Tanta University

Department of Computer and Automatic Control Engineering

Faculty of Engineering

On-Device Visual Question Answering (VQA) System Aimed for the Visually Impaired

Graduation Project

Prepared by

Ashraqat Hesham Ahmad

Mohamed Abdelrahman Nasser

Mostafa Alaa Nafie

Nadine Muhammad Fadlallah

Nada Adel Moussa

Supervised by

Moustafa Alzantot, Ph.D

July 2021

Table of Contents

Abstract

In this project, we propose the task of open-ended Visual Question Answering (VQA). Our main aim is to develop an AI-powered system to help empower blind and visually impaired individuals to be more independent and comfortable practicing their daily chores and activities (e.g., shopping, cooking, meeting people, playing sports) by enabling them to obtain information about images in the real world. VQA is multi-discipline Artificial Intelligence (AI) problem; it combines Computer Vision (CV), Natural Language Processing (NLP), and Knowledge Representation & Reasoning (KR). Today deep learning methods have become one of the most popular sub-fields of machine learning that are being employed to solve conventional artificial intelligence problems. Recent developments in computer vision and deep learning algorithms have enabled enormous progress in many computer vision tasks, such as image classification, object detection, face recognition, action, and activity recognition. Given enough data, deep neural networks (DNNs) outperform human performance to do image classification. With annotated realistic datasets rapidly increasing in size thanks to crowd-sourcing and advances in computational power, similar outcomes can be anticipated for other focused computer vision problems. Another remarkable progress achieved in deep neural networks has been applied to various NLP tasks such as part-of-speech (POS) tagging, named entity recognition, and semantic role labeling. In this project, we are aiming to implement a VQA system that uses the device's camera to capture an image of the user's surroundings and the microphone to accept the user's question(s) expressed in natural language. Given an image and a natural-language question about the image, the task is to understand both the image and question and respond to the user with an accurate natural-language answer using visual elements of the image and inference gathered from textual questions. In this review, we will address our solution to address the VQA problem taking into account the practical issues associated with such systems such as availability, reliability, and privacy.

1 Introduction

1.1 Motivation

Globally, among 7.79 billion people living in 2020, an estimated 49.1 million were blind and an estimated 33.6 million had severe visual impairment according to the Association for Research in Vision and Ophthalmology (ARVO) [1]. Vision impairment severely impacts the quality of life among adult populations. An individual with vision impairment is more likely to experience limitations in their independence, mobility, and productivity which then contribute to lower rates of workforce participation and higher rates of depression and anxiety [2].

1.2 Problem Statement

In this project, we aim to help individuals with visual impairment have a normal life without the burden of relying on others by developing a prototype of an assistive system that provides "Visual Question Answering (VQA) [3]". Our system utilizes a camera to capture and a microphone to accept a question expressed in natural language. The system will then use AI models to understand both the image and the question and produce an answer to the user through the device's speakers. However, to achieve this goal various practical challenges must be addressed such as availability, privacy, and reliability.

1) **Availability**: Ensuring that the system does not require the presence of either another human person or a high−quality Internet connection.
2) **Privacy:** Ensuring that the user's sensitive information (such as photos and videos) is not exposed to either another human or to remote servers.

1.3 Solution Overview

In our project, we purpose the VQA system as an android mobile application that queries an image from the user's mobile camera and a question expressed in natural language from the mobile's microphone and responds back with the answer via the mobile's speakers. In addition to that, the entire pipeline of question understanding, image understanding, and answer generation models are run offline and locally on the user's device. With that, we are able to achieve availability, privacy and speed as the machine learning model is run completely locally on the device without the need to talk to a remote server over the internet.

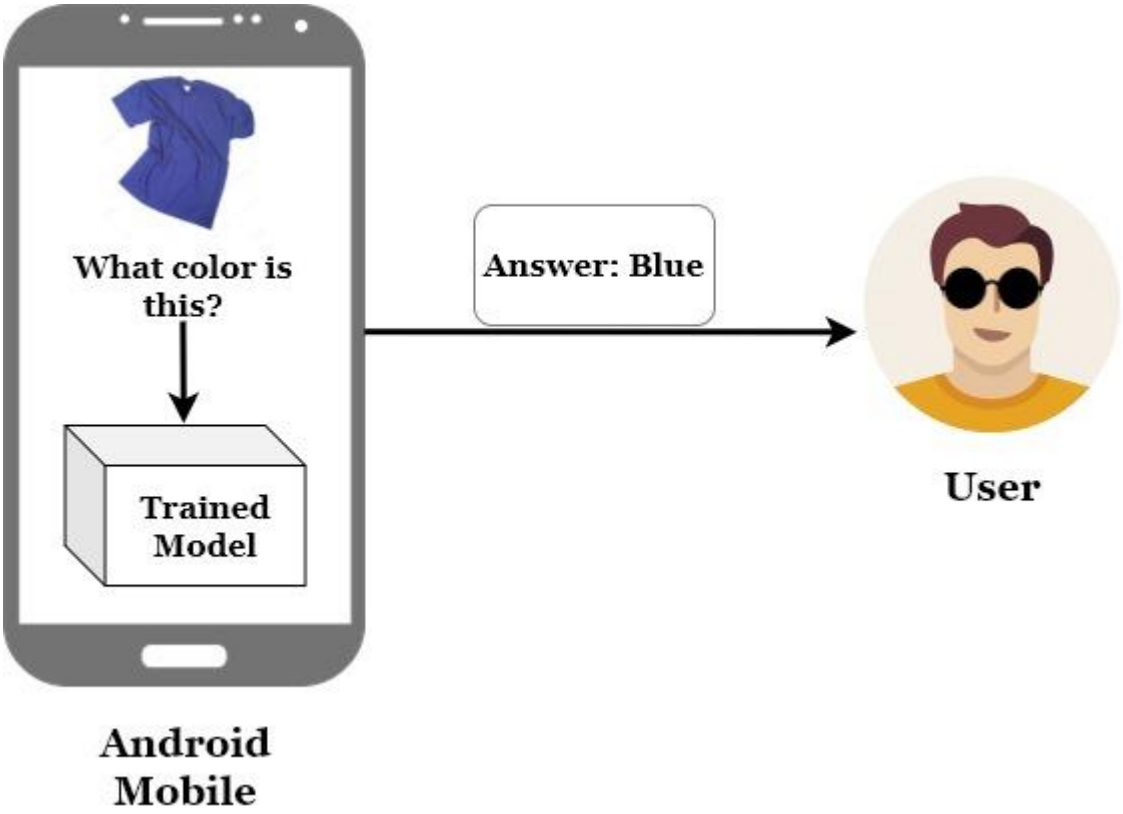


Figure 1: An entire VQA system run and deployed on a mobile device.

In this review, we place particular emphasis on exploring the VQA task, existing datasets, and methods for implementing and deploying VQA. In Section 2, we discuss the VQA problem and how it differs from other computer vision tasks, some of which also require the integration of vision and language (e.g., image captioning). Then, in Section 3, we describe currently available datasets for the VQA task with an emphasis on their strengths and weaknesses. We discuss how biases in some of these datasets severely limit their ability to assess algorithms. In section 4, we discuss possible future developments in vision and language tasks in the context of Edge AI [3] (i.e., deep learning models implemented on mobile and embedded devices). We then move to discuss in depth our chosen ML model for our project and the training results in section 5. In section 6, we discuss our approach on how we managed to deploy our model on the mobile phone. Finally, we discuss future work and advancmets that could be employed in our project.

2 A Vision and Language Approaches related to VQA

In this project, we introduce the task of free-form and open-ended Visual Question Answering (VQA). VQA is a computer vision task where a system takes as input an image and free-form, open-ended, natural-language question about the image and produces a natural-language answer as the output. Questions can be arbitrary and they encompass many sub-problems in computer vision, e.g.:

- Object recognition - What is in the image?
- Object detection - Are there any cats in the image?
- Attribute classification - What color is the cat?
- Scene classification - Is it sunny?
- Counting - How many cats are in the image?

Figure 2: Examples of free-form, open-ended questions collected for images and their produced answers

The main goal for the VQA system is to extract question-relevant semantic information from the images, which ranges from the detection of minute details to the inference of abstract scene attributes for the whole image, based on the question. While many computer vision problems involve extracting information from the images, they are limited in scope and generality compared to VQA. Object recognition, activity recognition, and scene classification can all be represented as image classification tasks, with today's best methods doing these using CNNs trained to classify images into particular semantic categories. The most successful of these is object recognition, where algorithms surpass humans in terms of accuracy. However, object recognition requires only classifying the dominant object in an image without knowledge of its spatial position or its role within the larger scene. Object detection involves the localization of specific semantic concepts (e.g., cars or people) by placing a bounding box around each instance of the object in an image. The best object detection methods all use deep CNNs [4]. Semantic segmentation takes the task of localization a step further by classifying each pixel as belonging to a particular semantic

class [5, 6]. Instance segmentation further builds upon localization by differentiating between separate instances of the same semantic class [7, 8, 9]

While semantic and instance segmentation are important computer vision problems that generalize object detection and recognition, but they are not sufficient for holistic scene understanding. One of the major problems they face is label ambiguity. For example, in Figure 3, the assigned semantic label for the position of the yellow cross can be 'bag', 'black,' or 'person.' The label depends on the task. Moreover, these approaches alone lack the ability to provide any contextual information about the scene. In this example, labeling a pixel as 'bag' does not inform us about whether it is being carried by the person, and labeling a pixel as 'person' does not tell us if the person is sitting, running, or skateboarding. This is in contrast with VQA, where a system is required to answer arbitrary questions about images, which may require reasoning about the relationships of objects with each other and the overall scene. The appropriate label is specified by the question.

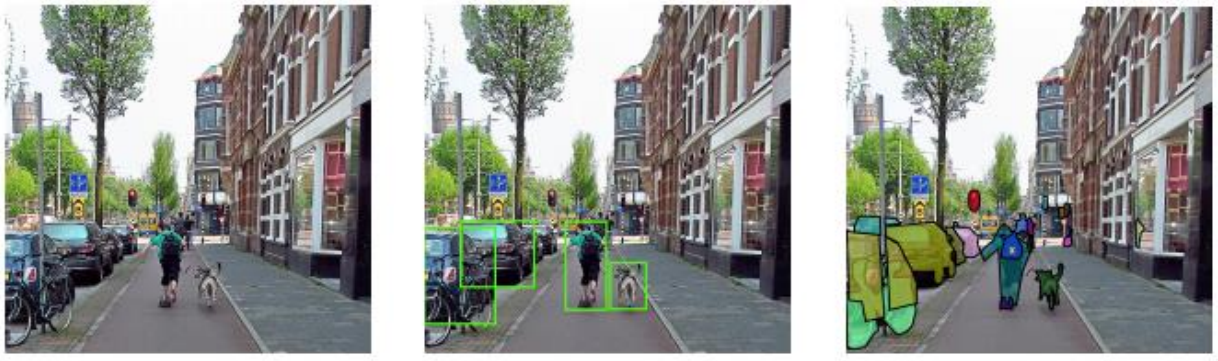


Figure3: Object detection, semantic segmentation, and image captioning compared to VQA. The middle figure shows the ideal output of a typical object detection system (i.e., a rectangular box around each object in the scene), and the right figure shows the semantic segmentation map from the COCO dataset [10]. Both tasks are unable to inform us about the actual context or role of that object within the scene. The captions for this COCO image range from very generic descriptions of the scene, e.g., A busy town sidewalk next to street parking and intersections., to a very focused discussion of a single activity without qualifying the overall scene, e.g., A woman jogging with a dog on a leash. Both are acceptable captions, but compared to VQA, significantly more information can be extracted. For the COCO-VQA dataset, the questions asked about this image are What kind of shoes is the skater wearing?, Urban or suburban?, and What animal is there?

Besides VQA, image captioning also combines vision with language, the goal here is to have an algorithm that is able to produce a detailed natural language description of a given image. To be able to achieve that, describing complex attributes and object relationships needs to be involved.

One major challenge of the image captioning task is the evaluation of captions. The ideal approach is evaluating using human judges, but this method is proven to be slow and expensive. For this reason, multiple automatic evaluation schemes have been proposed. The most widely used caption evaluation schemes are BLEU [11], ROUGE [12], METEOR [13], and CIDEr [14]. However, each of these metrics shows less quality in comparison to human captions.

In conclusion, when comparing image captioning to the VQA problem, image captioning arbitrarily choose the level of granularity of its image analysis whereas, in VQA, the level of granularity is specified by the nature of the question being asked. For example, 'What season is this?' will require understanding the entire scene, but 'What is the color of the dog standing behind the girl with a white dress?' would require attention to specific details of the scene.

3 Datasets for VQA

Beginning in 2014, five major datasets for VQA have been publicly released. These datasets enable VQA systems to be trained and evaluated. As of this review, the main datasets for VQA are DAQUAR[15] , COCO-QA [15], The VQA Dataset [16], FM-IQA [17], Visual7W[18], and Visual Genome [19]. All of the datasets -except DAQUAR- include images from the Microsoft Common Objects in Context (COCO) dataset , which consists of 328,000 images, 91 common object categories with over 2 million labeled instances, and an average of 5 captions per image.

An ideal VQA dataset is required to be sufficiently large to capture the variability within questions, images, and contexts that occur in real-world scenarios. It should also have a fair evaluation scheme that is difficult to 'game' and doing well on it indicates that an algorithm can answer a large

variety of question types about images that have definitive answers. If a dataset contains easily exploitable biases in the distribution of the questions or answers, it may be possible for an algorithm to perform well on the dataset without really solving the VQA problem. In the following subsections, we critically review the available datasets. We describe how the datasets were created and discuss their limitations.

3.1 DAQUAR

The DAtaset for QUestion Answering on Real-world images (DAQUAR) was the first major VQA dataset to be released. It is one of the smallest VQA datasets. It only contains 6795 training and 5673 testing QA pairs based on images from the NYU-DepthV2 Dataset [20]. The dataset is also available in an even smaller configuration consisting of only 37 object categories, known as DAQUAR-37 with 3825 training QA pairs and 297 testing QA pairs. While DAQUAR was a pioneering dataset for VQA, it is too small to successfully train and evaluate more complex models. Apart from the small size, DAQUAR contains exclusively indoor scenes, which constrains the variety of questions available. The images tend to have significant clutter and in some cases extreme lighting conditions (see Figure 4). This makes many questions difficult to answer, and even humans are only able to achieve 50.2% accuracy on the full dataset.

3.2 COCO-QA

In COCO-QA [21], QA pairs for each image are created using a Natural Language Processing (NLP) algorithm that derives them from the COCO image captions. For example, using the image caption A boy is playing Frisbee, it is possible to generate the question "What is the boy playing?" with "Frisbee" as the answer. COCO-QA contains 78,736 training and 38,948 testing QA pairs. Most questions ask about the object in the image (69.84%), with the other questions being about color (16.59%), counting (7.47%), and location (6.10%). All of the questions have a single-word answer, and there are only 435 unique answers. These constraints on the answers make evaluation relatively straightforward.

The biggest shortcoming of COCO-QA is due to flaws in the NLP algorithm that was used to generate the QA pairs. Where, longer sentences are broken into smaller chunks for ease of processing, but in many of these cases the algorithm does not cope well when a sentence contains clauses and grammatical variations. This results in awkwardly phrased questions, with many containing grammatical errors, and others being completely meaningless (see Figure 5). The other major shortcoming is that it only contains four kinds of questions, and these are limited to the kinds of things described in COCO's image captions.

COCO-QA: What does an intersection show on one side and two double-decker buses and a third vehicle,?
Ground Truth: Building

DAQUAR: What is behind the computer in the corner of the table?
Ground Truth: papers

Figure 4: Sample images from DAQUAR and the COCO-QA datasets and the corresponding QA pairs. A significant number of COCO-QA questions have grammatical errors and are nonsensical, whereas DAQUAR images are often marred with clutter and low-resolution images.

3.3 The VQA Dataset

The VQA Dataset [22] consists of both real images from COCO and abstract cartoon images. Most work on this dataset has focused solely on the portion containing real-world imagery from COCO, which we refer to as COCO-VQA. We refer to the synthetic portion of the dataset as SYNTH-VQA. COCO-VQA consists of three questions per image, with ten answers per question. Compared to other VQA datasets, COCO-VQA consists of a

relatively large number of questions (614,163 total, with 248,349 for training, 121,512 for validation, and 244,302 for testing).

SYNTH-VQA consists of 50,000 synthetic scenes that depict cartoon images in different simulated scenarios. Scenes are made from over 100 different objects, 30 different animal models, and 20 human cartoon models. The models span different ages, gender, facial expressions, and races to provide variation in appearance. SYNTH-VQA has 150,000 QA pairs with 3 questions per scene and 10 ground-truth answers per question. By using synthetic images, it becomes possible to create a more varied and balanced dataset.

Due to the diversity and size of the dataset, COCO-VQA has been widely used to evaluate algorithms. However, there are multiple problems with the dataset. COCO-VQA has a large variety of questions, but many of them can be accurately answered without using the image due to language biases. Relatively simple image-blind algorithms have achieved 49.6% accuracy on COCO-VQA using the question alone. The dataset also contains many subjective, opinion-seeking questions that do not have a single objective answer (see Figure 5). Several other practical issues also arise out of the dataset's biases. For example, 'yes/no' answers span about 38% of all questions, and almost 59% of them are answered with 'yes.' Combined with the evaluation metric used with COCO-VQA, these biases can make it difficult to assess whether an algorithm is truly solving the VQA problem using solely this dataset.

(a) Q: Would you like to fly in that? GT: yes (4x), no (6x). The VQA Dataset contains subjective questions that are prone to cause disagreement between annotators and also clearly lack a single objectively correct answer.

(b) Q: What color are the trees? GT: green. There are 73 total questions in the dataset asking this question. For 70 of those questions, the majority answer is green. Such questions can be often answered without information from the image.

(c) Q: Why would you say this woman is strong? GT: yes (5x), can lift up on arms, headstand, handstand, can stand on her head, she is standing upside down on stool. Questions seeking descriptive or explanatory answers can pose significant difficulty in evaluation.

Figure 5: Open ended QA pairs from The VQA Dataset for both real and abstract images.

3.4 FM-IQA

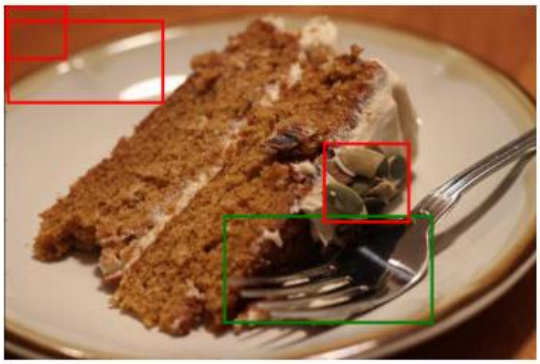
The Freestyle Multilingual Image Question Answering (FM-IQA) dataset is another dataset based on COCO. It contains human-generated answers and questions. The dataset was originally collected in Chinese, but English translations have been made available. Unlike COCO-QA and DAQUAR, this dataset also allowed for answers to be full sentences. This makes automatic evaluation with common metrics challenging. For this reason, the authors suggested using human judges for evaluation, where the judges are tasked with deciding whether or not the answer is provided by a human or not as well as assessing the quality of an answer on a scale of 0–2. This approach is impractical for most research groups and makes developing algorithms difficult.

3.5 Visual7W

The Visual7W dataset is a subset of Visual Genome. Visual7W contains 47,300 images from Visual Genome that are also present in COCO. Visual7W is named after the seven categories of questions it contains: What, Where, How, When, Who, Why, and Which. The dataset consists of two

distinct types of questions. The 'telling' questions are identical to Visual Genome questions and no binary questions, and the answer is text-based. The 'pointing' questions are the ones that begin with 'Which,' and for these questions, the algorithm has to select the correct bounding box among alternatives. An example pointing question is shown in Figure 4b. Visual7W uses a multiple-choice answer framework as the standard evaluation, with four possible answers being made available to an algorithm during evaluation.

(a) Example image from the Visual Genome dataset along with annotated image regions. This figure is taken from [35].
Free form QA: What does the sky look like?



(b) Example of the pointing QA task in Visual7W [34]. The bounding boxes are the given choices. Correct answer is shown in green
Q: Which object can you stab food with?

Figure 6: Visual7W is a subset of Visual Genome. Apart from the pointing task, all of the questions in Visual7W are sourced from Visual Genome data

4 The Future of Vision and Language tasks

Deep learning is the key enabler for many recent advances in artificial intelligence applications. While recent breakthroughs in deep learning research have enabled numerous mobile applications, the greater benefit will come when the artificial intelligence technologies becomes ubiquitous in mobile applications, such as automatous driving, affordable robots for home, and more intelligent personal assistance on mobile phones.

Deep learning and inference on the mobile device (i.e., TinyML [22]) shows several advantages when compared to the traditional mobile sensing and cloud computing paradigm. The advantages include:

1) Saving of communication bandwidth: the more computing done locally on mobile device, the less data needs to be sent to the cloud.

2) The reduction of cloud computing resource cost: due to the fact that computation on mobile devices becomes possible as mobile devices become more computationally powerful, there will be no need for the extra cost of renting or maintaining cloud computing resources.

3) The quick response time: if all computation can be performed locally, there will be no overhead in communication time or any concerns related to server reliability. For some applications, such as in the health care and military domains, this response time is very critical.

4) Mobile computing keeps the sensory data on the local device: this greatly improves user data privacy.

5 Related Work

With recent advancements in computer vision (CV) and natural language processing (NLP), visual question answering (VQA) becomes one of the most active research areas. Many papers have been published that suggest different model architectures for the VQA task based on neural networks. A commonly used approach is to extract a global image feature vector using a convolution neural network (CNN) and encode the corresponding question as a feature vector using a long short-term memory network (LSTM) [23] and then combine them to infer the answer.

In VQA: Visual question answering paper [3], the authors introduced the task of free-form and open-ended Visual Question Answering (VQA) and introduced a large dataset based on the MS COCO dataset for the task of visual question answering. They also set up an evaluation server and organized an annual challenge. They trained a model that followed the common approach by having an image channel that provides an embedding for the image using the VGGNet [24] and a question channel that provides an embedding for the question where An LSTM with one hidden layer is used to obtain 1024-dim embedding for the question. Each question word is encoded with 300-dim embedding. For the answer prediction, they used a Multi-Layer Perceptron (MLP), where the image

and question are fused together via element-wise multiplication to obtain a single embedding, which is then fed into a fully connected neural network classifier with 2 hidden layers and 1000 hidden units (dropout 0.5) in each layer with tanh non-linearity, followed by a softmax layer to obtain distribution over K = 1000 answers. The entire model is learned end-to-end with a cross-entropy loss.

In Stacked attention networks for image question answering paper [25], the authors found that the previous model often fails to give precise answers when such answers are related to a set of fine-grained regions in the image. They related that to the reason that an answer relates only to a small region of the image. Therefore, using the one global image feature vector to predict the answer could lead to sub-optimal results due to the noises introduced from regions that are irrelevant to the potential answer. They also found that answering a question from an image requires multi-step reasoning. In this paper, they proposed stacked attention networks (SANs) that allow multi-step reasoning for VQA. It applied the attention mechanism to the task of VQA, where the model pinpoints the regions of the image that highly relevant to the answer. The SAN consists of three major components: (1) the image model, which uses a CNN, specifically, the VGGNet, to extract high-level image representations, one vector for each region of the image. (2) The question model, which uses a CNN or an LSTM to extract a semantic vector of the question. (3) The stacked attention model, which locates the image regions that are relevant to the question for answer prediction. The SAN first uses the question vector to query the image vectors in the first visual attention layer, and then combines the question vector and the retrieved image vectors to form a refined query vector to query the image vectors again in the second layer.

The authors of Bottom-up and top-down attention for image captioning and visual question answering paper [26] proposed a mechanism that enables attention to be calculated at the level of objects rather than image regions. The paper introduced a model where the bottom-up mechanism is based on Faster R-CNN [27] proposes image regions, each with an associated feature vector, while the top-down mechanism determines feature weightings using the question representation as context.

Then the image embedding and the question embedding are fused together via element-wise multiplication, followed by a prediction of regression of scores over a set of candidate answers.

A recent paper is VisualBERT [28] where the authors proposed a simple and flexible model designed for capturing rich semantic in the image and associated text. VisualBERT integrates BERT, a recent Transformer-based model for natural language processing, and pre-trained object proposals systems such as Faster-RCNN and it can be applied to a variety of vision-and- language tasks including VQA. In this model, the image features extracted from object proposals are treated as unordered input tokens and fed into VisualBERT along with text. The text and image inputs are jointly processed by multiple Transformer layers in VisualBERT. The rich interaction among words and object proposals allow the model to capture the intricate associations between text and image.

Idea of using RNNs and CNNs to perform visual understanding tasks had also been presented here [29] Building three models for three tasks: image captioning, activity recognition, and video description. These models are based on a Long-term Recurrent Convolutional Network (LRCN) combining a deep hierarchical visual feature extractor (such as a CNN) with a model that can learn to recognize and synthesize temporal dynamics for tasks involving sequential data (inputs or outputs), visual, linguistic, or otherwise.

The approach of combining RNNs and CNNs to answer questions about visual content is followed here [37]. It uses GoogleNet architecture pre-trained on the ImageNet dataset to represent the image, the final dense layer is randomly initialized and trained along with the LSTM network which takes two inputs at each time step: the CNN output and a word of the question represented with one-hot vector encoding. The answer is then generated using a conditional language model that take the encoded image features and the question as input .The approach is evaluated on the DAQUAR dataset.

6 VQA Model and Training

6.1 Deep Learning Review

As we previously mentioned, many research papers suggest various model architectures for the VQA task based on neural networks. Before we head to discuss in detail the VQA model implemented for our project, we would first briefly review some deep learning concepts employed in our approach in developing the VQA system.

6.1.1 Multi-Layer Perceptrons

Multi-Layer Perceptrons (MLPs) are *fully connected* feedforward neural networks. This means there are no feedback signals flowing from later to earlier layers. MLPs compose several non-linear functions $f(x) = \hat{y}(h_3(h_2(h_1(x))))$ where $h_i(.)$ are called *hidden layers* and $\hat{y}(.)$ is the *output layer*. Each layer i in a MLP comprises multiple neurons j which are implemented as affine transformations $a^T x + b$ followed by non-linear activation functions (g):

$$h_{ij} = g(a_{ij}^T h_{i-1} + b_{ji})$$

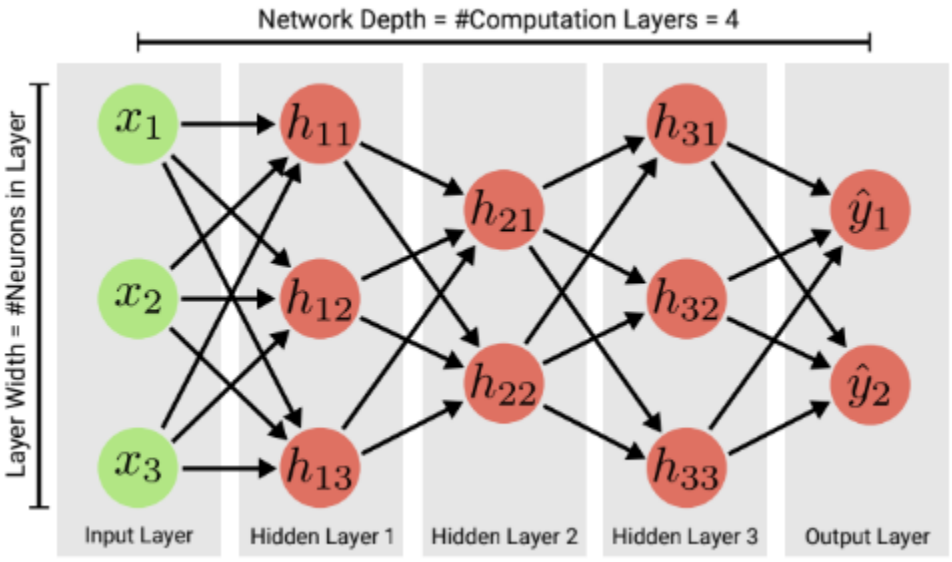


Figure 7: A multi-layer perceptron with 4 layers

MLPs use (mostly) nonlinear differentiable activation functions and are trained using backpropagation. This allows the model to learn more complex functions than a network trained using a linear activation function. Thus, the only utility of the hidden layers is to transform the input vectors into features that can be processed by the simple linear regression or logistic regression output layer as illustrated in Figure 8.

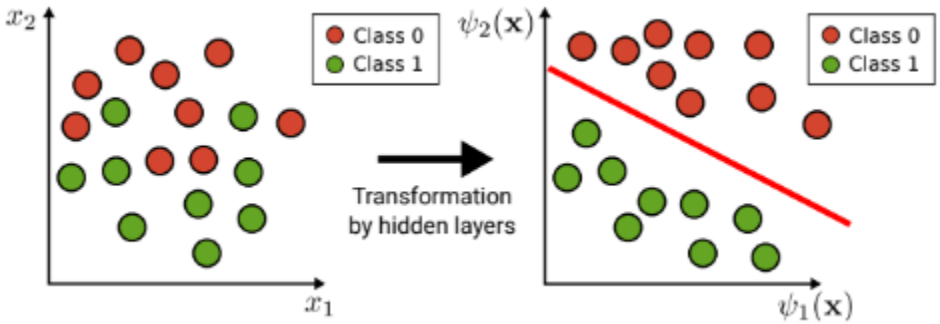


Figure 8: Hidden layers transform the input into better features

Some different activation functions that can be used are shown in Fig. 9.

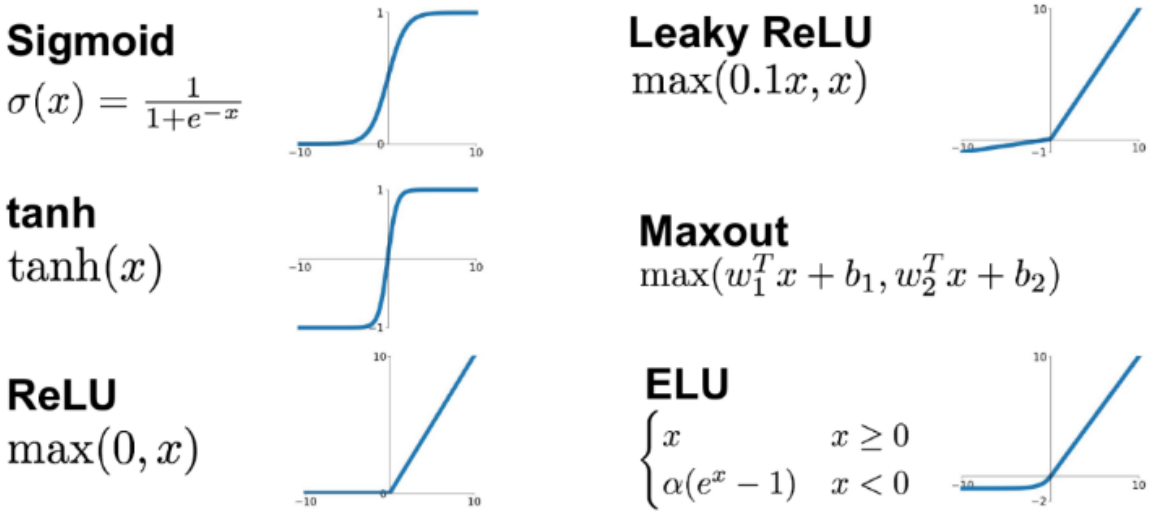


Figure 9: Activation functions that provide the needed nonlinearity

MLPs can be trained using the backpropagation algorithm and (stochastic) gradient descent as follows:

1. Initialize weights w, pick learning rate η and minibatch size $|X_{batch}|$
2. Draw (random) minibatch $X_{batch} \subseteq X$
3. For all elements $(x, y) \in X_{batch}$ of minibatch (in parallel) do:
   (a) Forward propagate x through network to calculate $h_1, h_2, \ldots, \hat{y}$
   (b) Backpropagate gradients through network to obtain $\nabla w L(\hat{y}, y)$
4. Update gradients: $w_{t+1} = w_{t+} - \eta \frac{1}{|X_{batch}|} \sum_{(x,y) \in X_{batch}} \nabla w L(\hat{y}, y)$
5. If validation error decreases, go to step 2, otherwise stop

The number of hidden layers and neurons per hidden layer are hyperparameters that can be selected to adjust the complexity of the model.

6.1.2 Optimization Challenges

While stochastic gradient descent (SGD) is one of the most common methods used to optimize deep learning networks. Recently however, a number of new optimizers have been proposed to tackle complex training scenarios where gradient descent methods behave poorly. One of the challenges of using SGD is inability for the algorithms to converge when using a fixed learning rate η, because of the noise that is added by using minibatches. SGD will always step over the optimum with a fixed learning rate, even when starting at the optimum (see Figure 10).

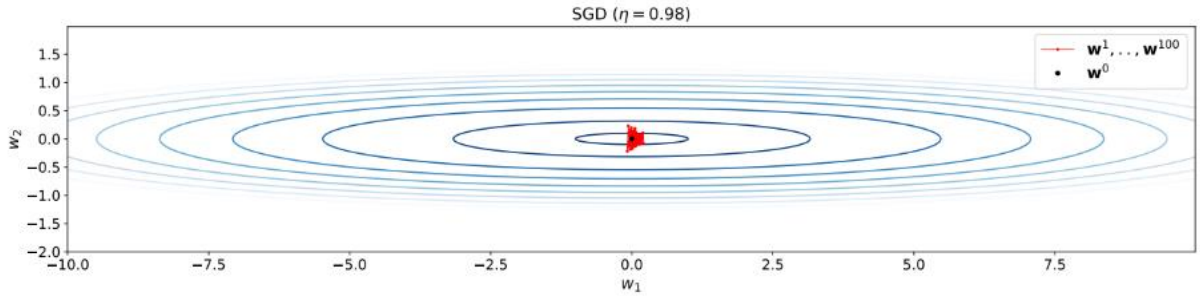


Figure 10: SGD not converging Good learning rate on the previous toy example shows non-convergence at optimum.

To help improve the update process of SGD, SGD with Momentum was introduced. By applying an exponential moving average value of gradients to the weight update, we intend to dampen the oscillation and accelerate the progress towards the optimum (see Figure 11). The velocity term m in the update equations:

$$m_{t+1} = \beta_1 m^t - (1 - \beta_1) \nabla w L_B(w^t)$$

$$w_{t+1} = w_t + m^{t+1}$$

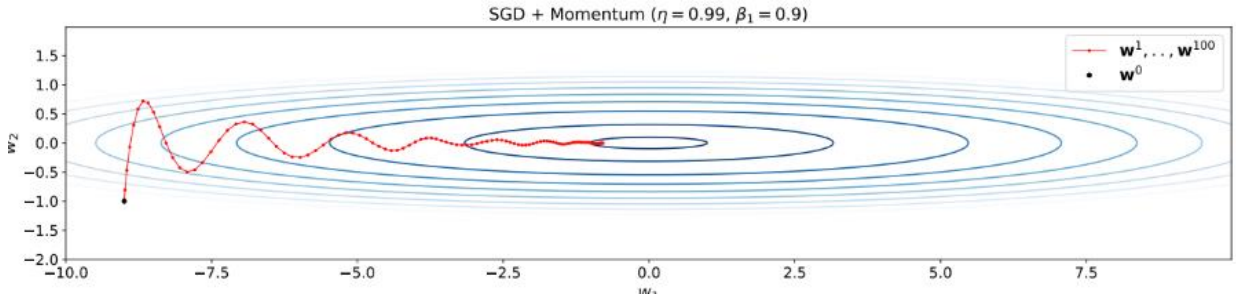


Figure 11: SGD not converging Good learning rate on the previous toy example shows non-convergence at optimum.

Another approach with the same motivation as momentum is RMSprop which stand for Root Mean Square Propagation. We want to have an even distribution of gradients on each weight dimension, in contrast to the standard SGD (see Fig. 12) with a very uneven gradient distribution. The idea of RMSprop is to divide the learning rate by a moving average

of squared gradients, which means that we change the learning rate per parameter. The moving average of squared gradients or the running variance v is used in this approach to adjust the per-weight step size.

$$v^{t+1} = \beta_2\, v^{\,t} - (1-\beta_1)(\nabla w L_B(w^t) \odot \nabla w L_B(w^t))$$

$$w_{t+1} = w_t - \eta \frac{\nabla w L_B(w^t)}{\sqrt{v^{t+1}} + \epsilon}$$

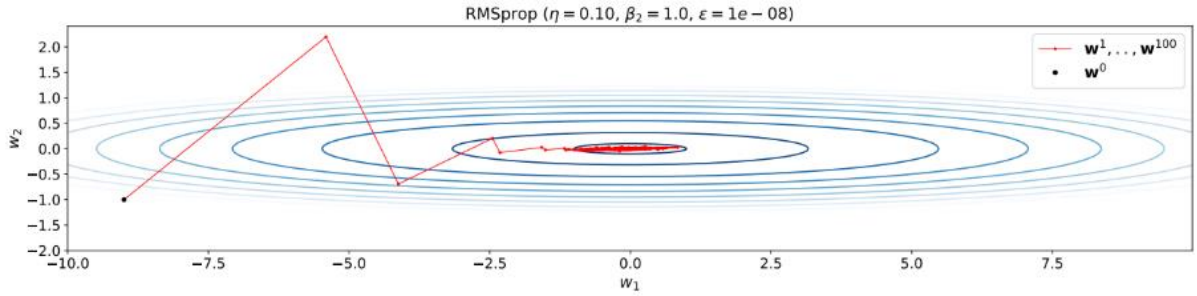


Figure 12: An example that shows the update behavior of RMSprop.

Leveraging the two previous approaches (see Figure 13), we end up with the most used and de facto default optimizer, *Adam,* due to its robustness. Given the update equation we can clearly see the influences of Momentum and RMSprop as both the first-moment velocity term $\boldsymbol{m}$ and the second-moment variance term $\boldsymbol{v}$ are used:

$$m^{t+1} = \beta_1\, m^{\,t} - (1-\beta_1)\nabla w L_B(w^t)$$

$$v^{t+1} = \beta_2\, v^t - (1-\beta_2)\,(\nabla w L_B(w^t) \odot \nabla w L_B(w^t))$$

$$w_{t+1} = w_t + m^{t+1}$$

$$\hat{m}^{\,t+1} = \frac{m^{t+1}}{1-\beta_1{}^{t+1}} \qquad \hat{v}^{\,t+1} = \frac{v^{t+1}}{1-\beta_2{}^{t+1}}$$

$$w_{t+1} = w_t - \eta \frac{\hat{m}^{\,t+1}}{\sqrt{v^{t+1}} + \epsilon}$$

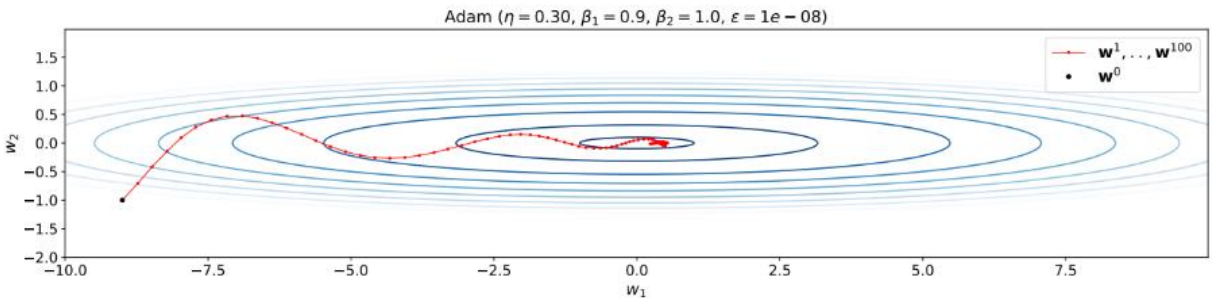


Figure 13: An example that shows the update behavior of Adam.

6.1.3 Convolutional Neural Networks

Convolutional networks, also known as convolutional neural networks, convnets or CNNs, are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels. Convolutional networks have been tremendously successful in practical applications. The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers [30].

6.1.3.1 Motivation

Convolution leverages three important ideas that can help improve a machine learning system: sparse interactions, parameter sharing and equivariant representations. Moreover, convolution provides a means for working with inputs of variable size. We now describe each of these ideas in turn [30]. Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means every output unit interacts with every input unit. Convolutional networks, however, typically have sparse interactions (also referred to as

sparse connectivity or sparse weights). This is accomplished by making the kernel smaller than the input (see Figure 14). For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output requires fewer operations.

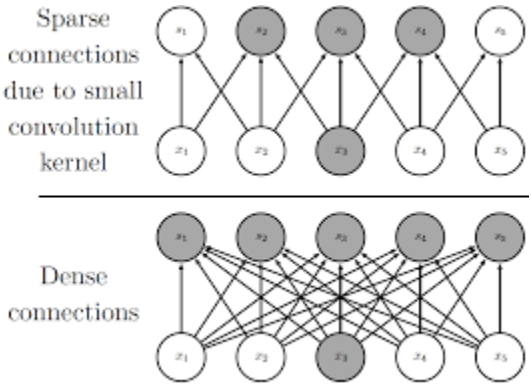


Figure 14: Sparse vs. Dense connectivity

Parameter sharing refers to using the same parameter for more than one function in a model (see Figure 15). In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. As a synonym for parameter sharing, one can say that a network has tied weights, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere. In a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather

than learning a separate set of parameters for every location, we learn only one set.

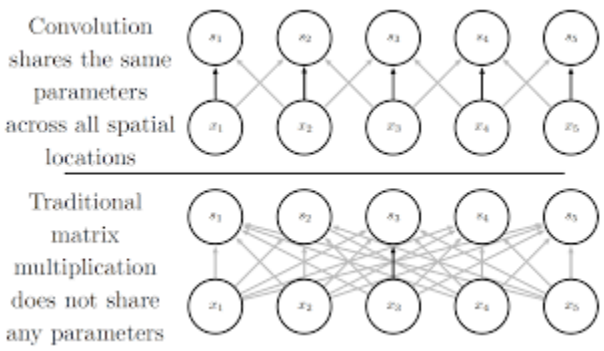


Figure 15: CNN vs. traditional NNs

In the case of convolution, the particular form of parameter sharing causes the layer to have a property called *equivariance* to translation. To say a function is equivariant means that if the input changes, the output changes in the same way. With images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output. This is useful for when we know that some function of a small number of neighboring pixels is useful when applied to multiple input locations. For example, when processing images, it is useful to detect edges in the first layer of a convolutional network. The same edges appear more or less everywhere in the image, so it is practical to share parameters across the entire image.

6.1.3.2 The Convolution Operation

In mathematics, the convolution between two functions, say $, g: R^d \rightarrow R$ is defined as:

$$(f * g)(x) = \int f(z)g(x - z)dz$$

That is, we measure the overlap between f and g when one function is "flipped" and shifted by x. Whenever we have discrete objects, the integral turns into a sum and we obtain the following definition:

$$(f * g)(i) = \sum_a f(a)g(i-a).$$

In convolutional network terminology, the first argument (function f) to the convolution is often referred to as the input and the second argument (function g) as the kernel. The output is sometimes referred to as the feature map.

In machine learning applications, the input is usually a multidimensional array of data and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. We will refer to these multidimensional arrays as tensors.

For two-dimensional tensors, we have a corresponding sum with indices (a, b) for f and $(i-a, j-b)$ for g respectively:

$$(f * g)(i,j) = \sum_a \sum_b f(a,b)g(i-a, j-b)$$

Convolution is commutative, meaning we can equivalently write:

$$(g * f)(i,j) = \sum_a \sum_b f(i-a, j-b)g(a,b)$$

Usually the latter formula is more straightforward to implement in a machine learning library, because there is less variation in the range of valid values of a and b.

Many neural network libraries implement a related function called the cross-correlation, which is the same as convolution but without flipping the kernel:

$$(f * g)(i,j) = \sum_a \sum_b f(i+a, j+b) g(a,b)$$

6.1.3.3 Convolution Layers

A convolutional layer cross-correlates the input and kernel and adds a scalar bias to produce an output. The two parameters of a convolutional layer are the kernel and the scalar bias. When training models based on convolutional layers, we typically initialize the kernels randomly, just as we would with a fully-connected layer [31].

Convolutional layer output is called a feature map (see Figure 16), as it can be regarded as the learned representations (features) in the spatial dimensions (e.g., width and height) to the subsequent layer. In CNNs, for any element of some layer, its receptive field refers to all the elements (from all the previous layers) that may affect the calculation of that element during the forward propagation [31].

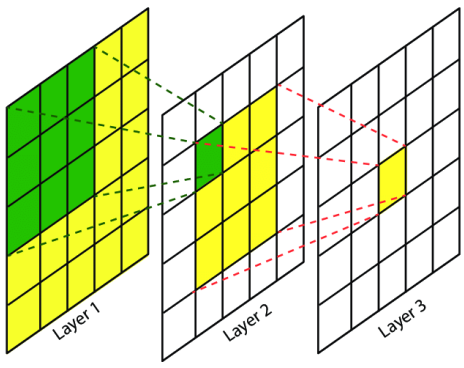


Figure 16: Feature mapping in CNNs

6.1.3.4 Padding:

The output shape of the convolutional layer is determined by the shape of the input and the shape of the convolution kernel. Assuming that the input shape $n_h \times n_w$and the convolution kernel shape $k_h \times k_w$, then the output shape will be $(n_w - k_w + 1)$.

One issue when applying convolutional layers is that we tend to lose pixels on the perimeter of our image. Since we typically use small kernels, for any given convolution, we might only lose a few pixels, but this can add up as we apply many successive convolutional layers. One straightforward solution to this problem is to add extra pixels of filler around the boundary of our input image, thus increasing the effective size of the image. Typically, we set the values of the extra pixels to zero (see Figure 17).

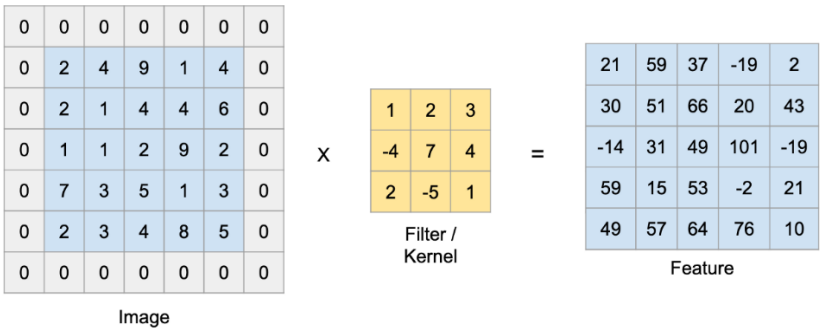


Figure 17: Padding with zero values

If we add a total of $p_h$rows of padding (roughly half on top and half on bottom) and a total of $p_w$columns of padding (roughly half on the left and half on the right), the output shape will be $⟦(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1)$. This means that the height and width of the output will increase by p_h and p_w, respectively.

In many cases, we will want to set $p_h = k_h - 1$ and $p_w = k_w - 1$ to give the input and output the same height and width. This will make it easier to predict the output shape of each layer when constructing the network.

CNNs commonly use convolution kernels with odd height and width values, such as 1, 3, 5, or 7. Choosing odd kernel sizes has the benefit that we can preserve the spatial dimensionality while padding with the same number of rows on top and bottom, and the same number of columns on left and right.

Moreover, this practice of using odd kernels and padding to precisely preserve dimensionality offers a clerical benefit. For any two-dimensional tensor X, when the kernel's size is odd and the number of padding rows and columns on all sides are the same, producing an output with the same height and width as the input, we know that the output $Y[i, j]$ is calculated by cross-correlation of the input and convolution kernel with the window centered on $X[i, j]$ [31].

6.1.3.5 Channels

In reality, images are not two-dimensional objects but rather third-order tensors, characterized by a height, width, and channel. Images consist of 3 channels: red, green, and blue. When the input data contain multiple channels, we need to construct a convolution kernel with the same number of input channels as the input data (see Figure 18), so that it can perform cross-correlation with the input data. Assuming that the number of channels for the input data is c_i, the number of input channels of the convolution kernel also needs to be c_i.

When is $c_i > 1$, we need a kernel that contains a tensor of shape $k_h \times k_w$ for *every* input channel. Concatenating these c_i tensors together yields

a convolution kernel of shape $k_h \times k_w \times c_i$. Since the input and convolution kernel each have c_i channels, we can perform a cross-correlation operation on the two-dimensional tensor of the input and the two-dimensional tensor of the convolution kernel for each channel, adding the c_i results together (summing over the channels) to yield a two-dimensional tensor.

Regardless of the number of input channels, we always end up with one output channel. To get an output with multiple channels, we can create a kernel tensor of shape $k_h \times k_w \times c_i$ for every output channel. Assume c_0 is the number of output channels, the shape of the convolution kernel is $k_h \times k_w \times c_i \times c_0$. In cross-correlation operations, the result on each output channel is calculated from the convolution kernel corresponding to that output channel and takes input from all channels in the input tensor.

It is very important to have multiple channels at each layer. In the most popular neural network architectures, we actually increase the channel dimension as we go higher up in the neural network, typically downsampling to trade off spatial resolution for greater channel depth. Intuitively, you could think of each channel as responding to some different set of features.

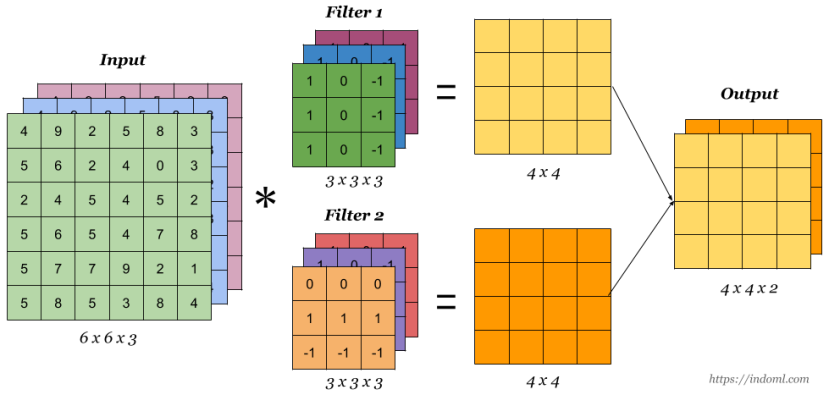


Figure 18: Example of convolution kernels with the same number of input channels as the input data.

6.1.3.6 Pooling

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the max pooling operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include the average of a rectangular neighborhood, the L^2 norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel.

In all cases, pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change [30].

Like convolutional layers, *pooling* operators consist of a fixed-shape window that is slid over all regions in the input according to its stride, computing a single output for each location traversed by the fixed-shape window (sometimes known as the *pooling window*). However, unlike the cross-correlation computation of the inputs and kernels in the convolutional layer, the pooling layer contains no parameters (there is no *kernel*). Instead, pooling operators are deterministic (see Figure 19), typically calculating either the maximum or the average value of the elements in the pooling window.

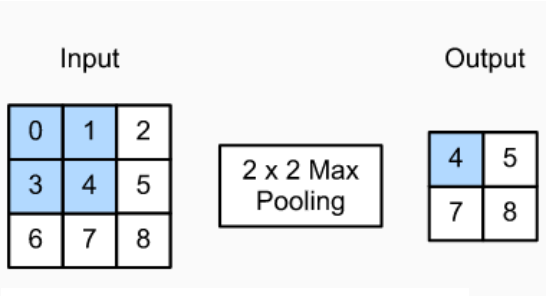


Figure 19: Example of pooling.

As with the cross-correlation operator, we can think of the pooling window as starting from the upper-left of the input tensor and sliding across the input tensor from left to right and top to bottom. At each location that the pooling window hits, it computes the maximum or average value of the input sub-tensor in the window, depending on whether max or average pooling is employed.

6.1.3.7 CNN Architectures: VGG16

Typical CNN architectures (see Figure 20) stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on. The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper (i.e., with more feature maps) thanks to the convolutional layers. At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities) [32].

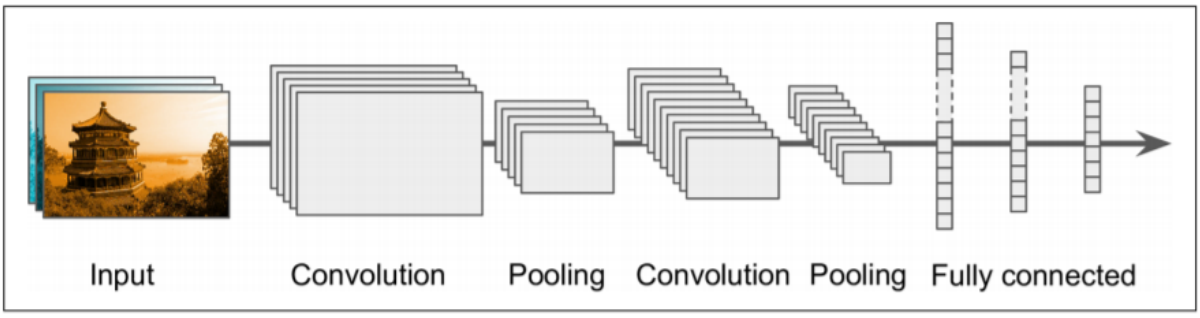


Figure 20: Typical CNN architecture.

In this section we will talk in detail about a specific popular CNN architecture which is the VGG16 model. **We will be using a pretrained VGG16 network to extract features from images in our VQA dataset. Moving forward these extracted features will represent the visual part in our proposed model**. We will explain what we mean by a pretrained network and feature extraction in the next section, but first let's take a look at the VGG16 network architecture.

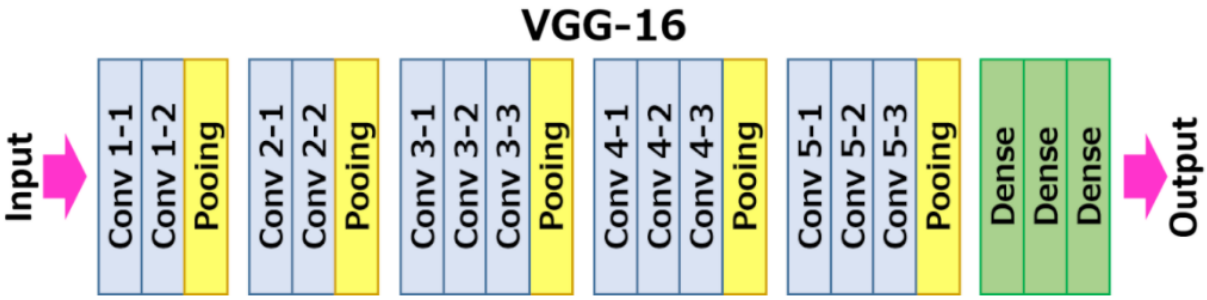


Figure 21: VGG-16 architecture.

The input to cov1 layer is of fixed size **224 x 224 RGB** image. The image is passed through a stack of convolutional (conv.) layers, where the filters were used with a very small receptive field: 3×3. The convolution stride is fixed to 1 pixel; the spatial padding of conv. layer input is such that the spatial resolution is preserved after convolution, i.e. the padding is 1-pixel for **3×3** conv. layers. Spatial pooling is carried out by five max-pooling layers, which follow some of the conv. layers (not all the conv. layers are followed by max-pooling). Max-pooling is performed over a **2×2** pixel window, with stride **2**. All hidden layers are equipped with the rectification (ReLU) non-linearity.

Three Fully-Connected (FC) layers follow a stack of convolutional layers (which has a different depth in different architectures): the first two have **4096** channels each, the third contains **1000** channels (one for each class). The final layer is the **soft-max** layer.

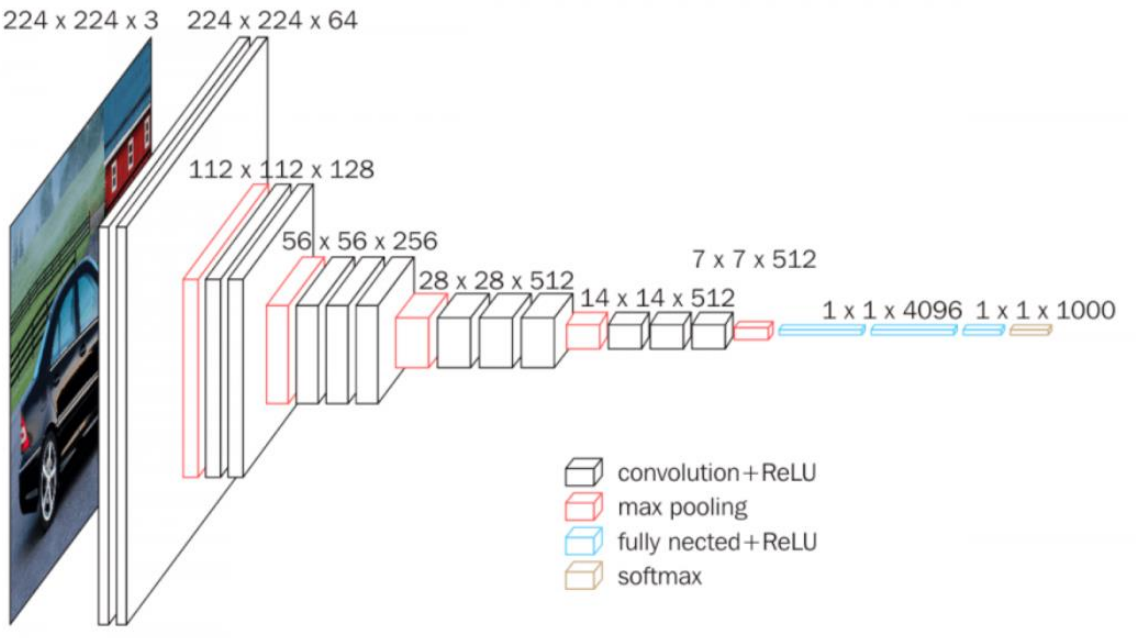


Figure 22: VGG-16 architecture.

VGG16 was trained on the ImageNet dataset. ImageNet is a dataset of over 15 million labeled high-resolution images belonging to roughly 22,000 categories. The images were collected from the web and labeled by human labelers using Amazon's Mechanical Turk crowd-sourcing tool. Starting in 2010, as part of the Pascal Visual Object Challenge, an annual competition called the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) has been held. ILSVRC uses a subset of ImageNet with roughly 1000 images in each of 1000 categories. At all, there are roughly 1.2 million training images, 50,000 validation images, and 150,000 testing images. ImageNet consists of variable-resolution images. Therefore, the images have been down-sampled to a fixed resolution of 256×256. Given a rectangular image, the image is rescaled and cropped out the central 256×256 patch from the resulting image [33].

6.1.3.8 Using a Pretrained Convnet

A pretrained network is a saved network that was previously trained on a large dataset, typically on a large-scale image-classification task. If this

original dataset is large enough and general enough, then the spatial hierarchy of features learned by the pretrained network can effectively act as a generic model of the visual world, and hence its features can prove useful for many different computer-vision problems, even though these new problems may involve completely different classes than those of the original task. There are two ways to use a pretrained network: feature extraction and fine-tuning. Feature extraction consists of using the representations learned by a previous network to extract interesting features from new samples. These features are then run through a new classifier, which is trained from scratch. The feature maps of a convnets are presence maps of generic concepts over a picture, which is likely to be useful regardless of the computer-vision problem at hand. The level of generality (and therefore reusability) of the representations extracted by specific convolution layers depends on the depth of the layer in the model. Layers that come earlier in the model extract local, highly generic feature maps (such as visual edges, colors, and textures), whereas layers that are higher up extract more-abstract concepts.

Fine-tuning consists of unfreezing a few of the top layers of a frozen model base used for feature extraction, and jointly training both the newly added part of the model and these top layers. This is called fine-tuning because it slightly adjusts the more abstract representations of the model being reused, in order to make them more relevant for the problem at hand [33].

7.1.4 Recurrent Neural Network

Recurrent neural networks (RNNs) are a family of artificial neural networks designed to process sequences of data, which makes them very useful when dealing with text.

Although a regular feedforward network can deal with short sequences, and with pretty long sequences, convolution neural networks can work quite well too; recurrent networks can scale up to much longer sequences than would be practical for other networks. They can also work with sequences of arbitrary lengths, rather than fixed-sized inputs like other networks.

6.1.4.1 Motivation

What makes recurrent neural networks specialized when dealing with text is that they take advantage of one of the early ideas found in machine learning and statistical models of the 1980s: sharing parameters across different parts of a model. Parameter sharing makes it possible to extend and apply the model to examples of different lengths and generalize across them. Such sharing is particularly important when a specific piece of information can occur at multiple positions within the sequence [30].

A recurrent network is very similar to a feedforward neural network, except that it has connections pointing backward or in other words, it feeds information from one node back to itself as shown in Figure 23 (left). At each time step t, the recurrent cell receives the inputs $x_{(t)}$ as well as the hidden state of the previous time step $h_{(t-1)}$ and produces a new hidden state $h_{(t)} = f_h\left(h_{(t-1)}, x_{(t)}\right)$, that hidden state will be fed back to the model at the next time step and depending on the hidden state, the cell produces an output $y_{(t)} = f_y(h_{(t)})$. We can unroll this network through time as shown in Figure 23 (right).

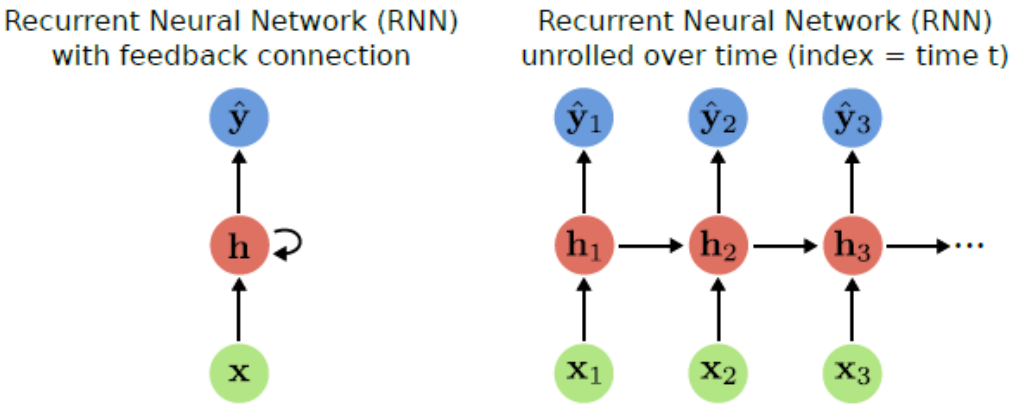


Figure 23: Recurrent neural network architecture.

6.1.4.2 Types of recurrent neural networks

Unlike feedforward neural networks where one input is mapped to one output, RNNs can process inputs and outputs of variable length, they can be categorized into four types based on the number of inputs and outputs (see Figure 23), each of which has its own set of applications:

- One to many: It is suitable for applications like image captioning.
- Many to one: it can be useful for sentiment classification or action recognition.
- Many to many (encoder-decoder): used in machine translation, where the encoder network produces a summary vector of the input, which is fed to the decoder network that produces a variable-length output based on that vector. In this architecture, the number of outputs can be different from the number of inputs.
- Many to many: used in video object tracking, where the model produces an output for each frame of the video. In this architecture, the number of outputs is equal to the number of inputs of the network

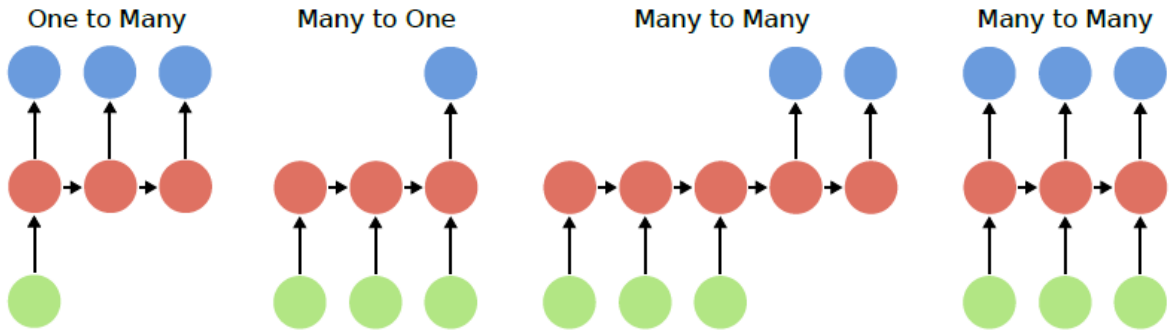
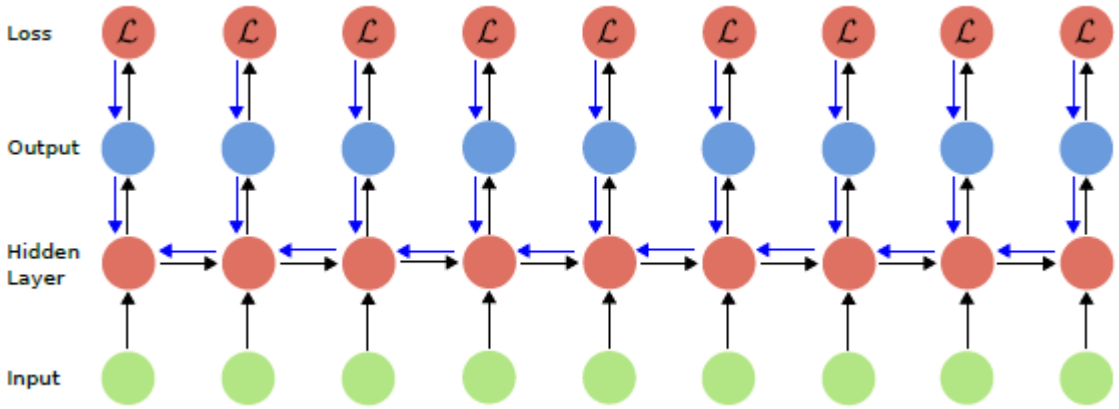


Figure 23: RNN types

6.1.4.3 Training RNNs

RNNs return an output at multiple times-steps with the number of time-steps not being fixed. At the same time, each output generates a loss. Thus, each of the losses at a time step needs to be taken into account when back-propagating. The first step is to unroll the graph through time and then the gradient derived from each loss at a time step is back-propagated to the previous time-step. There, it is added to the gradient of that time-step and so on. This process is repeated until the initial time-step is reached. Here, the weight parameters are updated. All of the hidden RNN cells share their parameters; hence, the gradients are accumulated. This approach is known as backpropagation through time (BPTT) (see Figure 24).



6.1.4.4 Multi-layer RNNs:

In a similar manner to feedforward neural networks, RNNs can be built with multiple layers. Each of these layers has its own set of weights. RNNs are often kept very shallow in practice, i.e., only a couple of layers deep. Multi-layer RNNs are constructed by adding a second cell/hidden layer after the first one, where the input goes in and produces a sequence of hidden states from the first RNN layer, which is then fed as input to the second RNN layer(see Figure 25).

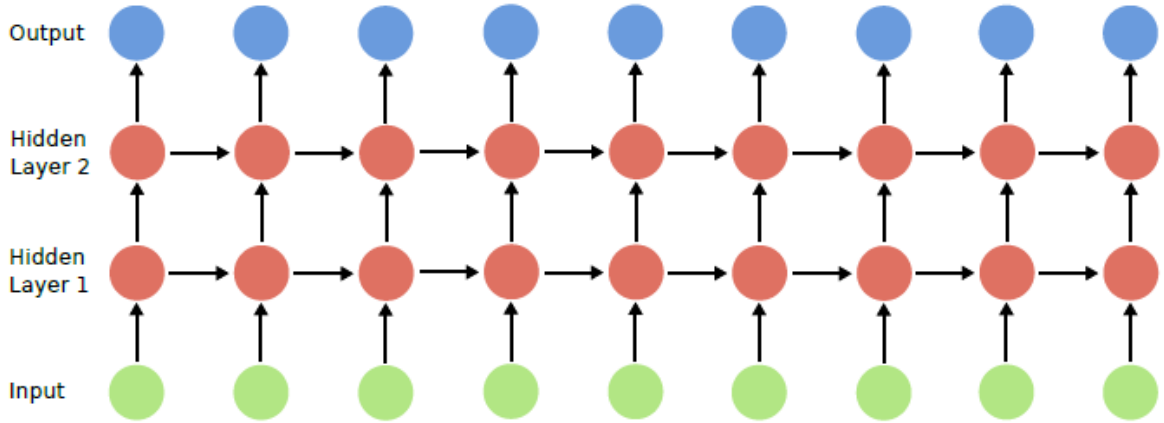


Figure 25: Multi-layer RNN with two layers.

6.1.4.5 Memory cells

Since the output of a recurrent cell at timestep t is a function of all the inputs from previous time steps, we can say it has a form of memory. The part of a neural network that preserves some state across time steps is called a memory cell (or simply a cell) [32].

1- Basic cell:

A single recurrent neuron or a layer of recurrent neurons is a basic cell. It is capable of learning only short patterns, typically about 10 steps long.

The cell's hidden state at time step t denoted $h_{(t)}$ is a function of some

input $x_{(t)}$ at that time step and the state of the previous time step $h_{(t-1)}$. In basic cells, the output is simply equal to the state.

$$h_{(t)} = \phi(W_{hh}h_{(t-1)} + W_{xh}x_{(t)} + b_h)$$

$$y_{(t)} = h_{(t)}$$

It turns out that the basic RNN is not very good at capturing long-term dependencies. Due to the vanishing gradient problem and the transformations that data goes through when traversing the RNN, some information is lost at each time step. Therefore after a while, the RNN's state contains virtually no trace of the first inputs. This can cause problems when trying to learn patterns from a long sequence of data. To overcome this problem, various types of cells with long-term memory have been introduced; one of them is the LSTM cell.

2- LSTM cell:

Long Short-Term Memory (LSTM) cell is a memory cell architecture that has a better gradient flow than the basic cell and it is capable of learning long-term dependencies (see Figure 26) [34].

It has two state vectors at each time step: the hidden state $h_{(t)}$, which we can think of as the short-term state and the cell state $c_{(t)}$ which is a long-term state. The key idea of that cell is that it can learn what to store in the long-term state, what to forget, and what to read from it. The LSTM cell takes the current input vector $x_{(t)}$and the previous short-term state $h_{(t-1)}$, and are fed to four different fully connected layers:

The main layer is the one that outputs $g_{(t)}$. It has the role of analyzing the current inputs $x_{(t)}$ and the previous short-term state $h_{(t-1)}$ and then the cell determines the most important parts to be stored in the long-term state and the rest is dropped.
The three other layers are gate controllers. Since they use the logistic activation function, their outputs are in the range from 0 to 1 and then

their outputs are fed to element-wise multiplication operations where they produce 0s to close the gate and produce 1s to open it, These gates are:

- The forget gate $f_{(t)}$: it controls which parts of the long-term state should be erased.
- The input gate $i_{(t)}$: it controls which parts of $g_{(t)}$ should be added to the long-term state.
- The output gate $o_{(t)}$: it controls what output and next short-term state to generate from the current internal cell state.

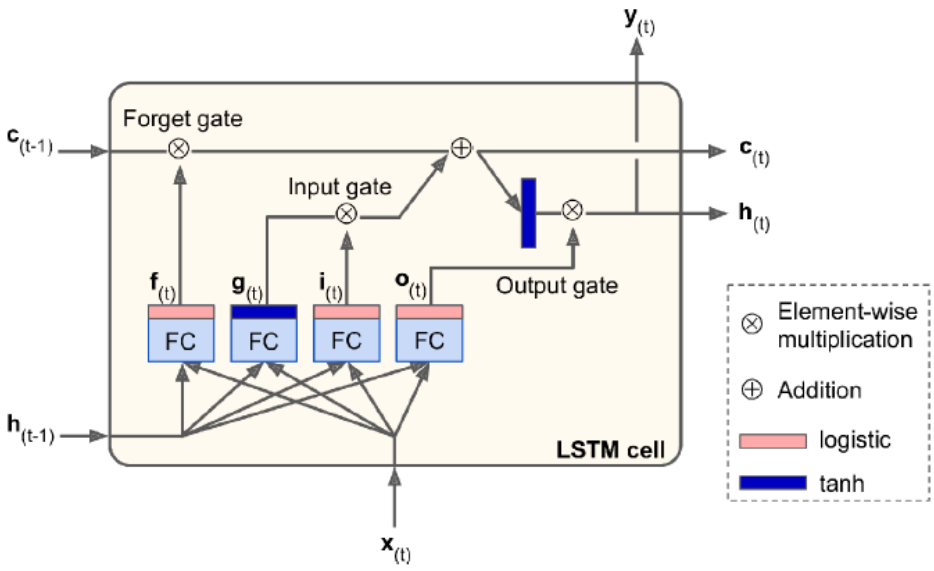


Figure 26: LSTM cell

$$g_{(t)} = \tanh(W_{xg}x_{(t)} + W_{hg}h_{(t-1)} + b_g$$

$$i_{(t)} = \sigma(W_{xi}x_{(t)} + W_{hi}\,h_{(t-1)} + b_i$$

$$f_{(t)} = \sigma(W_{xf}x_{(t)} + W_{hf}h_{(t-1)} + b_f$$

$$o_{(t)} = \sigma(W_{xo}x_{(t)} + W_{ho}h_{(t-1)} + b_o$$

$$c_{(t)} = f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)}$$

$$y_{(t)} = h_{(t)} = o_{(t)} \otimes \tanh(c_{(t)})$$

The key idea of LSTM is the cell state, the horizontal line running through between recurrent time-steps. We can imagine the cell state $c_{(t)}$ to be some kind of highway of information passing through straight down the entire chain, with only some minor linear interactions. Thus, even when there is a bunch of LSTMs stacked together, we can get an uninterrupted gradient flow where the gradients flow back through cell states instead of hidden states $h_{(t)}$ without vanishing in every time step.

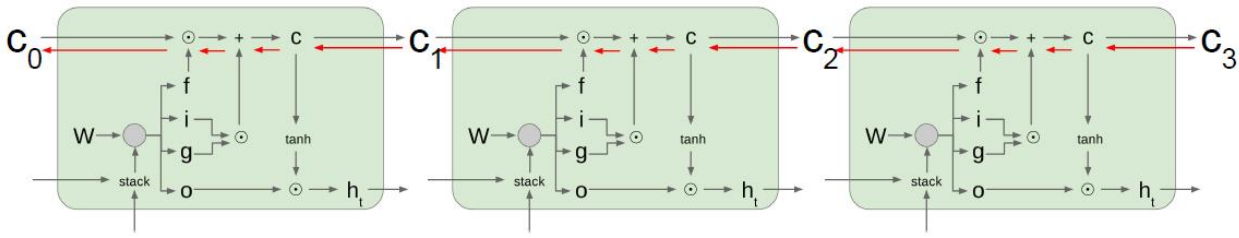


Figure 27: Stack of LSTM cells.

6.1.3.6 Word Representations

Machine learning algorithms cannot work with raw text directly; the text must be converted into numbers. The simplest way to represent words numerically is to represent each word in a sentence by a one-hot-encoded vector, which is a fixed-length vector that has 1 at the index of the word in the vocabulary and 0s everywhere else, but this method has several disadvantages:

- **Scalability issue:** As the number of words in our vocabulary increases, the dimensions of our one-hot-encoded vectors for each word will explode. This will lead to inefficiency in time and computational resources.

- **Sparsity issue**: Since the representation of the word will have 0s everywhere except for a single 1 at the correct location of the word. This will give the model a hard time learning the data.

- **No context is captured:** Since the one-hot encoding blindly creates vectors without taking into account the position and the context of the word in the sentence, the representation loses the contextual and semantic information.

We can instead use bag of words (BOW) representation, which represents a sentence by a fixed-length vector that holds the count of occurrences for each word in the vocabulary. This approach has similar disadvantages to those of the one-hot-encoding approach.

A better representation is word embeddings. Word Embeddings are a numerical vector representation of the text in the corpus that maps each word in the vocabulary to a set of real-valued vectors in a pre-defined N-dimensional space.

The Word Embeddings try to capture the semantic, contextual, and syntactic meaning of each word in the corpus vocabulary based on the usage of these words in sentences. Words that have similar semantic and contextual meaning also have similar vector representations while at the same time each word in the vocabulary will have a unique set of vector representations. An important aspect of these representations is the ability to solve word analogies of the form "A is to B what C is to X" using simple arithmetic. This is generally simplified as "King - Man + Woman = Queen".

The word embeddings representation maps each word to N-Dimensional space where N is pre-defined so it solves the scalability issue (see Figure 28). Each embedding vector is densely populated and that solves the

sparsity issue. These vectors are also learned in a way that captures the shared context and dependencies among the words. Therefore, word embeddings representation overcomes all the issues of BOW representation.

Instead of training word embeddings on our own using a limited corpus of text, a better approach is to reuse pre-trained word embeddings since they are trained on larger datasets, which makes them capture the semantic and syntactic meaning of words in a better way, which can boost the performance of the model. This is a form of transfer learning where a model developed for a task is reused as the starting point for a model on a second task.

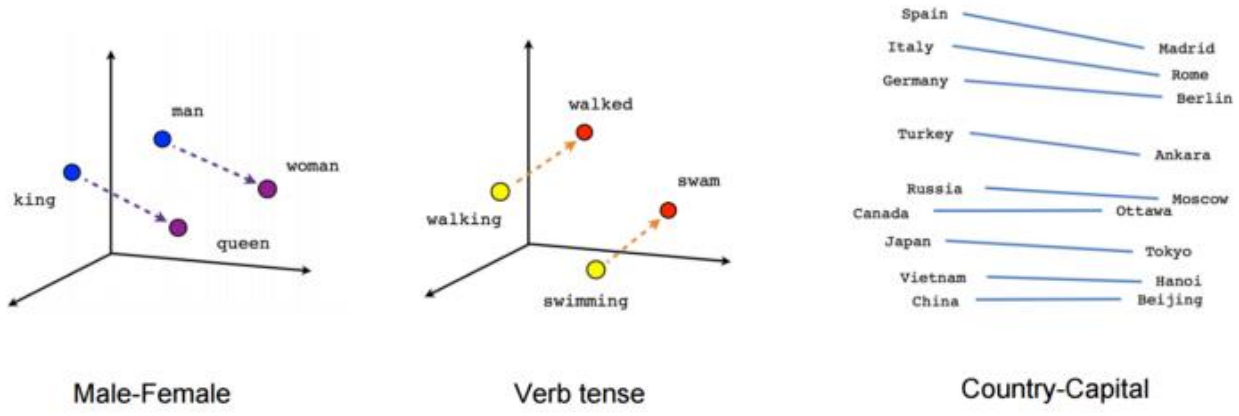


Figure 28: Word Embeddings.

GloVe word embeddings:

Global Vectors for word representation or GloVe [35] for short was introduced by Jeffrey Pennington, Richard Socher, Christopher D. Manning of Stanford University in the year 2014. It is an unsupervised learning algorithm for obtaining vector representation of words. Its goal

is to derive semantic relationships between words from their co-occurrence matrix.

During the training process, the model samples a pair of words i, j that appear close to each other in the text corpus.
It then computes $X_{i,j}$ which is the number of times i appears in the context of j.

The goal of the model is to minimize the following loss function:

$$\sum_{i=1}\sum_{j=1} f(X_{i,j})[e_i^T e_j + b_i - b_j - \log(X_{i,j})]^2$$

where e_i, e_j are embeddings of the words, b_i, b_j are biases and $f(X_{i,j})$ can be considered as a weighting function that gives less weight to the frequent stop words such as (the, is, a, …, etc) and it should vanish as $X_{i,j} \to 0$ to keep the value of the loss function finite. This loss function tries to bring words that co-occur together closer to each other.
The model uses the gradient descent technique to update the embeddings of the words at each step.

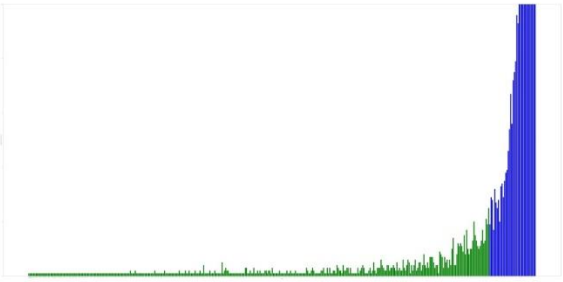
Ensemble Models

To combine the two input features which hold the visual representation of the image and the corresponding question to that image (after applying word embedding + LSTM). We decided to combine the two vectors using element-wise multiplication according to the accuracy result shown in this paper [1. reference]. The difference performance for two methods (concatenation compared to element-wise multiplication) was not that high for multiple-choice tasks but, element-wise multiplication performs better by1.24%.

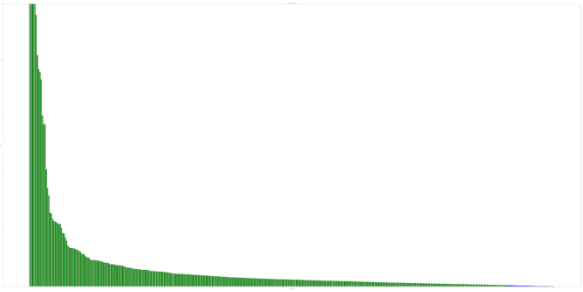
We made a vocabulary of 1000 (999 valid answer + < UNK >) words. To distinguish each word in our vocabulary from every other word we

used a one-hot vector technique to represent the answer. The one-hot vector consists of 0's in all cells except the cell that identifies the correct answer (has the value of 1). The top answers represent the most frequent answers that are likely to occur, using train and validation dataset to construct it. Where the train samples = 4,437,570 and the validation samples = 2,143,540. We consider only the most frequent ground truth answers so, we consider 443,757 and 214,354 answers for train and validation, respectively. After filtering it we got 439,661 answers for the train and 212,019 for validation with 26,480 unique answers. The top 999 answers are the answers that have word frequency of nearly 39; they are almost 574,814 answers so that they roughly represent 87.34% of our train/val dataset. Because of the huge number of unique words in the answers we tried to graph them in terms of frequency occurrence of particular words and if two or more words have the same frequency, we count it to this frequency. Figure 1: show the histogram distribution of frequencies, the x-axis represents the frequency component and y-axis represent the number of words in this frequency, the highest frequency shown from right to left in green and blue color, where green color means we consider this frequency in our answer's vocabulary. We can observe that the blue bars have a huge number of words which occur in low frequency. For example, the last sample has frequency of 1 and its y-axis has the value of 15,997 meaning that we have 15,997 words each one of them appeared for only one time.

The following chart shows Frequency vs Number of Words (according to frequency ordering):



Here, Number of Words vs Frequency (according to frequency ordering):



Softmax Layer

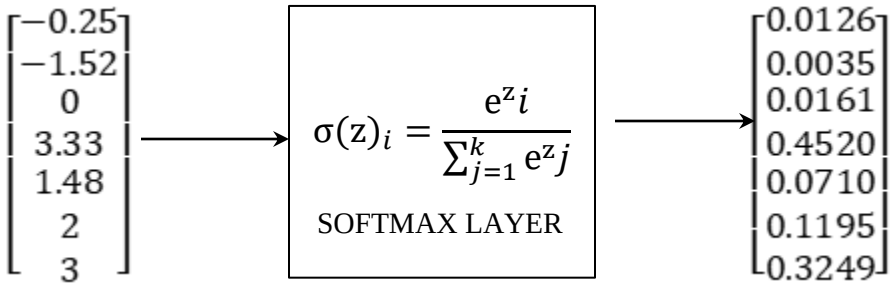
The softmax function, also known as normalized exponential function, is a generalization of the logistic function to multiple dimensions. It is often used in the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes.

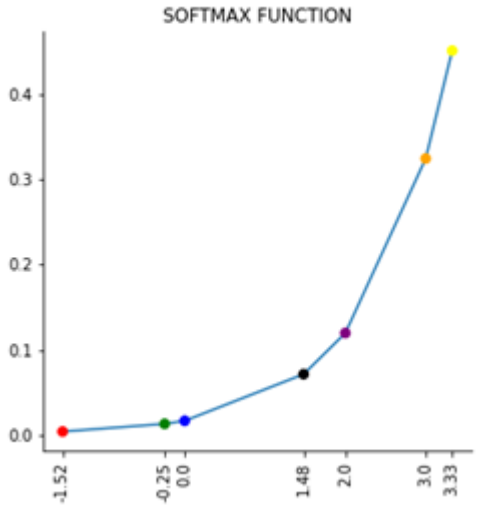
The softmax function takes as input a vector z of K real numbers and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers. That is, prior to applying softmax, some vector components could be negative, or greater than one; and might not sum to 1; but after applying softmax, each component will be in the interval [0,1] and the components will add up to 1, so that they can be interpreted as probabilities. Furthermore, the larger input components will correspond to larger probabilities.

The standard softmax $\boldsymbol{\sigma}$: $\mathbb{R}^K$ ➔ $[0, 1]^K$

$$\sigma(z)_i = \frac{e^z i}{\sum_{j=1}^{k} e^z j}$$

In simple words, it applies the standard exponential function to each element zi of the input vector **z** and normalizes these values by dividing by the sum of all these exponentials; this normalization ensures that the sum of the components of the output vector σ(z)i is 1





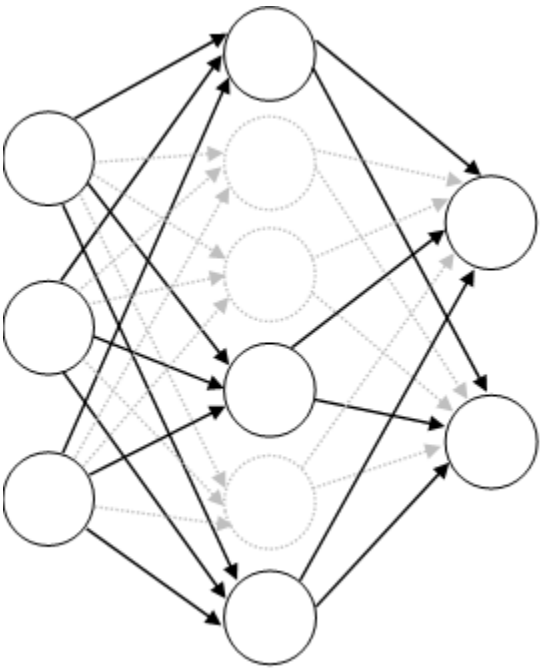
Dropout layer

The term "dropout" refers to dropping out units (both hidden and visible) in a neural network. In other words, dropout refers to ignoring (shutting down) neurons during the training phase. The criteria of choosing which neuron to be ignored is random by putting the dropout hyperparameter, which describes the probability of dropping each neuron in that layer. Technically, at each training stage, individual neurons are either dropped out of the net with probability *1-p* or kept with probability *p*, so that a reduced network is left. The incoming and outgoing edges for the dropped-out neuron are also removed. Dropping neurons is a useful technique for regularizing the model to overcome the over fitting problem. Regularization reduces overfitting by adding a

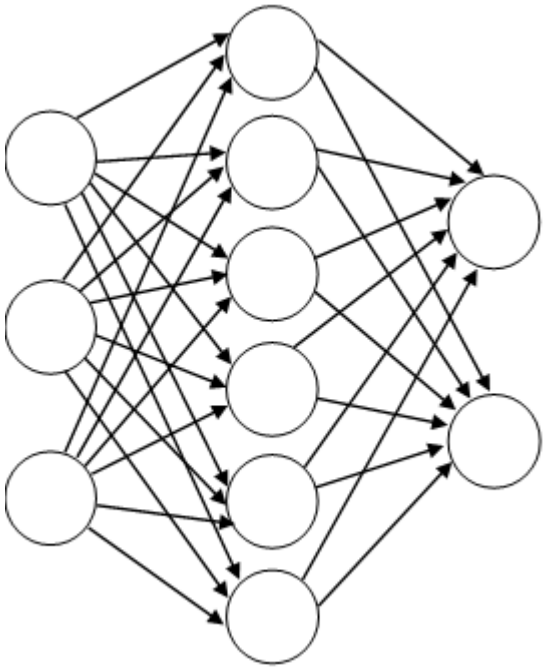
penalty to the loss function. By adding this penalty, the model is trained such that it does not learn an interdependent set of features weights. Dropout used to reduce interdependent learning amongst the neurons. Dropout forces a neural network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons. Dropout roughly doubles the number of iterations required to converge. However, training time for each epoch is less.

We have two different phases:

1. In the training phase each hidden layer, for each training sample, for each iteration, ignores (zero out) a random fraction -*p*- of nodes (and corresponding activations).
2. In the testing phase use all activations but reduce them by a factor p *(1-p)* (to account for the missing activations during training).



DROPOUT NEURAL NETWORK



STANDARD NEURAL NETWORK

6.2 VQA Model Architecture

Our model is based on artificial neural networks and consists of three parts:

(1)**Image encoder:** it takes the pre-processed image features extracted using *VGG-16* convolution neural network. These features were stored with the shape of (49, 512). The model flattens the image features and then feeds them to a fully connected layer with 1024 neurons and uses the $RELU$ activation function. This part outputs a 1024-dim embedding of the image.

(2)**Question encoder:** it takes a padded tensor of the vocabulary indices for each word in the question sentence. This tensor has the length of max_qu_length = 30. It uses an embedding layer initialized using pre-trained GloVe-300 word embeddings to replace each word in the sentence with its representative 300-dim vector. The word embeddings are then fed into a recurrent neural network with $LSTM$ cells that has a hidden state dimension of 1024. The hidden state of the last $LSTM$ cell provides a 1024-dim embedding of the question.

(3) **Answer predictor:** in this part, the image embedding and the question embedding are fused together using element-wise multiplication. The results of the multiplication are then fed into a drop-layer with a drop rate of 20%, which helps to prevent overfitting, and then into a fully connected layer of $K = 1000$ neurons and it uses *softmax* activation function to provide a probability distribution over K answers.

```
#image model
im_input = tf.keras.layers.Input(shape=(49, 512))
x1 = tf.keras.layers.Flatten()(im_input)
x1 = tf.keras.layers.Dense(1024, activation='relu')(x1)

#question model
vocab_size = question_vocab.vocab_size
q_input = tf.keras.layers.Input(shape=max_qu_length)
x2 = tf.keras.layers.Embedding(input_dim=vocab_size,
                               output_dim=300,
                               input_length=max_qu_length,
                               embeddings_initializer=tf.keras.initializers.Constant(embedding_matrix),
                               trainable=True)(q_input)
x2 = tf.keras.layers.LSTM(1024)(x2)

#model output
out = tf.keras.layers.Multiply()([x1, x2])
out = tf.keras.layers.Dropout(0.2)(out)
num_answers = answer_vocab.vocab_size
out = tf.keras.layers.Dense(num_answers, activation='softmax')(out)
```

Figure 29: VQA model code

The model was trained using ADAM optimizer with a learning rate of 0.001 on the categorical cross-entropy loss.

```
model = tf.keras.Model(inputs=[im_input, q_input], outputs=[out])
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
               metrics=['accuracy'])
```

Figure 30: Model optimizer and Loss function

After training the model on the training dataset, we merge the pre-trained VGG-16 network with the image part of our trained model to create the full model. The full model takes in two inputs:

(1) Pre-processed image of the shape (224, 224, 3).

(2) Pre-processed question tensor, which contains the indices of each word in the question padded to the length of 30.

The full model has one output, which is a vector that contains the probability distribution over the K=1000 answers.

```
img_model = tf.keras.applications.vgg16.VGG16(weights="imagenet", include_top=False)
vqa_model = tf.keras.models.load_model('/content/drive/MyDrive/checkpoints_features/checkpoint-10.h5')
```

```
img_input = tf.keras.layers.Input(shape=(224,224,3,))
image_features = img_model(img_input)
image_features_reshpaed = tf.keras.layers.Reshape((49, 512, ))(image_features)

question_input = tf.keras.layers.Input(shape=(30,))
output = vqa_model([image_features_reshpaed, question_input])
```

```
full_model = tf.keras.models.Model(inputs=[img_input, question_input], outputs=[output])
```

Figure 31: Combining VGG-16 with the rest of the model

The full model architecture is shown in the next figure:

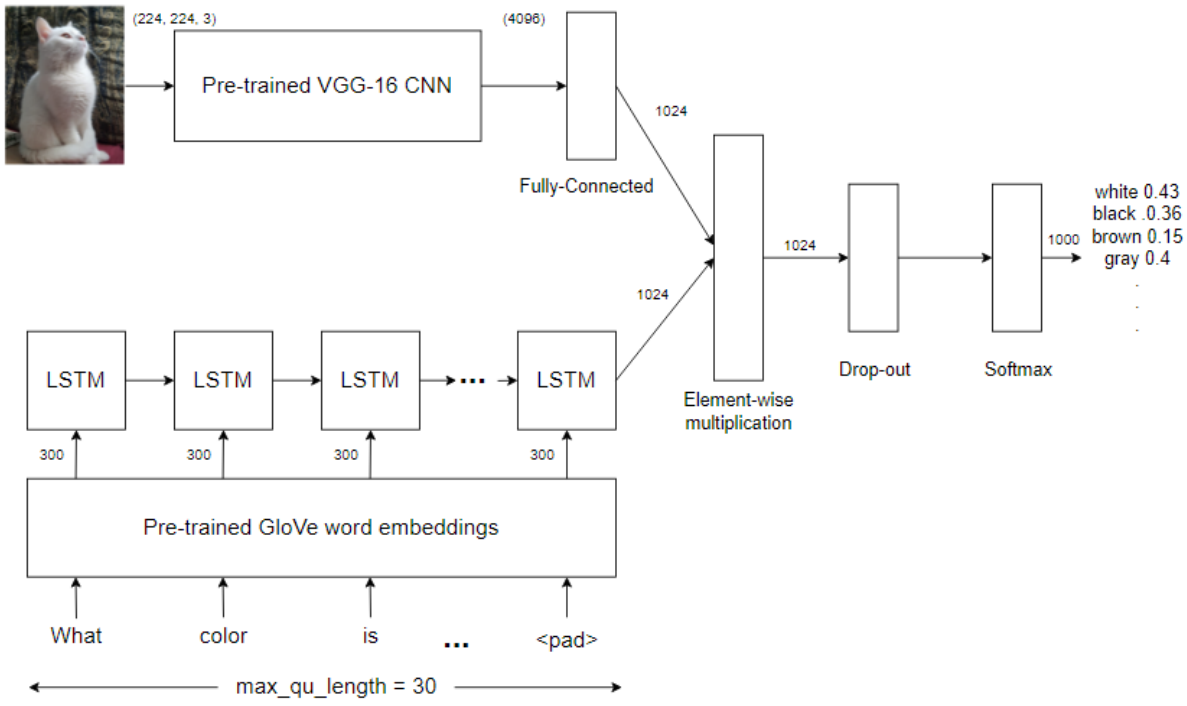


Figure 32: Full model architecture

The full model is later saved and converted to the tflite format to be able to run on the Android application.

6.3 Data Preproccesing

In this section we deal with our data to be preprocessed to feed it into our model, this chapter and the mentioned code inspired by [] but slightly different, data preprocess phase can be split into three main procedures:

1. Making our answer vocabulary
2. Making our question vocabulary
3. Preprocessing our images dataset
4. Extract the necessary information from the JSON files and save it as NumPy list.

Considering the size of the dataset and the need for high hardware specs, we decided to use google colab to preprocess and inspect our datasets. Its free, very high-performance specs for GPU runtime environment (Tesla T4 and Tesla K80) and off course high speed connectivity. The downside here is there is no guarantee that the session would be last or did not terminate after a certain time for both GPU runtime which about (6 ~ 7) hours or without GPU would last further for almost (~ 12) hours.

6.3.1 Answer Vocabulary

As we illustrated previously in the model description section, we build our answer vocabulary from top 1000 occurrence answers in the train and validation dataset. According to that we go through these steps:

1. Download annotations dataset json files, there are two files one for training which contains 4,437,570 answers and the other for validation with 2,143,540 answers, 10 answers for each question in the dataset. Answers in the form of a sentence of words meaning that each answer can be one word or more than one. We consider only the most frequent ground-truth answer labeled *'multiple_choice_answer'*.

"Snippet for annotations scheme"

```
{
  'question_type': 'what is this',
  'multiple_choice_answer': 'net',
  'answers':
  [
  {'answer': 'net', 'answer_confidence': 'maybe', 'answer_id': 1},
  {'answer': 'net', 'answer_confidence': 'yes', 'answer_id': 2},
  {'answer': 'net', 'answer_confidence': 'yes', 'answer_id': 3},
  {'answer': 'netting', 'answer_confidence': 'yes', 'answer_id': 4},
  {'answer': 'net', 'answer_confidence': 'yes', 'answer_id': 5},
  {'answer': 'net', 'answer_confidence': 'yes', 'answer_id': 6},
  {'answer': 'mesh', 'answer_confidence': 'maybe', 'answer_id': 7},
  {'answer': 'net', 'answer_confidence': 'yes', 'answer_id': 8},
  {'answer': 'net', 'answer_confidence': 'yes', 'answer_id': 9},
  {'answer': 'net', 'answer_confidence': 'yes', 'answer_id': 10}
  ],
  'image_id': 458752,
  'answer_type': 'other',
  'question_id': 458752000
}
```

2. For each answer we convert it to lowercase form then filter each answer to ignore anyone containing a non-alphanumeric character from our vocabulary. For simplicity we did not do any word stemming, lemmatization or stop keywords. *"Snippet for the ignored answers"*

   ```
   white, blue
   ...
   !
   she's happy
   what is this fruit?
   $5
   2223, 305
   ```

3. Dictionary data type to hold our answer frequency, where the keys are the answers, and the values are their frequency. If the answer

was in our dictionary keys, we count it by increasing its value. If not, we add it to our dictionary keys and initialize its value by 1. As a result we got 26,480 unique answers. *"Snippet for answers dictionary"*

```
(boy, 364)
(train, 777)
(shadow, 153)
(platform, 32)
(tabby, 147)
(monitor, 39)
(clock tower, 77)
(2010, 43)
(blue, 8188)
(cloudy, 530)
(gray, 3213)
(big ben, 68)
(clock, 317)
(for shade, 6)
(dirt, 399)
```

4. Sorting our dictionary by its values in descending order, we got our most occurrence answers first.

```
['<unk>', 'yes', 'no', '1', '2', 'white', '3', 'blue', 'red',
'black', '0', '4', 'brown', 'green', 'yellow', '5', 'gray',
'right', 'nothing', 'frisbee', 'left', 'baseball', 'tennis',
'6', 'orange', 'none', 'wood', 'pizza', 'bathroom', 'pink',
'kitchen', '10', '8', 'cat', '7', 'man', 'water', 'grass',
'dog', 'skiing', 'skateboarding', 'black and white']
```

5. Finally, we add the "<UNK>" answer at the top of the dictionary and save the first 1000 in the "txt" file to use it in different phases. *"Code snippet"*

```
def make_a_vocab(top_answer):

  answers = defaultdict(lambda :0)
  dataset = os.listdir(src_dir + '/Annotations')

  for file in dataset:
    path = os.path.join(src_dir, 'Annotations', file)

    try:
      with open(path, 'r') as f:
        data = json.load(f)
    except(IOError, SyntaxError):
      break

    annotations = data['annotations']
    for entry in annotations:
      vocab = entry['multiple_choice_answer']
      if re.search(r'[^\w\s]', vocab):
        continue
      answers[vocab] += 1

  # sort by max value to keep top-1000-answers occuring
  answers = sorted(answers, key=answers.get, reverse=True)
  # adding the <unk> answer as the first element in our answer-vocab
  top_answers = ['<unk>'] + answers[:top_answer-1]
  with open(des_dir + '/Annotations/annotation_vocabs.txt', 'w') as f :
    f.writelines([ans+'\n' for ans in top_answers])

  print(f'The number of total words of answers: {len(answers)}')
  print(f'Keep top {top_answers} answers into vocab' )
```

6.3.2 Question Vocabulary

It is like answer vocabulary with slightly different steps:
Downloaded questions dataset json files, they are four json files (test, test-dev, validation, and train). We got 443,757 questions for training, 214,354 for validation, 447,793 for test and 107,394 for test-dev in a total of 1,213,298

questions with 17,777 unique words. *"Snippet for question schema"*

```
{
  'image_id': 262144,
  'question': 'What credit card company is on the banner in the back-
ground?',
  'question_id': 262144005
}
```

To get each word of the question we split it with each non-alphanumeric character in the question. "Snippet for question after applying regular expression to split it"

```
quest['question'] >>>
What is this photo taken looking through?

regex.findall(quest['question'].lower()) >>>
[' ', ' ', ' ', ' ', ' ', ' ', '?']

regex.split(quest['question'].lower()) >>>
['what', 'is', 'this', 'photo', 'taken', 'looking', 'through', '']
```

Then add those tokens to our question array. After this step we got 7,463,853 words. To remove redundant words, we convert our array into a set then reordered it in ascending order. Now, we have 17,775 unique words. Finally, we insert the <PAD> and <UNK> in the top of our question vocabulary and save it as a "txt" file to use in different phases. *"Code snippet"*

```
def make_q_vocab():
# os.listdir returns a list of all entries in the directory
  dataset = os.listdir(src_dir + '/Questions')
  # \W matches any non-alphanumeric character;
  # this is equivalent to the class [^a-zA-Z0-9_];
  # + for one or more
  regex = re.compile(r'\W+')

  q_vocab = []
```

```
for file in dataset:
  path = os.path.join(src_dir, 'Questions', file)
  # list of questions type(q_data['questions']) == list of dict.
  # questions {'question_id', 'image_id', 'question'}
  try:
    with open(path, 'r') as f:
      print(f)
      q_data = json.load(f)
      question = q_data['questions']

  except(IOError, SyntaxError):
    break

  for idx, quest in enumerate(question):
  # split on non-
alphanumeric character (such as: spaces, ?, ...etc)
    split = regex.split(quest['question'].lower())

    # removing both the leading and the trailing characters
    # space is the default leading character to remove
    tmp = [w.strip() for w in split if len(w.strip()) > 0 ]

    # x = [1, 2, 3]
    # x.extend([4, 5]) >> [1, 2 ,3 ,4, 5]   ; iterable element
      # x.append([4, 5]) >> [1, 2, 3, [4, 5]] ; object

    q_vocab.extend(tmp)

    # sets, unlike lists or tuples,

# cannot have multiple occurrences of the same element and stor
e unordered values
  q_vocab = list(set(q_vocab))
  q_vocab.sort()
  q_vocab.insert(0, '<pad>')
  q_vocab.insert(1, '<unk>')

  if not os.path.exists(des_dir):
    os.makedirs(des_dir)

with open(des_dir + '/Questions/question_vocabs.txt', 'w') as f:
    f.writelines([v+'\n' for v in q_vocab])

  print(f"total word:{len(q_vocab)}")
```

6.3.3 Images

Dealing with images was a little bit harder than annotations and question because of their size and the colab disk and memory limitations. Also requiring downloading it each time we need it or if the session terminated. We tried to resize it and save it but it is still large and takes time in the training process. To overcome these issues we tried saving it as ".h5" and loading it from colab's disk. It was a huge improvement but still took almost an hour per epoch. Finally, we extracted the features from images, and it was the best solution for both memory usage and disk limitations. Features extraction will be described in more detailed in the model deployment.

6.3.3.1 Extracting Information

As a final step after preprocessing our data we need to make it easy to get the necessary information from JSON files for each datatype in our questions dataset *"train, val, test and test-dev"* the issue here that we did not have a separate annotations and images for test-dev datatype but it is related to our test images dataset so at first we must check if the data-subtype == "test-dev" then we assignment it to our test images dataset. *"Code snippet"*

```
with open(question, 'r') as f:
  data = json.load(f)

questions = data['questions']

if (data['data_subtype'] == 'test-dev2015'):
  filename = 'test2015'
else:
  filename = data['data_subtype']
```

Treating test dataset questions in a different way than train and validation because there is no answer here so we must flag/label it to deal with them if labeled this means that the given data is train or validation, so it has annotations, if not this means it is a test data. *"Code snippet"*

```
# true : if val or train
if labeled:
  template = annotation_dir + f'/*{filename}*.json'
  annotation_path = glob.glob(template)[0]
  with open(annotation_path) as f:
    annotations = json.load(f)['annotations']
  question_dict = {ans['question_id']: ans for ans in annotations}
```

The needed information for each question is differ how you will treat it but in general these are the most information you will need *('img_name', 'img_path', 'que', 'ans', 'img_id' and 'qu_id')* we go for two options here:

1. Make an information dictionary and keep the needed information for each particular question in our data set in it such as ('que_id', 'que', 'img_id', 'ans'). *"Snippet for info dictionary and its code"*

```
{
  'img_id': 458752,
  'img_name': 'COCO_train2014_000000458752.jpg'
  'qu_sentence': 'What is this photo taken looking through?'
  'qu_id': 458752000
  'ans': 'net'
}
```

```
for idx, qu in enumerate(questions):

  if (idx+1) % 10000 == 0:
    print(f'processing {data["data_subtype"]} data: {idx+1}/{len(questi
ons)}')

    qu_id = qu['question_id']
    qu_sentence = qu['question']
    img_id = qu['image_id']
    img_name = 'COCO_' + filename + '_{:0>12d}.jpg'.format(img_id)

    info = {
              'img_id': img_id,
              'img_name': img_name,
              'qu_sentence': qu_sentence,
              'qu_id': qu_id
           }

  if labeled:
    annotation_ans = question_dict[qu_id]['multiple_choice_answer']
    ans = match_top_ans(annotation_ans)
    info['ans'] = ans

  dataset[idx] = info
```

2. Making a 2D list holds that information while the first element of each raw entry holds the *'img_name'*, the second *'que'* and the last one *'ans'*. *"Snippet for for code and list"*

```
[['COCO_train2014_000000458752.jpg', 'What is this photo taken looking
through?', 'net']]
```

```
for idx, qu in enumerate(questions):
  if (idx+1) % 10000 == 0:
    print(f'processing {data["data_subtype"]} data: {idx+1}/{len(questi
ons)}')
    qu_id = qu['question_id']
    qu_sentence = qu['question']
    img_id = qu['image_id']
    img_name = 'COCO_' + filename + '_{:0>12d}.jpg'.format(img_id)

    info = [img_name, qu_sentence]

  if labeled:
    annotation_ans = question_dict[qu_id]['multiple_choice_answer']
    ans = match_top_ans(annotation_ans)
    info.append(ans)

  dataset[idx] = info
```

3. Finally we save it as NumPy files with extension *.npy* so we got four files like *"train.npy, val.npy, test.npy, test-dev.npy"*. *"Code snippet"*

```
processed_data = {}
for file in os.listdir(question_dir):
  try:
    datatype = file[20:-19]
    print(datatype)
    labeled = False if "test" in datatype else True
    question = os.path.join(question_dir, file)
    processed_data[datatype] = preprocessing(question, annotation_dir,
labeled)
  except (IOError, SyntaxError):
    pass

for key, value in processed_data.items():
  np.save(os.path.join(output_dir, f'{key}.npy'), np.array(value))
```

6.3.3.2 Feature Extraction

Using a pretrained VGG16 to extract features from images in our VQA dataset. The VGG16 model (pretrained on the ImageNet dataset) comes prepackaged with **Keras**. We will import it from the **keras.applications** module.

```
image_model =tf.keras.applications.vgg16.VGG16(weights="imagenet", include_top=False,input_shape=(224,224,3))
image_model.summary()
```

We pass three arguments to the constructor:

- **weights** specify the weight checkpoint from which to initialize the model.
- **include_top** refers to including (or not) the densely connected classifier on top of the network. By default, this densely connected classifier corresponds to the 1,000 classes from ImageNet. Because we only need to use the model for feature extraction, we do not need to include it.
- **input_shape** is the shape of the image tensors that you will feed to the network. This argument is purely optional: if you do not pass it, the network will be able to process inputs of any size.

We preprocess images in our VQA dataset to be resized to (224, 224, 3) and then feed them into the image model. Here is the detail of the VGG16 image model we will use to extract features from images in our VQA dataset:

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
```

The final feature map has shape (7, 7, 512). That is the output we will be saving on disk for each image in our training set.

The visual part (Image model) of our VQA model works as follows:

- We will first run this pretrained VGG16 model over images in our VQA dataset.
- The output feature map of shape (7, 7, 512) will be reshaped to (49, 512) and this output is recorded as a Numpy array on disk.
- We then use this Numpy array as the representation of our image going forward in our VQA proposed model.

This solution is fast and cheap to run because it only requires running the VGG16 pretrained model once for every input image, and the image model is by far the most expensive part of the pipeline.

6.4 Feeding our Data into the VQA Model

In this section we will go through steps to acquire data and pass it to a dataset API, to accomplish that we follow below steps (*all steps were done using google colab environment runtime*). After the preprocessing phase, we uploaded all files we had to our google drive so, we can easily acquire it into google colb by mounting our drive into colab or downloading files through *"!gdown-command"*. The final preprocessed data is *(imageFeatures_train.zip, imageFeatures_val.zip, train.npy, val.npy, test.npy, test-dev.npy, question_vocab.txt, answer_vocab.txt)*

1. Mounts google drive that holds our data.
2. Download our features from google drive into colab's disk, using this command

```
# Train
!gdown https://drive.google.com/uc?id=1qL6LnPlDBv9M0W0GZLu0_745UVE_uqXI
# Validation
!gdown https://drive.google.com/uc?id=13Okseytkvh5VRQm1cRcjJ0inouMFHS1i
```

3. Unzip both of the files and remove zip files to keep our disk space free.
4. Getting our vocabularies ready, so we created a vocab class that have three attributes, three methods and a constructor to initialize the vocb attributes by calling its "load_vocab" method.
    1. Attributes:
        1. List holds each entry (word) in the vocab.
        2. Dictionary where its keys are the words of the vocab, and its values are the index for that word.

3. Variable holds the size of the dictionary.

```
def __init__(self, vocab_file):
    self.vocab = self.load_vocab(vocab_file)
    self.vocab2idx = {word: idx for idx,
                      word in enumerate(self.vocab)}
    self.vocab_size = len(self.vocab)
```

2. Methods:

a. "load_vocab ()" which takes the path to the vocabulary txt file then, read each line into a list after removing any trailing space characters like "/n, /t"

```
def load_vocab(self, vocab_file):
    with open(vocab_file, 'r') as f:
      vocab = [line.strip() for line in f]
    return vocab
```

b. "word2idx ()" that returns the index for the given words. If the word is not in our dictionary, return the index for the '<UNK>' word. If it in our dictionary returns the index value.

```
def word2idx(self, word):
    if word in self.vocab2idx:
      return self.vocab2idx[word]
    else:
      return self.vocab2idx['<unk>']
```

c. "idx2word ()" returns the word corresponding to the given index by accessing the vocab list.

```
def idx2word(self, idx):
    return self.vocab[idx]
```

6.4.1 Building the VQA Dataset

Now we have our features loaded in our runtime environment and our answer and question vocabularies loaded into our program memory and they are ready to use. The next step is building our data set to feed it into our training_model. This model takes two inputs (one for image features while the other for the question) and one output. The image features input in a shape of (49, 512), the question in (MAX_QUE_LEN,) and the output in (len(answer_vocab),). So our dataset builder should consider these shapes. The main function for building our dataset is "build_dataset()" that takes the name of the dataset type (train or validation) and returns the data ready to use with "model.fit()". We will do that in the help of tensorflow dataset API and the mentioned below loading and preprocess functions. Dataset API will help with doing some operations on our dataset to get it ready by applying map function on the dataset also, useful for optimization of the input pipeline. The Dataset API will be discussed in more detail after this section. *"The entire code snippet for build_dataset()"*

```
def build_dataset(file_name):
  data_dir = os.path.join(INPUT_DIR, file_name)
  data = np.load(data_dir, allow_pickle=True).tolist()
  if 'train' in file_name:
    features_path = '/content/Features/train/ImagesFeatures'
  elif 'val' in file_name:
    features_path = '/content/Features/val/ImagesFeaturesVAL'
  features = []
  questions = []
  answers = []
  for element in data:
    features.append(os.path.join(features_path, element[0][:-
3] + 'npy'))
    questions.append(element[1])
    answers.append(element[2])
```

```
  dataset = tf.data.Dataset.from_tensor_slices((features, questions, an
swers))
  BATCH_SIZE = 128
  dataset = dataset.cache()
  dataset = dataset.map(lambda x, y, z: tf.py_function(func=preprocess,
inp=[x, y, z], Tout=(tf.float32,tf.int32,tf.int32)), num_parallel_calls
=tf.data.AUTOTUNE)
  dataset = dataset.map(get_tensors_ready)
  dataset = dataset.batch(BATCH_SIZE)
  dataset = dataset.prefetch(tf.data.experimental.AUTOTUNE)
  return dataset
```

6.4.1.1 Helping Functions

1. load_question (), which takes a particular question, making a numpy array "qu2idx" and filling it with the index of the '<PAD>' word in the question_vocab. Getting the tokens by passing this question to the tokenizer function which takes this question and returns its tokens as a list of tokens. Then iterate over this list of tokens, getting the corresponding index for each token in this list by invoking "question_vocab.word2idx(token)" and assign the returned index to the corresponding place in the "qu2idx" numpy array.

```
def load_question(question):
  qu_tokens = tokenizer(question)
  qu2idx = np.full(max_qu_length, question_vocab.word2idx('<pad>'))
  qu2idx[:len(qu_tokens)] = [question_vocab.word2idx(token) for token i
n qu_tokens]
  return qu2idx
```

2. load_answer (), similar to load_question but we did not need the tokenizer here since we treat the answer as a whole so in this function we get the one-hot vector representation for the answer by creating a numpy array of zeros with size of answer_vocab.vocab_size then assign the value of 1 to the index for the corresponding answer.

```
def load_answer(answer):
  answer_idx = answer_vocab.word2idx(answer)
  answer = np.zeros(answer_vocab.vocab_size)
  answer[answer_idx] = 1
  return answer
```

3. load_features (), it takes the path to the image feature and loads its corresponding feature numpy file.

```
def load_features(features_path):
  return np.load(features_path, allow_pickle=True)
```

4. All the above function are invoked from “preprocess ()” function that returns the (features, question_vector, answer_vector)

```
def preprocess(features_path, question, answer):
  features_path = features_path.numpy().decode('utf-8')
  question = question.numpy().decode('utf-8')
  answer = answer.numpy().decode('utf-8')

  features = load_features(features_path)
  question_vector = load_question(question)
  answer_vector = load_answer(answer)

  return (features, question_vector, answer_vector)
```

5. Getting our tensors in the required shape by applying “get_tensor_ready()” that returns our input/output tensors in the shape of ((x, y), z).

```
def get_tensors_ready(x,y,z):
  x.set_shape((49,512))
  y.set_shape((30,))
  z.set_shape((1000,))
  return ((x,y),z)
```

6.4.1.2 Dataset API

The tf.data API enables building complex input pipelines from simple, reusable pieces. For example, the pipeline for an image model might aggregate data from files in a distributed file system, apply random perturbations to each image, and merge randomly selected images into a batch for training. The pipeline for a text model might involve extracting symbols from raw text data, converting them to embedding identifiers with a lookup table, and batching together sequences of different lengths. The tf.data API makes it possible to handle large amounts of data, read from different data formats, and perform complex transformations.

The tf.data API introduces a **tf.data.datset** abstraction that represents a sequence of elements, in which each element consists of one or more components. For example, in an image pipeline, an element might be a single training example, with a pair of tensor components representing the image and its label.

We will use the tf.data.datset abstraction to build our input pipeline. Our model is a multiple-input single output model. We create a dataset object from three tensors:

- A tensor holding the images features locations on disk.
- A tensor holding the questions.
- A tensor holding the answers.

```
dataset = tf.data.Dataset.from_tensor_slices(((Images_features_paths , questions) , answers)))
```

The API will enable us to:

1- Randomly shuffle the elements of this dataset.

   This dataset fills a buffer with 1024 elements, then randomly samples elements from this buffer, replacing the selected elements with new elements.

```
dataset=dataset.shuffle(1024)
```

2- Map the paths in our dataset to the images features saved on disk.

```
def load_features(image_path):
   return np.load(image_path)
dataset=dataset.map(load_features)
```

3- Create batches of consecutive elements of this dataset.

```
dataset=dataset.batch(BATCH_SIZE)
```

4- Prefetch elements from the dataset. This allows later elements to be prepared while the current element is being processed.

```
dataset=dataset.prefetch(tf.data.AUTOTUNE)
```

We created a dataset object for the training data and a dataset object for the validation data, we then apply the previous methods to them and pass them to the training process

7 Model deployment:

This section describes the steps needed to convert the model into a format that can run on an Android application. For this purpose, we used the TensorFlow Lite library.

TensorFlow Lite library is a set of tools that enables on-device machine learning by helping developers run their models on mobile, embedded, and IoT devices. It lets us run machine-learned models on mobile devices with low latency. TFLite consists of two core components: converter and interpreter.
The converter will help us to convert deep learning models into the TFLite format and the interpreter makes our life easier while inferencing.

TensorFlow also provides another library called TensorFlow Lite Support. This library currently only supports Android. This library makes it easier to integrate models into the application. It provides a high-level API that helps transform raw input data into the form required by the model and interpret the model's output, reducing the amount of boilerplate code required.

The process of model deployment can be divided into two steps: the first step is to generate the TensorFlow Lite model and the second is step is to run inference using the generated model on the Android application.

1. Generate a TensorFlow Lite model:

A TensorFlow Lite model is represented by a special efficient portable format known as FlatBuffers, which is identified by the *.tflite* file extension. This format provides several advantages over TensorFlow's protocol buffer model format such as reduced size and faster inference. The TensorFlow lite model is optimized for on-device machine learning, by addressing 5 key constraints:

- Latency: there is no round-trip to a server.
- Privacy: no personal data leaves the device.
- Connectivity: Internet connectivity is not required.
- Size: reduce model and binary size.
- Power consumption: efficient inference and a lack of network connections.

To generate the TensorFlow Lite model, we followed the process recommended by the TensorFlow documentation, by first storing the model in SavedModel format. This is done through the low-level tf.saved_model API.
A SavedModel contains a complete TensorFlow program, including trained parameters (i.e, tf.Variables) and computation. It does not require the original model building code to run, which makes it useful for sharing or deploying with TFLite.

```
tf.saved_model.save(full_model, '/content/VQA/1/')
```

Figure 33: Saving the TensorFlow model using SavedModel format

Then, we converted the TensorFlow model into a TensorFlow Lite model

```
converter = tf.lite.TFLiteConverter.from_saved_model('/content/VQA/1/') # path to the SavedModel directory
tflite_model = converter.convert()

# Save the model.
with open('VQA_model.tflite', 'wb') as f:
  f.write(tflite_model)
```

Figure 34: Converting the SavedModel into TFLite model

by using the TensorFlow Lite Converter.

The third step was to use the TensorFlow Lite Python Interpreter to load the TFLite model.

```
interpreter = tf.lite.Interpreter(model_path="VQA_model.tflite")
interpreter.allocate_tensors()
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
```

Figure 15: Loading the TFLite model using python API

By inspecting the input_details of the TFLite model, we can see that the model has two input vectors; the first one is the question, which is an array of the shape {1, maximum question length} and has a datatype of *int64* which is suitable for the vocabulary index of each word in the question, the second input is for the image, which is an array of the shape {1, 224, 224, 3} and has a datatype of *float32* which makes it a suitable container of the pre-processed pixel values of the image.

```
input_details
```

```
[{'dtype': numpy.int64,
  'index': 0,
  'name': 'serving_default_input_3:0',
  'quantization': (0.0, 0),
  'quantization_parameters': {'quantized_dimension': 0,
   'scales': array([], dtype=float32),
   'zero_points': array([], dtype=int32)},
  'shape': array([ 1, 30], dtype=int32),
  'shape_signature': array([-1, 30], dtype=int32),
  'sparsity_parameters': {}},
 {'dtype': numpy.float32,
  'index': 1,
  'name': 'serving_default_input_2:0',
  'quantization': (0.0, 0),
  'quantization_parameters': {'quantized_dimension': 0,
   'scales': array([], dtype=float32),
   'zero_points': array([], dtype=int32)},
  'shape': array([  1, 224, 224,   3], dtype=int32),
  'shape_signature': array([ -1, 224, 224,   3], dtype=int32),
  'sparsity_parameters': {}}]
```

Figure 36: TFlite model input

Similarly, the output of the TFLite model is an array of the shape{1, 1000} and has a datatype of *float32* which makes it a suitable container to hold the prediction distribution over the top k=1000 answers.

```
output_details
```

```
[{'dtype': numpy.float32,
  'index': 83,
  'name': 'StatefulPartitionedCall:0',
  'quantization': (0.0, 0),
  'quantization_parameters': {'quantized_dimension': 0,
   'scales': array([], dtype=float32),
   'zero_points': array([], dtype=int32)},
  'shape': array([   1, 1000], dtype=int32),
  'shape_signature': array([  -1, 1000], dtype=int32),
  'sparsity_parameters': {}}]
```

Figure 37: TFlite model output

We then gave both the TensorFlow model and the converted TFLite model the same picture and question and compared the results of both models. Both models gave identical results, which meant that there were no errors in the conversion process.

```
pic = load_image('./pic4.jpg')
que = "What color is the cat ?"
que = load_question(que)
```

```
pic =  tf.expand_dims(pic, axis=0)
que = tf.expand_dims(que, axis=0)
```

```
ans = full_model([pic, que])
answers = tf.math.top_k(ans, 5)
for i in (answers.indices[0].numpy().tolist()):
  print(answer_vocab.idx2word(i), ans[0][i].numpy() * 100)
```

```
white 43.44096779823303
black 30.361929535865784
brown 15.473364293575287
gray 3.9943531155586243
black and white 0.8459299802780151
```

Figure 38 :TFLite model results using python Interpreter

```
input1_shape = input_details[0]['shape']
input2_shape = input_details[1]['shape']
interpreter.set_tensor(input_details[0]['index'], que)
interpreter.set_tensor(input_details[1]['index'], pic)
```

```
interpreter.invoke()
```

```
output_data = interpreter.get_tensor(output_details[0]['index'])
```

```
answers = tf.math.top_k(output_data, 5)
for i in (answers.indices[0].numpy().tolist()):
  print(answer_vocab.idx2word(i), ans[0][i].numpy() * 100)
```

```
white 43.44096779823303
black 30.361929535865784
brown 15.473364293575287
gray 3.9943531155586243
black and white 0.8459299802780151
```

Figure 39: TensorFlow model

2- Run inference on the Android application:

1. The first step is to add TensorFlow Lite libraries to the application. This can be done by adding the required libraries to the *build.gradle* file's dependencies section.

```
dependencies {

    implementation 'androidx.appcompat:appcompat:1.3.0'
    implementation 'com.google.android.material:material:1.3.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.0.4'

    //Add dependencies to download the required tflite libraries
    implementation('org.tensorflow:tensorflow-lite:0.0.0-nightly') {changing = true}
    implementation('org.tensorflow:tensorflow-lite-support:0.0.0-nightly') {changing = true}

    testImplementation 'junit:junit:4.+'
    androidTestImplementation 'androidx.test.ext:junit:1.1.2'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'
}
```

Figure 40: adding the TFLite libraries to dependencies section

2. The second step is to specify that the tflite file should not be compressed by the gradle script while building the application's APK file.

```
android {
    //Prevents the builder from compressing the tflite model
    aaptOptions {
        noCompress "tflite"
    }
```

Figure 42: Specify that the tflite file should not be compressed

3. The third step is to create an *assets* folder in the project's directory and add the generated TFLite model file to it.
We also add the *question_vocabs* text file which contains the words of the questions that the model had seen during training, and the

annotation_vocabs text file which contains the most frequent 1000 answers in the training set.

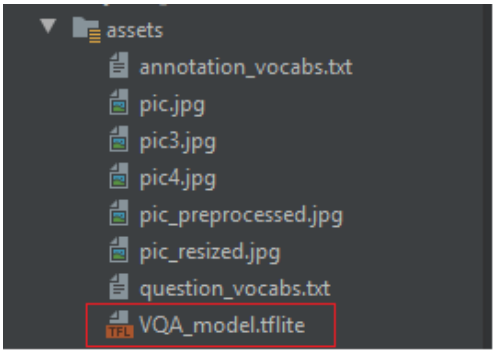


Figure 2: Add the TFLite model to the assets folder

4. The fourth step is to use the TFLite Support library to load the *tflite* model file into a *MappedByteBuffer* and then we create an Interpreter object and load it with the *MappedByteBuffer* we just created. This Interpreter object will allow us to run the model,

```
//Load the model into a MappedByteBuffer format
MappedByteBuffer VQA_model = FileUtil.loadMappedFile(activity, filePath: "VQA_model.tflite");
//Load the answers into a list of strings
answers = FileUtil.loadLabels(activity, filePath: "annotation_vocabs.txt");
//Create the interpreter object that performs inference with no options
classifier = new Interpreter(VQA_model, options: null);
```

Figure 43: Creating the interpreter

providing it a set of inputs.
We also use the TFLite Support Library to read the *annotation_vocabs* text file, which contains the top 1000 answers and loads them into a list of strings.

5. The fifth step is to create containers for both the inputs and the outputs of the model. Each input and output should be an array or multi-dimensional array of the supported primitive types or a raw *ByteBuffer* of the appropriate size. If the input is a *ByteBuffer*, its

order must be *ByteBuffer.nativeOrder( )*, which retrieves the native byte order of the underlying platform whether it was *BIG_INDIAN* or *LITTLE_INDIAN*.

Since *ByteBuffers* can be difficult to debug and manipulate, the TFLite Support library allows us to create tensors, which can be easier than dealing with *ByteBuffers* directly.

We used the interpreter object to retrieve information about the model's required inputs and output shapes and data types, this information is necessary to create the input and output containers.

The application then waits for the user to take a picture and enter his question about it.

```
int[] input_question_shape = classifier.getInputTensor(INPUT_QUESTION_INDEX).shape(); //{Batch_size, 30}
DataType input_question_type = classifier.getInputTensor(INPUT_QUESTION_INDEX).dataType();

int[] input_image_shape = classifier.getInputTensor(INPUT_IMAGE_INDEX).shape();     //{Batch_size, Height, Width, 3}
DataType input_data_type = classifier.getInputTensor(INPUT_IMAGE_INDEX).dataType();

int[] output_props_shape = classifier.getOutputTensor(OUTPUT_PROP_INDEX).shape();   //{Batch_size, Num_answers}
DataType output_props_type = classifier.getOutputTensor(OUTPUT_PROP_INDEX).dataType();

//Find the model's required input image size
img_resize_x = input_image_shape[1];
img_resize_y = input_image_shape[2];

//creates tensors for the input and the output
inputQuestionTensor = TensorBuffer.createFixedSize(input_question_shape, input_question_type);

//Creates a ByteBuffer that will hold the image data and can be read by the interpreter
//each pixel is represented by 3 colors, each of which is stored in 4 bytes (FLOAT32)
imgByteBuffer = ByteBuffer.allocateDirect(img_resize_x * img_resize_y * 3 * 4);
imgByteBuffer.order(ByteOrder.nativeOrder());

probability_tensor_buffer = TensorBuffer.createFixedSize(output_props_shape, output_props_type);
```

Figure 3: Creating the input and output containers

6. The sixth step after retrieving the picture and the question sentence from the user is to pre-process each of them in a similar manner to the pre-processing step done during the model training.

The *loadQuestion* function performs the task of pre-processing the input question and it works as follows:

The function first loads the *question_vocab*, which contains all of the question words seen in the training dataset. Then the question sentence is lowercased and tokenized into an array of words. Non-alphabetic characters are then removed from each word.

Another array of integers is created to hold the index of each word in the question.

This array has the length of the *max_question_length, which* was set to 30 during the model training. The function passes through each word in the question tokens array and then inserts the vocabulary index of that word into the indices array. If a word is not found in the vocabulary, it gets replaced with the index of the *<unk>* token. The rest of the array is padded with the index of the *<pad>* token. Finally, the question tensor is then loaded with the indices array.

```
private void loadQuestion(Context context, String sentence, TensorBuffer questionTensor) {
    question_vocab = new Vocab(context,  filename: "question_vocabs.txt");
    int max_question_length = 30;
    String[] tokens;
    int[] indices = new int[max_question_length];

    sentence = sentence.toLowerCase();
    tokens = sentence.split( regex: " ");

    for (int i = 0; i < tokens.length; i++) {
        String word = tokens[i];
        word = word.replaceAll( regex: "[^a-zA-Z0-9]",  replacement: ""); //remove non-alphabetic chars from each word
        if (word.length() > 0) {
            int index = question_vocab.word2idx(word);
            indices[i] = index;
        }
    }
    questionTensor.loadArray(indices);
}
```

Figure 45: Question pre-processing

We created *loadImage* function to perform pre-processing to the image taken from the camera. The image is first resized to the size expected by the VGG-16 network, which is (224, 224), then a copy of the pixel values of the image bitmap is stored into an integer array, where each element in that array is a packed integer value representing the color of a pixel in the form of αRGB.

```
private void loadImage(Bitmap bitmap, ByteBuffer imgBuffer) {

    Bitmap ResizedImage = Bitmap.createScaledBitmap(bitmap, img_resize_x, img_resize_y, filter: true);
    //An array that will hold the values stored in the bitmap
    int[] intValues = new int[img_resize_x * img_resize_y];
    //Returns in intValues[] a copy of the data in the bitmap. Each value is a packed int representing a Color(aRGB)
    ResizedImage.getPixels(intValues, offset: 0, ResizedImage.getWidth(), x: 0, y: 0, ResizedImage.getWidth(), ResizedImage.getHeight());

    imgBuffer.rewind();
    //loop through all pixels
    for (int i = 0; i < img_resize_x; i++)
        for(int j = 0; j < img_resize_y; j++)
        {
            int pixelValue = intValues[i * img_resize_x + j];
            float R = (pixelValue >> 16) & 0xFF;
            float G = (pixelValue >> 8) & 0xFF;
            float B = pixelValue & 0xFF;
            imgBuffer.putFloat(B - imageMean[2]);   //B
            imgBuffer.putFloat(G - imageMean[1]);   //G
            imgBuffer.putFloat(R - imageMean[0]);   //R
        }
    }
```

Figure 4: Image pre-processing

We then go through all the pixel values stored in that array, get each of the RGB components of that pixel, and zero-center them using the mean pixel values used in VGG-16 training on the ImageNet dataset, without scaling.
The pixel values are then loaded into the image *ByteBuffer* using the *BGR* format.

8. The last step after taking the picture and the question from the user and pre-processing each of them is to pass them to the model.

The *interpreter* API provides a method for running inference with a model that has multiple inputs or outputs. This method takes two arguments; the first argument is an array of input *ByteBuffers* and the second argument is a map that maps the index of each output to its corresponding *ByteBuffer*. In both cases, the ord er of inputs and outputs must match the order given to the TensorFlow converter. Finally, we map the results (predictions) to the corresponding answers by creating a TensorLabel object, initializing it with the list of answers strings and the output probability tensor, and invoking its getMapWithFloatValue method, which returns a map that stores each answer and its corresponding probability. We then store these answers into a list, sort them according to their probability, and return the top answer to the interface.

```
public Answer provideAnswer(Context context, final Bitmap bitmap, String question_sentence) {
    List<Answer> answers_list = new ArrayList<>();
    loadImage(bitmap, imgByteBuffer);
    loadQuestion(context, question_sentence, inputQuestionTensor);

    Object[] inputArray = {inputQuestionTensor.getBuffer(), imgByteBuffer};

    Map<Integer, Object> outputMap = new HashMap<>();
    outputMap.put(0, probability_tensor_buffer.getBuffer().rewind());

    classifier.runForMultipleInputsOutputs(inputArray, outputMap);

    //returns a map that has each label and its corresponding probability
    Map<String, Float> labelled_props = new TensorLabel(answers, probability_tensor_buffer).getMapWithFloatValue();

    //Adds every element in the labelled probabilities to the list of Recognition
    for (Map.Entry<String, Float> entry : labelled_props.entrySet()) {
        answers_list.add(new Answer(entry.getKey(), entry.getValue()));
    }
    //Sorts the list of predictions based on confidence score
    Collections.sort(answers_list);
    //return the top prediction
    return answers_list.get(0);
}
```

Figure 5: Running inference on the application

The inference on the Android application took a total time of nearly 1263 milliseconds (1.3 seconds).

```
long startTime = System.currentTimeMillis();
classifier.runForMultipleInputsOutputs(inputArray, outputMap);
long endTime = System.currentTimeMillis();
long timeElapsed = endTime - startTime;
Log.i( tag: "time",  msg: "Time elapsed: " + timeElapsed);
```

```
logcat
2021-07-12 23:32:01.000 28141-28141/com.example.vqa I/time: Time elapsed: 1263
```

Figure 6: Model inference time

Now, we can compare the results of running the model on Colab (to the left) to the results of running the converted TFLite model on Android (to the right) given the same image and question.

We can see that the converted TFLite model gives similar results with an error in the prediction probability of $\pm$ 1% in some of the predictions.

Where are they ?

```
beach 43.45899522304535
baseball field 15.753801167011261
office 7.792691886425018
```

```
answers_list = {ArrayList@32812} size = 1000
  0 = {Answer@32827} "beach (43.6%)"
  1 = {Answer@32828} "baseball field (14.8%)"
  2 = {Answer@32829} "office (7.5%)"
```

Figure 7: TF model results (left) vs TFLite model inference on Android results (right)

Conclusion

In this project, we proposed the task of open-ended Visual Question Answering (VQA). Our aim was to develop an AI-powered system to help empower blind and visually impaired individuals to be more independent and comfortable practicing their daily chores and activities (e.g.,shopping, cooking, playing sports and meeting people) by enabling them to obtain information about images in the real world. These kinds of projects that depend on deep learning algorithms require a huge amount of dataset to train your model on them. For this purpose, these are the major datasets for VQA that have been publicly released. Which enables the VQA systems to be trained and evaluated.,

After choosing the dataset, we need to get it preprocessed to satisfy our model requirements for the input and output shapes so preprocessing data can be split into three phases (making answer vocabulary, question vocabulary and extracting the features from our images) after this we collected the necessary information for each question and save them as numpy files (train.npy, test.npy, test-dev.npy, val.npy) this step is used for acquiring data in the dataset building phase. Finally, we described our model in two main parts. The first part is responsible for the visual part of the model to produce a visual representation that describes the image. For this part we used a VGG_16 pre-trained CNN model followed by a fully connected layer containing 1024 neurons. The second part is responsible for the question representation of the model using RNN with LSTM, using element-wise product operation to combine both parts then feed it into a softmax layer to produce the probability for each answer in our answer vocabulary.

After training and testing the model using tensorflow library we converted it using tensorflow lite so we can deploy it to android devices, as a demo our application consists of only one main activity that holds the entire application program with a minimal user interface asking the user to input a question about a captured picture we also add a speech recognition system that recognize the user's voice as an input and produce the transcription for it and pass it to our model instead of asking the user to manually enter his question.

Future Work

A successful and useful application requires continuous updates and improvements, and our application is not excluded from this rule. To make our application useful and to provide real aid to the visually impaired, we can help them get a better user experience with the application. We can make the application run as a service in the background and as the user shakes the phone in a particular pattern, the application would pop up and allow the user to take a picture and get his question about it. Instead of using Android Camera v1 API, which requires the user to touch the capture button at the bottom of the screen or the sound buttons, we can write a custom camera based on Android Camera v2 or Camera X API, where the user can capture an image by touching any part of the screen or using voice commands.

We can perform post-training quantization to reduce the inference time of the model. Post-training quantization is a conversion technique that can reduce model size while also improving CPU and hardware accelerator latency, with little degradation in model accuracy. We can quantize our already-trained float TensorFlow model while converting it to TensorFlow Lite format using the TensorFlow Lite Converter.We can try to train a more

complex model that could provide better accuracy, but it should still give a reasonable inference time on the Android device.

References

[1] Rupert R A Bourne, Jaimie Adelson, Seth Flaxman, Paul Briant, Michele Bottone, Theo Vos, Kovin Naidoo, Tasanee Braithwaite, Maria Cicinelli, Jost Jonas, Hans Limburg, Serge Resnikoff, Alex Silvester, Vinay Nangia, Hugh R Taylor; Global Prevalence of Blindness and Distance and Near Vision Impairment in 2020: progress towards the Vision 2020 targets and what the future holds.

[2] Vision Loss Expert Group of the Global Burden of Disease Study. Trends in prevalence of blindness and distance and near vision impairment over 30 years: an analysis for the Global Burden of Disease Study. Lancet Global Health 2020. doi: 10.1016/S2214-109X(20)30425-3

[3] Kafle, K., & Kanan, C. (2017). Visual question answering: Datasets, algorithms, and future challenges. Computer Vision and Image Understanding, 163, 3–20. https://doi.org/10.1016/j.cviu.2017.06.005

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[5] H. Noh, S. Hong, and B. Han, "Learning deconvolution network for semantic segmentation," in The IEEE Conference on Computer Vision and Pattern Recognition (CVPR),2015.

[6] N. Silberman, D. Sontag, and R. Fergus, "Instance segmentation of indoor scenes using a coverage loss," in European Conference on Computer Vision (ECCV), 2014.

[7] Z. Zhang, A. G. Schwing, S. Fidler, and R. Urtasun, "Monocular object instance segmentation and depth ordering with CNNs," in The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

[8] Z. Zhang, S. Fidler, and R. Urtasun, "Instance-level segmentation with deep densely

connected MRFs," in The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[9] A. Karpathy and L. Fei-Fei, "Deep visual-semantic alignments for generating image descriptions," in The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

[10] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Doll´ar, and C. L. Zitnick, "Microsoft coco: Common objects in context," in European Conference on Computer Vision (ECCV), 2014.

[11] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: a method for automatic evaluation of machine translation," in Annual Meeting of the Association for Computational Linguistics (ACL), 2002.

[12] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in Text summarization branches out: Proceedings of the ACL-04 workshop, vol. 8, Barcelona, Spain,

2004.

[13] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in Proceedings of the ACL workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization, vol. 29, pp. 65–72, 2005.

[14] R. Vedantam, C. Lawrence Zitnick, and D. Parikh, "Cider: Consensus-based image description evaluation," in The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

[15] M. Ren, R. Kiros, and R. Zemel, "Exploring models and data for image question answering," in Advances in Neural Information Processing Systems (NIPS), 2015.

[16] S. Antol, A. Agrawal, J. Lu, M. Mitchell, D. Batra, C. L. Zitnick, and D. Parikh, "VQA:Visual question answering," in The IEEE International Conference on Computer Vision (ICCV), 2015.

[17] H. Gao, J. Mao, J. Zhou, Z. Huang, L. Wang, and W. Xu, "Are you talking to a machine? Dataset and methods for multilingual image question answering," in Advances in Neural Information Processing Systems (NIPS), 2015.

[18] Y. Zhu, O. Groth, M. Bernstein, and L. Fei-Fei, "Visual7w: Grounded question answering in images," in The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[19] R. Krishna, Y. Zhu, O. Groth, J. Johnson, K. Hata, J. Kravitz, S. Chen, Y. Kalantidis, L.-J. Li, D. A. Shamma, et al., "Visual genome: Connecting language and vision using crowdsourced dense image annotations," International Journal of Computer Vision, vol. 123, no. 1, pp. 32–73, 2017.

[20] N. Silberman, D. Hoiem, P. Kohli, and R. Fergus, "Indoor segmentation and support inference from rgbd images," in European Conference on Computer Vision (ECCV),2012.

[21] M. Ren, R. Kiros, and R. Zemel, "Exploring models and data for image question answering," in Advances in Neural Information Processing Systems (NIPS), 2015.

[22] Stewart, M. P. R. (2020, October 3). Tiny Machine Learning: The Next AI Revolution - Towards Data Science. Medium. https://towardsdatascience.com/tiny-machine-learning-the-next-ai-revolution-495c26463868

[23] Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. Neural Computation, 9(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[24] Karen Simonyan, Andrew Zisserman: Very Deep Convolutional Networks for Large-Scale Image Recognition. ICLR 2015

[25] Yang, Zichao & He, Xiaodong & Gao, Jianfeng & Deng, li & Smola, Alex. (2016). Stacked Attention Networks for Image Question Answering. 21-29. 10.1109/CVPR.2016.10.

[26] Anderson, P., et al. Bottom-up and top-down attention for image captioning and visual question answering. in Proceedings of the IEEE conference on computer vision and pattern recognition. 2018.

[27] Ren, S., et al., Faster r-cnn: Towards real-time object detection with region proposal networks. Advances in neural information processing systems, 2015. 28: p. 91-99.

[28]Li, L.H., et al., Visualbert: A simple and performant baseline for vision and language. arXiv preprint arXiv:1908.03557, 2019.

[29] Donahue, J., et al., Long-term Recurrent Convolutional Networks for Visual Recognition and Description CoRR, 2014. arXiv: abs/1411.4389.

[30] GOODFELLOW, I. J.; BENGIO, Y.; COURVILLE, A. Deep Learning. Cambridge, MA, USA: MIT Press, 2016.

[31] ZHANG; LIPTON, A. A.; LI, Z. C. A.; SMOLA, M. A. *et al.* Dive into Deep Learning. 2011.

[32] GÉRON, A. Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems . . 1 ed. O'Reilly Media, 2017. 978-1491962299.

[33] HASSAN, M. U. VGG16-convolutional network for classification and detection. 2018. available at: https://neurohive.io/en/popular-networks/vgg16/.

[34] Hochreiter, S. and J. Schmidhuber, *Long short-term memory.* Neural computation, 1997. 9(8): p. 1735-1780.

[35] Pennington, J., R. Socher, and C.D. Manning. Glove: Global vectors for word representation. in Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). 2014.