FLOWER CLASSIFICATION

Team Members

MEGHA SAXENA(20BIT0366) - VIT VELLORE

SALONI AGARWAL(20BIT0389) - VIT VELLORE

Harshit K (20BCR7051) - VIT AP

Adarsh Rajesh (20BCE7215) - VIT AP

INTRODUCTION

In today's digital age, there is an abundance of visual information available in the form of images. Extracting meaningful text from images has become a crucial task in various domains, such as document analysis, automatic captioning, and image indexing. The advancements in deep learning and computer vision have led to the development of Convolutional Neural Networks (CNNs) as a powerful tool for image-to-text conversion.
In this project, we aim to develop an image caption generation system using CNNs. Our goal is to automatically generate accurate and meaningful captions that describe the content of an image. By leveraging the power of deep learning and the vast amounts of labeled image-text data, we can train a CNN-based model to learn the complex relationships between images and their corresponding textual descriptions. To train our model, we will leverage an extensive dataset of paired images and their corresponding captions. This dataset will be used to fine-tune a pre-trained CNN, such as the popular architectures like VGGNet or ResNet, by mapping images to their extracted features.
To accomplish this, we will leverage a deep learning framework to build and train our image captioning model. The model will be trained on a large-scale dataset containing images paired with corresponding human-generated captions. During the training process, the model will learn to associate the visual features extracted by the CNN with the textual information, enabling it to generate meaningful captions for unseen images.

Convolutional Neural Networks (CNNs) play a crucial role in image-to-text conversion tasks, such as optical character recognition (OCR) and image captioning. The purpose of using CNNs in image-to-text conversion is to extract meaningful and relevant information from images and transform it into textual representations.CNNs excel at analyzing visual data due to their unique architecture, which includes convolutional layers that perform feature extraction and pooling layers that downsample the extracted features. This hierarchical approach allows CNNs to capture spatial relationships and detect patterns in images effectively.

In OCR applications, CNNs are employed to recognize and extract text from images, enabling the conversion of printed or handwritten text into editable or searchable formats. By learning from large amounts of labeled image data, CNNs can identify individual characters or words and accurately transcribe them into text. Overall, CNNs are essential in image-to-text conversion as they enable the extraction of meaningful visual features, leading to accurate and reliable conversion of images into textual representations.

LITERATURE SURVEY

While Recurrent Neural Networks (RNNs) are commonly used in image-to-text conversion tasks like image captioning, they also have some limitations and challenges. RNNs process data sequentially, meaning they generate captions word by word in a sequential manner. This sequential processing can be computationally expensive and time-consuming, especially when dealing with long sequences or large datasets. It also limits parallelization and inhibits efficient hardware utilization. RNNs typically have limited contextual understanding and struggle to capture global dependencies within images.

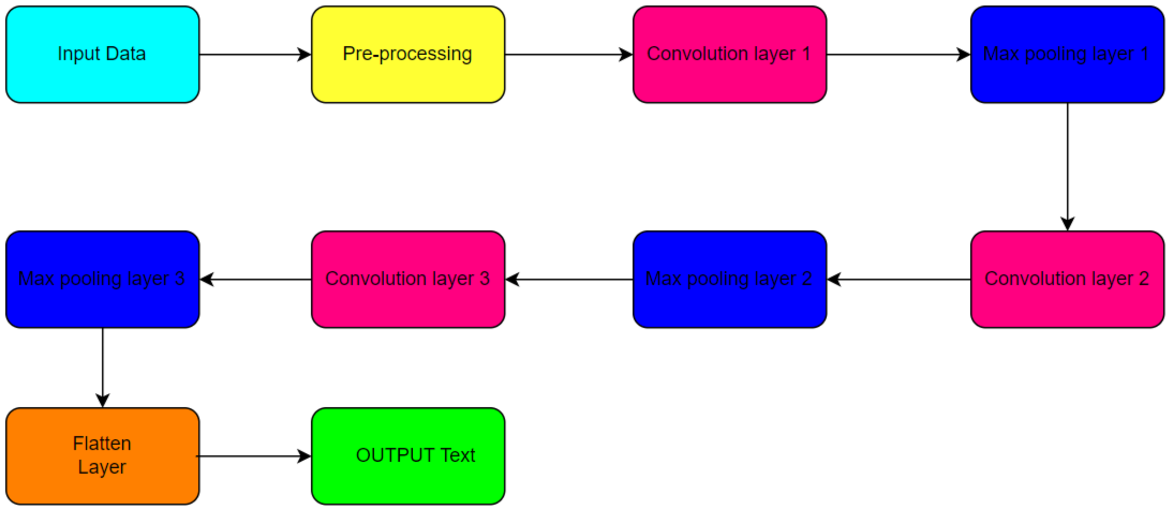
While they can model short-term dependencies effectively, they may fail to grasp the overall context and semantic relationships present in complex images. This can result in captions that are less coherent or fail to accurately describe the visual content. RNNs typically rely on pre-extracted visual features from Convolutional Neural Networks (CNNs) as inputs. While CNNs excel at extracting visual features, the separation of feature extraction (CNN) and sequential generation (RNN) can limit the ability to jointly learn visual and textual representations.

This separation may lead to a disconnect between the visual and textual modalities and can hinder the generation of precise and contextually grounded captions. Researchers and practitioners have made efforts to address some of these challenges by exploring alternative architectures, such as attention mechanisms and transformer-based models, which aim to improve contextual understanding and alleviate the sequential processing limitations of RNNs. However, these problems highlight the ongoing research and development needed to enhance the performance and capabilities of RNNs in image-to-text conversion tasks.

CNNs have been extensively studied and optimized for image classification, object detection, and other computer vision tasks. As a result, CNN-based models have achieved state-of-the-art performance on various image-related challenges, including image-to-text conversion tasks like captioning. RNNs, while effective for sequential processing and language-related tasks, have shown limitations in capturing visual information and generating accurate and contextually grounded captions.

It is worth noting that CNNs and RNNs are not mutually exclusive, and they are often combined in hybrid models for image-to-text conversions, such as the popular CNN-RNN architectures used in image captioning. However, CNNs form the foundational component for visual feature extraction in these models, highlighting their superiority in the initial stages of image-to-text conversion.

THEORETICAL ANALYSIS



When developing a Convolutional Neural Network (CNN), you typically need a specific set of software requirements. Here are the key requirements for building and training a CNN:

Python: CNNs are commonly implemented using Python due to its extensive libraries and frameworks for machine learning. Install Python and set up a suitable development environment, such as Anaconda or Miniconda.

Deep Learning Libraries: You will need deep learning libraries that provide high-level abstractions for building and training CNNs. The most popular libraries include:

TensorFlow: An open-source library developed by Google.
PyTorch: A popular deep-learning library maintained by Facebook.
Keras: A user-friendly, high-level API that can run on top of TensorFlow or other backends.
MXNet: A flexible deep learning framework with efficient GPU support.
Choose the library that best suits your needs and install it via the respective documentation.

Data Manipulation and Preprocessing: Python provides various libraries for data manipulation and preprocessing, which are essential for preparing your data before feeding it into a CNN. Popular libraries include NumPy, Pandas, and Scikit-learn.

Image Processing Libraries: CNNs often deal with image data, so you might need libraries for image processing and augmentation. OpenCV and Pillow are widely used libraries for image loading, transformation, and manipulation.

Google Colab or IDE: Choose an integrated development environment (IDE) or a notebook environment to write and execute your CNN code. Jupyter Notebook, JupyterLab, PyCharm, Google Colab, or Visual Studio Code are popular options.

Additional Libraries: Depending on your specific requirements, you might need additional libraries such as matplotlib for visualization, h5py for saving and loading model weights, or tqdm for progress bars during training.

It's important to note that these requirements can vary based on your project's specifications and the specific libraries or frameworks you choose. Make sure to refer to the documentation of the libraries you're using for detailed installation instructions and version compatibility.

EXPERIMENTAL INVESTIGATIONS

The provided code involves several steps, from importing libraries to deploying a model using Streamlit. Here's a detailed explanation of the entire process:

Importing Libraries:
The necessary libraries are imported, including TensorFlow and Streamlit. TensorFlow is a popular deep-learning framework, while Streamlit is a Python library used for building interactive web applications.

Dataset Upload and Flower Dataset Download:
The code assumes that a flower dataset is downloaded from the internet directly.
Creating the Training Dataset:
The code creates a training dataset for image classification using TensorFlow. It loads images from a directory, resizes them to 180x180 pixels, and splits them into training and validation sets. The training dataset is returned with a batch size of 32, and 20% of the data is used for validation and remaining 80% used for training.

Visualizing a Subset of Images:
A code snippet visualizes a subset of images from the training dataset. It creates a 3x3 grid of subplots and displays the first 9 images along with their corresponding labels. The imshow function is used to display each image, and the class names are shown as titles.

Creating a Sequential Model:
The code creates a sequential model for image classification using TensorFlow's Sequential API. The model consists of multiple layers stacked together in a sequential manner.

The layers.experimental.preprocessing.Rescaling layer scales the pixel values of the input images by dividing them by 255, normalizing the pixel values to a range between 0 and 1.
The layers.Conv2D layers apply convolutional filters to extract features from the input image. Multiple convolutional layers with increasing filter sizes (16, 32, and 64) and ReLU activation are used.

The layers.MaxPooling2D layers perform max pooling to downsample the feature maps.
The layers.Flatten layer flattens the 2D feature maps into a 1D vector.
The layers.Dense layers are fully connected layers. The model has a dense layer with 128 units and ReLU activation to capture high-level patterns, followed by a final dense layer with the number of classes units (num_classes). The softmax activation function is used in the last layer to produce class probabilities.
Compiling and Training the Model:
The model is compiled using the model.compile function. It specifies the optimizer (Adam), the loss function (SparseCategoricalCrossentropy), and the accuracy metric. The model is configured for training.
The model is then trained using the model.fit function. It takes the training dataset and the validation dataset as inputs and performs the specified number of epochs (10 in this case). During training, the model's parameters are updated based on the optimization algorithm, and loss and accuracy metrics are calculated. The training progress is stored in the history object.

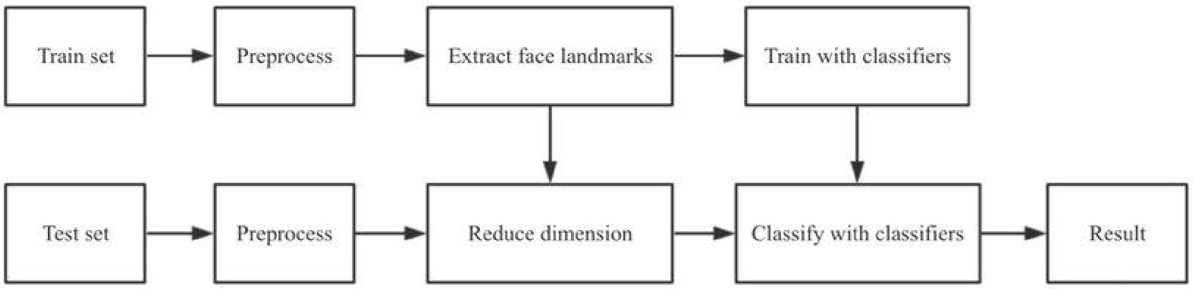
Saving the Model:
The code does not explicitly mention saving the model's architecture and weights. It might be missing from the provided snippet.

Streamlit Application:
The code also includes a Streamlit application for flower classification using a pre-trained model. The application allows users to upload an image file, preprocesses the image, and makes predictions using the loaded model.

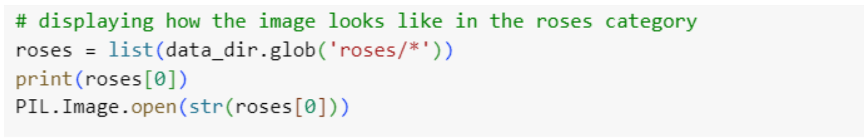
A function called load_model() is defined to load the pre-trained model using TensorFlow's load_model() function. The loaded model is cached using the @st.cache decorator to reduce the reload time.
The Streamlit application starts with a main heading "Flower Classification" using st.write().
An uploader component is added using st.file_uploader() to enable users to upload an image file (accepting "jpg" and "png" file types).
Inside an if statement, the import_and_predict() function is defined. It preprocesses the uploaded image by resizing it and converting it to a NumPy array. The preprocessed image is passed to the loaded model's predict() function, and the prediction result is returned.
If an image file is uploaded, it is displayed in the Streamlit app using st.image().
The import_and_predict() function is called with the uploaded image and the loaded model. The prediction result is stored in the predictions variable.
The prediction scores are converted to probabilities using the softmax function (tf.nn.softmax()), and the predicted class and corresponding confidence score are displayed using st.write().
Finally, the predicted class and confidence score are printed to the console using print().
ngrok Setup and Deployment:
The provided code includes commands to set up ngrok, a tool used to create a secure tunnel to the local machine. The ngrok tunnel allows accessing the Streamlit app remotely.
The command !wget downloads the ngrok binary for Linux from a specific URL.
The command !unzip extracts the downloaded ngrok binary.

The command get_ipython().system_raw('./ngrok http 8501 &') starts a background ngrok process that establishes a secure tunnel to the local machine's port 8501, which is the default port used by Streamlit.
The command !curl fetches the ngrok API endpoints, extracts the public URL of the ngrok tunnel, and prints it. This URL can be used to access the Streamlit app remotely.
Running the Streamlit App:
The command !streamlit run /content/app.py runs the Streamlit app located at /content/app.py.
The app starts and is served on the local machine at http://localhost:8501.
By following these steps, the code imports the necessary libraries, prepares the dataset, creates and trains a sequential model for image classification, sets up a Streamlit application for deploying the model, and utilizes ngrok to access the app remotely

FLOWCHART

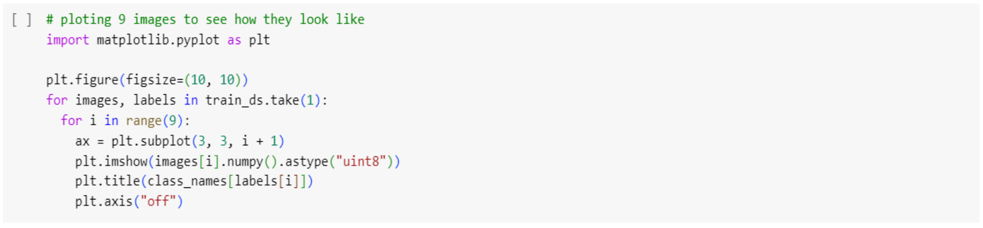


RESULT



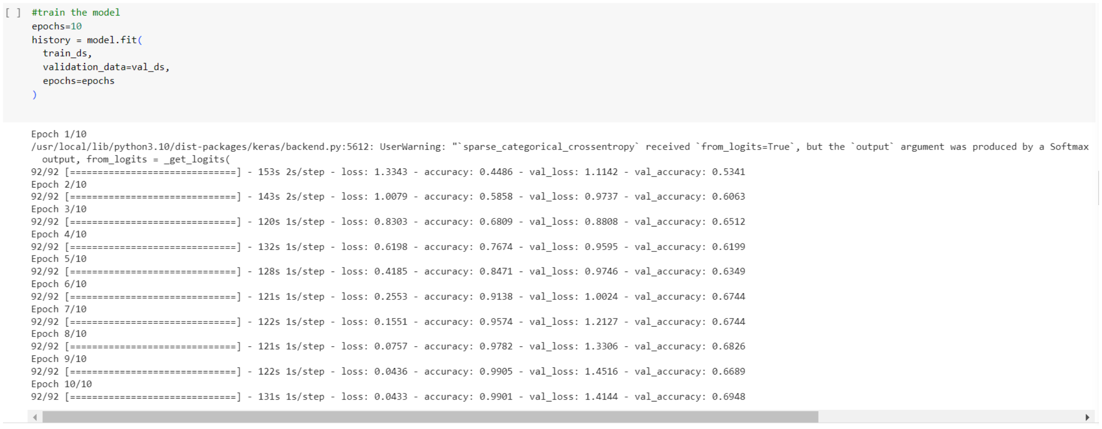
```
# displaying how the image looks like in the roses category
roses = list(data_dir.glob('roses/*'))
print(roses[0])
PIL.Image.open(str(roses[0]))
```

```
/root/.keras/datasets/flower_photos/roses/15424480096_45bb574b33.jpg
```



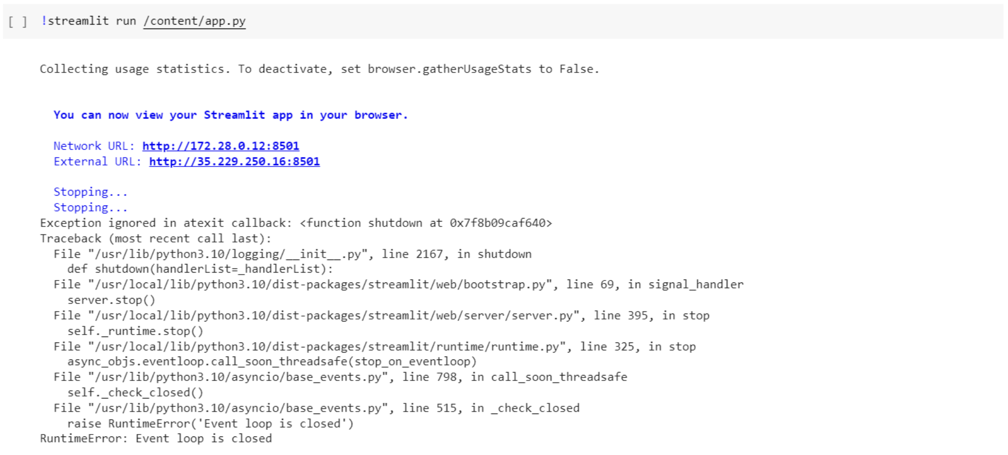
```
[ ] # ploting 9 images to see how they look like
    import matplotlib.pyplot as plt

    plt.figure(figsize=(10, 10))
    for images, labels in train_ds.take(1):
      for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title(class_names[labels[i]])
        plt.axis("off")
```



```
[ ] #train the model
    epochs=10
    history = model.fit(
      train_ds,
      validation_data=val_ds,
      epochs=epochs
    )

    Epoch 1/10
    /usr/local/lib/python3.10/dist-packages/keras/backend.py:5612: UserWarning: "`sparse_categorical_crossentropy` received `from_logits=True`, but the `output` argument was produced by a Softmax
      output, from_logits = _get_logits(
    92/92 [==============================] - 153s 2s/step - loss: 1.3343 - accuracy: 0.4486 - val_loss: 1.1142 - val_accuracy: 0.5341
    Epoch 2/10
    92/92 [==============================] - 143s 2s/step - loss: 1.0079 - accuracy: 0.5858 - val_loss: 0.9737 - val_accuracy: 0.6063
    Epoch 3/10
    92/92 [==============================] - 120s 1s/step - loss: 0.8303 - accuracy: 0.6809 - val_loss: 0.8808 - val_accuracy: 0.6512
    Epoch 4/10
    92/92 [==============================] - 132s 1s/step - loss: 0.6198 - accuracy: 0.7674 - val_loss: 0.9595 - val_accuracy: 0.6199
    Epoch 5/10
    92/92 [==============================] - 128s 1s/step - loss: 0.4185 - accuracy: 0.8471 - val_loss: 0.9746 - val_accuracy: 0.6349
    Epoch 6/10
    92/92 [==============================] - 121s 1s/step - loss: 0.2553 - accuracy: 0.9138 - val_loss: 1.0024 - val_accuracy: 0.6744
    Epoch 7/10
    92/92 [==============================] - 122s 1s/step - loss: 0.1551 - accuracy: 0.9574 - val_loss: 1.2127 - val_accuracy: 0.6744
    Epoch 8/10
    92/92 [==============================] - 121s 1s/step - loss: 0.0757 - accuracy: 0.9782 - val_loss: 1.3306 - val_accuracy: 0.6826
    Epoch 9/10
    92/92 [==============================] - 122s 1s/step - loss: 0.0436 - accuracy: 0.9905 - val_loss: 1.4516 - val_accuracy: 0.6689
    Epoch 10/10
    92/92 [==============================] - 131s 1s/step - loss: 0.0433 - accuracy: 0.9901 - val_loss: 1.4144 - val_accuracy: 0.6948
```



```
[ ] !streamlit run /content/app.py

    Collecting usage statistics. To deactivate, set browser.gatherUsageStats to False.


      You can now view your Streamlit app in your browser.

      Network URL: http://172.28.0.12:8501
      External URL: http://35.229.250.16:8501

      Stopping...
      Stopping...
    Exception ignored in atexit callback: <function shutdown at 0x7f8b09caf640>
    Traceback (most recent call last):
      File "/usr/lib/python3.10/logging/__init__.py", line 2167, in shutdown
        def shutdown(handlerList=_handlerList):
      File "/usr/local/lib/python3.10/dist-packages/streamlit/web/bootstrap.py", line 69, in signal_handler
        server.stop()
      File "/usr/local/lib/python3.10/dist-packages/streamlit/web/server/server.py", line 395, in stop
        self._runtime.stop()
      File "/usr/local/lib/python3.10/dist-packages/streamlit/runtime/runtime.py", line 325, in stop
        async_objs.eventloop.call_soon_threadsafe(stop_on_eventloop)
      File "/usr/lib/python3.10/asyncio/base_events.py", line 798, in call_soon_threadsafe
        self._check_closed()
      File "/usr/lib/python3.10/asyncio/base_events.py", line 515, in _check_closed
        raise RuntimeError('Event loop is closed')
    RuntimeError: Event loop is closed
```

ADVANTAGES

- Local feature extraction→ CNNs excel at capturing local spatial patterns in images. By using convolutional layers, they can learn to detect low-level features like edges, corners, and textures, which are crucial for understanding the content of an image. This ability makes CNNs effective in recognizing and localizing important visual elements that can be translated into meaningful text.

- Hierarchical representation→ CNNs can learn to extract features at multiple levels of abstraction. As the network gets deeper, higher-level features are learned, which capture more complex and abstract visual concepts. This hierarchical representation aligns well with the hierarchical structure of language, enabling CNNs to generate more semantically meaningful textual descriptions.
- Parameter sharing and translation invariance → CNNs leverage parameter sharing to reduce the number of learnable parameters, making them more efficient and scalable for image processing tasks. Additionally, CNNs are translation invariant, meaning that they can recognize the same pattern regardless of its location in the image. This property is valuable when converting images to text, as the textual description should be independent of the image's position or orientation.

DISADVANTAGES

- Lack of semantic understanding → CNNs primarily focus on local patterns and features within an image. While this can be effective for tasks like object recognition, it may not capture the overall semantic meaning of the image. Consequently, the generated text may lack a deep understanding of the image context.
- Difficulty with complex scenes → CNNs can struggle when faced with complex scenes containing multiple objects, occlusions, or cluttered backgrounds. They may fail to accurately identify and describe all relevant elements in the image, leading to incomplete or incorrect text generation.
- Sensitivity to image variations → CNNs are sensitive to various factors such as lighting conditions, scale, rotation, and viewpoint. Small changes in these factors can significantly affect the network's performance, leading to inaccurate or inconsistent text output.

APPLICATIONS

- Image Annotation →CNNs can be used to annotate images with descriptive tags or keywords. By training a CNN on a large dataset of labeled images, the network can learn to associate visual features with specific textual labels. Image annotation finds applications in content management systems, image search engines, and organizing large image databases.
- Image-to-Text Translation → CNNs can be utilized to translate images into natural language descriptions or instructions. For example, an image of a traffic sign could be translated into the corresponding text, such as "Stop" or "Yield." This has applications in areas like augmented reality, real-time image understanding, and automatic translation systems.
- Visual Sentiment Analysis → CNNs can be employed to analyze and understand the sentiment expressed in images. By training a CNN on a dataset of images labeled with the sentiment, the network can learn to classify images into different emotional categories, such as happy, sad, or angry. This has applications in social media analysis, market research, and sentiment-aware content filtering.

CONCLUSION
In conclusion, Convolutional Neural Networks (CNNs) have proven to be highly effective in image-to-text conversion tasks. CNNs excel at extracting meaningful features from images and can learn to recognize patterns, shapes, and objects within them. When applied to image-to-text conversion, CNNs are able to analyze and understand the visual content of an image and generate corresponding textual descriptions.
The use of CNNs in image-to-text conversion has led to significant advancements in various domains, such as computer vision, natural language processing, and accessibility. CNN-based models have demonstrated remarkable performance in tasks like image captioning, visual question answering, and generating textual descriptions for visually impaired individuals.
However, it's important to note that CNNs are not without limitations. They require large amounts of labeled training data and significant computational resources to train and fine-tune. Additionally, CNNs may struggle with handling certain types of images or challenging scenarios where objects are occluded, rotated, or appear in cluttered backgrounds.
Nonetheless, the continuous advancements in CNN architectures, such as the introduction of attention mechanisms and transformer models, have further improved the performance of image-to-text conversion systems. Overall, CNNs have revolutionized image-to-text conversion by bridging the gap between visual and textual information, opening up new possibilities for applications in areas like image indexing, content summarization, and assistive technologies.
With further advancements and refinements, CNN-based models hold great promise for enhancing our interactions with visual data and enabling.

FUTURE SCOPE
CNNs can be further improved to generate more accurate and contextually relevant captions for images. Future research may focus on developing models that better understand image semantics, relationships, and context to produce more informative and creative captions. CNNs have shown significant advancements in OCR tasks. However, there is still room for improvement in terms of accuracy, especially in challenging scenarios with low-quality images, different languages, and complex fonts. Future research may focus on developing robust CNN architectures that can handle such variations and improve OCR performance. CNNs have made significant progress in improving the efficiency and speed of image processing tasks. Future developments may focus on optimizing CNN architectures and designing efficient algorithms to enable real-time image-to-text conversion, which can be valuable in applications like live captioning, augmented reality, and assistive technologies.

Overall, the future scope for CNNs in image-to-text conversion is broad and encompasses areas such as improved accuracy, multimodal understanding, scene reasoning, OCR advancements, zero-shot learning, and real-time processing. Continued research and innovation in these areas will further enhance the capabilities of CNNs in converting images to text.

BIBLIOGRAPHY

1. Karpathy, A., & Fei-Fei, L. (2015). Deep visual-semantic alignments for generating image descriptions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 3128-3137). DOI: 10.1109/CVPR.2015.7298932
2. Vinyals, O., Toshev, A., Bengio, S., & Erhan, D. (2015). Show and tell: A neural image caption generator. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 3156-3164). DOI: 10.1109/CVPR.2015.7298935
3. Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhudinov, R., ... & Bengio, Y. (2015). Show, attend and tell: Neural image caption generation with visual attention. In International Conference on Machine Learning (ICML) (pp. 2048-2057).
4. Donahue, J., Hendricks, L. A., Guadarrama, S., Rohrbach, M., Venugopalan, S., Saenko, K., & Darrell, T. (2015). Long-term recurrent convolutional networks for visual recognition and description. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 2625-2634). DOI: 10.1109/CVPR.2015.7298890
5. Mao, J., Xu, W., Yang, Y., Wang, J., Huang, Z., & Yuille, A. L. (2014). Deep captioning with multimodal recurrent neural networks (m-RNN). In International Conference on Learning Representations (ICLR).
6. Karpathy, A., Joulin, A., & Fei-Fei, L. (2014). Deep fragment embeddings for bidirectional image sentence mapping. Advances in Neural Information Processing Systems (NIPS), 26, 1889-1897.
7. Kiros, R., Salakhutdinov, R., & Zemel, R. S. (2014). Unifying visual-semantic embeddings with multimodal neural language models. In Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI).
8. Chen, X., Fang, H., Lin, T. Y., Vedantam, R., Gupta, S., Dollár, P., & Zitnick, C. L. (2015). Microsoft COCO captions: Data collection and evaluation server. arXiv preprint arXiv:1504.00325.
9. Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster R-CNN: Towards real-time object detection with region proposal networks. In Advances in Neural Information Processing Systems (NIPS), 28, 91-99.
10. Wang, J., Yang, Y., Mao, J., Huang, Z., Huang, C., & Xu, W. (2016). CNN-RNN: A unified framework for multi-label image classification. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 2285-2294). DOI: 10.1109/CVPR.2016.252

APPENDIX A.

```
# import libraries
import matplotlib.pyplot as plt
import numpy as np
import os
```

```python
import PIL
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
```

```python
# dataset upload
import pathlib
dataset_url =
"https://storage.googleapis.com/download.tensorflow.org/example_images/flo
wer_photos.tgz"
data_dir = tf.keras.utils.get_file('flower_photos', origin=dataset_url,
untar=True)
data_dir = pathlib.Path(data_dir)
```

```python
# displaying how the image looks like in the roses category
roses = list(data_dir.glob('roses/*'))
print(roses[0])
PIL.Image.open(str(roses[0]))
# spliting data into training subsets
img_height,img_width=180,180
batch_size=32
train_ds = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
# validation subset
val_ds = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
# different class names in dataset
class_names = train_ds.class_names
```

```
print(class_names)
# ploting 9 images to see how they look like
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
  for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(images[i].numpy().astype("uint8"))
    plt.title(class_names[labels[i]])
    plt.axis("off")
#model
num_classes = 5

model = Sequential([
  layers.experimental.preprocessing.Rescaling(1./255,
input_shape=(img_height, img_width, 3)),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes,activation='softmax')
])

# using softmax function, we will get probability of the image belonging
to each class and the class with maximum probability is picked
#compile model
model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
#train the model
epochs=10
history = model.fit(
  train_ds,
  validation_data=val_ds,
  epochs=epochs
)

#save the model
tf.keras.models.save_model(model,'my_model2.hdf5')
'''
it will save the architecture of the model as well as the weight
'''
# for web server deployment, we are using streamlit
!pip install streamlit
%%writefile app.py
import streamlit as st
import tensorflow as tf
import streamlit as st

# to reduce time in reloading we use cache
@st.cache(allow_output_mutation=True)
def load_model():
  model=tf.keras.models.load_model('/content/my_model2.hdf5')
  return model
with st.spinner('Model is being loaded..'):
  model=load_model()

st.write("""
         # Flower Classification
         """
         )
         file = st.file_uploader("Please upload an brain scan file",
type=["jpg", "png"])
import cv2
from PIL import Image, ImageOps
import numpy as np
st.set_option('deprecation.showfileUploaderEncoding', False)
def import_and_predict(image_data, model):

        size = (180,180)
        image = ImageOps.fit(image_data, size, Image.ANTIALIAS)
image = np.asarray(image)
        img = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        #img_resize = (cv2.resize(img, dsize=(75, 75),
interpolation=cv2.INTER_CUBIC))/255.

        img_reshape = img[np.newaxis,...]

        prediction = model.predict(img_reshape)
                return prediction
if file is None:
    st.text("Please upload an image file")
else:
    image = Image.open(file)
    st.image(image, use_column_width=True)
    predictions = import_and_predict(image, model)
    score = tf.nn.softmax(predictions[0])
    st.write(prediction)
    st.write(score)
    print(
    "This image most likely belongs to {} with a {:.2f} percent
confidence."
    .format(class_names[np.argmax(score)], 100 * np.max(score))
)
!wget https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-amd64.zip
!unzip ngrok-stable-linux-amd64.zip
get_ipython().system_raw('./ngrok http 8501 &')
!curl -s http://localhost:4040/api/tunnels | python3 -c 'import sys, json;
print("Execute the next cell and then go to the following URL: " +
json.load(sys.stdin)["tunnels"][0]["public_url"])'
!streamlit run /content/app.py
```

Video Explaination :-
https://drive.google.com/drive/folders/1bzOiqovpClmDm-NIlqMc6mwl6tmg_QCg?usp=sharing