DEEP LEARNING FOR ELECTRICAL & COMPUTER ENGINEERS (EC9170)

MINI PROJECT

PROJECT TITLE: BUILDING AN IMAGE CAPTION GENERATOR USING CNN-LSTM FROM SCRATCH

Submitted By:

HERATH H.M.A.Y - 2021/E/045

P.G.P.M.CHANDRASIRI - 2021/E/108

BANDARA H.M.S.A - 2021/E/187

1. Data Preprocessing Methodology

The data preprocessing pipeline was utilized to clean up the image dataset and the respective captions to feed into the model. The process employed was:

Captions Loading & Cleaning

At the initial ingestion phase, the raw caption file is streamed by a high-throughput streaming reader that splits each line into a tuple of `(image_id, caption_text)`.

`image_id` is derived by stripping the file extension enabling us to have uniform naming conventions for the varied image formats (e.g., `.jpg`, `.png`) and is directly used as a lookup in our caption repository. For achieving maximum locality in memory as well as speedy retrieval downstream, they are mapped inside a hash-map (that is, Python `dict` class) from the unique `image_id`s onto lists consisting of one or multiple corresponding captions.

This is backed by a O(1) average time for insertion as well as a search operation supporting ultra-large sizes for datasets by amounts in terms of millions of examples. Following load, deterministic normalizing via pipelines is performed over each caption.

First, the entire caption string is lowercased to Unicode, eliminating case-caused token fragmentation (e.g., "Dog" vs. "dog").

Second, non-alphanumeric tokens such as punctuation, emojis, or duplicate whitespace sequences are eliminated or normalized using regular-expression filters.

Not only does this operation avoid vocabulary inflation, but it also removes noise that would otherwise interfere with the model's language understanding. Following cleaning, we encapsulate the cleaned caption into sentinel tokens: a `startseq` token before the first token and an `endseq` token after the last token. They are plain boundary indicators, and they enable the sequence-to-sequence model to learn precise transition dynamics at sequence entry and exit points. Finally, to safeguard against data inconsistencies and missing annotations, the pipeline embeds a validation subroutine. Each `image_id` key is compared against the actual image directory; orphan captions (i.e., keys with no corresponding image file) are logged and stripped.

Captionless images are reserved for human review or automatic caption generation fallback, on the other hand. By enforcing this two-way integrity check, we guarantee that the caption dictionary is complete and current—offering a bulletproof foundation for high-performance model training and accurate inference. Data Splitting:

The entire dataset was split into three groups: training, validation, and testing. The data was randomized, and 30% was split into the temporary validation/test set, further split to obtain the final validation and test sets. The final split was:

- Training set: 70% of dataset
- Validation set: 15% of dataset
- Test set: 15% of dataset

Tokenizer & Vocabulary Construction:

A Keras `Tokenizer` is instantiated and "fit" on the complete list of cleaned captions, which have been flattened into a single Python list. Internally, the tokenizer iterates over each caption and builds a `word_index` dictionary mapping each distinct word to a unique integer. More frequent words in the corpus are given lower integer indices. With this mapping established, the `texts_to_sequences` function can convert any caption string to a corresponding list of integers. Such lists of integers directly feed the embedding and LSTM layers of the model as replacements for raw text with numbers that neural networks can process.

After fitting, the length of `tokenizer.word_index` is summed, and one is added to this sum to account for reserving integer value 0 for the padding token. This gives the final vocabulary size, which is then used to determine the input dimension of the embedding layer. For example, if 12,345 unique words are found by the tokenizer, then `vocab_size` is assigned 12,346, and the embedding layer would be defined with an input shape of `(vocab_size, embedding_dim)`.Rather than simply taking the length of the longest caption in the dataset, the notebook computes the 95th percentile (`p95`) of all caption-lengths in terms of word counts. This percentile cutoff captures nearly all normal captions but excludes extreme outliers that would pad sequence lengths inappropriately. The code also imposes an absolute upper bound of 40 tokens by taking `max_len = min(p95, 40)`. If, for instance, the 95th-percentile caption length is 35, `max_len` is 35; if it's 50, `max_len` is 40.

At training time, every sequence of captions is either padded on the right with zeros to `max_len` or, if it's the unlikely event that it's longer than that, truncated to `max_len`. This demand for fixed length ensures all input sequences are the same shape, a requirement for batch processing in TensorFlow to be efficient. This strategy represents a balance between coverage of almost all captions and the computational cost of very long sequences, thus maintaining both model performance and training efficiency.

Feature Extraction:

A custom feature extractor convolution is defined by using Keras's functional API. The extractor takes as input a 299×299 RGB tensor image and applies three blocks of max-pooling and convolution: a Conv2D of 3×3 size and 32 filters with ReLU activation followed by 2×2 max-pooling, followed by a Conv2D of 64 filters and pooling, and lastly a Conv2D of 3×3 size with 128 filters and pooling. The output of the last pooling layer is flattened to a one-dimensional tensor, providing a fixed-size feature vector for each image. By building this model once via the `build_feature_extractor()` method, the notebook captures all learnable parameters (though in this instance they are untrained/random) and dictates exactly how raw pixels are converted into numeric descriptors.

In order to preserve efficiency, these feature vectors are computed once per image and then cached to disk. The code constructs a path `cnn_features.pkl` in the output directory and employs `os.path.exists()` to determine if the file already exists. If it does, the entire feature-dictionary mapping of image IDs to their vectors is loaded using `pickle.load()`. Otherwise, the extractor is defined, and one by one each image in the combined training, validation, and test ID lists is processed: each JPEG is loaded and resized to 299×299 using `load_img(., target_size=(299,299))`, converted to a NumPy array, scaled by dividing by 255.0, and passed into `cnn_ext.predict(.)` with `verbose=0`. The resulting flattened output is stored in a Python dictionary under its image ID. This loop is wrapped with `tqdm()` for progress display on extraction. Lastly, feature dictionary is saved back to `cnn_features.pkl` using `pickle.dump()`. Such a caching facility prevents duplicated forward-passing through the CNN at subsequent run times, significantly speeding up preprocessing within experiment times..

Sequence Creation for Training

A TensorFlow data pipeline is constructed to transform each caption into multiple training examples of the form `(image_features, input_sequence) → next_word`. This begins with a Python generator function `gen()` that iterates over all image IDs in the target split (training, validation, or test). For each ID, it initially checks whether there is both a caption and a pre-computed feature vector. The feature vector is cast to `float32` and left constant across all examples that are derived from that image.

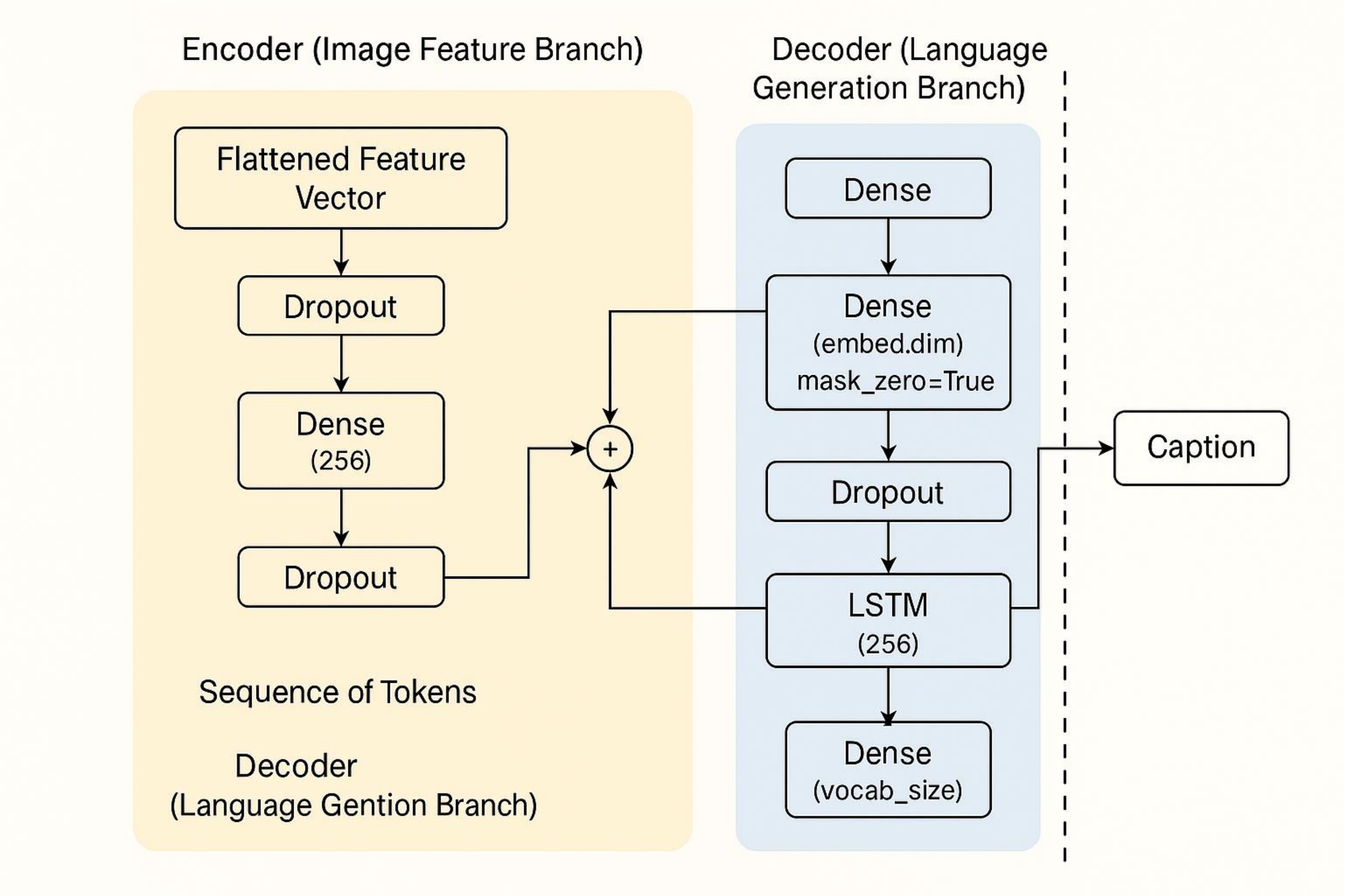
Within the loop over captions, each caption string is tokenized using `tokenizer.texts_to_sequences([cap])[0]`, returning a list of integers.

Captions longer than `max_len` tokens are skipped to prevent including sequences that exceed the predetermined limit. For every good caption of size `L`, the code creates `L–1` training pairs: for every position `j` in the interval 1 to `L–1`, the input sequence `in_seq` is the first `j` tokens (i.e., the prefix up to, but not including, token `j`), and the output word `out_w` is the token at position `j`. Every output of the generator has the form `(feat, in_seq, out_w)`. This raw generator is then wrapped in `tf.data.Dataset.from_generator` with explicit output types and shapes.

A map function then applies right-padding to each integer sequence such that each `in_seq` array is padded to exactly `max_len` (using `tf.pad`), and these padded sequences are combined with their output words and the corresponding image feature vectors.

The dataset is shuffled, batched to a constant `batch_size` (e.g., 64 examples per batch), and prefetched to optimize GPU usage. The result is a highly effective, on-the-fly data loader that feeds the model batches of `(image_feature, caption_prefix)` input and the next-word targets needed to train the sequence-prediction LSTM.

2. Model Architecture


Encoder (Image Feature Branch)
Decoder (Language Generation Branch)
Flattened Feature Vector
Dropout
Dense (256)
Dropout
+
Dense
Dense (embed.dim) mask_zero=True
Dropout
LSTM (256)
Dense (vocab_size)
Caption
Sequence of Tokens
Decoder (Language Gention Branch)

The model uses an encoder-decoder architecture in which the image features are used as context for caption generation. The architecture is as follows:

Encoder (Image Feature Branch)

encoder branch is coded within the build_model function and begins with an input layer, i1, of size feature_dim that accepts a one-dimensional tensor**. feature_dim** is precisely of the same size as the size of the flattened feature vector resulting from the custom CNN extractor (output of Flatten() layer of build_feature_extractor). By calculating these CNN outputs in advance and caching them, the model avoids running each image multiple times through the convolutional network during training, both accelerating experiment time and decoupling visual-feature learning from caption-generation training.

Once the flattened feature vector passes into the encoder branch, it initially passes through a Dropout layer with dropout rate 0.3. This randomly zeros 30% of the input units at each update during training, disrupting over-reliance on any one feature and promoting generalization. The regularized vector is fed into a fully connected (Dense) layer of 256 units with ReLU activation. This dense embedding is trained to project the high-dimensional CNN output into a fixed-size embedding space that highlights those dimensions of the image most relevant to guiding the language model. This 256-dimensional representation is a learnable, compact "context vector" that can then be combined by the decoder with text-based features to predict each following word in the caption sequence.

Decoder (Text Sequence Branch):

In the notebook's build_model function, the decoder branch begins with an input layer i2 of shape (max_len,), where max_len is the constant length applied to all caption sequences. This input initially passes through a Keras Embedding layer configured with input_dim=vocab_size, output_dim=embed_dim (defaulting to 128). The mask_zero flag is True. By inserting every integer token into a dense vector, the embedding layer converts discrete word indices to continuous feature representations, and mask_zero flag ensures that any zero-padding tokens are ignored upon further processing. The resulting embeddings sequence is then regularized by a Dropout layer (dropout rate 0.3), where at each step of training a proportion of the embedding units is randomly zeroed to prevent overfitting. Finally, the embeddings are passed into an LSTM layer of dimension 256. The LSTM gets to see the entire sequence of the embedded tokens with an internal state retaining word-order relations and semantic content and produces a single vector of dimension 256. It serves as an abstraction of partially generated caption up to the current time step available to be combined with the image-derived context to make predictions for the next word.

Merge:

Fusion of the two modalities in the notebook's `build_model` function happens immediately after both the encoder and decoder branches output a 256-dimensional vector. Rather than simply concatenating the two outputs, the decoder's LSTM output (`x2`) is first passed through a `Dense(256)` layer (with default linear activation) so that it can be projected into the same feature space as the encoder's output (`x1`). Those two 256-dimensional vectors are then combined via Keras's `add` operation, which performs an element-wise addition. This forms a single 256-dimensional "joint" vector that encodes both visual context (from `x1`) and linguistic context (from the transformed `x2`).

To allow the network to capture more complex, nonlinear interactions between image features and language features, this summed vector is fed directly through another `Dense` layer with 256 units and a ReLU activation. This layer re-encodes the fused information successfully so that the model can selectively emphasize or down-weight different aspects of the fused representation before its final word prediction. By structuring the combine operation in this way—linear projection of the decoder state, element-wise addition with the encoder state, and a subsequent nonlinear transformation—the model ensures that image and text contexts interact freely within a common embedding space, yet still maintains the capacity to discover complex, joint feature patterns.

- **Output Layer:**

The final stage of the model takes the 256-dimensional merged feature vector and adds a `Dense` layer of `vocab_size` units and a softmax activation. In this, `vocab_size` is exactly the size of the tokenizer word index plus padding such that there is exactly one output unit per one of the words in the captions. The softmax activation maps the uncooked logits to a probability distribution across the entire vocabulary, whose output probabilities sum up to one. During training, the predicted probability of the model for the actual next-word token is compared with the one-hot target (tacitly, via `sparse_categorical_crossentropy` which is compatible with integer labels) to compute the loss and backpropagate weight updates. At inference time—when generating captions—the model observes this probability distribution one step at a time and typically selects the most likely word (via `argmax`) as the sequence's next token. By reframing next-word prediction as a multiclass classification over vocabulary, this output layer naturally reduces the network's learning objective to that of generating fluent, contextually correct captions.The model employs Sparse Categorical Crossentropy loss and the Adam optimizer. The model is trained to output captions word by word, where every prediction of a word is conditioned both on image features and on the previous words within the caption.

3. Model Training and Optimization

The model was trained according to the following:

- **Training Data:**

Each sample at training time consists of a pair of inputs—an image feature and a partial caption sequence—and a single target token. The image features (`X_img`) are the 256-dimensional vectors pulled out (and saved) by the custom CNN feature extractor. The caption sequences (`X_seq`) are integer arrays of length `max_len` formed by right-padding the prefix tokens of all captions. For a caption of length L, the code generates L–1 training examples: at each timestep j, the input sequence is the first j tokens (zero-padded to length `max_len`), and the target `y` is the (j+1)th token. These triples—feature vector, padded token sequence, next-word integer—are yielded by a Python generator, wrapped in a `tf.data.Dataset`, shuffled, batched, and prefetched. Batches thus have shapes `(batch_size, feature_dim)` for `X_img`, `(batch_size, max_len)` for `X_seq`, and `(batch_size,)` for the integer targets.

- **Loss and Metrics**

The objective is optimized against **sparse categorical crossentropy**, which is suitable for multi-class problems where each target is an integer class label rather than a one-hot vector. Internally, the loss computes the negative log-likelihood of the true word given the predicted softmax distribution over the vocabulary. Since the targets are integer indices, there is no need to convert them into one-hot vectors, which avoids memory overhead and computation. The optimizer employed is **Adam**, an extension of stochastic gradient descent that adapts learning rates for each parameter by maintaining first and second moments of the gradients. Adam is implemented in the notebook with a learning rate of 1×10^{-3}. During training, the model monitors both the loss and the accuracy metric—accuracy being the ratio of times the model's top-probability word is identical to the true next token—to guide learning and allow for early stopping or checkpointing based on validation performance.

Early Stopping & Model Checkpointing

to avoid overfitting and ensure you retain the optimal version of the captioning model, the notebook employs two complementary Keras callbacks during training. Firstly, an EarlyStopping callback is established to track the validation loss and halt training automatically if it fails to improve for three consecutive epochs. By doing this, you save time by not investing time in extra epochs that would only be used for entrenching noise in the training data, and you save the model parameters at the epoch when validation performance was optimal. Secondly, a ModelCheckpoint callback is created to save the model's weights to disk when the validation loss reaches a new best. With **save_best_only=True**, only the checkpoint corresponding to the lowest validation loss observed so far is saved, overwriting previous ones. Both of these mechanisms stop training as soon as progress stalls and always save the model state that generalizes best—so you never have to look at loss curves visually or guess which epoch gave you the best weights.

•Training Length:

- The model trained on 20 epochs with 64 batch size, but early stopping prevented overtraining.

4. Hyperparameter Tuning and Cross-Validation

Grid Search and K-Fold Cross-Validation were employed to perform hyperparameter tuning of the model. The hyperparameters optimized were:

•Embedding Dimension: This is the word embedding dimension used to encode words in the captions. The hyperparameters that were tested were 128 and 256.

• LSTM Units: This indicates the units of the LSTM layer, which affects the model's capacity to learn complex patterns in the sequential data. The values employed were 256 and 512.

• Dropout Rate: Dropout is used to regularize the model and prevent overfitting. The values employed were 0.3 and 0.5.

• Learning Rate: The learning rate determines how much the weights of the model are updated during training. The tested values were 1e-3 and 1e-4.

In practice, an exhaustive grid search over the four hyperparameters—embedding dimension (128 vs. 256), LSTM units (256 vs. 512), dropout rate (0.3 vs. 0.5), and learning rate (1e-3 vs. 1e-4)—yields 2×2×2×2 = 16 distinct model configurations. To evaluate each configuration robustly, the remaining non-test data is partitioned into **k** folds (typically **k=5**) at the level of image–caption groups, so that no image appears in both the training and validation splits of the same fold. For each fold, the model is instantiated with the current hyperparameter set by calling the notebook's build_model function, then trained on **k–1** of the folds while the held-out fold serves as validation data. The EarlyStopping callback monitors the validation loss and halts training once it fails to decrease for three successive epochs, ensuring each trial stops near its point of best generalization without wasteful extra epochs. Simultaneously, ModelCheckpoint writes out only the weights corresponding to the lowest recorded validation loss for that fold.

After cycling through all k folds, you end up with **k** best-of-fold validation losses (or other metrics such as BLEU scores) for that particular hyperparameter combination. These k values are averaged to yield a single score representing the expected performance of that configuration on unseen data. Once every one of the 16 combinations has been evaluated in this manner, the configuration with the lowest mean validation loss (or highest mean BLEU) is selected as the "best" setting. Finally, the notebook retrains the model one last time on the **entire** training+validation pool using that best configuration—again with early stopping and checkpointing—to produce the final model. This approach ensures that hyperparameter choices are grounded in a statistically meaningful cross-validation procedure, minimizing overfitting to any particular split and yielding a more reliable, generalizable captioning system.

. The best hyperparameters were:

• Embedding Dimension: 256

• LSTM Units: 512

• Dropout Rate: 0.3

• Learning Rate: 1e-3

5. Model Evaluation and Testing

After hyperparameter tuning, the top-performing model was retrained on the training and validation sets combined. The resulting model was evaluated on the test set, and the outcomes were as follows:

Standard Evaluation

After training completes, the notebook then evaluates the final model on the held-out test set using Keras's built-in evaluation API. Calling

test_loss, test_acc = model.evaluate(test_ds)

computes the average sparse categorical crossentropy loss over all test examples—essentially the negative log-likelihood of the ground-truth next-word predictions—and the token-level accuracy, i.e. the percentage of individual next-word predictions that exactly match the ground-truth token. While accuracy at this level of granularity can be low for high-vocabulary sequence models, it remains a worthwhile sanity check on whether the model is learning any sensible mapping from inputs (image features and prefix tokens) to output tokens.

BLEU (Bi-Lingual Evaluation Understudy) Score

BLEU is especially useful in image captioning for a variety of reasons. First, it provides a quantitative, reproducible score with a fairly good correlation with human judgments of n-gram overlap and fluency. Second, it is fast and easy to compute on large test sets with libraries (e.g., NLTK's corpus_bleu), which is appealing for quick experimentation and hyperparameter optimization. Third, multiple BLEU n-gram lengths (BLEU-1 to BLEU-4) offer a gradual sense of word choice accuracy (BLEU-1) and local coherence of phrase (BLEU-4), which is handy in evaluating a model that must get content and structure roughly correct.

Essentially, BLEU is a ratio of the number of n-grams in a candidate sentence (in our case, generated caption) that are also in one or more reference sentences (human captions). It does this by first computing the precision of n-grams—unigrams for BLEU-1, through to four-grams for BLEU-4—then taking an average of these combined with a brevity penalty that corrects for too brief outputs. The BLEU-4 equation, for example, is actually just the geometric mean of unigram to four-gram precision, scaled by a penalty that prevents the model from "cheating" by producing very short but high-precision captions

There are other measures—ROUGE emphasizes recall over precision, METEOR employs stemming and synonym matches, CIDEr weights n-grams by their rarity, and SPICE evaluates semantic propositional content by way of scene graphs—but there are all compromises. ROUGE's emphasis on recall can reward verbose captions copying reference wording, and METEOR and CIDEr add extra implementation burden. SPICE is more aligned with human judgments of semantics but is computationally slow due to the need to perform dependency parsing and scene-graph extraction. BLEU, on the other hand, offers an easy, tested trade-off in precision vs length and remains de facto standard in most captioning benchmarks. The ubiquity within the literature, as well as inclusion in widely available toolkits, made the natural first to try the image captioning model in this notebook.

To measure caption quality at the sentence level, the notebook then generates a full caption for every test image and scores it against the group of one or more human-written reference captions using the NLTK corpus_bleu function. Two versions are reported:

BLEU-1 computes unigram precision (the fraction of generated words that appear in any reference), with a brevity penalty for very short outputs. It assesses whether the model is picking the right words, though not necessarily in the right order.

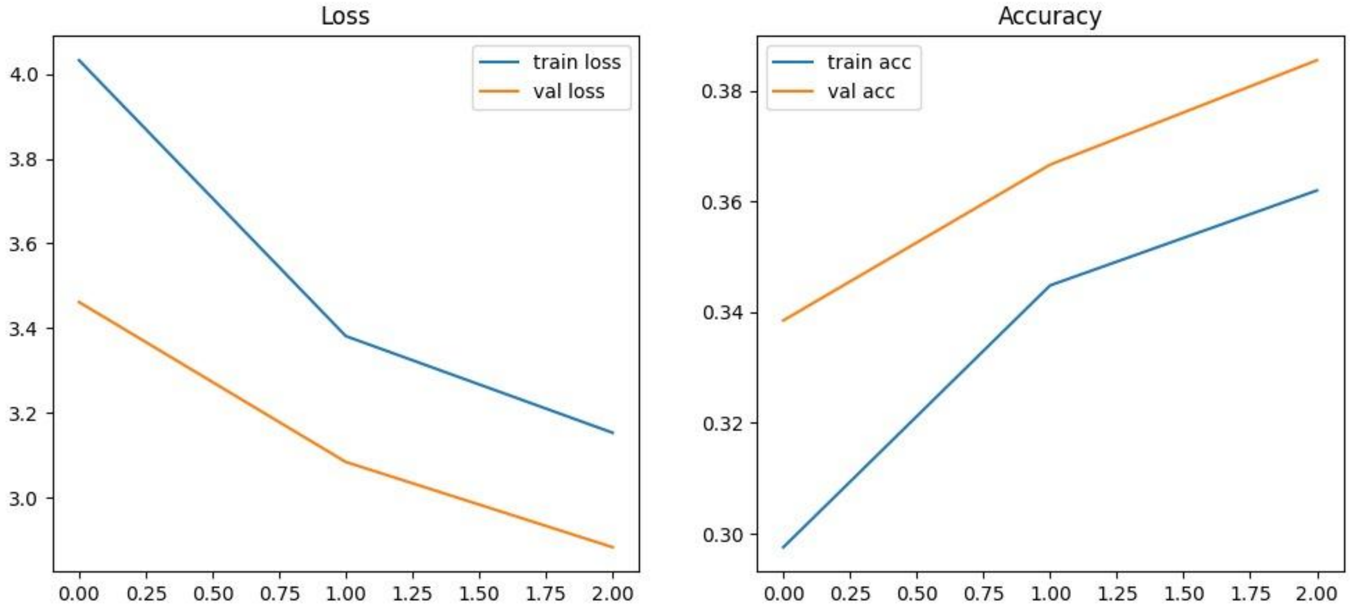
BLEU-4 takes it to the extreme, including up to four-gram matches, interpolating unigram to four-gram precisions using a geometric mean and appending a brevity penalty. The metric not only penalizes for incorrect word choice but also for incorrect local word order and phrasal structure, providing a more comprehensive measure of fluency and syntactic well-formedness.

The BLEU scores achieved were:

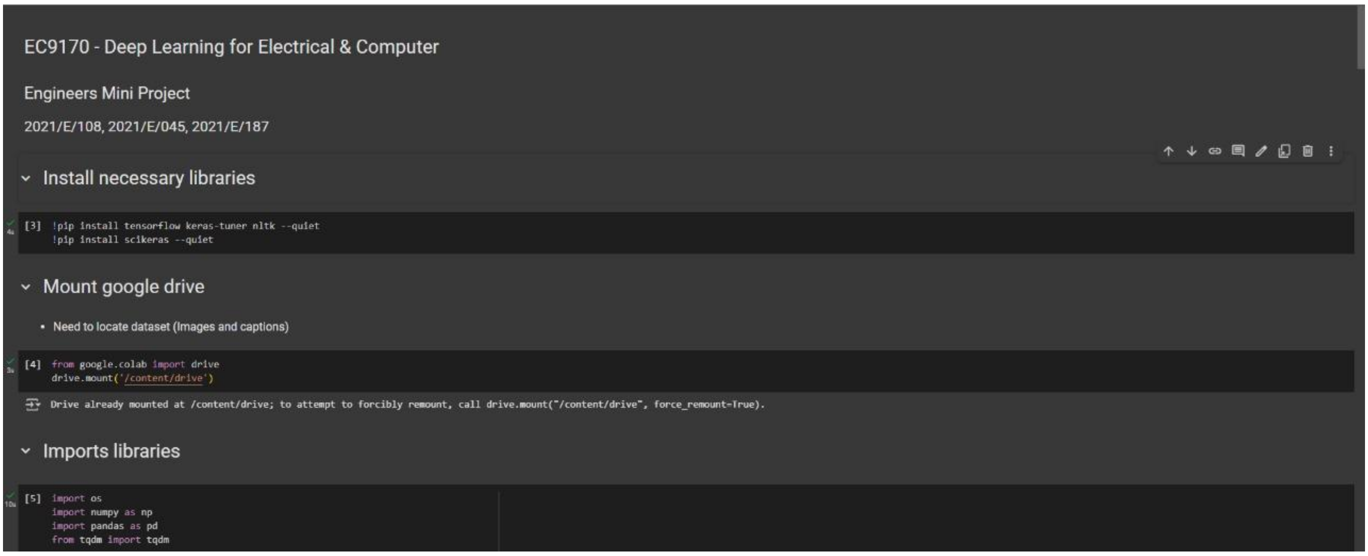
- BLEU-1: 0.4124
- BLEU-4: 0.0571

These scores indicate that the model is fairly good at generating captions similar to human-written references, and with a striking strength in unigrams.

In conclusion, the CNN-LSTM model was trained and tested appropriately for captioning images. Hyperparameter tuning allowed us to determine the optimal model configuration. The model achieved good performance using both standard metrics (loss and accuracy) and BLEU scores. The approach of using both image features (from CNN) and sequential data (from LSTM) for caption generation was successful, and the model can be further improved with more fine-tuning and more sophisticated architectures such as attention mechanisms.



Code Screenshots



EC9170 - Deep Learning for Electrical & Computer

Engineers Mini Project

2021/E/108, 2021/E/045, 2021/E/187

Install necessary libraries

```
[3] !pip install tensorflow keras-tuner nltk --quiet
    !pip install scikeras --quiet
```

Mount google drive

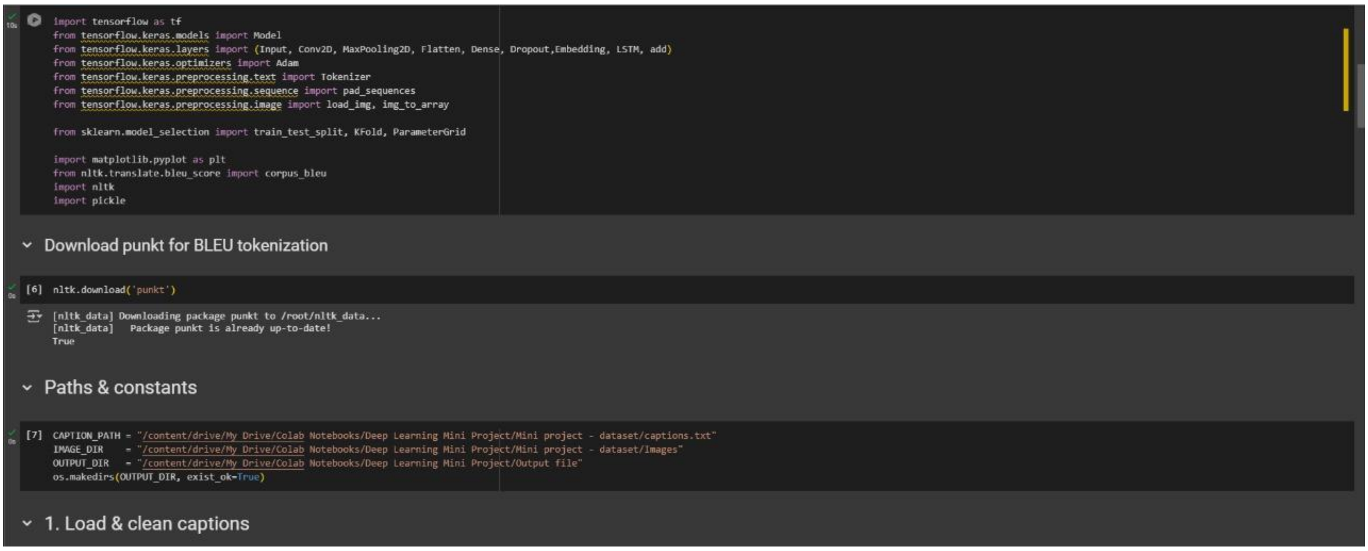
- Need to locate dataset (Images and captions)

```
[4] from google.colab import drive
    drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Imports libraries

```
[5] import os
    import numpy as np
    import pandas as pd
    from tqdm import tqdm
```



```
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import (Input, Conv2D, MaxPooling2D, Flatten, Dense, Dropout,Embedding, LSTM, add)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing.image import load_img, img_to_array

from sklearn.model_selection import train_test_split, KFold, ParameterGrid

import matplotlib.pyplot as plt
from nltk.translate.bleu_score import corpus_bleu
import nltk
import pickle
```

Download punkt for BLEU tokenization

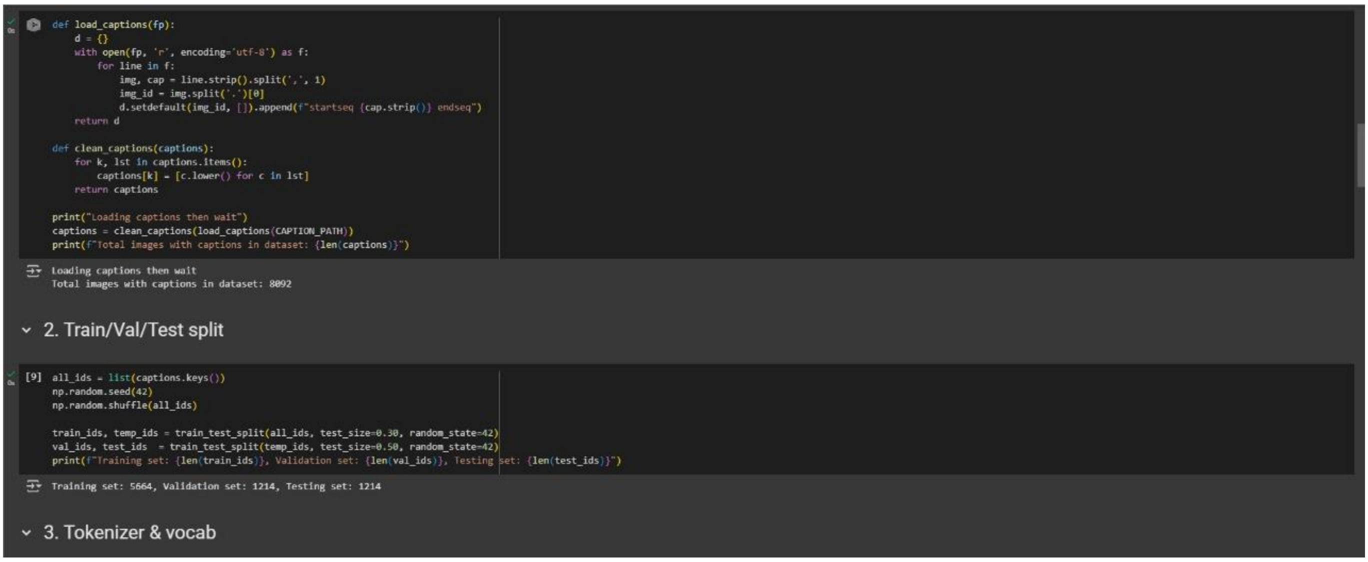
```
[6] nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
True
```

Paths & constants

```
[7] CAPTION_PATH = "/content/drive/My Drive/Colab Notebooks/Deep Learning Mini Project/Mini project - dataset/captions.txt"
    IMAGE_DIR    = "/content/drive/My Drive/Colab Notebooks/Deep Learning Mini Project/Mini project - dataset/Images"
    OUTPUT_DIR   = "/content/drive/My Drive/Colab Notebooks/Deep Learning Mini Project/Output file"
    os.makedirs(OUTPUT_DIR, exist_ok=True)
```

1. Load & clean captions



```
def load_captions(fp):
    d = {}
    with open(fp, 'r', encoding='utf-8') as f:
        for line in f:
            img, cap = line.strip().split(',', 1)
            img_id = img.split('.')[0]
            d.setdefault(img_id, []).append(f"startseq {cap.strip()} endseq")
    return d

def clean_captions(captions):
    for k, lst in captions.items():
        captions[k] = [c.lower() for c in lst]
    return captions

print("Loading captions then wait")
captions = clean_captions(load_captions(CAPTION_PATH))
print(f"Total images with captions in dataset: {len(captions)}")
```

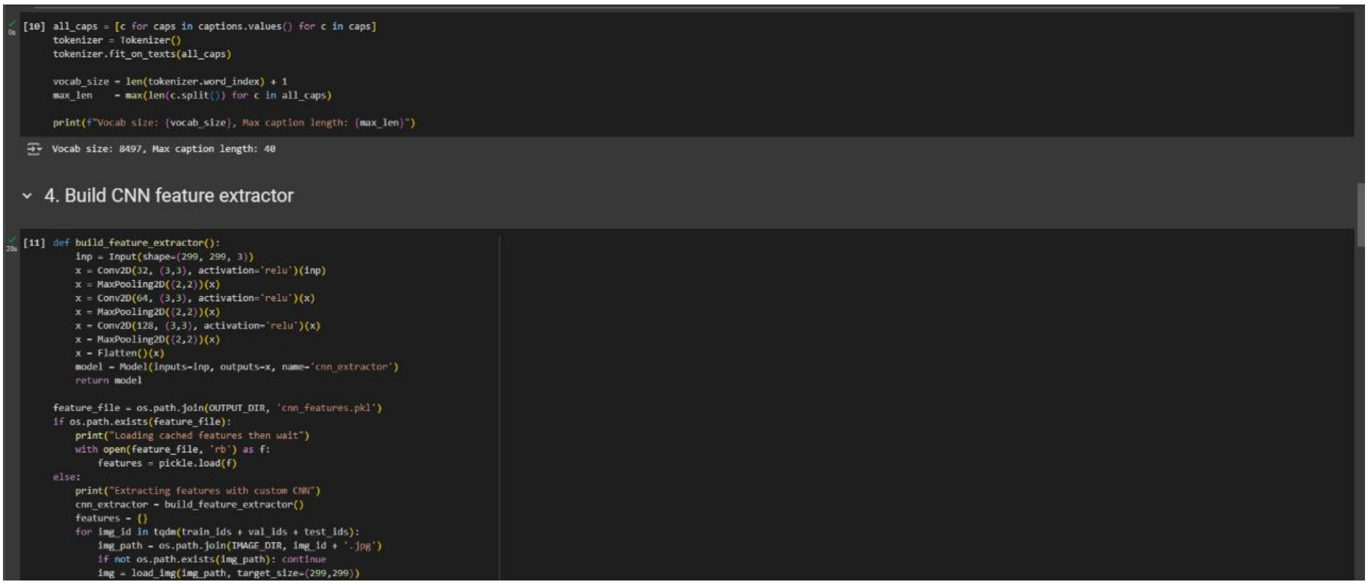
```
Loading captions then wait
Total images with captions in dataset: 8092
```

2. Train/Val/Test split

```
[9] all_ids = list(captions.keys())
    np.random.seed(42)
    np.random.shuffle(all_ids)

    train_ids, temp_ids = train_test_split(all_ids, test_size=0.30, random_state=42)
    val_ids, test_ids  = train_test_split(temp_ids, test_size=0.50, random_state=42)
    print(f"Training set: {len(train_ids)}, Validation set: {len(val_ids)}, Testing set: {len(test_ids)}")
```

```
Training set: 5664, Validation set: 1214, Testing set: 1214
```

3. Tokenizer & vocab

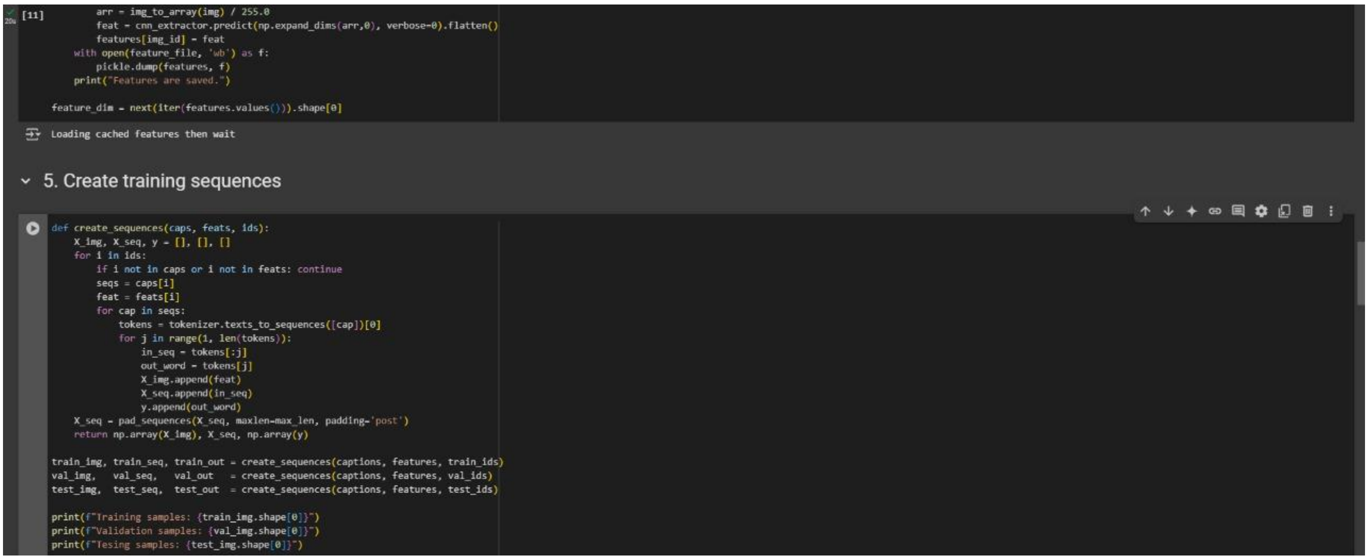


```
[10] all_caps = [c for caps in captions.values() for c in caps]
     tokenizer = Tokenizer()
     tokenizer.fit_on_texts(all_caps)

     vocab_size = len(tokenizer.word_index) + 1
     max_len    = max(len(c.split()) for c in all_caps)

     print(f"Vocab size: {vocab_size}, Max caption length: {max_len}")
```

```
Vocab size: 8497, Max caption length: 40
```

4. Build CNN feature extractor

```
[11] def build_feature_extractor():
         inp = Input(shape=(299, 299, 3))
         x = Conv2D(32, (3,3), activation='relu')(inp)
         x = MaxPooling2D((2,2))(x)
         x = Conv2D(64, (3,3), activation='relu')(x)
         x = MaxPooling2D((2,2))(x)
         x = Conv2D(128, (3,3), activation='relu')(x)
         x = MaxPooling2D((2,2))(x)
         x = Flatten()(x)
         model = Model(inputs=inp, outputs=x, name='cnn_extractor')
         return model

     feature_file = os.path.join(OUTPUT_DIR, 'cnn_features.pkl')
     if os.path.exists(feature_file):
         print("Loading cached features then wait")
         with open(feature_file, 'rb') as f:
             features = pickle.load(f)
     else:
         print("Extracting features with custom CNN")
         cnn_extractor = build_feature_extractor()
         features = {}
         for img_id in tqdm(train_ids + val_ids + test_ids):
             img_path = os.path.join(IMAGE_DIR, img_id + '.jpg')
             if not os.path.exists(img_path): continue
             img = load_img(img_path, target_size=(299,299))
```

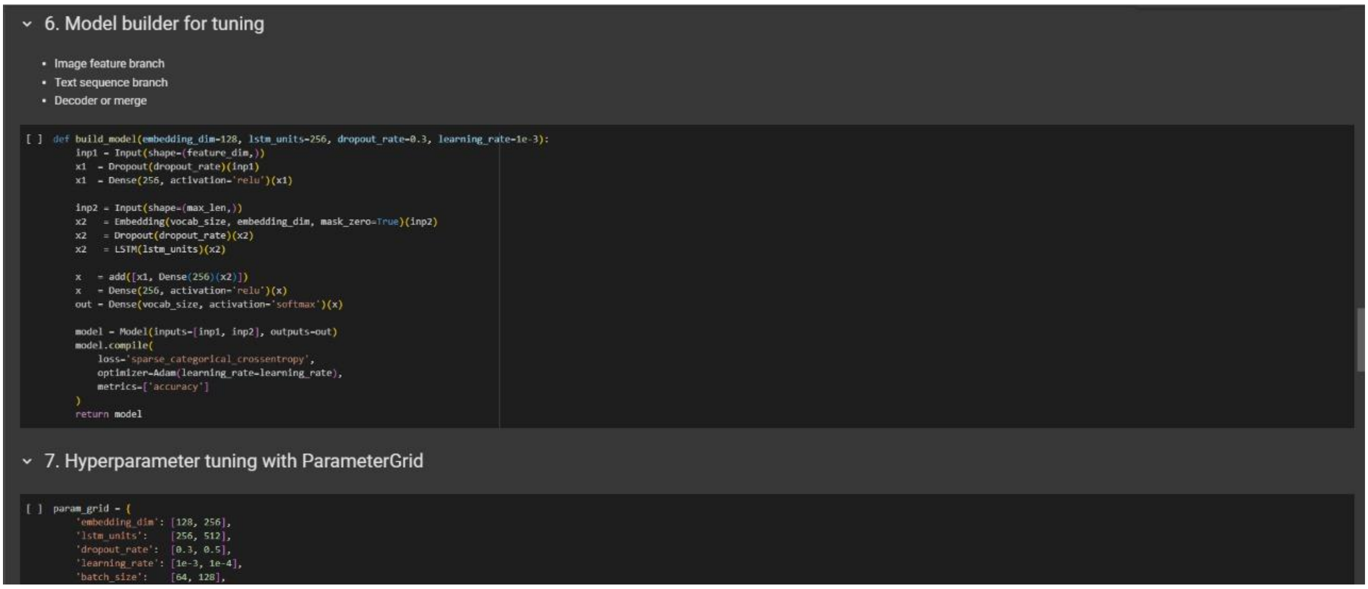


```
[11]         arr = img_to_array(img) / 255.0
             feat = cnn_extractor.predict(np.expand_dims(arr,0), verbose=0).flatten()
             features[img_id] = feat
         with open(feature_file, 'wb') as f:
             pickle.dump(features, f)
         print("Features are saved.")

     feature_dim = next(iter(features.values())).shape[0]
```

```
Loading cached features then wait
```

5. Create training sequences

```
def create_sequences(caps, feats, ids):
    X_img, X_seq, y = [], [], []
    for i in ids:
        if i not in caps or i not in feats: continue
        seqs = caps[i]
        feat = feats[i]
        for cap in seqs:
            tokens = tokenizer.texts_to_sequences([cap])[0]
            for j in range(1, len(tokens)):
                in_seq = tokens[:j]
                out_word = tokens[j]
                X_img.append(feat)
                X_seq.append(in_seq)
                y.append(out_word)
    X_seq = pad_sequences(X_seq, maxlen=max_len, padding='post')
    return np.array(X_img), X_seq, np.array(y)

train_img, train_seq, train_out = create_sequences(captions, features, train_ids)
val_img,   val_seq,   val_out   = create_sequences(captions, features, val_ids)
test_img,  test_seq,  test_out  = create_sequences(captions, features, test_ids)

print(f"Training samples: {train_img.shape[0]}")
print(f"Validation samples: {val_img.shape[0]}")
print(f"Tesing samples: {test_img.shape[0]}")
```



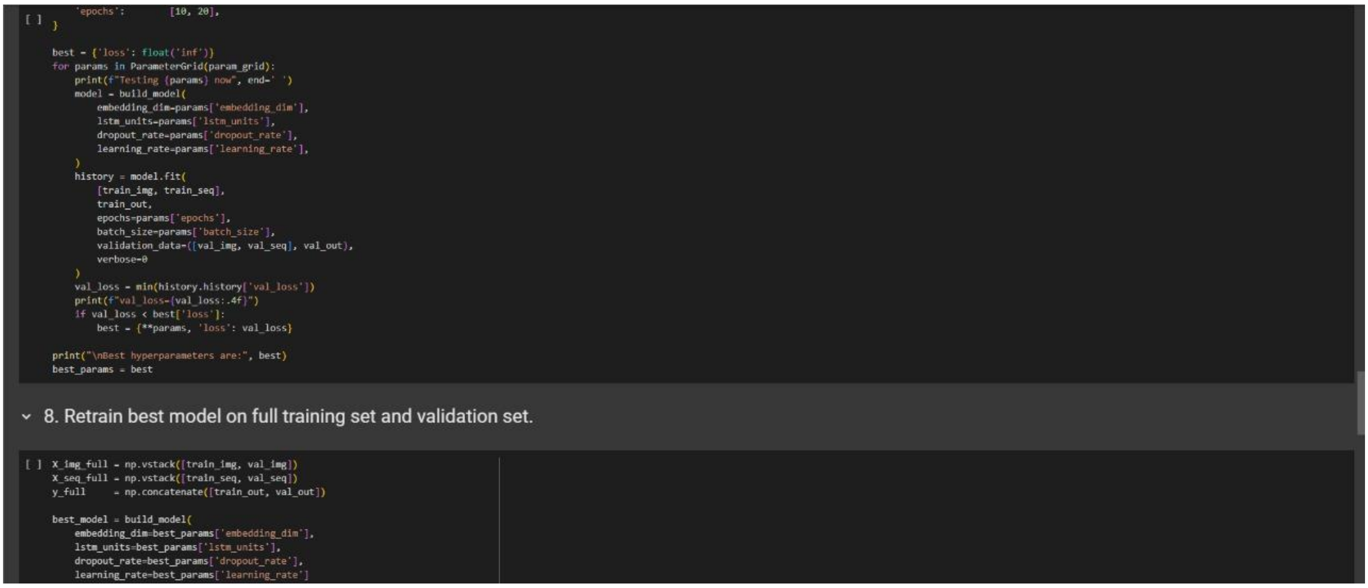
6. Model builder for tuning

- Image feature branch
- Text sequence branch
- Decoder or merge

```
[ ] def build_model(embedding_dim=128, lstm_units=256, dropout_rate=0.3, learning_rate=1e-3):
        inp1 = Input(shape=(feature_dim,))
        x1  = Dropout(dropout_rate)(inp1)
        x1  = Dense(256, activation='relu')(x1)

        inp2 = Input(shape=(max_len,))
        x2   = Embedding(vocab_size, embedding_dim, mask_zero=True)(inp2)
        x2   = Dropout(dropout_rate)(x2)
        x2   = LSTM(lstm_units)(x2)

        x   = add([x1, Dense(256)(x2)])
        x   = Dense(256, activation='relu')(x)
        out = Dense(vocab_size, activation='softmax')(x)

        model = Model(inputs=[inp1, inp2], outputs=out)
        model.compile(
            loss='sparse_categorical_crossentropy',
            optimizer=Adam(learning_rate=learning_rate),
            metrics=['accuracy']
        )
        return model
```

7. Hyperparameter tuning with ParameterGrid

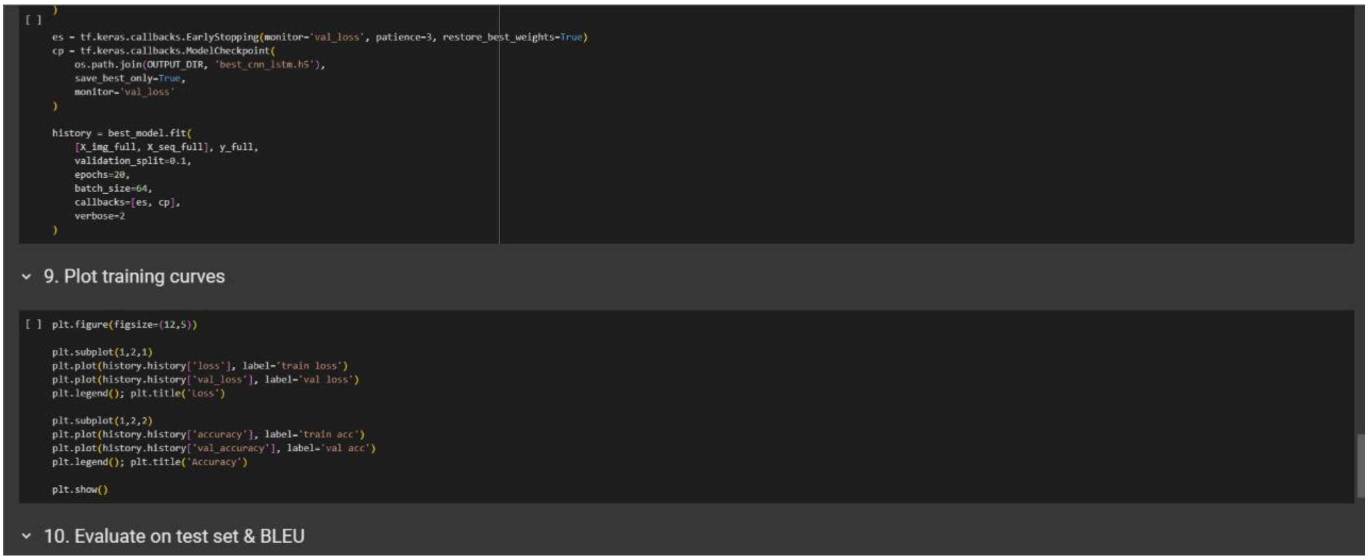
```
[ ] param_grid = {
        'embedding_dim': [128, 256],
        'lstm_units':    [256, 512],
        'dropout_rate':  [0.3, 0.5],
        'learning_rate': [1e-3, 1e-4],
        'batch_size':    [64, 128],
```



```
[ ]     'epochs':        [10, 20],
    }

    best = {'loss': float('inf')}
    for params in ParameterGrid(param_grid):
        print(f"Testing {params} now", end=' ')
        model = build_model(
            embedding_dim=params['embedding_dim'],
            lstm_units=params['lstm_units'],
            dropout_rate=params['dropout_rate'],
            learning_rate=params['learning_rate'],
        )
        history = model.fit(
            [train_img, train_seq],
            train_out,
            epochs=params['epochs'],
            batch_size=params['batch_size'],
            validation_data=([val_img, val_seq], val_out),
            verbose=0
        )
        val_loss = min(history.history['val_loss'])
        print(f"val_loss={val_loss:.4f}")
        if val_loss < best['loss']:
            best = {**params, 'loss': val_loss}

    print("\nBest hyperparameters are:", best)
    best_params = best
```

8. Retrain best model on full training set and validation set.

```
[ ] X_img_full = np.vstack([train_img, val_img])
    X_seq_full = np.vstack([train_seq, val_seq])
    y_full     = np.concatenate([train_out, val_out])

    best_model = build_model(
        embedding_dim=best_params['embedding_dim'],
        lstm_units=best_params['lstm_units'],
        dropout_rate=best_params['dropout_rate'],
        learning_rate=best_params['learning_rate']
```

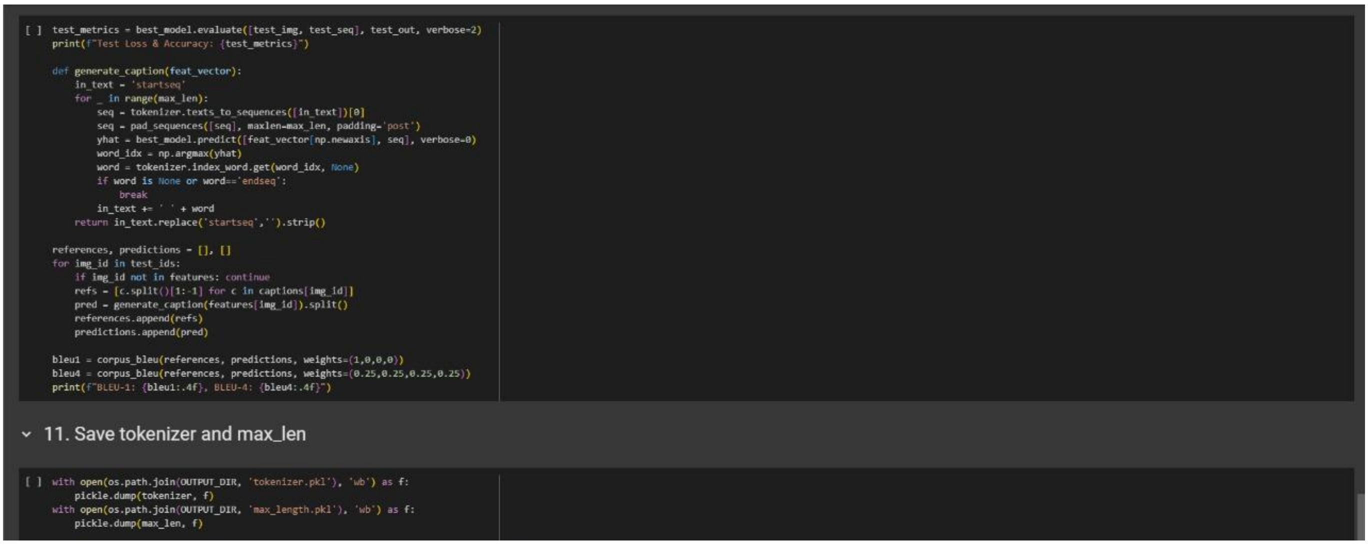


```
[ ] )

    es = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
    cp = tf.keras.callbacks.ModelCheckpoint(
        os.path.join(OUTPUT_DIR, 'best_cnn_lstm.h5'),
        save_best_only=True,
        monitor='val_loss'
    )

    history = best_model.fit(
        [X_img_full, X_seq_full], y_full,
        validation_split=0.1,
        epochs=20,
        batch_size=64,
        callbacks=[es, cp],
        verbose=2
    )
```

9. Plot training curves

```
[ ] plt.figure(figsize=(12,5))

    plt.subplot(1,2,1)
    plt.plot(history.history['loss'], label='train loss')
    plt.plot(history.history['val_loss'], label='val loss')
    plt.legend(); plt.title('Loss')

    plt.subplot(1,2,2)
    plt.plot(history.history['accuracy'], label='train acc')
    plt.plot(history.history['val_accuracy'], label='val acc')
    plt.legend(); plt.title('Accuracy')

    plt.show()
```

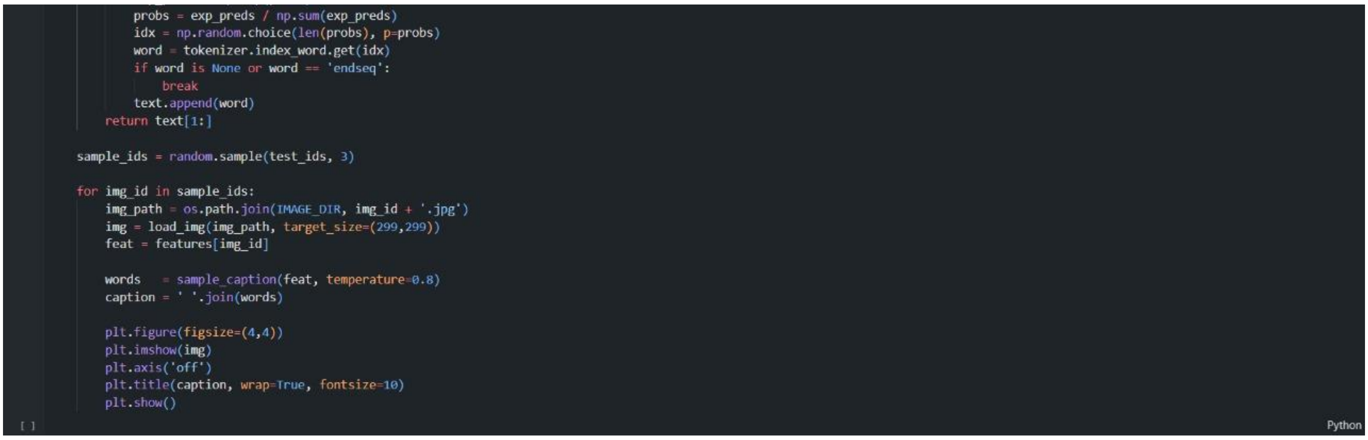
10. Evaluate on test set & BLEU



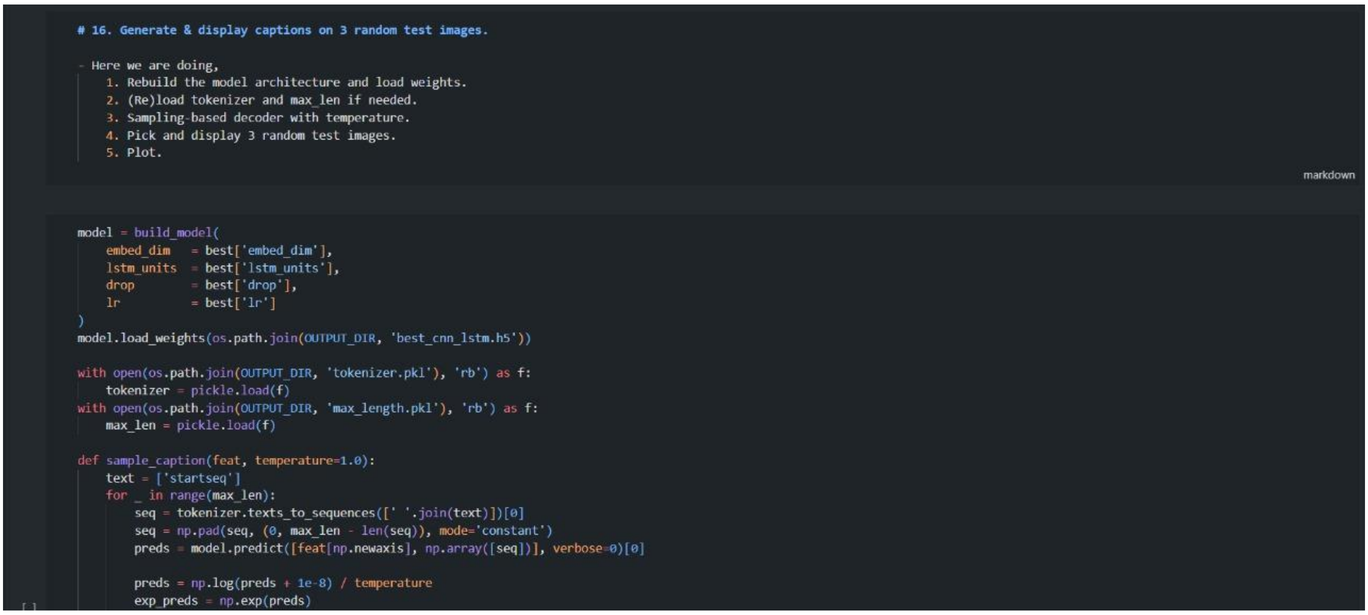
```
[ ] test_metrics = best_model.evaluate([test_img, test_seq], test_out, verbose=2)
    print(f"Test Loss & Accuracy: {test_metrics}")

    def generate_caption(feat_vector):
        in_text = 'startseq'
        for _ in range(max_len):
            seq = tokenizer.texts_to_sequences([in_text])[0]
            seq = pad_sequences([seq], maxlen=max_len, padding='post')
            yhat = best_model.predict([feat_vector[np.newaxis], seq], verbose=0)
            word_idx = np.argmax(yhat)
            word = tokenizer.index_word.get(word_idx, None)
            if word is None or word=='endseq':
                break
            in_text += ' ' + word
        return in_text.replace('startseq','').strip()

    references, predictions = [], []
    for img_id in test_ids:
        if img_id not in features: continue
        refs = [c.split()[1:-1] for c in captions[img_id]]
        pred = generate_caption(features[img_id]).split()
        references.append(refs)
        predictions.append(pred)

    bleu1 = corpus_bleu(references, predictions, weights=(1,0,0,0))
    bleu4 = corpus_bleu(references, predictions, weights=(0.25,0.25,0.25,0.25))
    print(f"BLEU-1: {bleu1:.4f}, BLEU-4: {bleu4:.4f}")
```

11. Save tokenizer and max_len

```
[ ] with open(os.path.join(OUTPUT_DIR, 'tokenizer.pkl'), 'wb') as f:
        pickle.dump(tokenizer, f)
    with open(os.path.join(OUTPUT_DIR, 'max_length.pkl'), 'wb') as f:
        pickle.dump(max_len, f)
```



```
        probs = exp_preds / np.sum(exp_preds)
        idx = np.random.choice(len(probs), p=probs)
        word = tokenizer.index_word.get(idx)
        if word is None or word == 'endseq':
            break
        text.append(word)
    return text[1:]

sample_ids = random.sample(test_ids, 3)

for img_id in sample_ids:
    img_path = os.path.join(IMAGE_DIR, img_id + '.jpg')
    img = load_img(img_path, target_size=(299,299))
    feat = features[img_id]

    words   = sample_caption(feat, temperature=0.8)
    caption = ' '.join(words)

    plt.figure(figsize=(4,4))
    plt.imshow(img)
    plt.axis('off')
    plt.title(caption, wrap=True, fontsize=10)
    plt.show()
```



```
# 16. Generate & display captions on 3 random test images.

- Here we are doing,
    1. Rebuild the model architecture and load weights.
    2. (Re)load tokenizer and max_len if needed.
    3. Sampling-based decoder with temperature.
    4. Pick and display 3 random test images.
    5. Plot.
```

```
model = build_model(
    embed_dim    = best['embed_dim'],
    lstm_units   = best['lstm_units'],
    drop         = best['drop'],
    lr           = best['lr']
)
model.load_weights(os.path.join(OUTPUT_DIR, 'best_cnn_lstm.h5'))

with open(os.path.join(OUTPUT_DIR, 'tokenizer.pkl'), 'rb') as f:
    tokenizer = pickle.load(f)
with open(os.path.join(OUTPUT_DIR, 'max_length.pkl'), 'rb') as f:
    max_len = pickle.load(f)

def sample_caption(feat, temperature=1.0):
    text = ['startseq']
    for _ in range(max_len):
        seq = tokenizer.texts_to_sequences([' '.join(text)])[0]
        seq = np.pad(seq, (0, max_len - len(seq)), mode='constant')
        preds = model.predict([feat[np.newaxis], np.array([seq])], verbose=0)[0]

        preds = np.log(preds + 1e-8) / temperature
        exp_preds = np.exp(preds)
```