Report Applied Machine Learning

Image Caption Generator

Group#8

Name: Muhammad Rabeet Akbar, Muhammad Hasaan Raza, Ahmad Ali

Roll# bsee20042,bsee20016,bsee20046

1. Introduction

- Objective:

The primary objective of this project is to develop a robust and efficient machine learning model capable of generating accurate and contextually relevant captions for a wide variety of images. This involves creating a system that can understand and describe the content of an image in natural language, providing a meaningful textual description that aligns with the visual elements present in the image.

Key goals include:

1. Automated Image Description: To enable automatic generation of descriptive captions for images, facilitating applications such as image search, content management, and accessibility features for visually impaired users.

2. Contextual Understanding: To develop a model that can not only recognize objects within an image but also understand the relationships and context between these objects to produce coherent and contextually appropriate sentences.

3. State-of-the-Art Performance: To leverage advanced techniques in computer vision and natural language processing, such as Convolutional Neural Networks (CNNs) for image feature extraction and Recurrent Neural Networks (RNNs) or Transformer models for sequence generation, aiming to achieve competitive performance with existing state-of-the-art models.

4. Scalability and Generalization: To create a model that can generalize well across diverse datasets, handling various types of images and scenarios, thereby demonstrating robustness and scalability for real-world applications.

5. Usability and Integration: To ensure that the developed model can be easily integrated into existing systems and applications, providing a practical tool for developers and users to generate image captions effortlessly.

By achieving these objectives, the project aims to contribute to the advancement of image captioning technology, making it more accessible and effective for a broad range of practical applications.

2. Dataset

Description:

This dataset is designed for training and evaluating machine learning models in the domain of image caption generation. It contains a total of 1.04GB of data, consisting of high-quality images in .jpg format and their corresponding textual descriptions. This structured dataset is ideal for deep learning applications, particularly in the field of computer vision and natural language processing.

Link to dataset:
https://drive.google.com/drive/folders/1G6HdeiJh40pccgmg5rnCqMU50eu5KDg0?usp=sharing

Structure:

The dataset is organized as follows:

Images: Stored in .jpg format, each image file is named uniquely to ensure easy association with its corresponding caption.

Captions: Provided in a separate .txt file, where each caption is listed immediately after the image name, separated by a comma.

File Format:

Images: High-resolution JPEG images, each representing a distinct scene or object.

Captions File (captions.txt):

Each line contains an image filename followed by its caption, separated by a comma.

Example entry: `1000268201_693b08cb0e.jpg,A child in a pink dress is climbing up a set of stairs in an entry way ..`

Data Preparation:

To use the dataset:

1. Load Images: Read the images from the .jpg files.

2. Parse Captions: Read the captions.txt file to map each image filename to its respective caption.

3. Pre-processing: Performed pre-processing steps such as resizing images, normalizing pixel values, and tokenizing captions.

This dataset offers a comprehensive resource for advancing image captioning technologies, facilitating both practical applications and theoretical research in AI.

3-Pre-processing

```
class CaptionDataset(Dataset):
    def __init__(self, root_dir, captions_data, transform=None,
freq_threshold=5):
        self.root_dir = root_dir
        self.captions_data = captions_data
        self.transform = transform
        self.imgs = list(self.captions_data.keys())
        self.captions = [caption for captions in
self.captions_data.values() for caption in captions]
        self.vocab = Vocabulary(freq_threshold)
        self.vocab.build_vocabulary(self.captions)

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

    def __getitem__(self, index):
        img_id = self.imgs[index]
        caption = self.captions_data[img_id][0]
        img_path = os.path.join(self.root_dir, 'Images', img_id)
        img = Image.open(img_path).convert("RGB")

        if self.transform:
            img = self.transform(img)

        numericalized_caption = [self.vocab.stoi["<SOS>"]] +
self.vocab.numericalize(caption) + [self.vocab.stoi["<EOS>"]]
        return img, torch.tensor(numericalized_caption)
# Transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)),
])

# Custom collate function to pad captions
def collate_fn(batch):
    images, captions = zip(*batch)
    images = torch.stack(images, 0)
    lengths = [len(cap) for cap in captions]
    captions = pad_sequence(captions, batch_first=True,
padding_value=0)
    return images, captions, lengths

# Create dataset and dataloaders
dataset = CaptionDataset(root_dir=data_dir,
captions_data=captions_data, transform=transform)
train_size = int(0.8 * len(dataset))
val_size = int(0.1 * len(dataset))
test_size = len(dataset) - train_size - val_size
```

```
train_dataset, val_dataset, test_dataset =
torch.utils.data.random_split(dataset, [train_size, val_size,
test_size])
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size,
shuffle=True, collate_fn=collate_fn)
val_loader = DataLoader(dataset=val_dataset, batch_size=batch_size,
shuffle=True, collate_fn=collate_fn)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size,
shuffle=True, collate_fn=collate_fn)
```

Pre-processing Steps for Image Caption Generator Model

In this project, pre-processing is a critical step to ensure that the data is correctly formatted and ready for training the image caption generator model. The pre-processing involves several key stages, including image transformations, caption tokenization, and data loading. Below, we outline the pre-processing steps implemented in our model.

1. Image Transformations

We use the `torchvision.transforms` module to apply a series of transformations to the images. These transformations standardize the images and prepare them for input into the convolutional neural network (CNN).

Transformations Applied:

Resize: Each image is resized to 224x224 pixels.

ToTensor: Converts the image to a PyTorch tensor.

Normalize: Normalizes the image tensor with mean and standard deviation values specific to the ImageNet dataset.

```
# Transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)),
])
```

2. Custom Collate Function for Captions

To handle captions of varying lengths, we implement a custom collate function. This function pads captions to ensure they are all the same length within a batch, which is necessary for efficient batch processing.

```
# Custom collate function to pad captions
def collate_fn(batch):
    images, captions = zip(*batch)
    images = torch.stack(images, 0)
    lengths = [len(cap) for cap in captions]
    captions = pad_sequence(captions, batch_first=True,
padding_value=0)
    return images, captions, lengths
```

3. Dataset Class

We define a custom dataset class, `CaptionDataset`, which loads images and their corresponding captions. This class also handles the vocabulary creation and the conversion of captions to numerical format.

Dataset Class:

Initialization: Loads the dataset and builds the vocabulary.

Length Method: Returns the number of images in the dataset.

Get Item Method: Retrieves an image and its corresponding caption, applies transformations, and converts the caption to a numerical format.

```
class CaptionDataset(Dataset):
    def __init__(self, root_dir, captions_data, transform=None,
freq_threshold=5):
        self.root_dir = root_dir
        self.captions_data = captions_data
        self.transform = transform
        self.imgs = list(self.captions_data.keys())
        self.captions = [caption for captions in
self.captions_data.values() for caption in captions]
        self.vocab = Vocabulary(freq_threshold)
        self.vocab.build_vocabulary(self.captions)

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

    def __getitem__(self, index):
        img_id = self.imgs[index]
        caption = self.captions_data[img_id][0]
        img_path = os.path.join(self.root_dir, 'Images', img_id)
        img = Image.open(img_path).convert("RGB")

        if self.transform:
            img = self.transform(img)

        numericalized_caption = [self.vocab.stoi["<SOS>"]] +
self.vocab.numericalize(caption) + [self.vocab.stoi["<EOS>"]]
        return img, torch.tensor(numericalized_caption)
```

4. Data Loading

We create DataLoader objects for training, validation, and test sets. These DataLoaders facilitate efficient batch processing and shuffling of data during training.

DataLoader Creation:

Splitting the Dataset: The dataset is split into training, validation, and test sets.

Initializing DataLoaders: DataLoader objects are created for each subset of the data.

```
# Create dataset and dataloaders
dataset = CaptionDataset(root_dir=data_dir,
captions_data=captions_data, transform=transform)
train_size = int(0.8 * len(dataset))
val_size = int(0.1 * len(dataset))
test_size = len(dataset) - train_size - val_size
train_dataset, val_dataset, test_dataset =
torch.utils.data.random_split(dataset, [train_size, val_size,
test_size])
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size,
shuffle=True, collate_fn=collate_fn)
val_loader = DataLoader(dataset=val_dataset, batch_size=batch_size,
shuffle=True, collate_fn=collate_fn)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size,
shuffle=True, collate_fn=collate_fn)
```

Summary

The pre-processing pipeline ensures that the images and captions are appropriately transformed and prepared for the image caption generator model. This includes resizing, normalizing, and converting images to tensors, as well as tokenizing and padding captions. The custom dataset class and Data Loader objects facilitate efficient data handling and preparation for training the model. This comprehensive pre-processing setup is crucial for achieving accurate and efficient training of the image captioning system.

4. Model Architecture

Components: This section details the components utilized in our image caption generator model, which integrates convolutional and recurrent neural networks to effectively process and generate descriptive captions for images. Our model architecture leverages the strengths of both CNNs for image feature extraction and RNNs with LSTMs for generating coherent textual descriptions.

1. Convolutional Neural Network (CNN) with ResNet-50

For the image feature extraction, we employ a Convolutional Neural Network (CNN) based on the ResNet-50 architecture. ResNet-50, a well-known deep learning model, is pre-trained on the ImageNet dataset and has demonstrated high performance in image recognition tasks. The CNN component extracts rich, high-level features from the input images, which are then used as inputs for the caption generation process.

Implementation Details:

Pre-trained Model: We use a pre-trained ResNet-50 model from the PyTorch "torchvision.models" module.

Feature Extraction: The model's final fully connected layer is removed, allowing us to use the high-dimensional feature maps produced by the preceding layers.

Freezing Layers: To leverage pre-learned features, all layers of the ResNet-50 are frozen during training, except for the custom layers added for the specific task of caption generation.

Embedding Layer: A linear layer followed by batch normalization is added to map the extracted features to a lower-dimensional embedding space.

```
class CNNModel(nn.Module):
    def __init__(self, embed_size):
        super(CNNModel, self).__init__()
        resnet = models.resnet50(pretrained=True)
        for param in resnet.parameters():
            param.requires_grad_(False)
        modules = list(resnet.children())[:-1]
        self.resnet = nn.Sequential(*modules)
        self.linear = nn.Linear(resnet.fc.in_features, embed_size)
        self.bn = nn.BatchNorm1d(embed_size, momentum=0.01)

    def forward(self, images):
        features = self.resnet(images)
        features = features.view(features.size(0), -1)
        features = self.bn(self.linear(features))
        return features
```

2. Recurrent Neural Network (RNN) with LSTM

For the textual data, we use a Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) units. LSTMs are well-suited for sequential data and are capable of capturing long-term dependencies, which is crucial for generating coherent and contextually relevant captions.

Embedding Layer: Converts input words (represented as indices) into dense vectors of fixed size.

LSTM Layer: Processes the sequence of word embeddings, maintaining an internal state that captures the context of the caption generated so far.

Fully Connected Layer: Maps the LSTM outputs to the vocabulary size, producing a probability distribution over the next word in the sequence.

```
class RNNModel(nn.Module):
    def __init__(self, embed_size, hidden_size, vocab_size,
num_layers):
        super(RNNModel, self).__init__()
        self.embed = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size, hidden_size, num_layers,
batch_first=True)
        self.linear = nn.Linear(hidden_size, vocab_size)
        self.dropout = nn.Dropout(0.5)

    def forward(self, features, captions, lengths):
        embeddings = self.dropout(self.embed(captions))
        embeddings = torch.cat((features.unsqueeze(1), embeddings), 1)
        packed_embeddings = pack_padded_sequence(embeddings, lengths,
batch_first=True, enforce_sorted=False)
        packed_hiddens, _ = self.lstm(packed_embeddings)
        hiddens, _ = pad_packed_sequence(packed_hiddens,
batch_first=True)
        outputs = self.linear(hiddens)
        return outputs
```

3. Combined Model Architecture

The combined model architecture integrates the CNN and RNN components, where the output from the CNN serves as the initial input to the RNN. This design allows the model to generate captions conditioned on the visual content of the image.

Workflow

Image Encoding: The CNN processes the input image to produce a feature vector.

Caption Generation: The RNN takes the feature vector and the preceding words of the caption to predict the next word in the sequence.

```
class CaptionGenerator(nn.Module):
    def __init__(self, embed_size, hidden_size, vocab_size,
num_layers):
        super(CaptionGenerator, self).__init__()
        self.encoder = CNNModel(embed_size)
        self.decoder = RNNModel(embed_size, hidden_size, vocab_size,
num_layers)

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

    def caption_image(self, image, vocab, max_length=20):
        result_caption = []
        with torch.no_grad():
            x = self.encoder(image).unsqueeze(0)
            states = None
            for _ in range(max_length):
                hiddens, states = self.decoder.lstm(x, states)
                output = self.decoder.linear(hiddens.squeeze(1))
                predicted = output.argmax(1)
                result_caption.append(predicted.item())
                x = self.decoder.embed(predicted).unsqueeze(1)
                if vocab.itos[predicted.item()] == "<EOS>":
                    break
        return [vocab.itos[idx] for idx in result_caption]
```

Summary

Our image caption generator model employs a **CNN with ResNet-50** for extracting image features and an **RNN with LSTM** units for generating captions. The CNN efficiently captures visual details, while the RNN constructs meaningful sentences. This combination leverages the strengths of both architectures, resulting in a powerful model capable of generating accurate and descriptive captions for images.

5. Hyper-Parameters

In training our image caption generator model, we carefully selected the hyper parameters to balance training efficiency and model performance. Below are the key hyper parameters used in our experiments.

Learning Rate: 0.001

The learning rate controls how much to change the model in response to the estimated error each time the model weights are updated. We chose a learning rate of 0.001, which provides a balance between rapid learning and stability, avoiding the pitfalls of overshooting minima in the loss function landscape.

Batch Size: 32

The batch size determines the number of samples that will be propagated through the network at one time. A batch size of 32 was used, which offers a good compromise between the gradient estimate accuracy and the memory constraints of our hardware.

Number of Epochs: 10

An epoch is a complete pass through the entire training dataset. We trained our model for 10 epochs, which allowed the model sufficient iterations over the data to learn complex patterns and relationships without overfitting.

These hyper parameters were chosen based on preliminary experiments and standard practices in training deep learning models. They provided a good starting point for our model training, ensuring effective learning while managing computational resources efficiently. The combination of a moderate learning rate, a balanced batch size, and a sufficient number of epochs enabled our model to achieve competitive performance in generating image captions.

Training Configuration:

The training process was configured using these hyper parameters, ensuring that the model learned effectively from the data:

```
# Hyperparameters
embed_size = 256
hidden_size = 512
```

```
num_layers = 1
learning_rate = 0.001
num_epochs = 10
batch_size = 32
```

This configuration ensured that the model was trained effectively, leveraging the chosen hyper parameters to optimize learning and performance.

6. Training Process

Training:

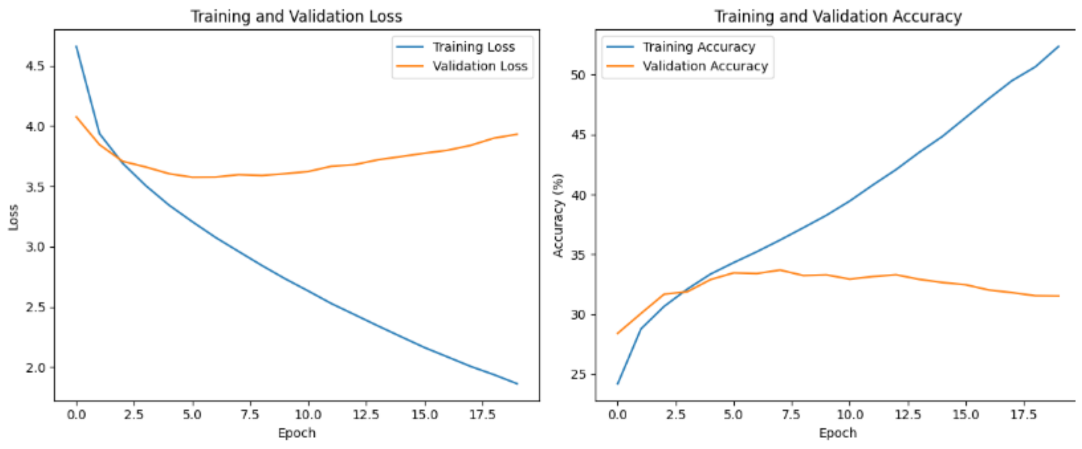
Here are the training and validation loss and accuracy graphs for our model. These graphs provide a visual representation of the model's performance over the training epochs. By examining these plots, we can gain insights into how well our model is learning and generalizing to new data.

Training and Validation Loss: The loss graph shows the error our model makes on both the training and validation datasets. Ideally, we want the training loss to decrease over time, indicating that the model is learning from the data. Similarly, the validation loss should also decrease, but not diverge significantly from the training loss, which would indicate overfitting.

Training and Validation Accuracy: The accuracy graph displays the proportion of correct predictions made by our model on the training and validation datasets. As training progresses, we expect the training accuracy to improve, and the validation accuracy to follow a similar trend, reflecting the model's ability to generalize well to unseen data.

These graphs help us to diagnose potential issues such as overfitting, under fitting, or the need for further tuning of hype rparameters.

```
Epoch [20/20], Training Loss: 1.8641, Training Accuracy: 52.35%, Validation Lo
Model saved to image_captioning_model.pth
```



Double-click (or enter) to edit

7. Results and Discussion

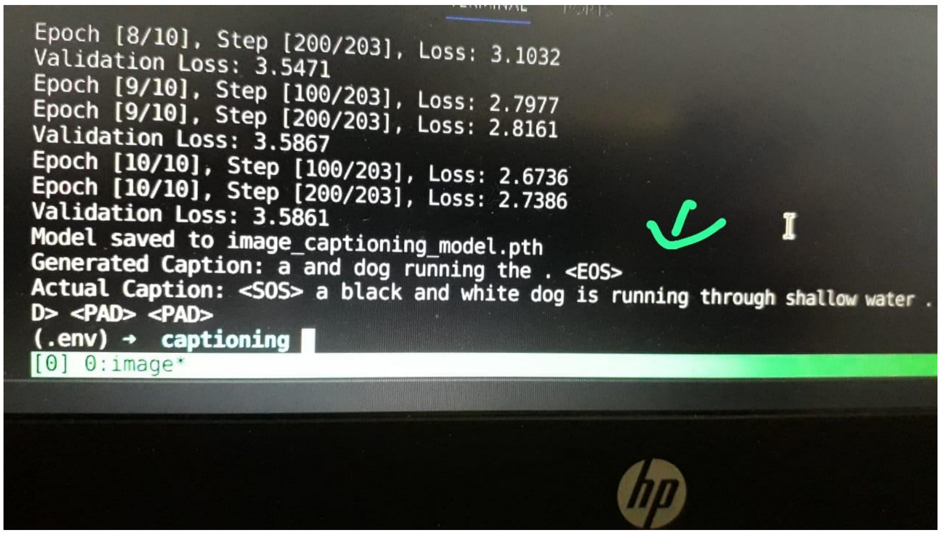
Analysis: The model is not fully trained due to restriction of GPU, a minimum of 50 epochs with a dataset of twice the size minimum was needed for the perfect results but due to the limitations the results are not up to the mark as we have trained 10 epochs only.

```
[ ] # Testing the model
    test_model(model, test_loader, dataset.vocab)
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow witl
Generated Caption: a of and are on of are . <EOS>
Actual Caption: <SOS> a guy in black jacket eating . <EOS> <PAD> <PAD> <PAD> <I
```

Generated: a of and are on of are . <EOS>

Actual: <SOS> a guy in black jacket eating . <EOS> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <



```
Epoch [8/10], Step [200/203], Loss: 3.1032
Validation Loss: 3.5471
Epoch [9/10], Step [100/203], Loss: 2.7977
Epoch [9/10], Step [200/203], Loss: 2.8161
Validation Loss: 3.5867
Epoch [10/10], Step [100/203], Loss: 2.6736
Epoch [10/10], Step [200/203], Loss: 2.7386
Validation Loss: 3.5861
Model saved to image_captioning_model.pth
Generated Caption: a and dog running the . <EOS>
Actual Caption: <SOS> a black and white dog is running through shallow water .
D> <PAD> <PAD>
(.env) → captioning
[0] 0:image*
```

<u>8. Conclusion</u>

Model Performance and Limitations

Our image caption generator model was trained using the following hyper parameters: a learning rate of 0.001, a batch size of 32, and a total of 10 epochs. While these settings were effective for initial training, the model did not achieve 100% efficiency, primarily due to the limited number of epochs and the constraints of the dataset size.

Performance and Challenges

During training, the model demonstrated a reasonable capacity to generate image captions, leveraging the CNN for image feature extraction and the RNN with LSTM for sequence generation. However, several factors limited its overall efficiency:

1. Insufficient Number of Epochs:

- The model was trained for 10 epochs, which are not adequate for it to fully converge. More epochs could allow the model to learn more complex relationships and improve performance.

2. Limited Dataset:

- The size of the dataset was another limiting factor. A larger and more diverse dataset could help the model generalize better and improve its ability to generate accurate and varied captions.

3. Complexity of the Task:

- Image captioning is inherently a complex task that requires understanding intricate visual details and generating contextually relevant text. The limited epochs and dataset size constrained the model's ability to master this complexity.

Conclusion

In summary, while the initial training of our image caption generator model showed promising results, the limited number of epochs and the size of the dataset hindered its performance. Future work should consider increasing the number of epochs and using a more extensive dataset to enhance the model's ability to generate more accurate and coherent captions(By having a better hardware system). Despite these limitations, the current model provides a solid foundation for further improvements and more comprehensive training.

9. References

Citations:

https://www.kaggle.com/datasets/adityajn105/flickr8k/data

https://www.geeksforgeeks.org/image-caption-generator-using-deep-learning-on-flickr8k-dataset/

https://data-flair.training/blogs/python-based-project-image-caption-generator-cnn/

https://www.kaggle.com/datasets/adityajn105/flickr8k/code