requirements.txt` lists all Python dependencies of the project and is read by pip in the installation script.

backend/ Holds all the Python files. Model and encoder interfaces are placed in their own directories `model_wrappers/` and `feature_extractors/`. `templates/` contains HTML files, in accordance with the convention of Flask, the web framework we use. `lib/` contains third-party Python libraries that are not accessible as Python packages, in particular Neural Monkey.

frontend/ Contains all the JavaScript source files located in the `sources/` subdirectory. The `package.json` and generated `package-lock.json` files hold information about npm packages. For our purposes, they mainly store the list of frontend dependencies. The `webpack.config.js` file is webpack's configuration file; it defines where to look for files that should be added to the bundle, where to

place the output, and how to process special files, such as fonts and CSS. After bundling, the resulting output source is located in `dist/main.js`. `.babelrc` contains Babel directives (4.2.2).

4.2.2 Overview of Used Technologies

Macaque utilises several technologies to manage the interoperability between the backend and the frontend and to simplify the development process and code structure.

Node.js Node.js[5] is a JavaScript runtime. It allows JavaScript programs to run from the console, without the help of a browser. It also provides a standard library to facilitate, for example, file system manipulation or networking. We use Node.js for running several other tools associated with the development process.

npm Npm[6] is Node.js' package manager. We use it for dependencies management and package installation.

pip Pip[7] is Python's npm. We use it to manage the project's Python-based dependencies.

webpack Webpack[8] is a complex tool but at its core it is a JavaScript module bundler - it bundles multiple source files into one bundle, so it allows better project code organization. Additionally, by using a configuration file, one can specify preprocessing options for the sources. In this way, a loader can be used to transpile a ECMAScript6 JavaScript file into a browser-compatible script or to directly import a CSS file into a JavaScript module.

React React[9] is a JavaScript framework for building interactive user interfaces. It introduces a component-based model that provides an elegant way to build UIs by composing reusable components. A React application is tree of components with data flowing down, towards the leaves of the tree. Additionally, components can encapsulate their own state and rerendering of parts of the UI is done only when necessary.

JSX JSX is a superset of JavaScript extended by a XML-like syntax for describing React elements[10].

babel Babel[11] is a compiler from new versions of JavaScript (mainly ECMAScript6) into backwards compatible versions of the language. We use it as a part of out webpack pipeline for this reason.

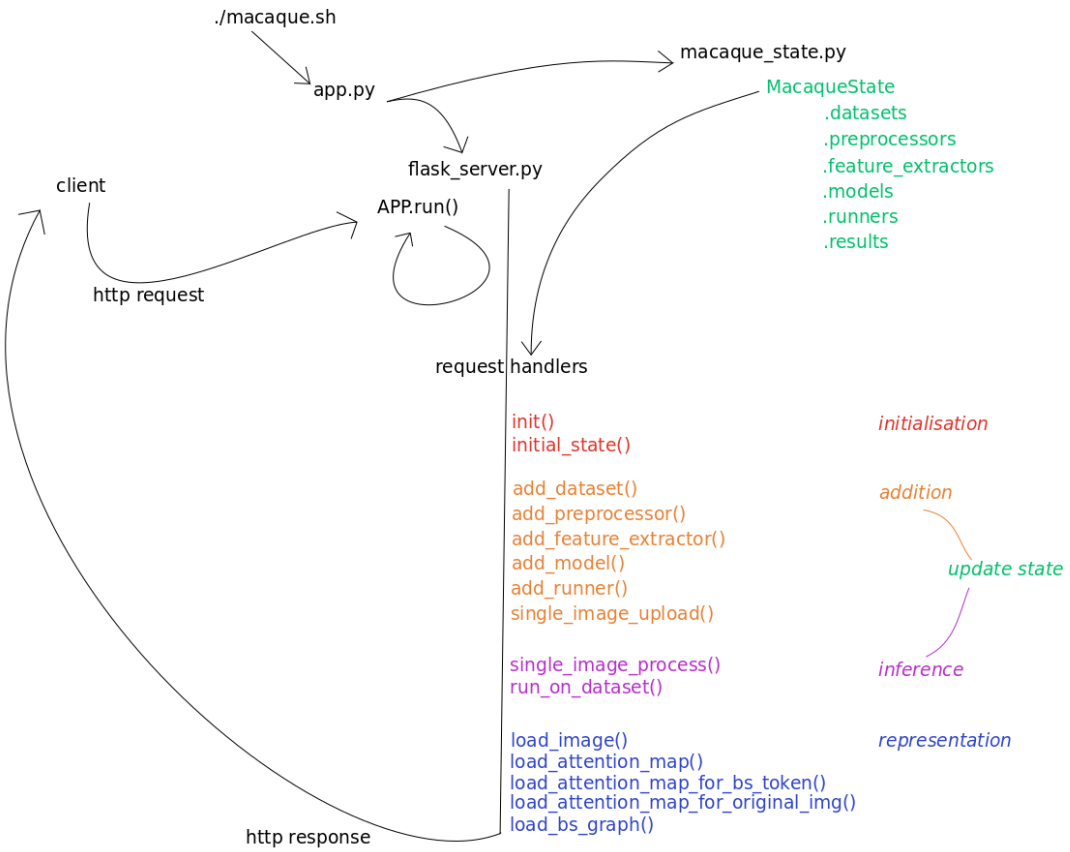
[5] `https://nodejs.org`
[6] `https://npmjs.com`
[7] `https://pypi.org/project/pip`
[8] `https://webpack.js.org`
[9] `https://reactjs.org`
[10] `https://reactjs.org/docs/introducing-jsx.html`.
[11] `https://babeljs.io`

d3.js D3.js[12] is a JavaScript library for data visualizations. We use it to build the beam search output graph.

4.2.3 The Backend

Overview of Functioning. The backend side of Macaque is responsible for running a web server and handling user requests. This involves loading datasets, instantiating models, running these models on datasets, collecting the results and transforming some of the data collected from the model's output into visualizations, which are sent over to the browser, to be displayed to the user.

Figure 4.8: A scheme of the backend execution. User request can be divided into four types – initialisatio, addition, inference and representation. Initialisation requests are performed once at the beginning of execution, addition and inference requests change application state and create new objects such as datasets, and perform model inference respectively. Representation requests produce data for visualization.



Chronologically, upon launching the program, the initial state is constructed, and updated after processing configuration files found in the configuration directory. The web server is launched and the rest consists of it fulfiling user requests.

[12] `https://d3js.org`

The web server maps the http requests to handler functions in `flask_server.py`. Conceptually, user requests can be categorized into *initialisation*, *addition*, *inference* and *representation*, based on "what they request".

Initialisation request are only performed at the start of the application, when the user enters the address where Macaque is running into the web browser's search bar. Addition requests result from the user requesting to add a new dataset, preprocessor, encoder, model or runner. Inference requests trigger processing of a dataset by a runner, responding the results of the run. Representation requests are requests for data that are subsequently displayed in the browser as visualizations of certain features. The data is already present, such as the image file of a dataset instance, or a computed attention alignment stored in the results, but the datum either resides on the server-side, or needs to be transformed into a form that is suitable for display.

Web Framework and Server. Macaque is a web application. Its backend is written in Python. A standardized way for universal web servers to communicate with Python applications is defined in the Web Server Gateway Interface (WSGI). The Python application has to fulfil the application side of the interface and then it is guaranteed to be able to run on most servers by using modules like CGI.

Flask[13] is a web framework in Python that abstracts over the WSGI and simplyfies working with http: it routes URLs to handler functions, supports variable sections in URLs, builds the http responses, provides a global Request object that accesses data from the http request and can send over files from the file system upon receiving a path.

We chose Flask from the choice of Python web frameworks because it is lightweight, easy to use, satisfies out functionality requirements and provides its own simple web server that we employ. The built-in server has disadvantages, for example, that it si single-threaded. We would consider an alternative as a future improvement (4.2.5).

Application State. The global state of the application is held in an instance of the `MacaqueState` class. It holds references to all of the datasets, preprocessors, encoders, models, runners and the results. At the beginning of the program's execution, configuration files found in the configuration directory are read and the objects defined in them are added to the state. Subsequently, the state is updated trough addition or inference requests made by the user.

Datasets. Datasets are implemented by the `Dataset` class. The `Dataset` class holds all the elements of the dataset, general information about the dataset and provides means to iterate over it by implementing Python's Iterator interface.

The image data is loaded on demand by calling the `load_images` method on the dataset instance. This method is called on batches of data, yielded by iteration, resulting in a lazy loading implementation.

The dataset instance holds general information such as the batch size and the number of elements. Additionally it has an `elements` attribute, which is a list of `DataInstance` instances.

[13] `https://flask.palletsprojects.com/`

The `DataInstance` class represents a single dataset element. It stores the path to the image data, the image data, upon loading, and optionally the source caption.

Preprocessing of Images. Preprocessing of images is handled by the `Preprocessing` class. It stores information about the target dimensions of the image and the method of preprocessing to be applied on the image. The methods available are listed in section 4.1.5. Operations on images are done in either their Numpy Array or Pillow[14] Image representations.

Feature Extracting. The `FeatureExtractor` class provides an interface for interacting with the encoder. It has a single method, called `extract_features` that expects a Numpy Array of images and returns a Numpy Array of features.

For every type of encoder there is a `FeatureExtractor` child class: the `KerasFeatureExtractor` class for encoders from Keras' Applications module, `NeuralMonkeyFeatureExtracotr` for interfacing with Neural Monkey's ImageNet encoders, and `PluginFeatureExtractor` for user-defined encoders.

Model Wrappers. Similarly to feature extractors, interfacing with models in Macaque is done trough a `ModelWrapper` class with descendant classes for specific model types. The `NeuralMonkeyModelWrapper` class manages the Neural Monkey Experiment, delegating data and transforming the results. The `PluginModelWrapper` loads and executes models provided by the user via a plugin that satisfies the interface described in section 4.1.10. All `ModelWrapper` child classes implement the `run` method that expects inputs as a Numpy Array and returns a list of dictionaries holding the results. The structure of the output dictionaries is identical to the one in section 4.1.10.

Runners. The `Runner` class connects the preprocessor, encoder and model into a single executable model pipeline that receives a `Dataset` instance into its `run` method, and produces output, that is subsequently sent over to the server.

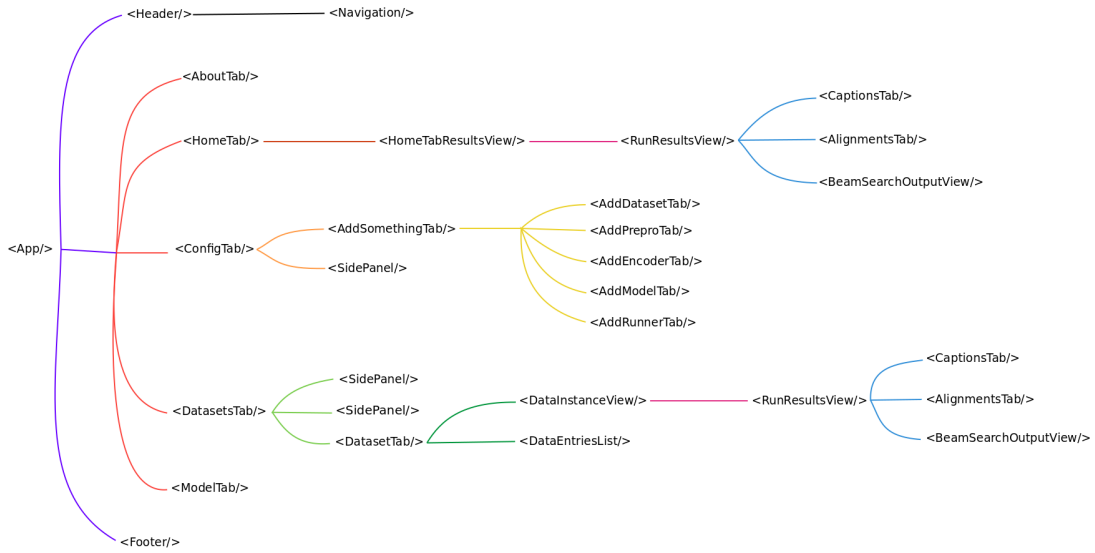
Attention Map Visualizations. The procedure we use to visualize attention alignments is based on Xu et al. (2015). The code that implements this algorithm is located in `visualizations.py`. We upscale the attention map to match the proportions of the image. Since the proportions of the map are usually significantly smaller than the image's (for example, 14x14, compared to 224x224), we smooth the upscaled attention map by a Gaussian Blur filter. We then paste the attention onto the image. Image operations are performed using the Pillow Python library.

Beam Search Output Graphs. We implement a unifying structure for beam search output in the `BeamSearchOutputGraph`, located in `beam_search_output.py`.

Beam search decoding typically outputs the *beamsize* number of captions. Optionally, the attention alignments for each caption (an attention map for each

[14] `https://pillow.readthedocs.io`

Figure 4.9: A graph of the frontend's central components relationships.



token of the caption), and the scores – either caption or token loss function values. However, if we want to reconstruct the tree of hypotheses, we need either all the partial hypotheses generated during decoding, or information about the branching – which hypothesis does each of the *beamsize* generated tokens at timestep t extend. This can be stored in a two-dimensional array; the value at index (i, j) is the number of the hypothesis the j-th token generated at timestep i extends. The `BeamSearchOutputGraph` expects in its constructor, the tokens, scores, alignments and an array of parent hypothesis indexes, and from this information constructs a tree structure. On user request, its instance is serialized using `BeamSearchOutputGraphEncoder` and sent over to the browser.

Validation. All user provided configurations are subject to validation before attempts at instantiation are made. Validation employs functions found in `validation.py`. User configurations can be either read from configuration files or input trough the frontend interface. In the case of configuration files, if an invalid value is detected, or data is anyhow invalid, the configuration file is skipped with a warning printed to standard output. In the case of input coming from the browser, the user only provides values, the form is legitimate since it is constructed programatically. If invalid values are detected for a certain key, a message describing the problem is added to an error log dictionary. The error log is sent back to the client instead of the result of the demanded operation. The error log is then displayed to the user in the browser.

4.2.4 The Frontend

The frontend side of Macaque is built using React. Some familiarity with React is therefore a prerequisite to understanding how the frontend works. For that reason, we will shortly explain React's basic concepts and how it is used to create interactive UIs before we proceed.

React Components. React is a JavaScript library for building interactive user interfaces. Central to this are *components*. A React application is built from components. A component is a either a JavaScript object with a special `render()` method or a function. The return value from the function or from the object's `render()` method is either another component or DOM elements. React applications have a root component which maps to the entire DOM tree and is injected by React into the HTML file.

Additionally, data can flow trough the tree. Components receive data from other components higher up the tree in the form of *props*. In the case of function components, props are passed as arguments. In the case of class (object) components, props are set as a property.

Class components can also have local state. Changing the props or state of a component will trigger its re-rendering (either calling its `render()` method, or invoking the function component). Automatic rendering triggered by changes in certain special data is the key to React's interactive UIs.[15]

Overview of Functioning. We use webpack to bundle all the JavaScript source files into one `main.js` file. This file is refered to from the `index.html` that is loaded by the browser after the user uses it to make the initial http request. The browser then loads `main.js` and builds and renders the DOM tree. We don't declare HTML tags in the HTML file or use plain JavaScript to create them after it is loaded. Instead, we define React components (4.2.4) that represent parts of the user interface. This structure of components forms a tree with DOM elements in the leaves. React then injects the structure into the HTML.

The root component, called `<App/>` holds the application state. Its constructor fetches the initial state (after processing the configuration files happens) from the backend. The component's `render()` method builds the DOM, passing event handlers and parts of its state to child components.

The design of the UI components is simple. The `<App/>` component renders a header of the page `<Header/>`, that contains a navigation menu `<Navigation/>`. The header is complemented by `<Footer/>` and the main tab is placed between them.

The value of the main tab is different, based on the selected item from the navigation menu. It can be either `<AboutTab/>`, `<ModelTab/>`, `<HomeTab/>`, `<ConfigTab>` or `<DatasetsTab/>`.

<HomeTab/> The `<HomeTab/>` consists notably of an input form for uploading an image, a runner selection menu `<RunnersMenu/>` and the results view `<HomeTabResultsView/>`. `<HomeTabResultsView/>` is just a wrapper that provides additional CSS to the `<RunResultsView/>`. The uploaded image is sent over to the backend, processed by the runner selected in the `<RunnersMenu/>` and the results are sent back, displayed by `<RunResultsView/>`.

<DatasetsTab/> The `<DatasetsTab/>` has a side panel made of two `<SidePanel/>` instances – one for listing available datasets, and the second for

[15]There is a lot more to React. We refer the reader to its well written documentation at `https://reactjs.org/docs/`.

runners. The side panels are complemented by the `<DatasetTab/>` which lists the dataset elements and displays the results. The list of instances is represented by `<DataEntriesList/>`, while `<DataInstanceView/>` displays the single instance view.

<ConfigTab/> The `<ConfigTab/>` consists of a side menu implemented by `<SidePanel/>`, that allows the user to select between input forms for different entities (datasets, models, etc.), and the input form in the middle of the page. The input form is either `<AddDatasetTab/>`, `<AddPreproTab/>`, `<AddEncoderTab/>`, `<AddModelTab/>` or `<AddRunnerTab/>`.

All the input forms receive a special event handler in their props from `<App/>`. When the user hits the subbmit button in the form, the configuration is sent to the server where an attempt to instantiate the object is made. The result from this operation is sent back to the browser. If the operation was unsuccessful, the form component displays an error message as `<ErrorTab/>`. If it was successful, the result is passed as an argument to the callback function provided by `<App/>`.

<ModelTab/> The `<ModelTab/>` displays formated information about the runners. Since Macaque can be deployed online, users don't have to be familiar with the models being used. It is interesting to try to understand why a model performs as it does and having the basic facts about its architecture and training settings is the first step. Descriptions can only be added to runners in configuration files by adding `about=<path/to/description>` to the `[runner]` section.

<AboutTab/> The `<AboutTab/>` simply displays a short description of the application.

4.2.5 Future Improvements

In this section we propose some of the things that could be improved or added to Macaque in the future.

Independent Web server. Macaque utilises Flask's simple built-in web server. It currently does not offer the possibility of running without it. Flask's server is single-threaded, and not designed to withstand any kind of security attack, or running for extended-periods of time. These limitations could certainly be undesired in a number of contexts.

Evaluation. Originally, we planned to add this feature, but in the end didn't due to time constraints. Including an evaluation feature that would compute popular or user provided metrics on the dataset results against reference captions would be a natural extension.

Datasets Online. When running Macaque on public IP addresses, the one feature missing is adding datasets. This is caused by an early design decision that we would not transfer data trough the network if they are present on the system

that runs the application, instead utilising Python standard library functions to access them. We could add an exception for this case but that would require nontrivial intervention in the current codebase, which we didn't undergo for time constraints.

More General Alignments Structure. The <AlignmentsTab/> React component expects only one set of alignments for each caption produced by the model. This makes sense when there is only one attention module in the neural network. But network architectures, such as the Transformer can include mutliple layers of attention, producing multiple attention alignments per caption. The <AlignmentsTab/>could be rewritten relatively easily to handle this more general case. However, training a model to demonstrate this feature would be more difficult than changing the code. For the lack of it, we decided to allocate time to other aspects of this thesis.

Machine Translation. Currently, there is a much greater community behind machine translation than image captioning or multimodal translation. Extending Macaque to this task could attract more people to the application, having a larger impact.

4.2.6 Testing

We provide tests for the backend side of the application. Tests utilise the pytest[16] testing framework and can be run by executing `./test.sh` from the command line, at the root directory of the project.

[16] `https://docs.pytest.org/en/latest/`

Conclusion

In consonance with the goals of this thesis, we have presented Macaque, an application for building intuitions about the workings of deep learning models on image captioning and multimodal translation. Macaque's primary contribution is to allow exploration of the inference process beneath attention mechanisms and beam search decoding. It allows to do so in a convenient and straightforward way by simple usage and intuitive visualizations. As such it is an unique application in the spectrum of tools developed to aid machine learning research and model interpretability.

Possibilites for further improvments of Macaque include adding more features, such as attention alignment visualizations for more complex model architectures, support for evaluation of results by common metrics, or extending the list of supported tasks by machine translation.

In the experimental part of this thesis, we proposed a new attention function formulation, called scaled general attention. In experiments covering image captioning and multimodal machine translation, we explored four model architectures and found that scaled general attention outperforms general attention, while achieving comparable results to additive attention. We evaluated our results both quantitatively, and qualitatively using Macaque.

Bibliography

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv e-prints*, art. arXiv:1409.0473, September 2014.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3(null):1137–1155, March 2003. ISSN 1532-4435.

Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *arXiv e-prints*, art. arXiv:1409.1259, September 2014a.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv e-prints*, art. arXiv:1406.1078, June 2014b.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv e-prints*, art. arXiv:1412.3555, December 2014.

J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

Desmond Elliott, Stella Frank, Khalil Sima'an, and Lucia Specia. Multi30K: Multilingual English-German Image Descriptions. *arXiv e-prints*, art. arXiv:1605.00459, May 2016.

Jeffrey L. Elman. Finding structure in time. *COGNITIVE SCIENCE*, 14(2): 179–211, 1990.

Ali Farhadi, Mohsen Hejrati, Amin Sadeghi, Peter Young, Cyrus Rashtchian, Julia Hockenmaier, and David Forsyth. Every picture tells a story: Generating sentences from images. volume 6314, pages 15–29, 09 2010. doi: 10.1007/978-3-642-15561-1_2.

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

A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, 2013.

Jindřich Helcl and Jindřich Libovický. CUNI System for the WMT17 Multimodal Translation Task. *arXiv e-prints*, art. arXiv:1707.04550, July 2017.

Jindřich Helcl, Jindřich Libovický, and Dušan Variš. CUNI System for the WMT18 Multimodal Translation Task. *arXiv e-prints*, art. arXiv:1811.04697, November 2018.

Jindřich Helcl and Jindřich Libovický. Neural monkey: An open-source tool for sequence learning. *The Prague Bulletin of Mathematical Linguistics*, (107): 5–17, 2017. ISSN 0032-6585.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 2342–2350. JMLR.org, 2015.

Andrej Karpathy and Li Fei-Fei. Deep Visual-Semantic Alignments for Generating Image Descriptions. *arXiv e-prints*, art. arXiv:1412.2306, December 2014.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, art. arXiv:1412.6980, December 2014.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

G. Kulkarni, V. Premraj, S. Dhar, S. Li, Y. Choi, A. C. Berg, and T. L. Berg. Baby talk: Understanding and generating simple image descriptions. In *CVPR 2011*, pages 1601–1608, 2011.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86, pages 2278–2324, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.7665.

Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft COCO: Common Objects in Context. *arXiv e-prints*, art. arXiv:1405.0312, May 2014.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective Approaches to Attention-based Neural Machine Translation. *arXiv e-prints*, art. arXiv:1508.04025, August 2015.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL https://www.aclweb.org/anthology/P02-1040.

Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. How to Construct Deep Recurrent Neural Networks. *arXiv e-prints*, art. arXiv:1312.6026, December 2013.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2018. URL `https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf`.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*, page 696–699. MIT Press, Cambridge, MA, USA, 1988. ISBN 0262010976.

Piyush Sharma, Nan Ding, Sebastian Goodman, and Radu Soricut. Conceptual captions: A cleaned, hypernymed, image alt-text dataset for automatic image captioning. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2556–2565, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1238. URL `https://www.aclweb.org/anthology/P18-1238`.

Jonathan Shen, Ruoming Pang, Ron J. Weiss, Mike Schuster, Navdeep Jaitly, Zongheng Yang, Zhifeng Chen, Yu Zhang, Yuxuan Wang, RJ Skerry-Ryan, Rif A. Saurous, Yannis Agiomyrgiannakis, and Yonghui Wu. Natural TTS Synthesis by Conditioning WaveNet on Mel Spectrogram Predictions. *arXiv e-prints*, art. arXiv:1712.05884, December 2017.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. *arXiv e-prints*, art. arXiv:1409.3215, September 2014.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv e-prints*, art. arXiv:1312.6199, December 2013.

A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950. ISSN 00264423. URL `http://www.jstor.org/stable/2251299`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *arXiv e-prints*, art. arXiv:1706.03762, June 2017.

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. *arXiv e-prints*, art. arXiv:1502.03044, February 2015.

Peter Young, Alice Lai, Micah Hodosh, and Julia Hockenmaier. From image descriptions to visual denotations: New similarity metrics for semantic inference over event descriptions. *Transactions of the Association for Computational Linguistics*, 2:67–78, 2014. doi: 10.1162/tacl_a_00166. URL `https://www.aclweb.org/anthology/Q14-1006`.

List of Figures

List of Tables

A. Appendix

A.1 Configuration Files Templates

Below are templates for all possible section types in Macaque's configuration files. The templates can be also found in `models/templates/templates.ini`. Fields that are not listed as optional are mandatory.

Figure A.1: A template for creating a dataset in Macaque. The *sources*, *references*, *srcCaps* fields are optional.

```
[dataset]
name=
batchSize=
prefix=
sources=
references=
srcCaps=
```

Figure A.2: A template for creating a preprocessor in Macaque.

```
[prepro]
name=
targetWidth=
targetHeight=
mode=
```

Figure A.3: A template for creating a plugin encoder in Macaque.

```
[encoder]
name=
type=plugin
path=
```

Figure A.4: A template for creating a Keras encoder in Macaque. The *checkpoint* field is optional.

```
[encoder]
name=
type=keras
network=
layer=
checkpoint=
```

Figure A.5: A template for creating a TensorFlow Slim encoder in Macaque.

```
[encoder]
name=
type=tfSlim
network=
layer=
checkpoint=
```

Figure A.6: A template for creating a plugin model in Macaque.

```
[model]
name=
type=plugin
runsOnFeatures=
path=
```

Figure A.7: A template for creating a Neural Monkey model in Macaque. The *srcCaptionSeries*, *attnSeries* and *bsSeries* are optional.

```
[model]
name=
type=neuralmonkey
runsOnFeatures=
configPath=
varsPath=
dataSeries=
greedySeries=
srcCaptionSeries=
attnSeries=
bsSeries=
```

Figure A.8: A template for creating a runner in Macaque. The *prepro*, *encoder* and *about* fields are optional.

```
[runner]
name=
prepro=
encoder=
model=
about=
```