Final Project - Report

Deep Learning, Spring 2024

Instructor:

Prof. Alina Vereshchaka

Done By:

Devendra Mandava-dmandava-50534296

Sethu Thakkilapati-sethutha-50540870

Hemanth Kongara-hkongara 50540967

Project Title: Enhancing Image Understanding through Visual Question Answering

Team Member	Checkpoint [Contribution %]	Final [Contribution %]
Devendra Mandava	33.3	33.3
Sethu Thakkilapati	33.3	33.3
Hemanth Kongara	33.3	33.3

University at Buffalo
Graduate School of Education

Step-1:

Introduction

In our project, Enhancing Image Understanding through Visual Question Answering. A critical component is the development of comprehensive vocabularies for both the questions and answers. These vocabularies are essential for processing and understanding the vast array of questions and answers that can be posed about images.

Methodology

The project involves processing large datasets to extract and organize the vocabulary necessary for both questions and answers. Here is an outline of the steps followed:

Reading Data Files: The questions and annotations are stored in JSON files. Each file is parsed to extract relevant details.

Text Preprocessing:

- **Questions**: We tokenize the questions to split them into words. The words are cleaned and filtered to ensure they are meaningful (non-empty).
- **Answers:** For answers, we focus on the most frequent ones, considering only those that do not contain special characters and limiting our vocabulary to the top 1000 answers.

Vocabulary Creation:

- A unique list of words is created from the extracted data. This list is then sorted and written into a text file, forming our question vocabulary.
- Similarly, the top answers are identified and stored separately.

Data Labeling: For labeled datasets, each question is paired with its corresponding answer from the annotations. This is crucial for training and validation purposes.

Implementation

The implementation is done using Python, leveraging libraries such as json for parsing, re for regular expressions (used in tokenizing and cleaning), and os for file handling. The processed vocabularies are stored under predefined directories.

Generating Query Vocabulary

A function generate_query_vocabulary is defined to handle the extraction and processing of words from the questions. This function reads all question files, tokenizes each question, removes duplicates, and saves the sorted list of words.

```python
def generate_query_vocabulary():
    dataset_files = os.listdir(orig_path + '/questions')
    regex_pattern = re.compile(r'\W+')

    query_words = []

    for file_name in dataset_files:
        file_path = os.path.join(orig_path, 'questions', file_name)
        with open(file_path, 'r') as file_obj:
            question_data = json.load(file_obj)
            questions = question_data['questions']

        for query in questions:
            split_question = regex_pattern.split(query['question'].lower())
            cleaned_words = [word.strip() for word in split_question if word]
            query_words.extend(cleaned_words)

    query_words = list(set(query_words))
    query_words.sort()
    query_words.insert(0, '')
    query_words.insert(1, '')

    vocab_file_path = os.path.join(dest_path, 'Questions', 'question_vocabs.txt')
    with open(vocab_file_path, 'w') as file_obj:
        file_obj.writelines([word + '\\n' for word in query_words])
```

Creating Answer Vocabulary

The create_answer_vocabulary function processes the answers, counting the frequency of each unique answer across the dataset and selecting the top 1000 most frequent answers.

```python
def create_answer_vocabulary(max_answers_count):
    answer_counts = defaultdict(int)
    annotation_files = os.listdir(orig_path + '/annotations')

    for file_name in annotation_files:
        file_path = os.path.join(orig_path, 'annotations', file_name)
        with open(file_path, 'r') as file_obj:
            annotations = json.load(file_obj)['annotations']

        for entry in annotations:
            answer = entry['multiple_choice_answer']
            if not re.search(r'[^\w\s]', answer):
                answer_counts[answer] += 1

    top_answers = [''] + sorted(answer_counts, key=answer_counts.get, reverse=True)[:max_answers_count-1]
    vocab_file_path = os.path.join(dest_path, 'Annotations', 'annotation_vocabs.txt')
    with open(vocab_file_path, 'w') as file_obj:
        file_obj.writelines([ans + '\n' for ans in top_answers])
```

Outcomes
The preprocessing scripts successfully generated comprehensive vocabularies and prepared the dataset. Some key statistics include:

- Total words in the question vocabulary: 331,640
- Number of unique answers processed before filtering: 26,480
- Top 1000 answers retained for the vocabulary

Challenges and Resolutions
A significant challenge was managing the size of the data and ensuring efficient processing. This was addressed by streamlining file operations and optimizing data handling procedures. Additionally, handling unknown answers in the dataset required careful consideration to maintain the integrity of the training process.

Conclusion of step-1
The creation of vocabularies and the preprocessing of the data are crucial steps in developing a robust VQA system. These processes ensure that the system has a solid foundation of words and phrases to work with, enhancing its ability to understand and respond to a variety of questions about images accurately. This setup lays the groundwork for the subsequent phases of the project, which will involve model training and evaluation.

Step-2: Validation Image Feature Extraction

Objective
The primary objective of this phase was to download the validation images from the COCO dataset, preprocess these images, and extract features using the VGG19 model, a pre-trained convolutional neural network that is highly effective in image feature recognition tasks.

Implementation Details

Data Acquisition

The validation images were downloaded from the COCO dataset using the following command:

```
!wget http://images.cocodataset.org/zips/val2014.zip
!unzip val2014.zip -d /content/images
```

This command fetches and extracts approximately 40,504 validation images into a specified directory for processing.

Image Processing

Each image was processed to conform to the input requirements of the VGG19 model. The processing steps included:

Reading the image data: Using TensorFlow's tf.io.read_file.
Decoding the JPEG image: tf.image.decode_jpeg.
Resizing the image: Standardizing to 224x224 pixels, a requirement for VGG19 input.
Preprocessing: Using tf.keras.applications.vgg19.preprocess_input to prepare images for model input.

Here is a snippet of the image processing function:

```
def process_img_data(file_loc):
    raw_img_data = tf.io.read_file(file_loc)
    decoded_img = tf.image.decode_jpeg(raw_img_data, channels=3)
    resized_img = tf.image.resize(decoded_img, (224, 224))
    preprocessed_img = tf.keras.applications.vgg19.preprocess_input(resized_img)
    return preprocessed_img, file_loc
```

Feature Extraction

The processed images were then fed into the VGG19 model to extract features. The VGG19 model was loaded with pre-trained ImageNet weights, excluding the top classification layers:

```
def create_feature_extractor_model():
    vgg19_model = tf.keras.applications.VGG19(include_top=False, weights='imagenet')
    feature_extractor_model = tf.keras.Model(vgg19_model.input, vgg19_model.layers[-1].output)
    return feature_extractor_model
```

The extracted features were reshaped and saved as NumPy files for each image, facilitating quick loading during model training.

Batch Processing

The images were processed in batches to optimize the computation time and resource utilization. TensorFlow's tf.data.Dataset API was used to handle the data efficiently:

```
image_dataset = tf.data.Dataset.from_tensor_slices(unique_paths)
image_dataset = image_dataset.map(process_img_data, num_parallel_calls=tf.data.experimental.AUTOTUNE).batch(64)
```

Results
The feature extraction process processed all 40,504 validation images, resulting in a comprehensive dataset of image features stored as NumPy files. These features will be instrumental in training the VQA model.

Challenges Encountered

- Large Data Volume: Handling tens of thousands of images and their features required careful management of computing resources.
- Integration Issues: Initial challenges in ensuring the correct paths for saving and retrieving data were resolved through meticulous file management.

Step-3: Image Feature Extraction for Training Data:

Objective
The primary goal of this phase was to download and process the training images from the COCO dataset, then utilize the VGG19 neural network model to extract detailed features from these images. These features are essential for the model to accurately associate visual data with the corresponding textual questions and answers in the training phase.

Implementation Details

Data Acquisition

The training images were downloaded and extracted using the following commands:

```
!wget http://images.cocodataset.org/zips/train2014.zip
!unzip train2014.zip -d /content/images && rm train2014.zip
```

This setup ensures that all training images are readily available for preprocessing and feature extraction.

Image Processing

- Each image underwent several preprocessing steps to standardize its format before feature extraction:

- Reading the image data: Utilizing TensorFlow's tf.io.read_file.
- Decoding the image: Transforming the raw data into a usable JPEG format with tf.image.decode_jpeg.
- Resizing: Adjusting each image to 224x224 pixels to match the VGG19 input specifications.
- Preprocessing: Applying the tf.keras.applications.vgg19.preprocess_input to normalize the image data.

Here is the function used for processing the images:

```
def process_img_data(file_loc):
    raw_img_data = tf.io.read_file(file_loc)
    decoded_img = tf.image.decode_jpeg(raw_img_data, channels=3)
    resized_img = tf.image.resize(decoded_img, (224, 224))
    preprocessed_img = tf.keras.applications.vgg19.preprocess_input(resized_img)
    return preprocessed_img, file_loc
```

Feature Extraction

The processed images were input into the VGG19 model to extract deep features. The VGG19 model was chosen for its effectiveness in capturing intricate details necessary for complex image understanding tasks:

```
def create_feature_extractor_model():
    vgg19_model = tf.keras.applications.VGG19(include_top=False, weights='imagenet')
    feature_extractor_model = tf.keras.Model(vgg19_model.input, vgg19_model.layers[-1].output)
    print("Image model ready")
    return feature_extractor_model
```

The features from each image were saved as NumPy files, which allows for efficient storage and accessibility.

Batch Processing

To handle the large volume of images efficiently, the processing was conducted in batches:

```
image_dataset = tf.data.Dataset.from_tensor_slices(unique_paths)
image_dataset = image_dataset.map(process_img_data, num_parallel_calls=tf.data.experimental.AUTOTUNE).batch(64)
```

Results

The feature extraction process was successfully completed for 82,783 training images, ensuring a comprehensive dataset for training the VQA model. The features were stored in a structured manner to facilitate easy integration into the training pipeline.

Challenges Encountered
Handling Large Datasets: Managing the high volume of data required robust computational resources and efficient data handling strategies.

System Performance: Ensuring that the system performed optimally under the load of processing tens of thousands of images was crucial.

Step-4 Modelling:

Data Preparation

The dataset comprises image features extracted and stored as numpy files from the COCO dataset, alongside corresponding questions and answers. Image features were processed using a pre-trained VGG19 model to capture the visual context. The questions and answers were vectorized using embeddings and one-hot encoding respectively, prepared through custom-built vocab classes and tokenizer functions.

Model Architecture
Two model architectures were experimented with:

RNN-Based Model:
- **Image Model:** Flatten layer followed by a dense layer (512 units, 'tanh' activation).
- **Question Model:** Embedding layer followed by a Simple RNN layer (512 units).
- **Output:** Combined using multiplication, followed by a dense layer with softmax activation for classification.

LSTM-Based Model:
- **Image Model:** Similar to the RNN model.
- **Question Model:** Embedding layer followed by an LSTM layer where both the hidden state and cell state are used.
- **Output**: Similar to the RNN model but uses additional dropout layers.

The models were compiled with the Adam optimizer and categorical cross-entropy loss function, aimed at multi-class classification of the answers.

The decision to choose RNN-based and LSTM-based models for the Visual Question Answering (VQA) project primarily stems from the specific needs of processing sequential data and the inherent characteristics of the problem domain, which involves interpreting and correlating features from both visual (images) and textual (questions) inputs. Here's a detailed explanation of why these models were selected:

1. RNN-Based Model:

a. Handling Sequential Data:

Recurrent Neural Networks (RNNs) are fundamentally designed to work with sequences - whether it be text (questions) or time-series data. In the context of VQA, questions are inherently sequential, where the meaning depends significantly on the order of the words. RNNs process these sequences iteratively, maintaining an internal state that encapsulates information about the sequence processed so far, making them suitable for this aspect of VQA.

b. Integration of Image and Text Data:

The architecture involving a Flatten layer for images and an RNN for questions allows for a straightforward but effective integration strategy. After flattening the spatial features extracted by a pre-trained VGG19 from images, the network passes it through dense layers to create a feature vector. This vector effectively condenses the image information into a form that can be combined with the encoded question data, processed through the RNN, ensuring that the model can correlate features from both modalities.

c. Output Mechanism:

Using a multiplication (or sometimes concatenation) layer to combine the outputs from the image and text processing branches before the final classification allows the model to learn correlations between specific visual features and textual queries, which is crucial for generating accurate answers based on both context clues.

2. LSTM-Based Model:

a. Advanced Sequence Processing:

Long Short-Term Memory networks (LSTMs) are an extension of RNNs that are better at capturing long-range dependencies in sequence data. This capability is particularly beneficial for VQA where the question might be complex, and understanding context is crucial for accurately interpreting the image in light of the question. The LSTM helps in retaining important information throughout longer sequences, thus mitigating the vanishing gradient problem common in standard RNNs.

b. Utilizing Hidden and Cell States:

LSTMs manage two state vectors (the hidden state and the cell state), which provide a richer representation of the data as it flows through the network. By utilizing both states in the VQA model, the architecture harnesses a deeper understanding and a more nuanced memory of both past visual and textual data, enhancing the model's ability to predict more accurate answers.

c. Robustness with Dropout:
Incorporating dropout layers in the LSTM-based model helps in preventing overfitting. This is especially useful in VQA tasks where the model complexity is high due to the multimodal nature of the input data. Dropout ensures that the model remains generalizable and performs well not just on the training data but also on unseen validation data.

Rationale Behind Model Choices:
Both RNNs and LSTMs offer frameworks for effectively integrating diverse data types inherent to VQA (images and text) and align well with the sequential processing required for text. The choice to experiment with both architectures allows for comparative analysis on how each model handles the complexities of VQA tasks, specifically in terms of learning dependencies within the data and managing the integration of learned features from both domains to generate accurate responses.

Training and Validation

Models were trained using data preprocessed into feature sets of image data, question data, and answer labels. Training involved multiple epochs with batch processing to optimize performance and manage computational resources efficiently.

Key Observations:

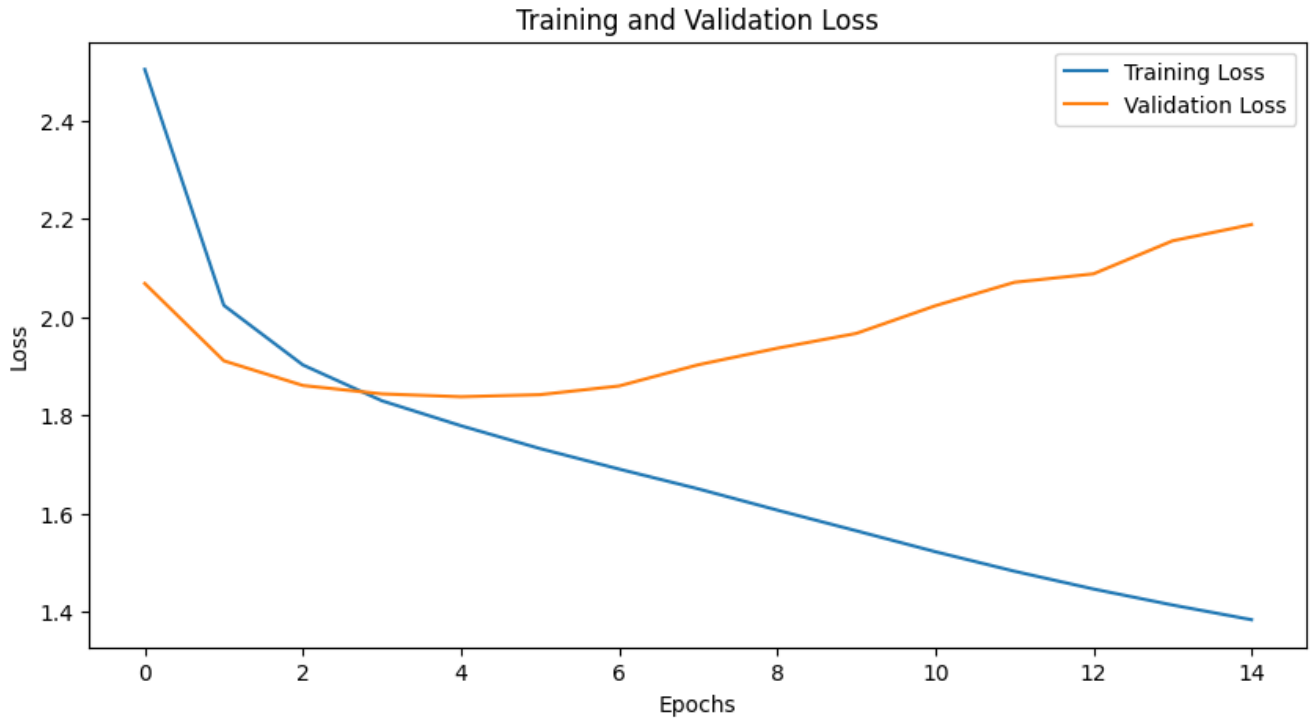
- The LSTM model showed better handling of sequential data, reflecting in a higher accuracy and stability during training compared to the RNN model.
- The RNN model struggled with performance, likely due to the vanishing gradient problem common in simple RNNs, which was evident from the fluctuating loss and accuracy rates.
- Both models showed signs of overfitting as the number of epochs increased, indicated by increasing validation loss despite improving training loss.

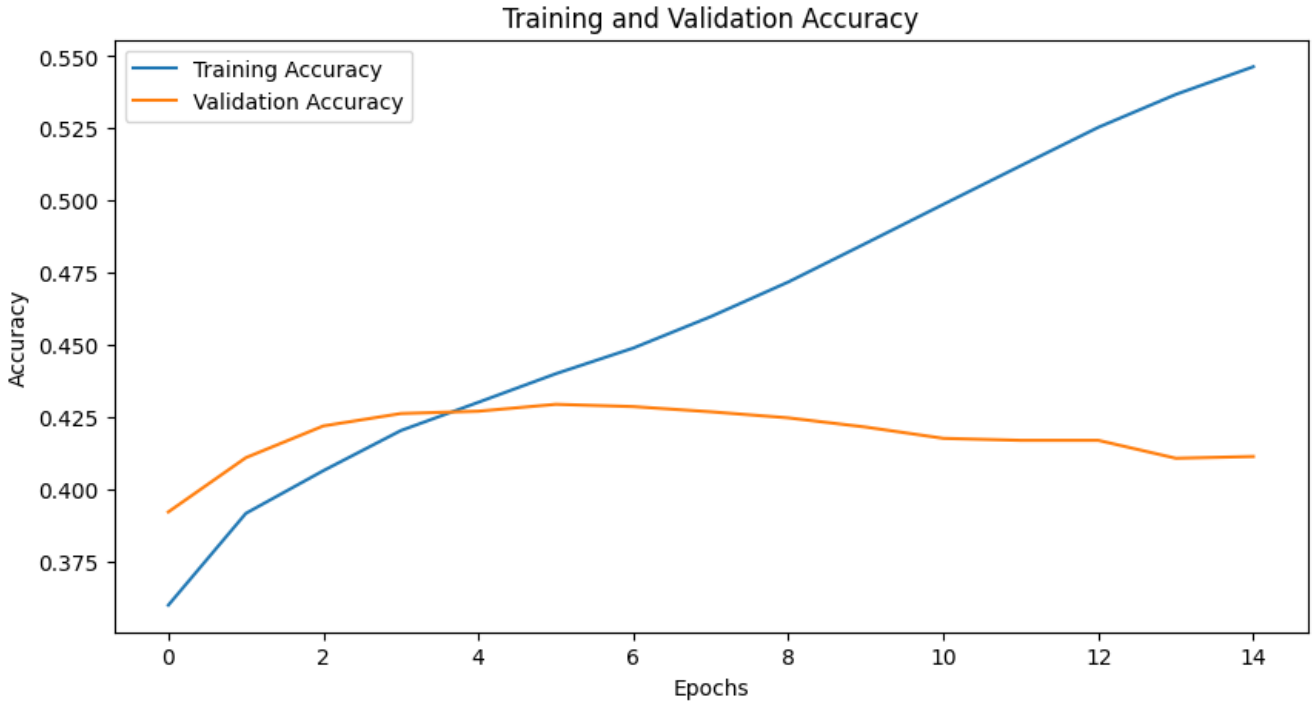
Evaluation

Model performance was evaluated based on the accuracy metric and the loss during both training and validation phases. It was observed that:

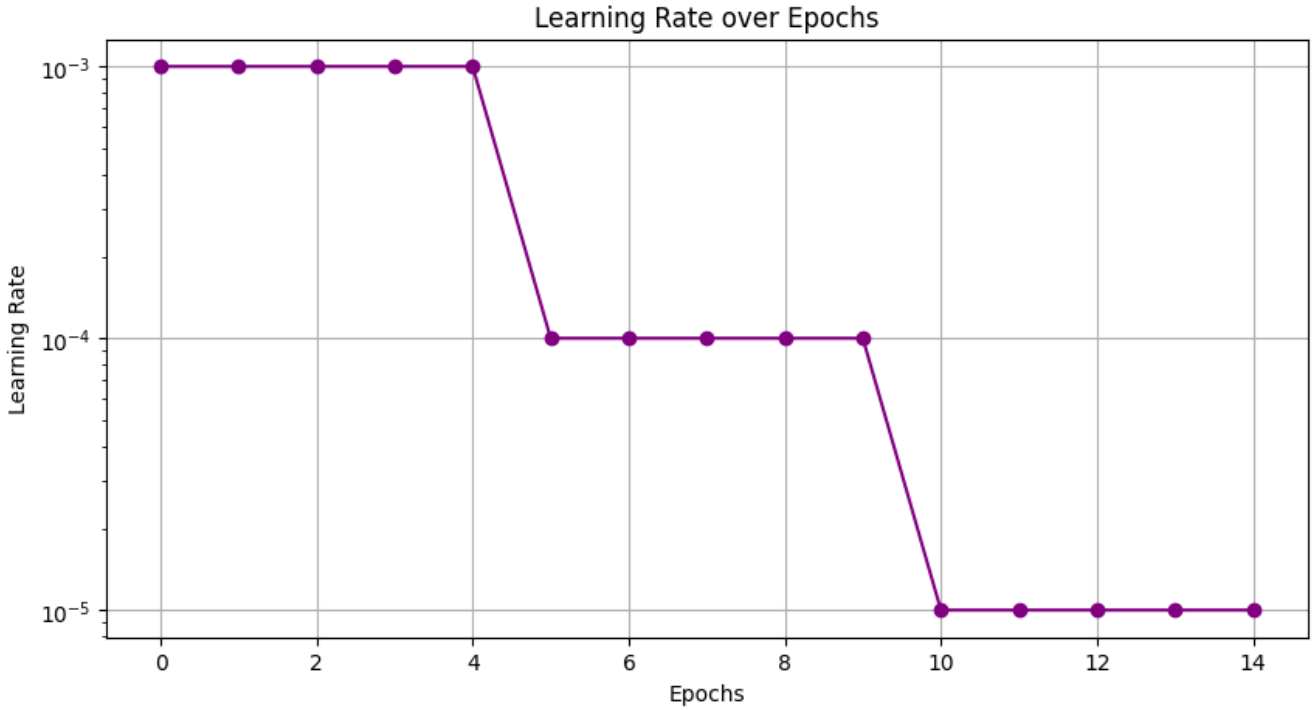
- The LSTM model generally outperformed the RNN model in terms of accuracy and stability.
- Performance on the validation set was suboptimal for both models, suggesting potential overfitting.

Plots:




Training and Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy
0.550
0.525
0.500
0.475
0.450
0.425
0.400
0.375
0
2
4
6
8
10
12
14
Epochs


Rate of Change in Training Loss
Rate of Change in Loss
−0.1
−0.2
−0.3
−0.4
−0.5
0
2
4
6
8
10
12
Epochs


Learning Rate over Epochs
Learning Rate
10^{-3}
10^{-4}
10^{-5}
0
2
4
6
8
10
12
14
Epochs

Results:

```
[16]  img = load_image('/content/imgtest1.jpg')
      que = load_question('what is this')

      # Resize and cast data as required by the input_details
      img = tf.expand_dims(img, axis=0).numpy().astype(input_details[1]['dtype'])
      que = tf.expand_dims(que, axis=0).numpy().astype(input_details[0]['dtype'])
```

```
[17]  interpreter.set_tensor(input_details[0]['index'], que)
      interpreter.set_tensor(input_details[1]['index'], img)
```

```
[18]  interpreter.invoke()
```

```
[19]  output_data = interpreter.get_tensor(output_details[0]['index'])
      predicted_label_idx = tf.argmax(output_data, axis=1).numpy()[0]
      predicted_label = answer_vocab.idx2word(predicted_label_idx)
```

```
      print(f"Predicted answer: {predicted_label}")
```

```
Predicted answer: laptop
```

```
[37]  img = load_image('/content/testimg2.jpeg')
      que = load_question('what is the object?')

      # Resize and cast data as required by the input_details
      img = tf.expand_dims(img, axis=0).numpy().astype(input_details[1]['dtype'])
      que = tf.expand_dims(que, axis=0).numpy().astype(input_details[0]['dtype'])
```

```
[38]  interpreter.set_tensor(input_details[0]['index'], que)
      interpreter.set_tensor(input_details[1]['index'], img)
```

```
[39]  interpreter.invoke()
```

```
[40]  output_data = interpreter.get_tensor(output_details[0]['index'])
      predicted_label_idx = tf.argmax(output_data, axis=1).numpy()[0]
      predicted_label = answer_vocab.idx2word(predicted_label_idx)
```

```
      print(f"Predicted answer: {predicted_label}")

Predicted answer: trees
```


```
[43]  img = load_image('/content/pic.jpg')
      que = load_question('what colour is the shirt?')

      # Resize and cast data as required by the input_details
      img = tf.expand_dims(img, axis=0).numpy().astype(input_details[1]['dtype'])
      que = tf.expand_dims(que, axis=0).numpy().astype(input_details[0]['dtype'])
```

```
[44]  interpreter.set_tensor(input_details[0]['index'], que)
      interpreter.set_tensor(input_details[1]['index'], img)
```

```
[45]  interpreter.invoke()
```

```
[46]  output_data = interpreter.get_tensor(output_details[0]['index'])
      predicted_label_idx = tf.argmax(output_data, axis=1).numpy()[0]
      predicted_label = answer_vocab.idx2word(predicted_label_idx)
```

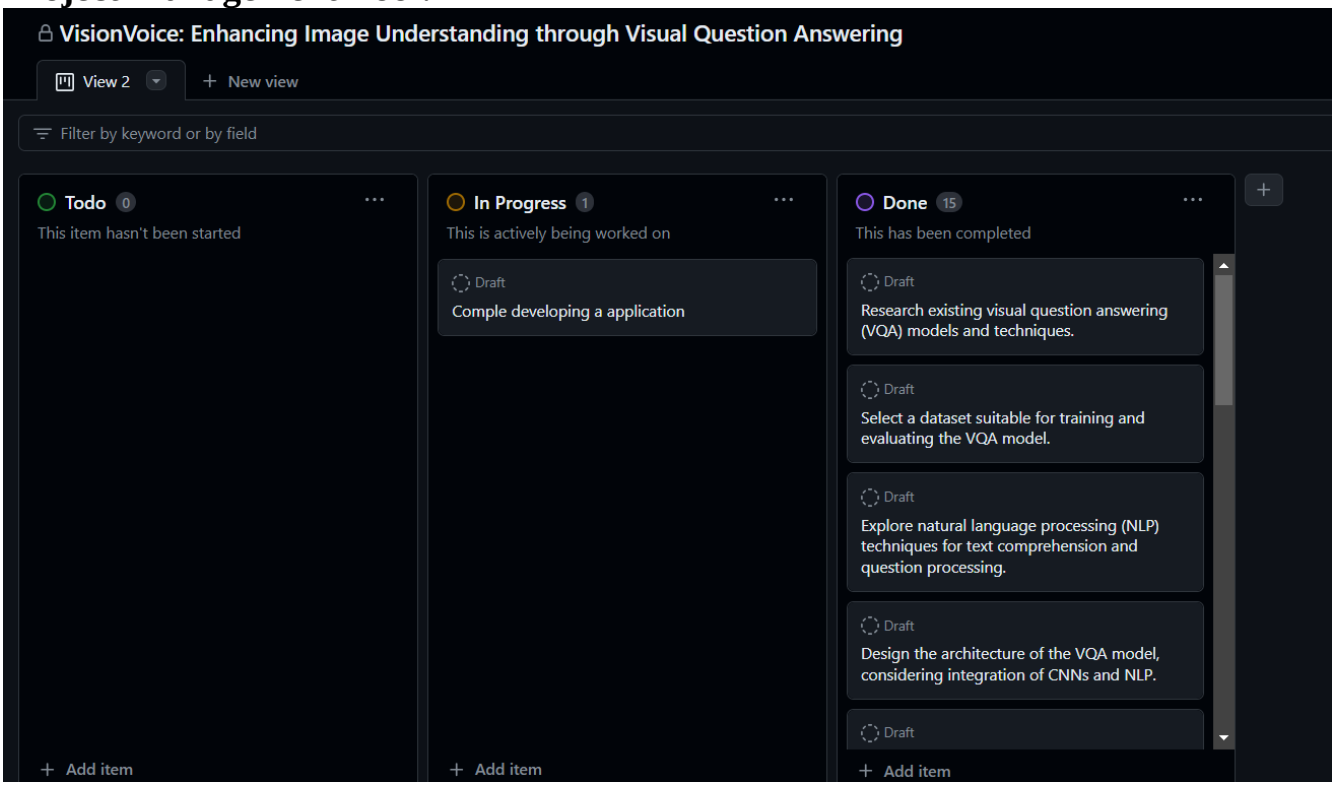
```
      print(f"Predicted answer: {predicted_label}")
```

```
Predicted answer: red and yellow
```

Real-world deep learning application:

Visual Question Answering (VQA) is a transformative deep learning application that merges computer vision and natural language processing to answer questions about images. This technology enhances accessibility for the visually impaired by providing verbal answers to visual questions, supporting interactive learning tools in education, and improving customer engagement in retail by enabling intuitive product queries. The project utilizes real-world images from comprehensive datasets like Visual Genome and COCO, ensuring the model's effectiveness across diverse real-life scenarios. By focusing on VQA, this deep learning initiative directly contributes to practical advancements in AI, making digital content more accessible and interactive across various sectors.

Project Management Tool:



Project Files:https://drive.google.com/drive/folders/1tblAlnADpXt3-dYrBzoT53p5o82G3MwM?usp=sharing

References:

1. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network https://arxiv.org/abs/1808.03314
2. Long short-term memory (LSTM) recurrent neural network for muscle activity detection https://jneuroengrehab.biomedcentral.com/articles/10.1186/s12984-021-00945-w
3. Understanding LSTM -- a tutorial into Long Short-Term Memory Recurrent Neural Networks https://arxiv.org/abs/1909.09586
4. Long short term memory (LSTM) recurrent neural network (RNN) for discharge level prediction and forecast in Cimandiri river, Indonesia https://iopscience.iop.org/article/10.1088/1755-1315/299/1/012037