Sri Lanka Institute of Information Technology

Faculty of Graduate Studies and Research

Master of Science in Information Technology Specializing in Enterprise Applications Development

Fundamentals of Cognitive Computing - SE5110 Primary Module Assignment

Student Name: D.M.D.W.R Warnasooriya
Student Reg No: MS24019736

Github Link

Project source and models stored in this github public repository

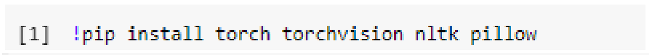
https://github.com/warnasooriya/fcc-primary-assignment

Phase 1: Image Captioning

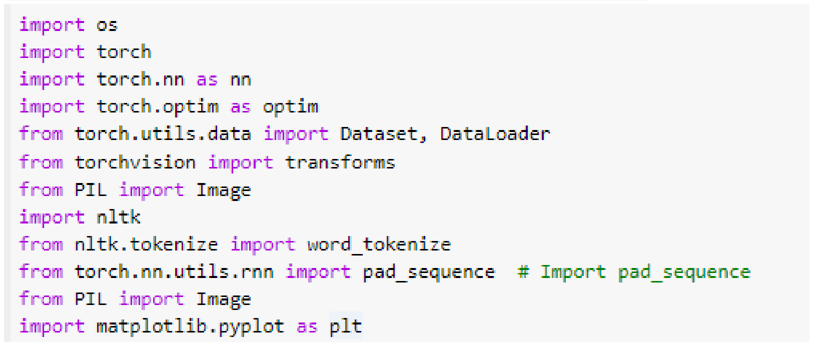
In this assignment phase 1, I created image captioning model from scratch using the flicker8k dataset

Setup and Imports:

imported Python libraries like torch, PIL, nltk, etc., which are used for handling tensors, images, and text data.

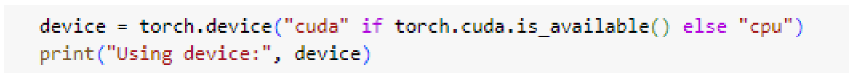


```
[1]  !pip install torch torchvision nltk pillow
```



```
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image
import nltk
from nltk.tokenize import word_tokenize
from torch.nn.utils.rnn import pad_sequence  # Import pad_sequence
from PIL import Image
import matplotlib.pyplot as plt
```

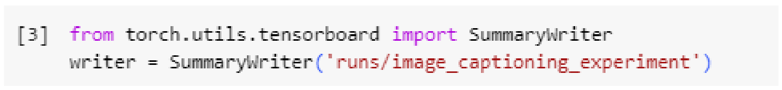
set up the device (GPU or CPU) for training depending on what is available



```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)
```

TensorBoard Setup

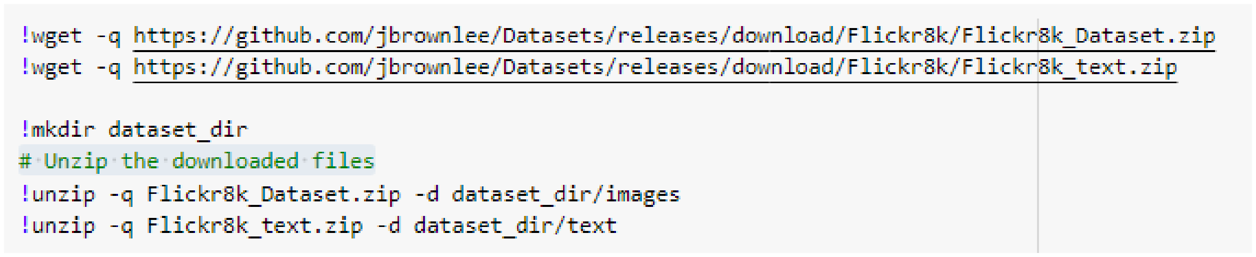
- **`SummaryWriter`** from TensorBoard is initialized to log training and validation metrics, which helps in visualizing the training process in TensorBoard.



```
[3]  from torch.utils.tensorboard import SummaryWriter
     writer = SummaryWriter('runs/image_captioning_experiment')
```

Data Downloading and Unzipping

download Flickr8k dataset from public location it contains images and captions in separately, after downloaded extract this dataset to local location in google colabs

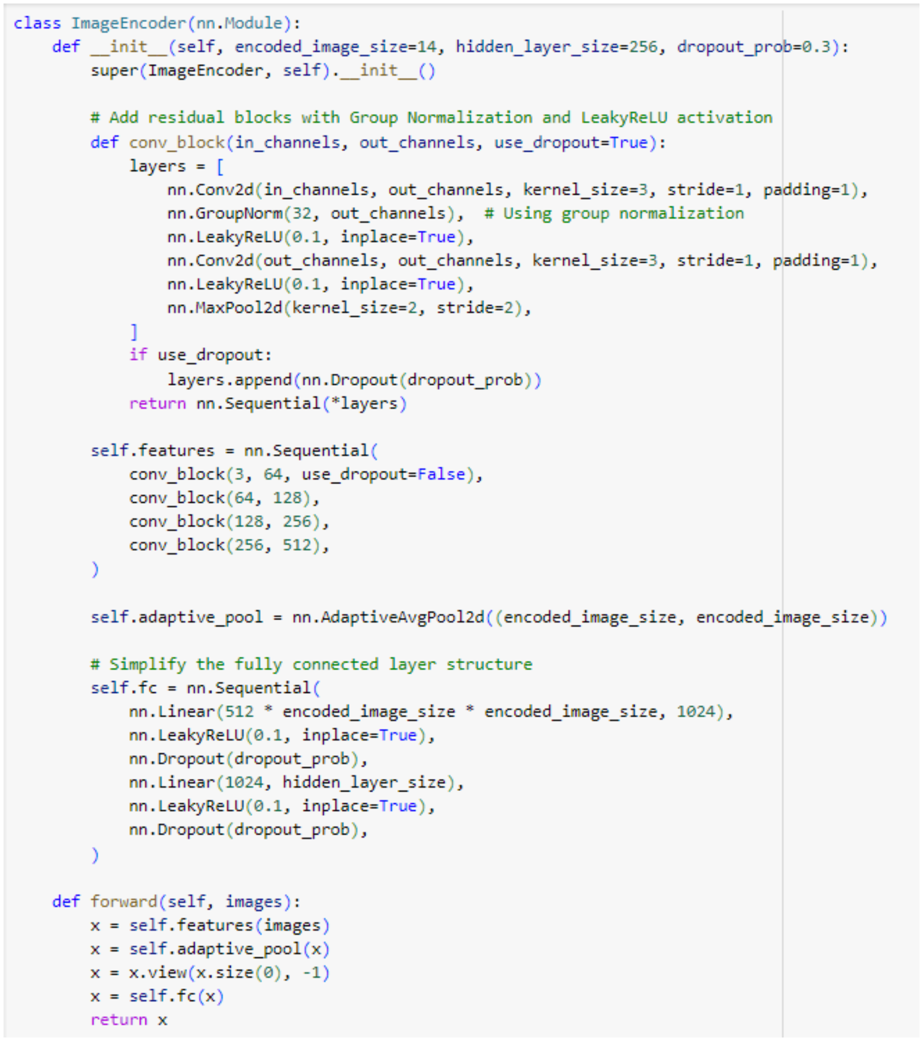


```
!wget -q https://github.com/jbrownlee/Datasets/releases/download/Flickr8k/Flickr8k_Dataset.zip
!wget -q https://github.com/jbrownlee/Datasets/releases/download/Flickr8k/Flickr8k_text.zip

!mkdir dataset_dir
# Unzip the downloaded files
!unzip -q Flickr8k_Dataset.zip -d dataset_dir/images
!unzip -q Flickr8k_text.zip -d dataset_dir/text
```

Model Definition

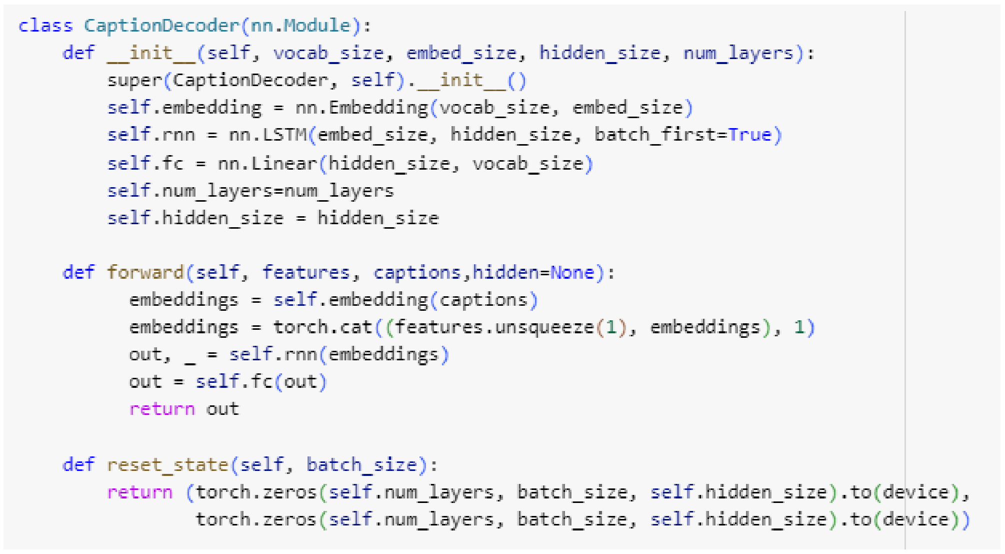
- Two main classes, **ImageEncoder** and **CaptionDecoder**, are defined:

  ImageEncoder: This class uses CNN to extract features from images.



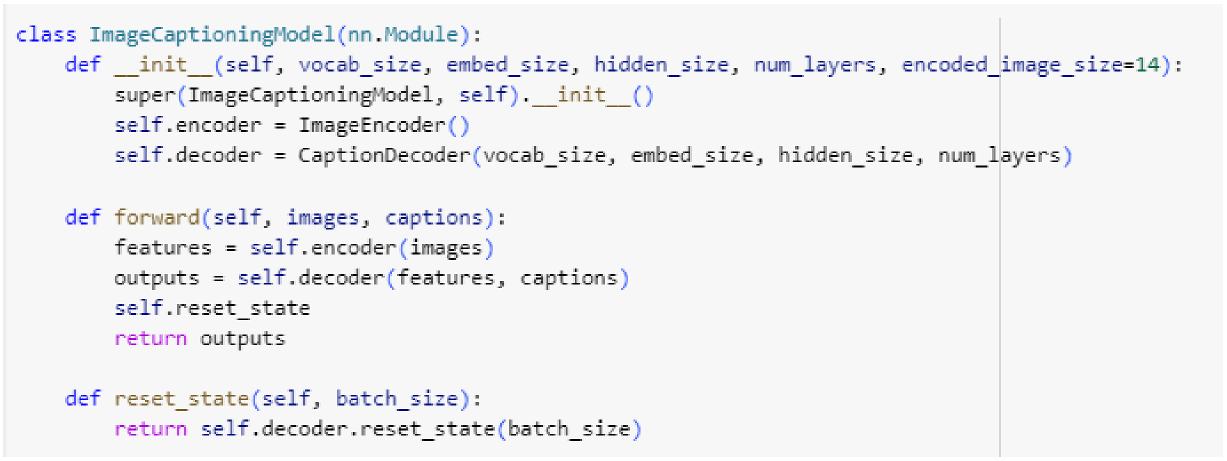
```
class ImageEncoder(nn.Module):
    def __init__(self, encoded_image_size=14, hidden_layer_size=256, dropout_prob=0.3):
        super(ImageEncoder, self).__init__()

        # Add residual blocks with Group Normalization and LeakyReLU activation
        def conv_block(in_channels, out_channels, use_dropout=True):
            layers = [
                nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1),
                nn.GroupNorm(32, out_channels),  # Using group normalization
                nn.LeakyReLU(0.1, inplace=True),
                nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1),
                nn.LeakyReLU(0.1, inplace=True),
                nn.MaxPool2d(kernel_size=2, stride=2),
            ]
            if use_dropout:
                layers.append(nn.Dropout(dropout_prob))
            return nn.Sequential(*layers)

        self.features = nn.Sequential(
            conv_block(3, 64, use_dropout=False),
            conv_block(64, 128),
            conv_block(128, 256),
            conv_block(256, 512),
        )

        self.adaptive_pool = nn.AdaptiveAvgPool2d((encoded_image_size, encoded_image_size))

        # Simplify the fully connected layer structure
        self.fc = nn.Sequential(
            nn.Linear(512 * encoded_image_size * encoded_image_size, 1024),
            nn.LeakyReLU(0.1, inplace=True),
            nn.Dropout(dropout_prob),
            nn.Linear(1024, hidden_layer_size),
            nn.LeakyReLU(0.1, inplace=True),
            nn.Dropout(dropout_prob),
        )

    def forward(self, images):
        x = self.features(images)
        x = self.adaptive_pool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

CaptionDecoder: This class uses an LSTM (a type of recurrent neural network) to generate captions based on the features provided by the encoder.



```
class CaptionDecoder(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size, num_layers):
        super(CaptionDecoder, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.LSTM(embed_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)
        self.num_layers=num_layers
        self.hidden_size = hidden_size

    def forward(self, features, captions,hidden=None):
          embeddings = self.embedding(captions)
          embeddings = torch.cat((features.unsqueeze(1), embeddings), 1)
          out, _ = self.rnn(embeddings)
          out = self.fc(out)
          return out

    def reset_state(self, batch_size):
        return (torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device),
                torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device))
```

- These classes are combined into a single **ImageCaptioningModel** that handles the entire flow from image input to text output.

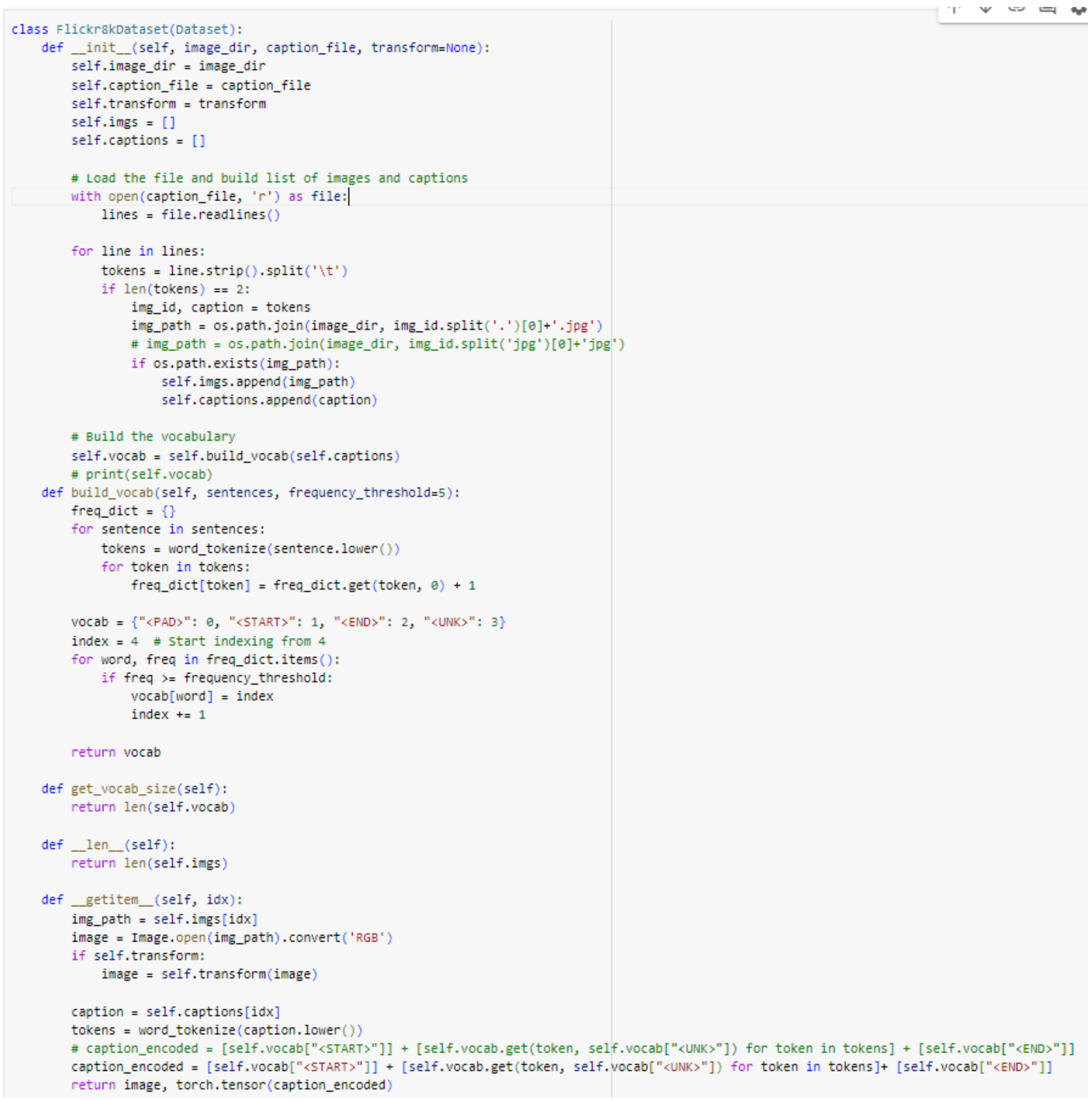


```
class ImageCaptioningModel(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size, num_layers, encoded_image_size=14):
        super(ImageCaptioningModel, self).__init__()
        self.encoder = ImageEncoder()
        self.decoder = CaptionDecoder(vocab_size, embed_size, hidden_size, num_layers)

    def forward(self, images, captions):
        features = self.encoder(images)
        outputs = self.decoder(features, captions)
        self.reset_state
        return outputs

    def reset_state(self, batch_size):
        return self.decoder.reset_state(batch_size)
```

Dataset Preparation

The **Flickr8kDataset** class is created to handle loading and preprocessing of image and text data:

- reads image paths and captions, checks for file existence, and processes the captions into numerical tokens using a vocabulary built from the dataset.
- applies transformations to images (resizing, normalization) to prepare them for the model.



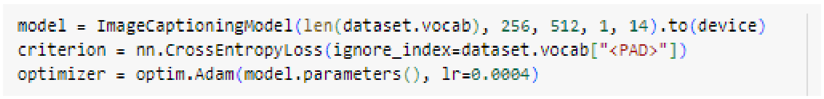
```python
class Flickr8kDataset(Dataset):
    def __init__(self, image_dir, caption_file, transform=None):
        self.image_dir = image_dir
        self.caption_file = caption_file
        self.transform = transform
        self.imgs = []
        self.captions = []

        # Load the file and build list of images and captions
        with open(caption_file, 'r') as file:
            lines = file.readlines()

        for line in lines:
            tokens = line.strip().split('\t')
            if len(tokens) == 2:
                img_id, caption = tokens
                img_path = os.path.join(image_dir, img_id.split('.')[0]+'.jpg')
                # img_path = os.path.join(image_dir, img_id.split('jpg')[0]+'jpg')
                if os.path.exists(img_path):
                    self.imgs.append(img_path)
                    self.captions.append(caption)

        # Build the vocabulary
        self.vocab = self.build_vocab(self.captions)
        # print(self.vocab)
    def build_vocab(self, sentences, frequency_threshold=5):
        freq_dict = {}
        for sentence in sentences:
            tokens = word_tokenize(sentence.lower())
            for token in tokens:
                freq_dict[token] = freq_dict.get(token, 0) + 1

        vocab = {"<PAD>": 0, "<START>": 1, "<END>": 2, "<UNK>": 3}
        index = 4  # Start indexing from 4
        for word, freq in freq_dict.items():
            if freq >= frequency_threshold:
                vocab[word] = index
                index += 1

        return vocab

    def get_vocab_size(self):
        return len(self.vocab)

    def __len__(self):
        return len(self.imgs)

    def __getitem__(self, idx):
        img_path = self.imgs[idx]
        image = Image.open(img_path).convert('RGB')
        if self.transform:
            image = self.transform(image)

        caption = self.captions[idx]
        tokens = word_tokenize(caption.lower())
        # caption_encoded = [self.vocab["<START>"]] + [self.vocab.get(token, self.vocab["<UNK>"]) for token in tokens] + [self.vocab["<END>"]]
        caption_encoded = [self.vocab["<START>"]] + [self.vocab.get(token, self.vocab["<UNK>"]) for token in tokens]+ [self.vocab["<END>"]]
        return image, torch.tensor(caption_encoded)
```

Data Loader Setup

DataLoaders for training and validation are set up using subsets of the dataset to feed data in batches during the training process.

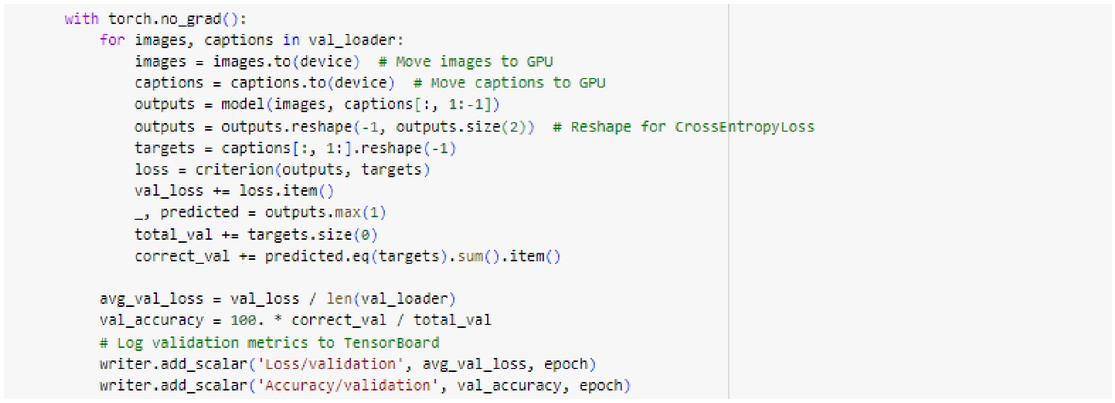
Training Preparation

- implement the image captioning model, criterion (loss function), and optimizer.



```
model = ImageCaptioningModel(len(dataset.vocab), 256, 512, 1, 14).to(device)
criterion = nn.CrossEntropyLoss(ignore_index=dataset.vocab["<PAD>"])
optimizer = optim.Adam(model.parameters(), lr=0.0004)
```

- The training loop is set up to train the model using the training data loader and validate using the validation data loader. Training and validation losses and accuracies are computed and logged to TensorBoard.



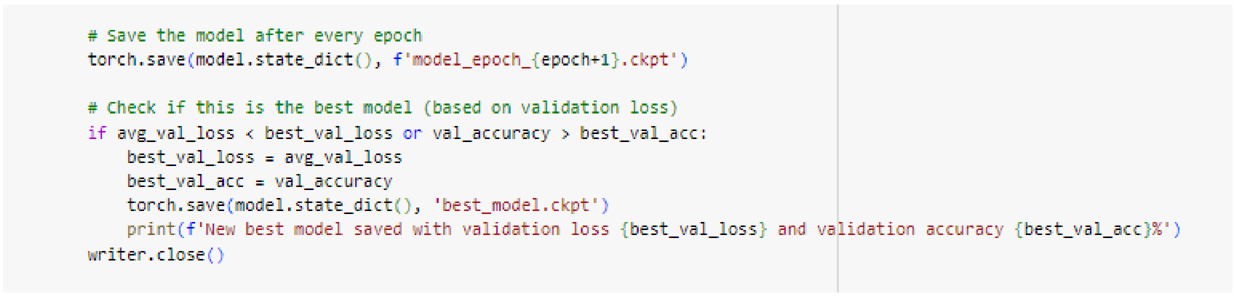
```
        with torch.no_grad():
            for images, captions in val_loader:
                images = images.to(device)  # Move images to GPU
                captions = captions.to(device)  # Move captions to GPU
                outputs = model(images, captions[:, 1:-1])
                outputs = outputs.reshape(-1, outputs.size(2))  # Reshape for CrossEntropyLoss
                targets = captions[:, 1:].reshape(-1)
                loss = criterion(outputs, targets)
                val_loss += loss.item()
                _, predicted = outputs.max(1)
                total_val += targets.size(0)
                correct_val += predicted.eq(targets).sum().item()

            avg_val_loss = val_loss / len(val_loader)
            val_accuracy = 100. * correct_val / total_val
            # Log validation metrics to TensorBoard
            writer.add_scalar('Loss/validation', avg_val_loss, epoch)
            writer.add_scalar('Accuracy/validation', val_accuracy, epoch)
```

Model Training

Model trained for a specified number of epochs, during each epoch feeds batches of images and captions through the model, compute the loss, update the model's weights, and evaluate model performance with the validation dataset

Saving and Loading the Model

After each epoch save the model based on it's performance , in this code store first epoch model as best model and then compare every model epoch performances with the best model performances and if current epoch optimized best model then it take as best model and save to local storage

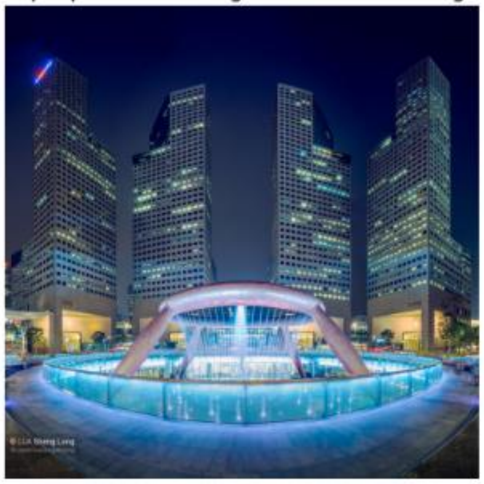


```
    # Save the model after every epoch
    torch.save(model.state_dict(), f'model_epoch_{epoch+1}.ckpt')

    # Check if this is the best model (based on validation loss)
    if avg_val_loss < best_val_loss or val_accuracy > best_val_acc:
        best_val_loss = avg_val_loss
        best_val_acc = val_accuracy
        torch.save(model.state_dict(), 'best_model.ckpt')
        print(f'New best model saved with validation loss {best_val_loss} and validation accuracy {best_val_acc}%')
    writer.close()
```

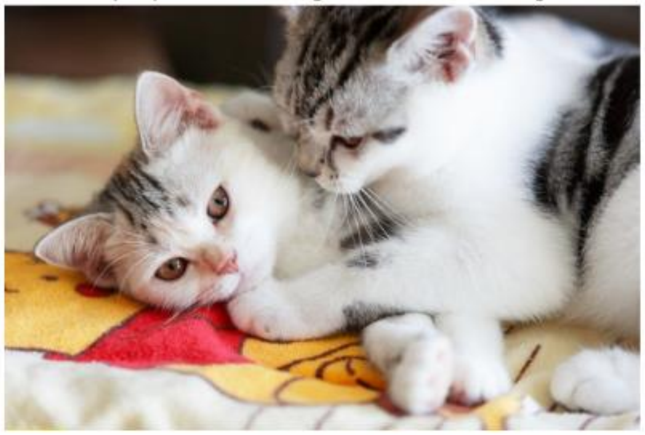
Image Caption Prediction

- The trained model is loaded and used to generate captions for new images.
- An image is processed, passed through the model, and the output tokens are converted back to text to form the caption.

```
of people are standing in front of a building .
```



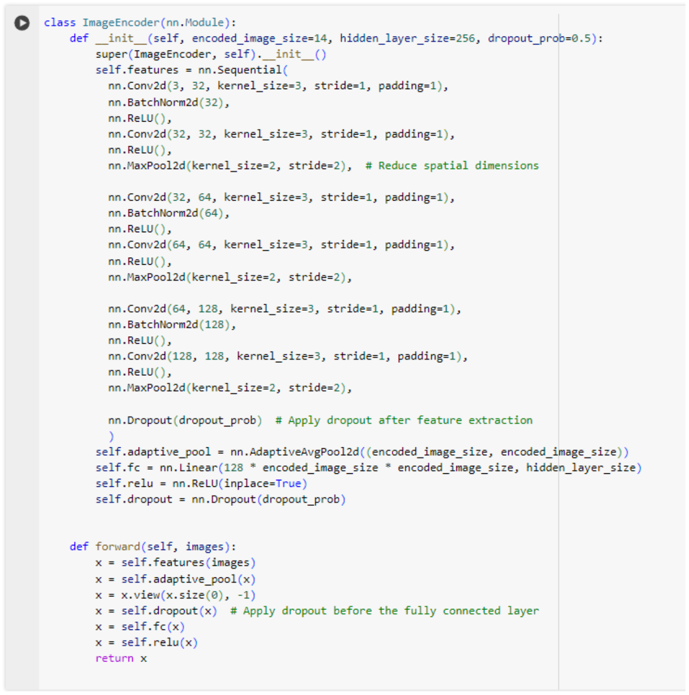
```
of people are standing in front of a building .
```



Experiment with different neural network architectures

Here is the 1st architecture outline:

1. Input Layer: Accepts the input images.
2. Convolutional Layers:
   Multiple blocks of Conv2d + BatchNorm2d + ReLU followed by MaxPool2d layers. These blocks progressively reduce the spatial dimensions and increase the depth of feature maps.
3. Adaptive Pooling Layer: Reduces the feature map to a specified size.
4. Flatten Layer: Converts the 2D feature maps into a 1D feature vector.
5. Dropout Layer: Applied before the fully connected layer for regularization.
6. Fully Connected Layer: Transforms the feature vector to the desired output size.
7. Output Activation: ReLU activation applied at the end.



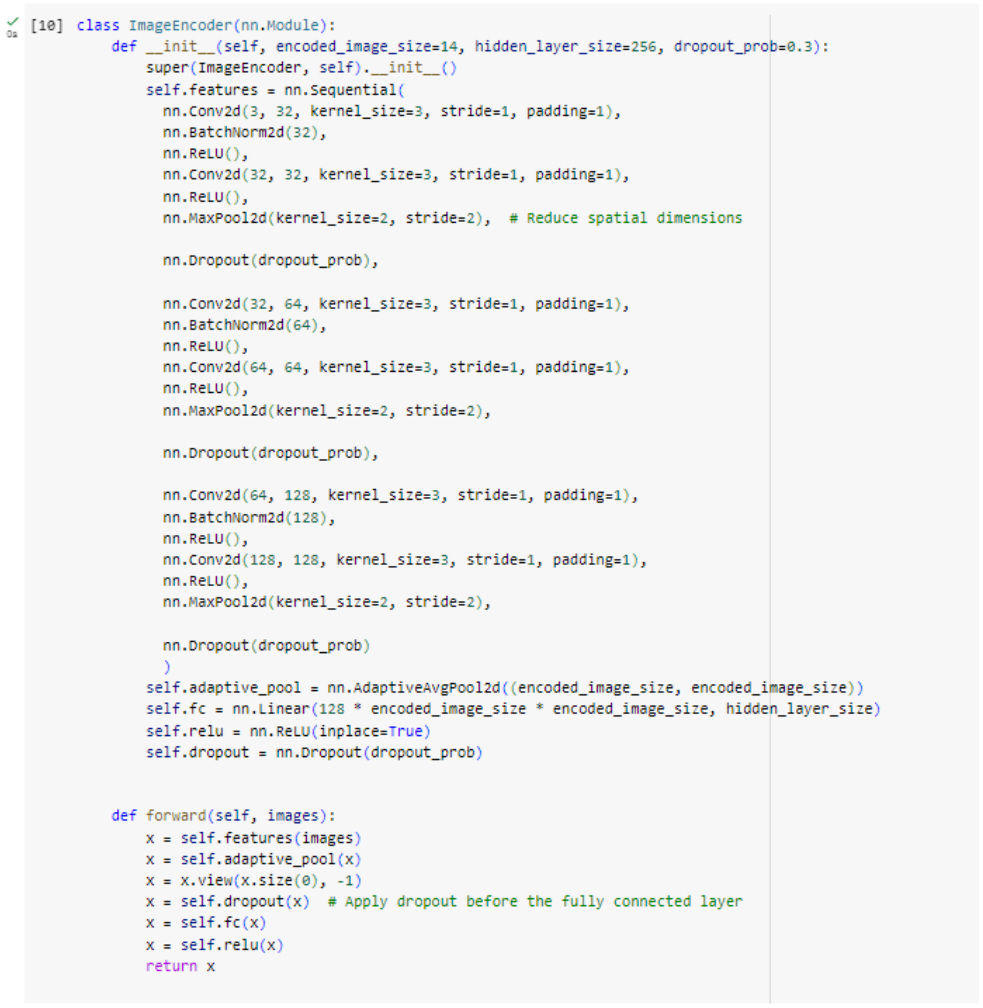
```python
class ImageEncoder(nn.Module):
    def __init__(self, encoded_image_size=14, hidden_layer_size=256, dropout_prob=0.5):
        super(ImageEncoder, self).__init__()
        self.features = nn.Sequential(
          nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1),
          nn.BatchNorm2d(32),
          nn.ReLU(),
          nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=1),
          nn.ReLU(),
          nn.MaxPool2d(kernel_size=2, stride=2),  # Reduce spatial dimensions

          nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
          nn.BatchNorm2d(64),
          nn.ReLU(),
          nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
          nn.ReLU(),
          nn.MaxPool2d(kernel_size=2, stride=2),

          nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
          nn.BatchNorm2d(128),
          nn.ReLU(),
          nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
          nn.ReLU(),
          nn.MaxPool2d(kernel_size=2, stride=2),

          nn.Dropout(dropout_prob)  # Apply dropout after feature extraction
          )
        self.adaptive_pool = nn.AdaptiveAvgPool2d((encoded_image_size, encoded_image_size))
        self.fc = nn.Linear(128 * encoded_image_size * encoded_image_size, hidden_layer_size)
        self.relu = nn.ReLU(inplace=True)
        self.dropout = nn.Dropout(dropout_prob)


    def forward(self, images):
        x = self.features(images)
        x = self.adaptive_pool(x)
        x = x.view(x.size(0), -1)
        x = self.dropout(x)  # Apply dropout before the fully connected layer
        x = self.fc(x)
        x = self.relu(x)
        return x
```

Model Training & Validation Results

```
Epoch 1, Training Loss: 3.2619, Validation Loss: 2.9435, Training Accuracy: 21.18%, Validation Accuracy: 22.82%
New best model saved with validation loss 2.943451478076075 and validation accuracy 22.817868276182264%
Epoch 2, Training Loss: 2.7357, Validation Loss: 2.8445, Training Accuracy: 23.47%, Validation Accuracy: 23.38%
New best model saved with validation loss 2.8444652312357905 and validation accuracy 23.382746346532244%
Epoch 3, Training Loss: 2.4959, Validation Loss: 2.8326, Training Accuracy: 24.81%, Validation Accuracy: 23.60%
New best model saved with validation loss 2.8326161878382266 and validation accuracy 23.601070062567707%
Epoch 4, Training Loss: 2.2845, Validation Loss: 2.8767, Training Accuracy: 26.30%, Validation Accuracy: 23.49%
Epoch 5, Training Loss: 2.0931, Validation Loss: 2.9456, Training Accuracy: 28.11%, Validation Accuracy: 23.29%
Epoch 6, Training Loss: 1.9241, Validation Loss: 3.0219, Training Accuracy: 29.87%, Validation Accuracy: 22.96%
Epoch 7, Training Loss: 1.7829, Validation Loss: 3.1186, Training Accuracy: 31.56%, Validation Accuracy: 22.54%
Epoch 8, Training Loss: 1.6612, Validation Loss: 3.2212, Training Accuracy: 33.18%, Validation Accuracy: 22.55%
Epoch 9, Training Loss: 1.5648, Validation Loss: 3.3195, Training Accuracy: 34.72%, Validation Accuracy: 22.24%
Epoch 10, Training Loss: 1.4855, Validation Loss: 3.4235, Training Accuracy: 35.73%, Validation Accuracy: 22.13%
```

Here is the 2nd architecture outline:

1. Initial Layers:
   - Convolutional Layers: Two Conv2d layers with 32 filters each, kernel size of 3, stride of 1, and padding of 1. These layers help in extracting various features from the input image at a finer level.
   - Batch Normalization (BatchNorm2d): Applied after each convolutional layer to normalize the outputs, helping to stabilize and speed up training.
   - Activation (ReLU): Applied after each batch normalization to introduce non-linearity into the model, allowing for more complex patterns to be learned.
   - Pooling Layer (MaxPool2d): Reduces the spatial dimensions (down sampling the feature maps) which decreases the computational cost and controls overfitting.
2. Additional Convolutional Blocks:
3. Each block includes similar layers (convolution, batch normalization, activation) with an increasing number of filters (64, then 128), allowing the network to capture more complex features at various levels.
4. Followed by max pooling to reduce dimensionality after feature extraction.
5. Dropout (Dropout):
6. Applied after pooling in each block with a dropout probability of 0.3. This is used to prevent overfitting by randomly zeroing out some of the features during training.
7. Adaptive Pooling (AdaptiveAvgPool2d):
8. This layer dynamically adjusts the pooling operation to achieve a fixed output size (as specified by encoded_image_size), making the model adaptable to various input image sizes.
9. Flatten and Fully Connected Layers:
10. Flattening: Converts the 2D feature maps into a 1D feature vector.
11. Fully Connected Layer (Linear): Transforms the flattened feature vector to the desired output size (hidden_layer_size).
12. Activation (ReLU): Ensures the non-linearity after the fully connected layer.
13. Output Dropout:
14. Another dropout layer is applied before the final output to further aid in preventing overfitting.



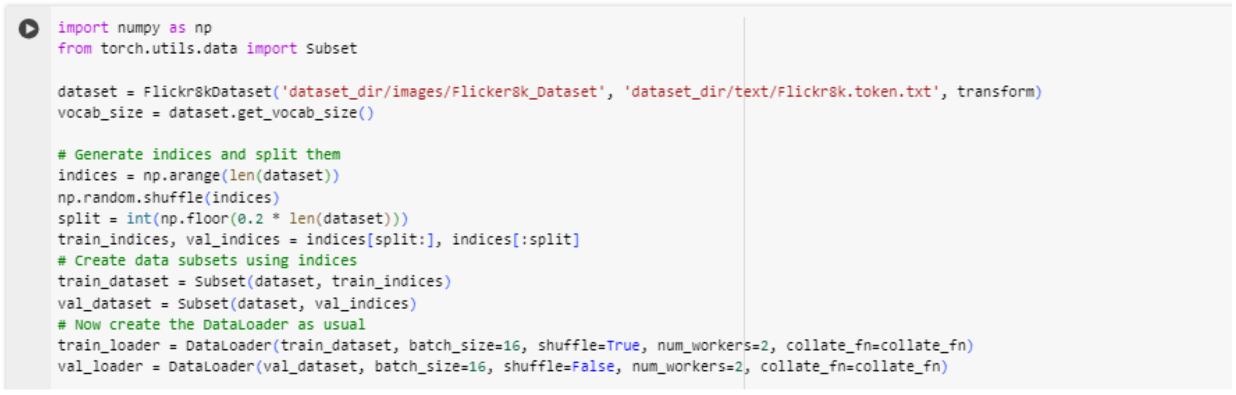
```
[10] class ImageEncoder(nn.Module):
         def __init__(self, encoded_image_size=14, hidden_layer_size=256, dropout_prob=0.3):
             super(ImageEncoder, self).__init__()
             self.features = nn.Sequential(
               nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1),
               nn.BatchNorm2d(32),
               nn.ReLU(),
               nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=1),
               nn.ReLU(),
               nn.MaxPool2d(kernel_size=2, stride=2),  # Reduce spatial dimensions

               nn.Dropout(dropout_prob),

               nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
               nn.BatchNorm2d(64),
               nn.ReLU(),
               nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
               nn.ReLU(),
               nn.MaxPool2d(kernel_size=2, stride=2),

               nn.Dropout(dropout_prob),

               nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
               nn.BatchNorm2d(128),
               nn.ReLU(),
               nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
               nn.ReLU(),
               nn.MaxPool2d(kernel_size=2, stride=2),

               nn.Dropout(dropout_prob)
               )
             self.adaptive_pool = nn.AdaptiveAvgPool2d((encoded_image_size, encoded_image_size))
             self.fc = nn.Linear(128 * encoded_image_size * encoded_image_size, hidden_layer_size)
             self.relu = nn.ReLU(inplace=True)
             self.dropout = nn.Dropout(dropout_prob)


         def forward(self, images):
             x = self.features(images)
             x = self.adaptive_pool(x)
             x = x.view(x.size(0), -1)
             x = self.dropout(x)  # Apply dropout before the fully connected layer
             x = self.fc(x)
             x = self.relu(x)
             return x
```

Model Training & Validation Results

```
Epoch 1, Training Loss: 3.2511, Validation Loss: 2.9468, Training Accuracy: 21.25%, Validation Accuracy: 22.78%
New best model saved with validation loss 2.9467590938914907 and validation accuracy 22.783591898238534%
Epoch 2, Training Loss: 2.7345, Validation Loss: 2.8465, Training Accuracy: 23.58%, Validation Accuracy: 23.37%
New best model saved with validation loss 2.8465474500015318 and validation accuracy 23.369831222468473%
Epoch 3, Training Loss: 2.4973, Validation Loss: 2.8388, Training Accuracy: 24.87%, Validation Accuracy: 23.47%
New best model saved with validation loss 2.8387964054529844 and validation accuracy 23.47487035091868%
Epoch 4, Training Loss: 2.2937, Validation Loss: 2.8838, Training Accuracy: 26.25%, Validation Accuracy: 23.30%
Epoch 5, Training Loss: 2.1059, Validation Loss: 2.9563, Training Accuracy: 27.97%, Validation Accuracy: 22.95%
Epoch 6, Training Loss: 1.9398, Validation Loss: 3.0360, Training Accuracy: 29.81%, Validation Accuracy: 22.82%
Epoch 7, Training Loss: 1.7973, Validation Loss: 3.1192, Training Accuracy: 31.51%, Validation Accuracy: 22.58%
Epoch 8, Training Loss: 1.6841, Validation Loss: 3.2233, Training Accuracy: 32.91%, Validation Accuracy: 22.36%
Epoch 9, Training Loss: 1.5843, Validation Loss: 3.3161, Training Accuracy: 34.26%, Validation Accuracy: 22.20%
Epoch 10, Training Loss: 1.4986, Validation Loss: 3.4318, Training Accuracy: 35.52%, Validation Accuracy: 21.88%
```

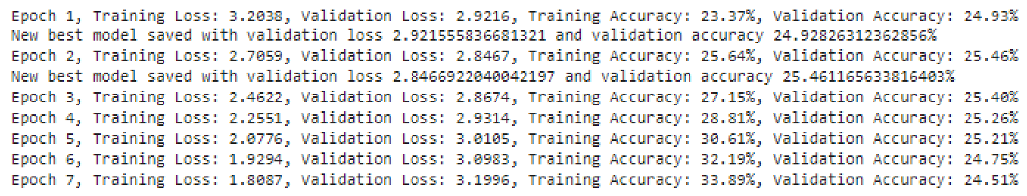
Experiment with different hyperparameters and regularization methods

Reduce batch size from 32 to 16 & Learning Rate 0.002



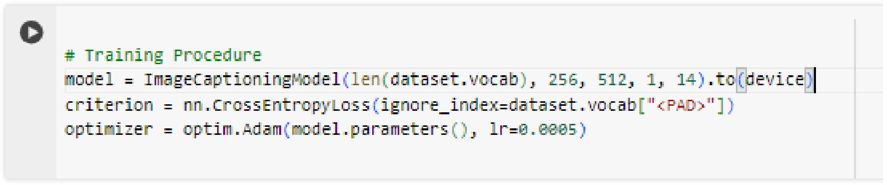
```
import numpy as np
from torch.utils.data import Subset

dataset = Flickr8kDataset('dataset_dir/images/Flicker8k_Dataset', 'dataset_dir/text/Flickr8k.token.txt', transform)
vocab_size = dataset.get_vocab_size()

# Generate indices and split them
indices = np.arange(len(dataset))
np.random.shuffle(indices)
split = int(np.floor(0.2 * len(dataset)))
train_indices, val_indices = indices[split:], indices[:split]
# Create data subsets using indices
train_dataset = Subset(dataset, train_indices)
val_dataset = Subset(dataset, val_indices)
# Now create the DataLoader as usual
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True, num_workers=2, collate_fn=collate_fn)
val_loader = DataLoader(val_dataset, batch_size=16, shuffle=False, num_workers=2, collate_fn=collate_fn)
```

Model Training & Validation Results



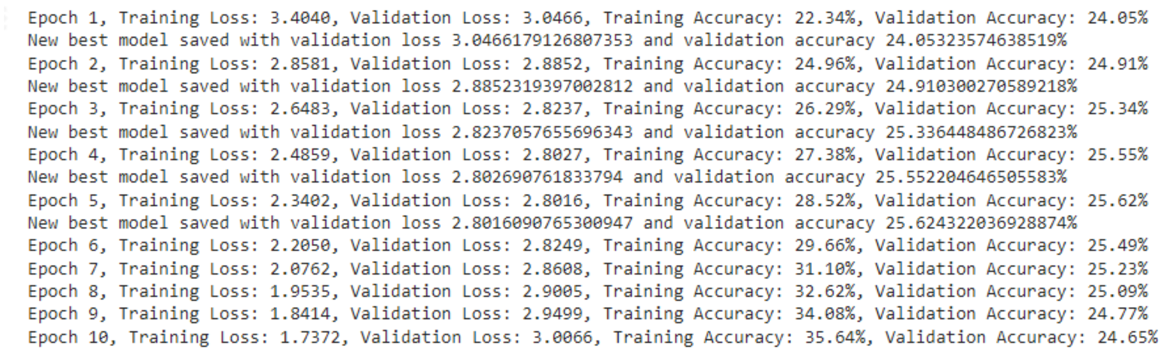
```
Epoch 1, Training Loss: 3.2038, Validation Loss: 2.9216, Training Accuracy: 23.37%, Validation Accuracy: 24.93%
New best model saved with validation loss 2.921555836681321 and validation accuracy 24.92826312362856%
Epoch 2, Training Loss: 2.7059, Validation Loss: 2.8467, Training Accuracy: 25.64%, Validation Accuracy: 25.46%
New best model saved with validation loss 2.8466922040042197 and validation accuracy 25.461165633816403%
Epoch 3, Training Loss: 2.4622, Validation Loss: 2.8674, Training Accuracy: 27.15%, Validation Accuracy: 25.40%
Epoch 4, Training Loss: 2.2551, Validation Loss: 2.9314, Training Accuracy: 28.81%, Validation Accuracy: 25.26%
Epoch 5, Training Loss: 2.0776, Validation Loss: 3.0105, Training Accuracy: 30.61%, Validation Accuracy: 25.21%
Epoch 6, Training Loss: 1.9294, Validation Loss: 3.0983, Training Accuracy: 32.19%, Validation Accuracy: 24.75%
Epoch 7, Training Loss: 1.8087, Validation Loss: 3.1996, Training Accuracy: 33.89%, Validation Accuracy: 24.51%
```

Change learning rate 0.002 to 0.0005



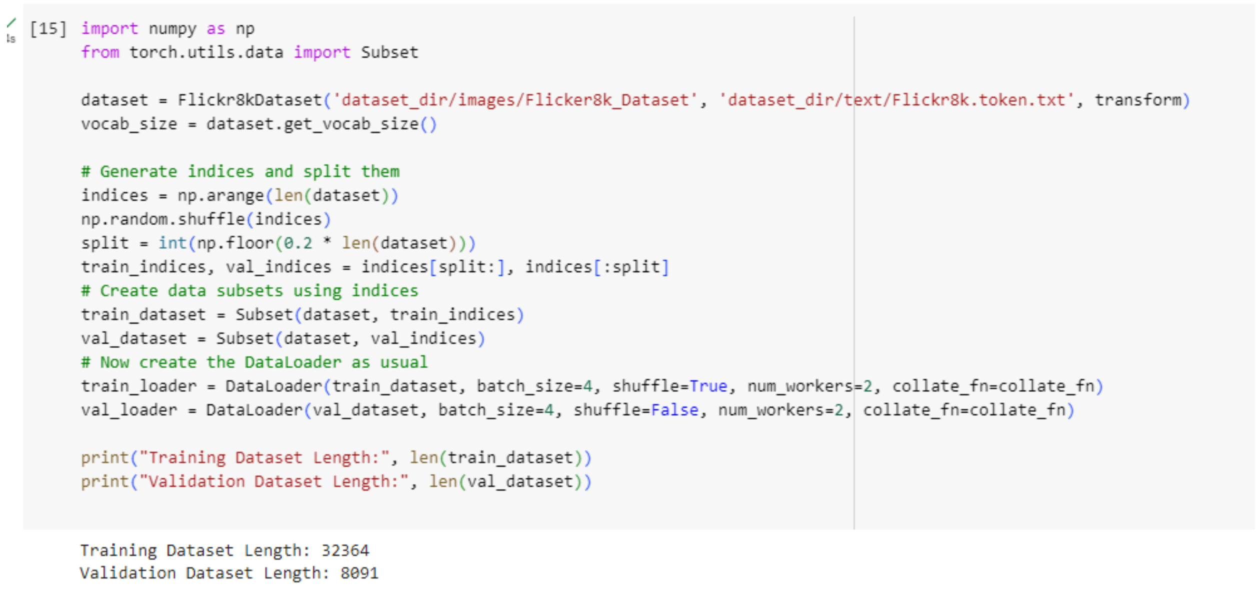
```
# Training Procedure
model = ImageCaptioningModel(len(dataset.vocab), 256, 512, 1, 14).to(device)
criterion = nn.CrossEntropyLoss(ignore_index=dataset.vocab["<PAD>"])
optimizer = optim.Adam(model.parameters(), lr=0.0005)
```

Model Training & Validation Results



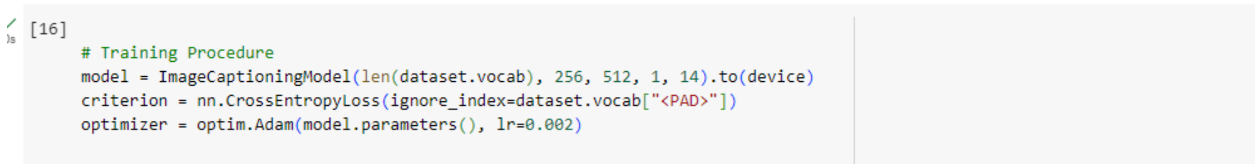
```
Epoch 1, Training Loss: 3.4040, Validation Loss: 3.0466, Training Accuracy: 22.34%, Validation Accuracy: 24.05%
New best model saved with validation loss 3.0466179126807353 and validation accuracy 24.053235746638519%
Epoch 2, Training Loss: 2.8581, Validation Loss: 2.8852, Training Accuracy: 24.96%, Validation Accuracy: 24.91%
New best model saved with validation loss 2.8852319397002812 and validation accuracy 24.910300270589218%
Epoch 3, Training Loss: 2.6483, Validation Loss: 2.8237, Training Accuracy: 26.29%, Validation Accuracy: 25.34%
New best model saved with validation loss 2.8237057655696343 and validation accuracy 25.336448486726823%
Epoch 4, Training Loss: 2.4859, Validation Loss: 2.8027, Training Accuracy: 27.38%, Validation Accuracy: 25.55%
New best model saved with validation loss 2.802690761833794 and validation accuracy 25.552204646505583%
Epoch 5, Training Loss: 2.3402, Validation Loss: 2.8016, Training Accuracy: 28.52%, Validation Accuracy: 25.62%
New best model saved with validation loss 2.8016090765300947 and validation accuracy 25.624322036928874%
Epoch 6, Training Loss: 2.2050, Validation Loss: 2.8249, Training Accuracy: 29.66%, Validation Accuracy: 25.49%
Epoch 7, Training Loss: 2.0762, Validation Loss: 2.8608, Training Accuracy: 31.10%, Validation Accuracy: 25.23%
Epoch 8, Training Loss: 1.9535, Validation Loss: 2.9005, Training Accuracy: 32.62%, Validation Accuracy: 25.09%
Epoch 9, Training Loss: 1.8414, Validation Loss: 2.9499, Training Accuracy: 34.08%, Validation Accuracy: 24.77%
Epoch 10, Training Loss: 1.7372, Validation Loss: 3.0066, Training Accuracy: 35.64%, Validation Accuracy: 24.65%
```

Reverse Learning rate and reduce batch size to 4

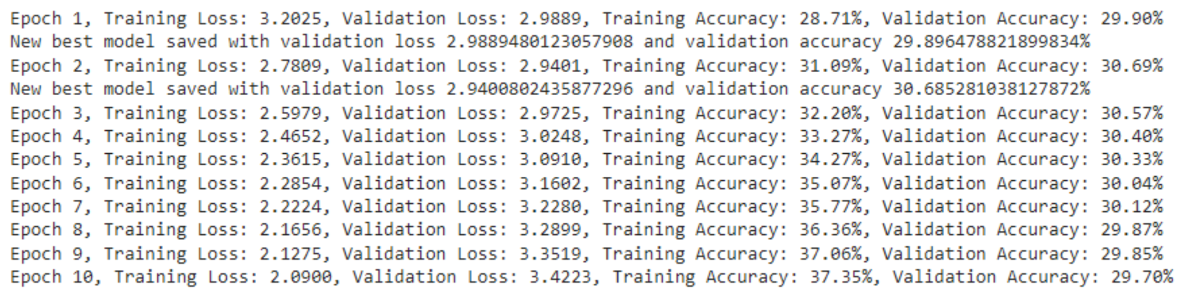


```
[15] import numpy as np
     from torch.utils.data import Subset

     dataset = Flickr8kDataset('dataset_dir/images/Flicker8k_Dataset', 'dataset_dir/text/Flickr8k.token.txt', transform)
     vocab_size = dataset.get_vocab_size()

     # Generate indices and split them
     indices = np.arange(len(dataset))
     np.random.shuffle(indices)
     split = int(np.floor(0.2 * len(dataset)))
     train_indices, val_indices = indices[split:], indices[:split]
     # Create data subsets using indices
     train_dataset = Subset(dataset, train_indices)
     val_dataset = Subset(dataset, val_indices)
     # Now create the DataLoader as usual
     train_loader = DataLoader(train_dataset, batch_size=4, shuffle=True, num_workers=2, collate_fn=collate_fn)
     val_loader = DataLoader(val_dataset, batch_size=4, shuffle=False, num_workers=2, collate_fn=collate_fn)

     print("Training Dataset Length:", len(train_dataset))
     print("Validation Dataset Length:", len(val_dataset))
```

```
Training Dataset Length: 32364
Validation Dataset Length: 8091
```



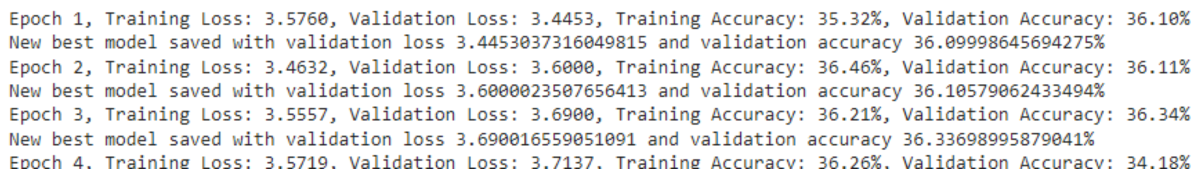
```
[16]
     # Training Procedure
     model = ImageCaptioningModel(len(dataset.vocab), 256, 512, 1, 14).to(device)
     criterion = nn.CrossEntropyLoss(ignore_index=dataset.vocab["<PAD>"])
     optimizer = optim.Adam(model.parameters(), lr=0.002)
```



```
Epoch 1, Training Loss: 3.2025, Validation Loss: 2.9889, Training Accuracy: 28.71%, Validation Accuracy: 29.90%
New best model saved with validation loss 2.9889480123057908 and validation accuracy 29.896478821899834%
Epoch 2, Training Loss: 2.7809, Validation Loss: 2.9401, Training Accuracy: 31.09%, Validation Accuracy: 30.69%
New best model saved with validation loss 2.9400802435877296 and validation accuracy 30.685281038127872%
Epoch 3, Training Loss: 2.5979, Validation Loss: 2.9725, Training Accuracy: 32.20%, Validation Accuracy: 30.57%
Epoch 4, Training Loss: 2.4652, Validation Loss: 3.0248, Training Accuracy: 33.27%, Validation Accuracy: 30.40%
Epoch 5, Training Loss: 2.3615, Validation Loss: 3.0910, Training Accuracy: 34.27%, Validation Accuracy: 30.33%
Epoch 6, Training Loss: 2.2854, Validation Loss: 3.1602, Training Accuracy: 35.07%, Validation Accuracy: 30.04%
Epoch 7, Training Loss: 2.2224, Validation Loss: 3.2280, Training Accuracy: 35.77%, Validation Accuracy: 30.12%
Epoch 8, Training Loss: 2.1656, Validation Loss: 3.2899, Training Accuracy: 36.36%, Validation Accuracy: 29.87%
Epoch 9, Training Loss: 2.1275, Validation Loss: 3.3519, Training Accuracy: 37.06%, Validation Accuracy: 29.85%
Epoch 10, Training Loss: 2.0900, Validation Loss: 3.4223, Training Accuracy: 37.35%, Validation Accuracy: 29.70%
```

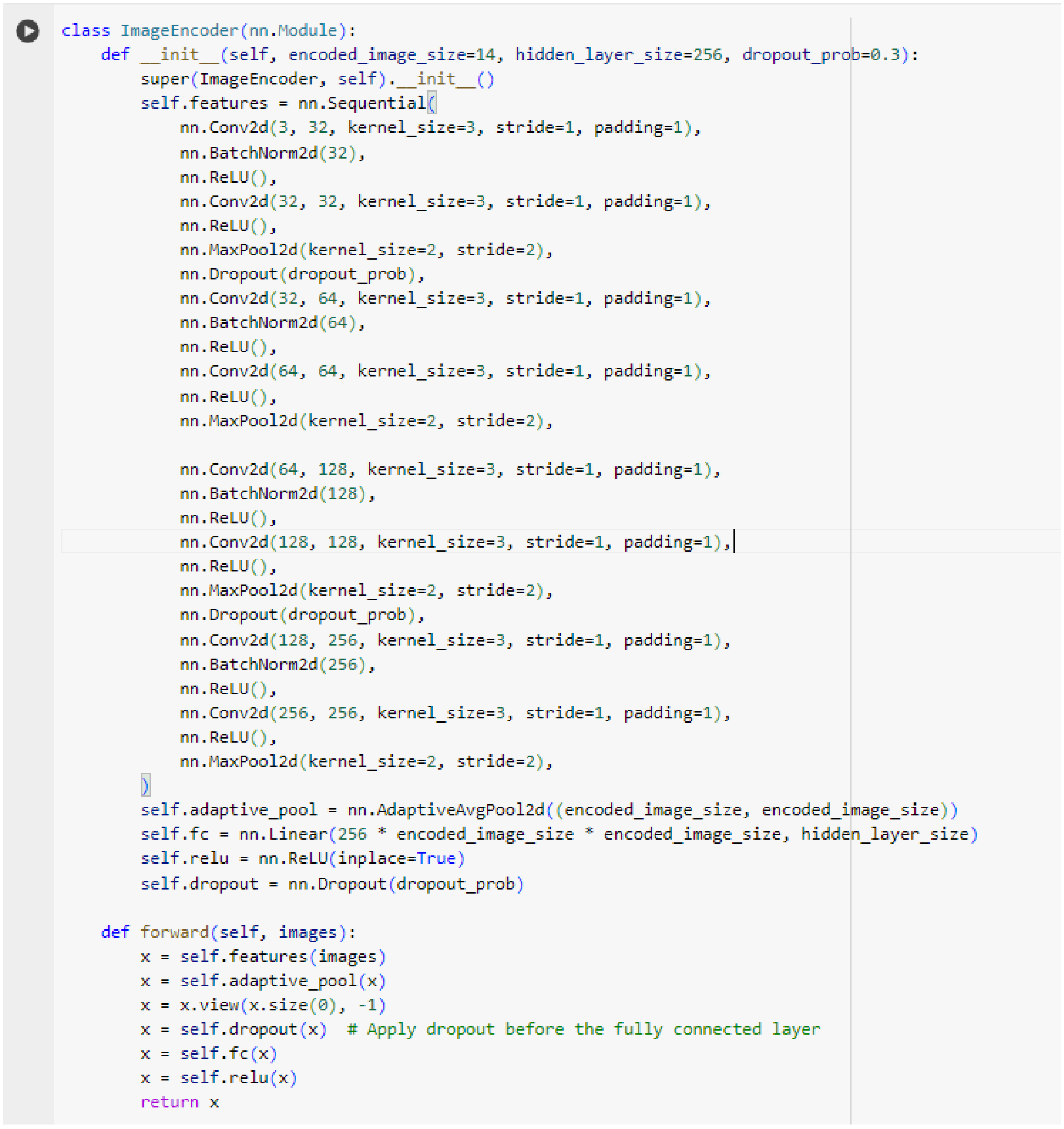
Batch Size = 1

Learning Rate = 0.003



```
Epoch 1, Training Loss: 3.5760, Validation Loss: 3.4453, Training Accuracy: 35.32%, Validation Accuracy: 36.10%
New best model saved with validation loss 3.4453037316049815 and validation accuracy 36.09998645694275%
Epoch 2, Training Loss: 3.4632, Validation Loss: 3.6000, Training Accuracy: 36.46%, Validation Accuracy: 36.11%
New best model saved with validation loss 3.6000023507656413 and validation accuracy 36.10579062433494%
Epoch 3, Training Loss: 3.5557, Validation Loss: 3.6900, Training Accuracy: 36.21%, Validation Accuracy: 36.34%
New best model saved with validation loss 3.690016559051091 and validation accuracy 36.33698995879041%
Epoch 4, Training Loss: 3.5719, Validation Loss: 3.7137, Training Accuracy: 36.26%, Validation Accuracy: 34.18%
```

Learning rate   = 5e-5



```python
class ImageEncoder(nn.Module):
    def __init__(self, encoded_image_size=14, hidden_layer_size=256, dropout_prob=0.3):
        super(ImageEncoder, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(dropout_prob),
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(dropout_prob),
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.adaptive_pool = nn.AdaptiveAvgPool2d((encoded_image_size, encoded_image_size))
        self.fc = nn.Linear(256 * encoded_image_size * encoded_image_size, hidden_layer_size)
        self.relu = nn.ReLU(inplace=True)
        self.dropout = nn.Dropout(dropout_prob)

    def forward(self, images):
        x = self.features(images)
        x = self.adaptive_pool(x)
        x = x.view(x.size(0), -1)
        x = self.dropout(x)  # Apply dropout before the fully connected layer
        x = self.fc(x)
        x = self.relu(x)
        return x
```

```
Epoch 1, Training Loss: 3.8892, Validation Loss: 3.4625, Training Accuracy: 24.50%, Validation Accuracy: 27.12%
New best model saved with validation loss 3.462541566764843 and validation accuracy 27.12462666304643%
Epoch 2, Training Loss: 3.3012, Validation Loss: 3.2264, Training Accuracy: 27.90%, Validation Accuracy: 28.60%
New best model saved with validation loss 3.226351773567973 and validation accuracy 28.599628681084017%
Epoch 3, Training Loss: 3.1034, Validation Loss: 3.1045, Training Accuracy: 29.09%, Validation Accuracy: 29.47%
New best model saved with validation loss 3.104534246222665 and validation accuracy 29.47068708676094%
Epoch 4, Training Loss: 2.9796, Validation Loss: 3.0234, Training Accuracy: 29.91%, Validation Accuracy: 29.97%
New best model saved with validation loss 3.023403839315451 and validation accuracy 29.96675741720542%
Epoch 5, Training Loss: 2.8858, Validation Loss: 2.9674, Training Accuracy: 30.52%, Validation Accuracy: 30.37%
New best model saved with validation loss 2.967435947928285 and validation accuracy 30.3725664301282%
Epoch 6, Training Loss: 2.8087, Validation Loss: 2.9271, Training Accuracy: 31.08%, Validation Accuracy: 30.66%
New best model saved with validation loss 2.9271023318252527 and validation accuracy 30.65949468338825%
Epoch 7, Training Loss: 2.7427, Validation Loss: 2.8990, Training Accuracy: 31.60%, Validation Accuracy: 30.84%
New best model saved with validation loss 2.898959975636247 and validation accuracy 30.836348159182805%
Epoch 8, Training Loss: 2.6857, Validation Loss: 2.8753, Training Accuracy: 31.98%, Validation Accuracy: 31.04%
New best model saved with validation loss 2.875324533708132 and validation accuracy 31.04328874081793%
Epoch 9, Training Loss: 2.6324, Validation Loss: 2.8571, Training Accuracy: 32.49%, Validation Accuracy: 31.26%
New best model saved with validation loss 2.8571399140935454 and validation accuracy 31.263438295748912%
Epoch 10, Training Loss: 2.5855, Validation Loss: 2.8439, Training Accuracy: 32.83%, Validation Accuracy: 31.26%
New best model saved with validation loss 2.84388864700357 and validation accuracy 31.2612368001996%
```

Try with Change Optimizer to RMSprop and Learning Rate: 0.001

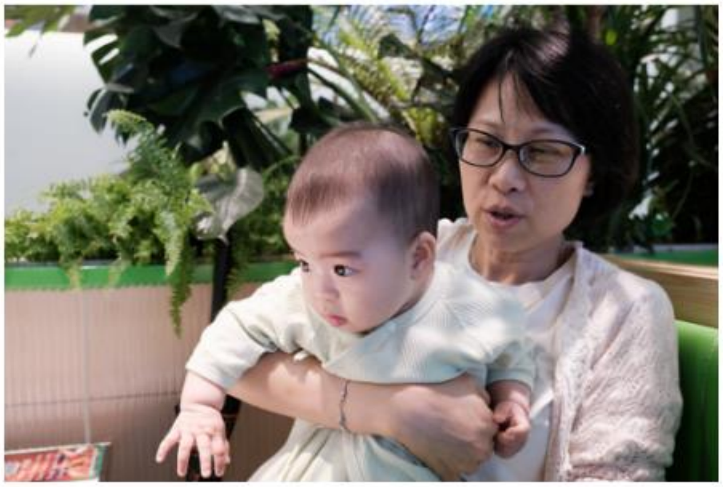
```python
model = ImageCaptioningModel(len(dataset.vocab), 256, 512, 1, 14).to(device)
criterion = nn.CrossEntropyLoss(ignore_index=dataset.vocab["<PAD>"])
optimizer = optim.RMSprop(model.parameters(), lr=0.001)
```

Model Training & Validation Results

```
Epoch 1, Training Loss: 3.3096, Validation Loss: 3.0034, Training Accuracy: 19.43%, Validation Accuracy: 21.09%
New best model saved with validation loss 3.0033720816214253 and validation accuracy 21.086496967951323%
Epoch 2, Training Loss: 2.8380, Validation Loss: 2.8749, Training Accuracy: 21.51%, Validation Accuracy: 21.68%
New best model saved with validation loss 2.8749297754032406 and validation accuracy 21.68210551220792%
```

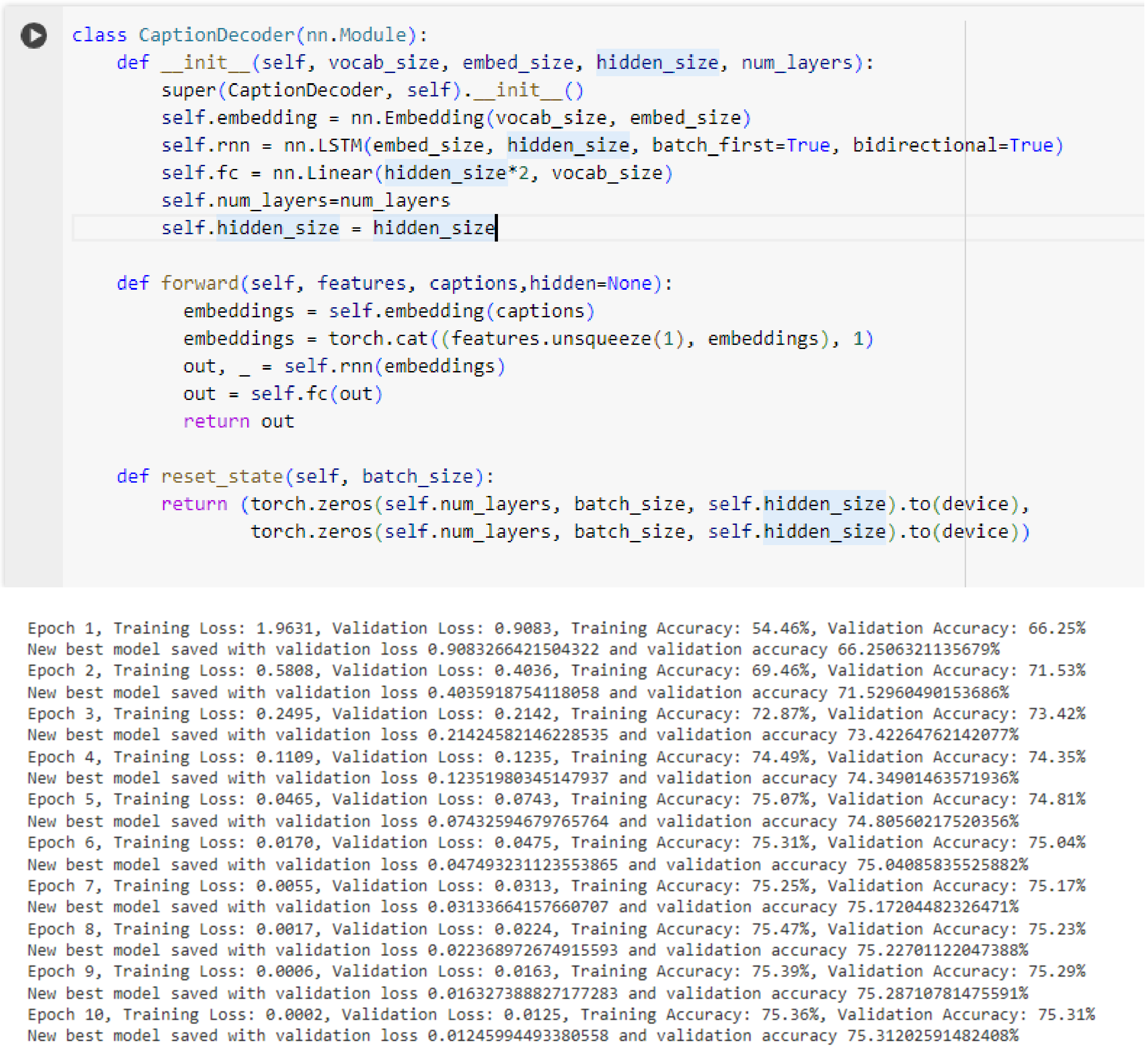
Modal Usage – Caption Generation with trained Model

<UNK> in a race .



Changes for Bidirectional LSTM

To make the LSTM in the decoder bidirectional, you simply need to add the **bidirectional=True** parameter to the LSTM constructor and adjust the subsequent layers to handle the doubled output size.

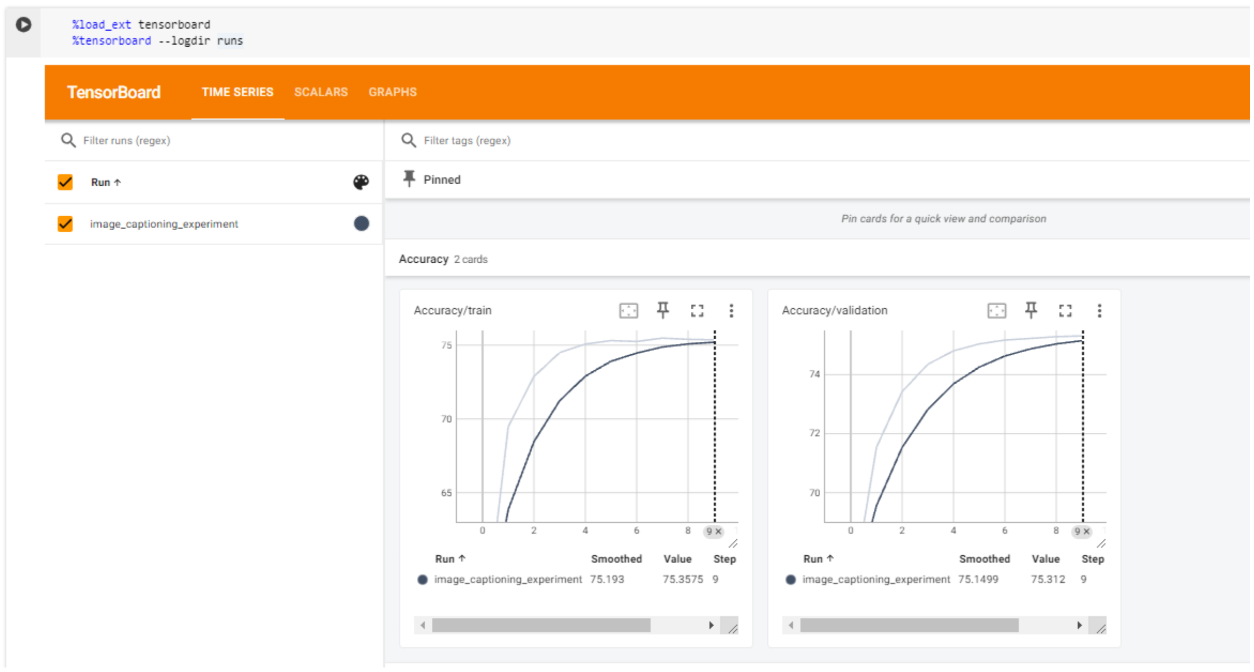


```
class CaptionDecoder(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size, num_layers):
        super(CaptionDecoder, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.LSTM(embed_size, hidden_size, batch_first=True, bidirectional=True)
        self.fc = nn.Linear(hidden_size*2, vocab_size)
        self.num_layers=num_layers
        self.hidden_size = hidden_size

    def forward(self, features, captions,hidden=None):
          embeddings = self.embedding(captions)
          embeddings = torch.cat((features.unsqueeze(1), embeddings), 1)
          out, _ = self.rnn(embeddings)
          out = self.fc(out)
          return out

    def reset_state(self, batch_size):
        return (torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device),
                torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device))
```

```
Epoch 1, Training Loss: 1.9631, Validation Loss: 0.9083, Training Accuracy: 54.46%, Validation Accuracy: 66.25%
New best model saved with validation loss 0.9083266421504322 and validation accuracy 66.2506321135679%
Epoch 2, Training Loss: 0.5808, Validation Loss: 0.4036, Training Accuracy: 69.46%, Validation Accuracy: 71.53%
New best model saved with validation loss 0.4035918754118058 and validation accuracy 71.52960490153686%
Epoch 3, Training Loss: 0.2495, Validation Loss: 0.2142, Training Accuracy: 72.87%, Validation Accuracy: 73.42%
New best model saved with validation loss 0.21424582146228535 and validation accuracy 73.42264762142077%
Epoch 4, Training Loss: 0.1109, Validation Loss: 0.1235, Training Accuracy: 74.49%, Validation Accuracy: 74.35%
New best model saved with validation loss 0.12351980345147937 and validation accuracy 74.34901463571936%
Epoch 5, Training Loss: 0.0465, Validation Loss: 0.0743, Training Accuracy: 75.07%, Validation Accuracy: 74.81%
New best model saved with validation loss 0.07432594679765764 and validation accuracy 74.80560217520356%
Epoch 6, Training Loss: 0.0170, Validation Loss: 0.0475, Training Accuracy: 75.31%, Validation Accuracy: 75.04%
New best model saved with validation loss 0.047493231123553865 and validation accuracy 75.04085835525882%
Epoch 7, Training Loss: 0.0055, Validation Loss: 0.0313, Training Accuracy: 75.25%, Validation Accuracy: 75.17%
New best model saved with validation loss 0.03133664157660707 and validation accuracy 75.17204482326471%
Epoch 8, Training Loss: 0.0017, Validation Loss: 0.0224, Training Accuracy: 75.47%, Validation Accuracy: 75.23%
New best model saved with validation loss 0.022368972674915593 and validation accuracy 75.22701122047388%
Epoch 9, Training Loss: 0.0006, Validation Loss: 0.0163, Training Accuracy: 75.39%, Validation Accuracy: 75.29%
New best model saved with validation loss 0.016327388827177283 and validation accuracy 75.28710781475591%
Epoch 10, Training Loss: 0.0002, Validation Loss: 0.0125, Training Accuracy: 75.36%, Validation Accuracy: 75.31%
New best model saved with validation loss 0.01245994493380558 and validation accuracy 75.31202591482408%
```

Implementing Tensor Board for visualizing the training and validation process of this Model

Tensor Board is a tool that helps visualize the training and validation of machine learning models. It allows you to monitor metrics like loss and accuracy in real-time, making it easier to see how well your model is performing and to make necessary adjustments. This tool is particularly useful for comparing different training runs, visualizing model architectures, and understanding changes in data like weights and biases during training. Essentially, Tensor Board helps you understand and optimize your models more effectively.



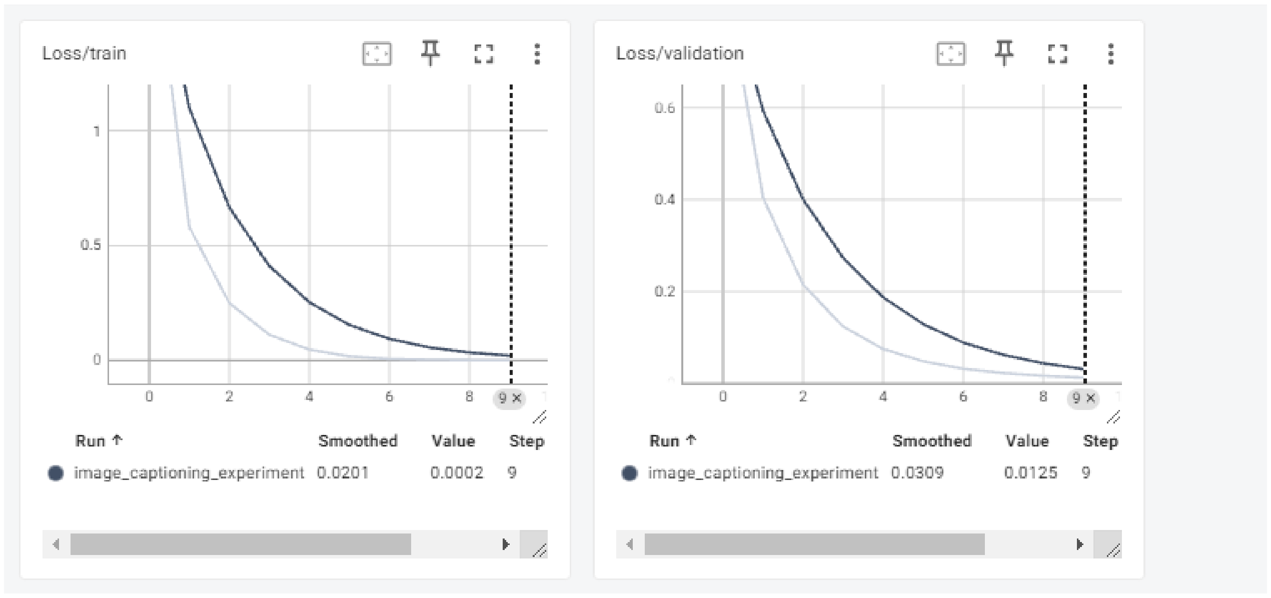
```
%load_ext tensorboard
%tensorboard --logdir runs
```

Accuracy/train

```
Run ↑                          Smoothed   Value     Step
● image_captioning_experiment  75.193     75.3575   9
```

Accuracy/validation

```
Run ↑                          Smoothed   Value     Step
● image_captioning_experiment  75.1499    75.312    9
```

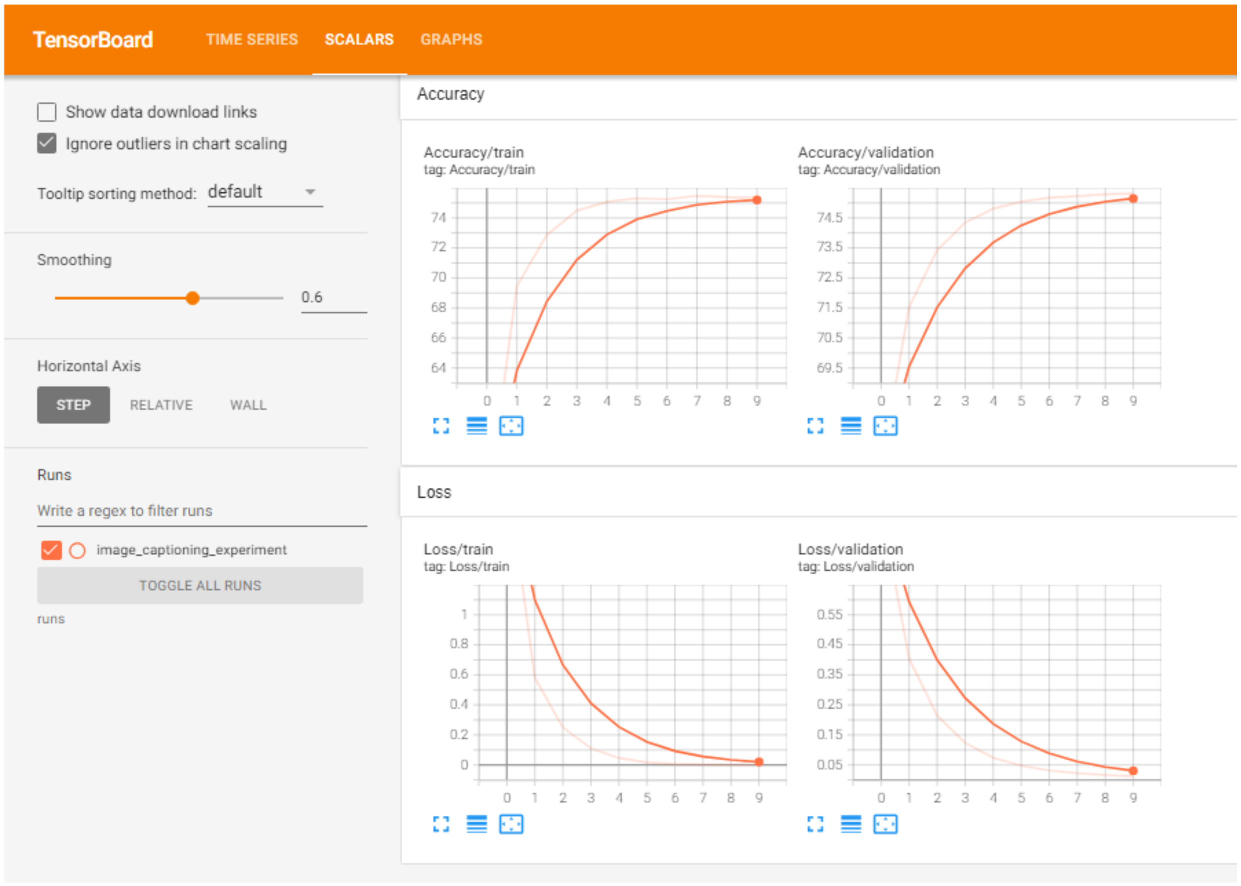


Loss/train

```
Run ↑                          Smoothed   Value     Step
● image_captioning_experiment  0.0201     0.0002    9
```

Loss/validation

```
Run ↑                          Smoothed   Value     Step
● image_captioning_experiment  0.0309     0.0125    9
```



Phase 1 conclusion

In this assignment i used google colab for code and executions , in this image captioning model created from scratch by utilizing Flickr8k dataset it trains an image captioning model that combines CNN and RNN, specially LSTM units to generate descriptive captions from image . This model is trained to extract significant features from images and then translate these features into coherent, contextual captions using natural language.

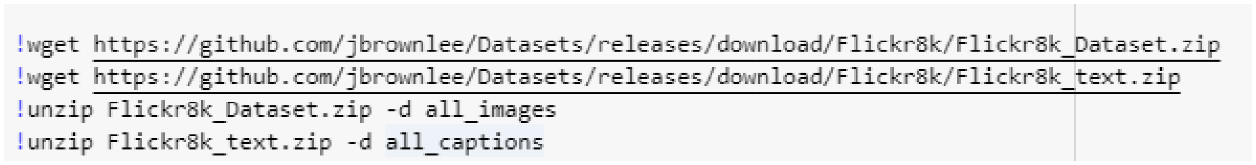
Implement TensorBoard for monitoring training and validation performance provides valuable insights that are crucial for model optimization. and also, i tried with various model network design and various hyper parameter changes for getting performance in this image captioning model

Phase 2: Image Generation

I used google colab for development environment because it allows free GPU processing power that is helpful to training my model free of charge with limited time period.

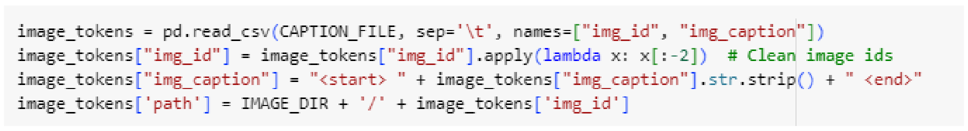
Overview of the Process

Data Acquisition: download image dataset and captions from public online source (Flickr8k) and extract to local directory in google colabs



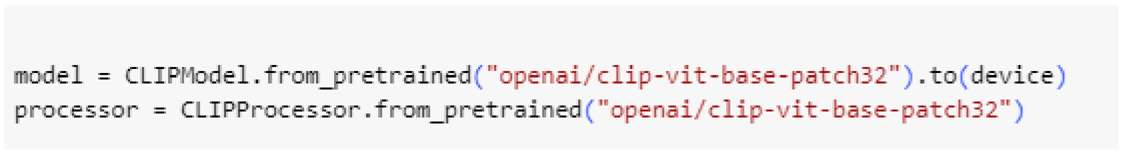
```
!wget https://github.com/jbrownlee/Datasets/releases/download/Flickr8k/Flickr8k_Dataset.zip
!wget https://github.com/jbrownlee/Datasets/releases/download/Flickr8k/Flickr8k_text.zip
!unzip Flickr8k_Dataset.zip -d all_images
!unzip Flickr8k_text.zip -d all_captions
```

Preparation of the Data: clean images name and ids and captions processed adding special tokens ('<start>' and '<end>') for identify beginning and end of each caption this helps to provide structured format and then Model can process effectively



```
image_tokens = pd.read_csv(CAPTION_FILE, sep='\t', names=["img_id", "img_caption"])
image_tokens["img_id"] = image_tokens["img_id"].apply(lambda x: x[:-2])  # Clean image ids
image_tokens["img_caption"] = "<start> " + image_tokens["img_caption"].str.strip() + " <end>"
image_tokens['path'] = IMAGE_DIR + '/' + image_tokens['img_id']
```

Loading AI Models: this assignment uses pre-trained models from the 'transformers' library specifically a CLIP model , this model designed for understanding both visual and textual data.



```
model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32").to(device)
processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
```

Embedding Generation:

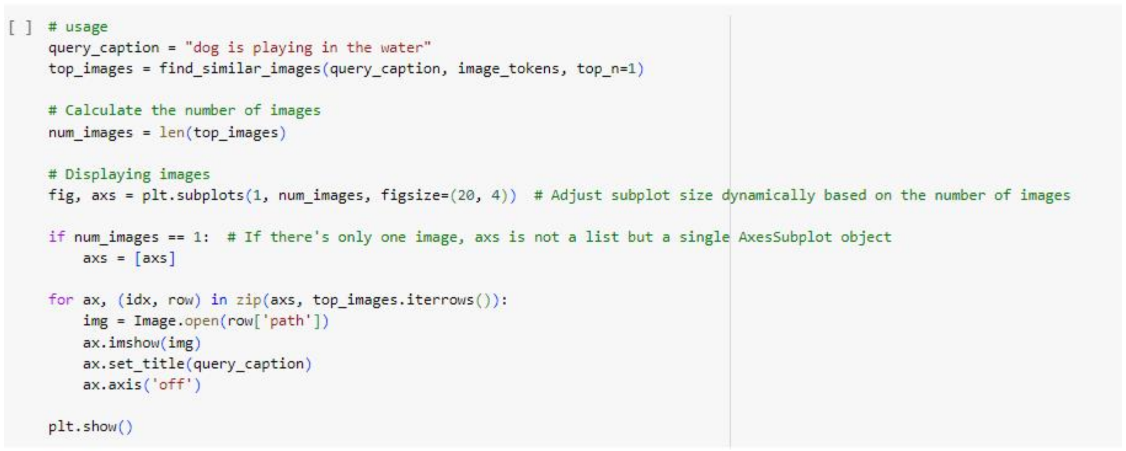
- **Text Embeddings**: this program process text data in baches (to manage memory usage on GPU) to convert text captions into embeddings
- **Image Embeddings**: similarly, image data is processed to generate embeddings, every image loaded, transformed into standard format, and then passed through the Model to get its embeddings

Image Retrieval Based on Text Queries:

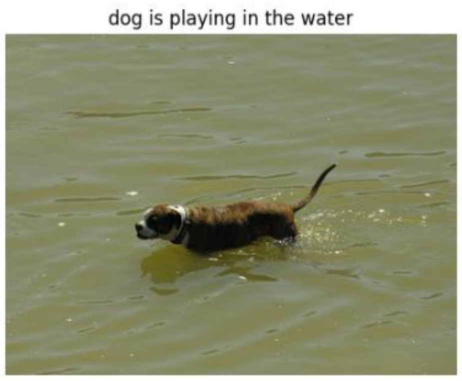
- Using description can retrieve image related to the provided description using a query function

Display Results : the matched images are retrieving to the top_images variable and display image using the `matplotlib.pyplot library`

Usage & Output



```
[ ] # usage
    query_caption = "dog is playing in the water"
    top_images = find_similar_images(query_caption, image_tokens, top_n=1)

    # Calculate the number of images
    num_images = len(top_images)

    # Displaying images
    fig, axs = plt.subplots(1, num_images, figsize=(20, 4))  # Adjust subplot size dynamically based on the number of images

    if num_images == 1:  # If there's only one image, axs is not a list but a single AxesSubplot object
        axs = [axs]

    for ax, (idx, row) in zip(axs, top_images.iterrows()):
        img = Image.open(row['path'])
        ax.imshow(img)
        ax.set_title(query_caption)
        ax.axis('off')

    plt.show()
```



Phase 2 Conclusion

This assignment generates images using text, doing by the CLIP pre-trained model, initially downloaded dataset and extract it to local directory and pre-process datasets for the structured format and embedding generation for text and images and finally visualize image according to provided description , In order to locate the most comparable photos based on text or image queries, a similarity search function is built. Finally, create a model that can take in the text and generate images