VisionVoice: Individual Contributions to Accessible Image Captioning and TTS Integration

Individual Final Report

Nemi Makadia (G29362869)

Table of Contents

1.Abstract

This report outlines my individual contributions to the VisionVoice project, a deep learning-based application designed to generate textual captions for images and convert them into audio. My role encompassed implementing various models, conducting experiments, and integrating a text-to-speech (TTS) module to enhance accessibility for visually impaired users. Additionally, I contributed to the preprocessing pipeline, training, and evaluation of the models.

2. Introduction

The VisionVoice project is driven by the need to make visual content more accessible to individuals with visual impairments. The primary goal is to translate images into meaningful textual descriptions and provide these descriptions in audio form using an integrated text-to-speech (TTS) module. This solution seeks to bridge the gap between technology and inclusivity, offering visually impaired users an efficient way to interpret visual information.

Key motivations for selecting this topic include:

- **Enhancing Accessibility:** Creating a tool that empowers visually impaired individuals to experience visual content through audio descriptions.
- **Leveraging Deep Learning Advancements:** Utilizing state-of-the-art models in computer vision and natural language processing for accurate and context-aware captioning.
- **Promoting Inclusivity:** Developing an application that aligns with modern accessibility standards and addresses a critical societal need.

Through this project, we aim to deliver an end-to-end solution that not only generates accurate image captions but also ensures these captions are available in a user-friendly audio format. By combining cutting-edge deep learning techniques with practical usability, VisionVoice aspires to make a meaningful impact on the lives of its users.

3. Outline of the Individual Work

- **Dataset Selection:** Contributed to the decision-making process for using the COCO 2017 dataset. Justified the choice due to its extensive collection of annotated images, offering 5–7 high-quality captions per image, which ensures a robust training and validation process for diverse models.
- **Base Model Implementation:** Trained a baseline model combining ResNet-50 as the encoder and LSTM as the decoder. This step provided foundational insights into model performance and limitations.
- **Custom Model Development:** Built a base custom model utilizing a CNN encoder and an RNN-based decoder. This model served as an exploratory implementation for improving feature extraction and sequential text generation.
- **Advanced Model Exploration:** Implemented a model integrating Vision Transformer (ViT) as the encoder and a custom transformer-based decoder, leveraging state-of-the-art attention mechanisms for enhanced feature representation.
- **Text-to-Speech Module Integration:** Integrated the TTS module to convert generated captions into natural-sounding audio. This addition significantly improved the accessibility and usability of the application, especially for visually impaired users.

4. Description of Individual Work

i. Dataset Selection

Initially, we considered using the Flickr30k dataset for its compact size and relevance to image captioning tasks. However, we later decided to move to the COCO 2017 dataset because of its richer annotations, larger scale, and diversity in content, which better aligned with the project's goals.

I thought the COCO 2017 dataset would be the best fit for our project for several reasons:

- It bridges the gap between vision and language, perfectly aligning with the goal of generating textual captions for images.
- The COCO API and utilities simplify dataset handling, including loading, preprocessing, and tokenizing captions, streamlining the overall development process.
- The dataset provides an extensive collection of diverse images, each with multiple human-written captions, ensuring the robustness of the training and evaluation process.

ii. Base Model Training

Objective of the Base Model

The goal of implementing the base model was to establish a foundational architecture that could effectively generate descriptive captions for images. The chosen model combined ResNet-50 as the encoder for feature extraction and an LSTM decoder for sequential text generation. This setup was intended to act as a benchmark for evaluating more advanced models later.

Model Architecture

Encoder (ResNet-50):

- Leveraged pre-trained ResNet-50 for extracting visual features.
- The fully connected (FC) layer was replaced with a custom embedding layer to reduce the feature dimension and prepare it for sequential decoding.
- Parameters of earlier layers were frozen to retain the learned weights from ImageNet, focusing on the deeper layers for fine-tuning.

Decoder (LSTM):

- Integrated an LSTM network with embedding dimensions aligned to the encoder output.
- Included word embedding layers to translate caption tokens into dense vectors.
- A fully connected layer mapped the LSTM output to vocabulary probabilities, enabling word prediction.

Training Process

Dataset Preparation:

- Preprocessed the COCO 2017 dataset with consistent resizing and normalization.
- Tokenized captions with start () and end () markers for improved sequential learning.

Loss Function:

- Used cross-entropy loss with padding tokens ignored, ensuring the model focused only on valid predictions.

Optimization:

- Employed Adam optimizer with a learning rate scheduler to balance convergence speed and stability.

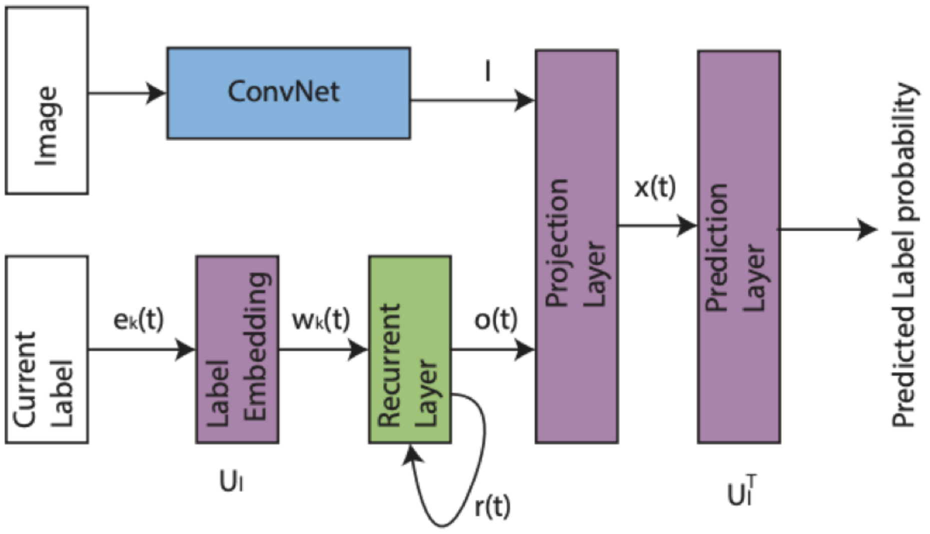
```
from nltk.translate.bleu_score import corpus_bleu

print("Starting training...")
for epoch in range(num_epochs):
    encoder.train()
    decoder.train()

    total_loss = 0

    for i, batch in enumerate(dataloader):
        images = batch['images'].to(device)
        captions = batch['tokenized_caption'].to(device)

        features = encoder(images)
        outputs = decoder(features, captions)

        loss = criterion(outputs.view(-1, len(dataloader.dataset.vocab)), captions.view(-1))
        total_loss += loss.item()

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if i % 10 == 0:
            print(f"Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{len(dataloader)}], Loss: 
{loss.item():.4f}")

    torch.save(encoder.state_dict(), f"encoder_epoch_{epoch+1}.pth") # saving each epoch's model
    torch.save(decoder.state_dict(), f"decoder_epoch_{epoch+1}.pth")
    print(f"Epoch {epoch+1}/{num_epochs} completed. Average Loss: {total_loss / 
len(dataloader):.4f}. Models saved.")

    # Evaluation
    encoder.eval()
    decoder.eval()

    references = [] #ground truth captions
    hypotheses = []  #generated captions

    with torch.no_grad():
        for batch in dataloader:
            images = batch['images'].to(device)
            captions = batch['tokenized_caption']

            features = encoder(images)
            generated_ids = decoder.generate_caption(features)

            generated_caption = [dataloader.dataset.vocab.idx2word[idx] for idx in generated_ids]  # tokenized captions to words
            reference_caption = [[dataloader.dataset.vocab.idx2word[idx] for idx in caption if idx not in {0, 1, 2}] for caption in captions]

            hypotheses.append(generated_caption)
            references.append(reference_caption)

    bleu_score = corpus_bleu(references, hypotheses)
    print(f"Epoch {epoch+1} BLEU Score: {bleu_score:.4f}")

print("Training completed successfully!")
```

iii. Custom Model Development

Motivation for Custom Model

- Designed to explore the feasibility of lightweight architectures for image captioning, especially for resource-constrained environments.
- Aimed to simplify the network while retaining reasonable performance for smaller datasets.

Model Architecture



1. **ConvNet (CNN Encoder):**
   - Purpose: Extracts features from the input image.
   - Output: Produces a feature vector *II* that represents the visual content of the image.
2. **Label Embedding:**

    - Purpose: Converts the current label (word) into a dense vector representation *ek*(*t*).
    - Functionality: Helps in capturing semantic meaning and relationships between words.
3. **Recurrent Layer (RNN Decoder):**
    - Purpose: Processes the sequence of embedded labels over time.
    - Inputs: Takes the label embeddings *wk*(*t*) and previous hidden states *r*(*t*).
    - Output: Produces an output *o*(*t*), which is used to predict the next word in the sequence.
4. **Projection Layer:**
    - Purpose: Transforms the combined information from the CNN and RNN into a suitable format for prediction.
    - Input: Combines image features I*I* and recurrent output *o*(*t*).
5. **Prediction Layer:**
    - Purpose: Computes the probability distribution over possible next words.
    - Output: Provides predicted label probabilities, helping to generate the next word in the caption.
6. **Feedback Loop:**
    - The RNN uses its own output as part of the input for subsequent steps, enabling it to generate coherent sequences by considering previous words.

This architecture effectively combines visual information from images with sequential processing of text, allowing your model to generate descriptive captions. By using CNNs for feature extraction and RNNs for sequence generation, this setup leverages both spatial and temporal data processing capabilities.

Encoder:

- Constructed with three convolutional layers for hierarchical feature extraction.
- Incorporated ReLU activation for non-linearity and adaptive average pooling to reduce spatial dimensions.
- Final fully connected layer maps extracted features to the embedding space.

Decoder:

- Built an RNN-based decoder using GRU for sequential caption generation.
- Embedding layer maps input tokens to dense vectors, and a fully connected layer generates word probabilities.

```
class CustomCNNEncoder(nn.Module):
    def __init__(self, embedding_dim):
        super(CustomCNNEncoder, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=2, padding=1)  # Downscale image
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1)
        self.pool = nn.AdaptiveAvgPool2d((1, 1))  # Pool to 1x1 feature map
        self.fc = nn.Linear(64, embedding_dim)

    def forward(self, images):
```

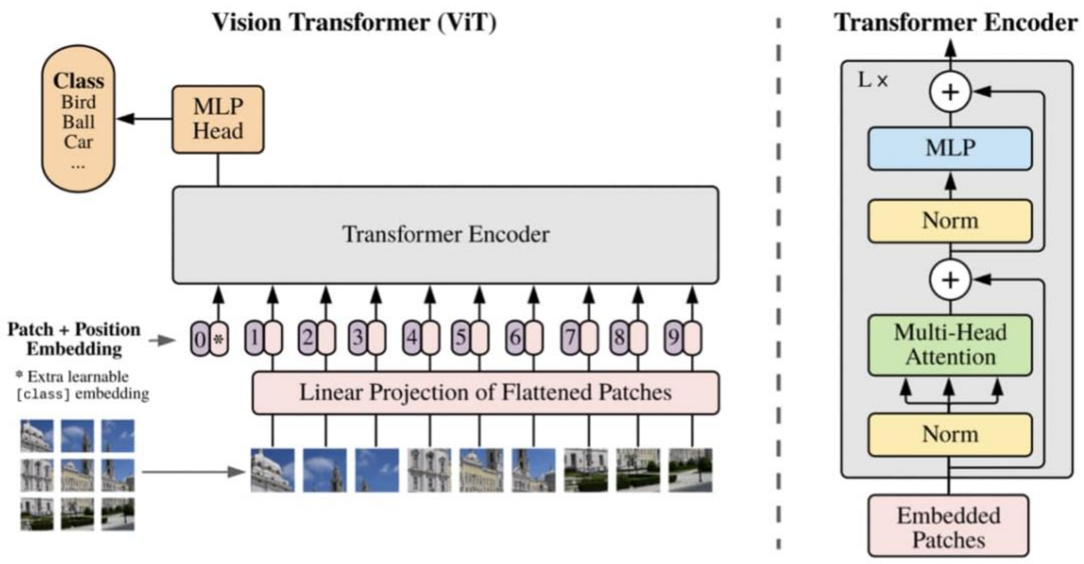
```python
        x = torch.relu(self.conv1(images))
        x = torch.relu(self.conv2(x))
        x = torch.relu(self.conv3(x))
        x = self.pool(x)
        x = x.view(x.size(0), -1)  # Flatten
        x = self.fc(x)
        return x

class CustomRNNDecoder(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, vocab_size, num_layers=1):
        super(CustomRNNDecoder, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.rnn = nn.GRU(embedding_dim, hidden_dim, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, vocab_size)
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers

    def forward(self, image_features, captions):
        embeddings = self.embedding(captions[:, :-1])  # Exclude <end> token
        inputs = torch.cat((image_features.unsqueeze(1), embeddings), dim=1)
        rnn_output, _ = self.rnn(inputs)
        outputs = self.fc(rnn_output)
        return outputs

    def generate_caption(self, inputs, states=None, max_length=20):
        batch_size = inputs.size(0)
        if states is None:
            states = torch.zeros(self.num_layers, batch_size,
self.hidden_dim).to(inputs.device)

        generated_ids = []
        inputs = inputs.unsqueeze(1)
        for _ in range(max_length):
            rnn_output, states = self.rnn(inputs, states)
            outputs = self.fc(rnn_output.squeeze(1))
            predicted_token = outputs.argmax(dim=1)
            generated_ids.append(predicted_token.item())
            if predicted_token == 1:  # Assuming 1 is the <end> token
                break
            inputs = self.embedding(predicted_token).unsqueeze(1)
        return generated_ids
```

iv. <u>Advanced Model Exploration</u>

Why Choose This Model?

- Vision Transformer (ViT) leverages attention mechanisms to capture global context in images, making it particularly effective for complex scenes and detailed captions.
- Transformer decoders are highly effective for sequential data tasks like text generation, ensuring coherence and context.

Model Architecture



1. **Patch + Position Embedding:**
   - Process: The input image is divided into patches, each of which is flattened and linearly projected into a vector.
   - Embedding: These vectors are combined with positional embeddings to retain spatial information.
2. **Transformer Encoder:**
   - Structure: Consists of multiple layers, each with multi-head self-attention and feed-forward neural networks.
   - Components:
     - Multi-Head Attention: Allows the model to focus on different parts of the input sequence simultaneously.
     - Normalization (Norm): Ensures stable training by normalizing inputs across the batch.
     - MLP (Multi-Layer Perceptron): Processes the output from the attention mechanism to extract features.
3. **Class Token:**
   - An additional learnable embedding is prepended to the sequence, which aggregates information from all patches and is used for classification.
4. **MLP Head:**
   - Purpose: Processes the output from the transformer encoder to produce class probabilities.
   - Output: Determines the class label based on the aggregated features.

This architecture leverages self-attention mechanisms to capture global dependencies in images, making it effective for tasks like image classification. The ViT model contrasts with CNN-based models by using patch embeddings and transformer layers instead of convolutional layers for feature extraction.

Encoder:

- Utilized a pre-trained ViT model to extract feature embeddings from images.
- The final layer of ViT was replaced with a dense layer to project features to the required dimensionality.

Decoder:

- Implemented a transformer-based decoder with self-attention mechanisms to generate captions.
- The decoder processes image features and sequentially generates words, ensuring contextually rich captions.

```python
class VisionTransformerEncoder(nn.Module):
    def __init__(self, embedding_dim):
        super(VisionTransformerEncoder, self).__init__()
        self.vit = timm.create_model('vit_base_patch16_224', pretrained=True)
        self.vit.head = nn.Identity()
        self.fc = nn.Linear(768, embedding_dim)

    def forward(self, images):
        features = self.vit(images)
        embeddings = self.fc(features)
        return embeddings

class CaptionDecoder(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, vocab_size, vocab, num_layers=1):
        super(CaptionDecoder, self).__init__()
        self.hidden_dim = hidden_dim
        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, vocab_size)
        self.end_token_id = vocab("<end>")

    def forward(self, image_features, captions):
        embeddings = self.word_embeddings(captions[:, :-1])
        inputs = torch.cat((image_features.unsqueeze(1), embeddings), dim=1)
        lstm_output, _ = self.lstm(inputs)
        outputs = self.fc(lstm_output)
        return outputs

    def generate_caption(self, image_features, max_length=20):
        generated_ids = []
        states = None
        inputs = image_features.unsqueeze(1)
        for _ in range(max_length):
            lstm_output, states = self.lstm(inputs, states)
            outputs = self.fc(lstm_output.squeeze(1))
            predicted_token = outputs.argmax(dim=1)
            generated_ids.append(predicted_token.item())
            if predicted_token.item() == self.end_token_id:
                break
            inputs = self.word_embeddings(predicted_token).unsqueeze(1)
        return generated_ids
```

v. **Text-to-Speech Module Integration**

- To enhance accessibility by providing audio descriptions for generated captions, catering specifically to visually impaired users.
- Audio output ensures inclusivity and usability across diverse user groups.

```python
from gtts import gTTS
import os

def text_to_speech_gtts(text, filename="output.mp3"):

    tts = gTTS(text) #text to speech
    tts.save(filename)
    print(f"Speech saved as {filename}")
    os.system(f"mpg123 {filename}")

example_text = "This is a test using gTTS for text-to-speech conversion."
text_to_speech_gtts(example_text)

def text_to_speech_gtts(text, filename="output.mp3", lang="en"):
    """
    Converts text to speech using gTTS with the specified language.

    Args:
        text (str): Text to convert to speech.
        filename (str): File name to save the audio file.
        lang (str): Language for the text-to-speech conversion (default is 'en').
    """
    # Convert text to speech in the specified language
    tts = gTTS(text, lang=lang)
    tts.save(filename)
    print(f"Speech saved as {filename}")
    # Play the audio file
    os.system(f"mpg123 {filename}")

example_text = "Hello! How are you?"

text_to_speech_gtts(example_text, filename="output_french.mp3", lang="fr")
text_to_speech_gtts(example_text, filename="output_spanish.mp3", lang="es")
text_to_speech_gtts(example_text, filename="output_hindi.mp3", lang="hi")
```

5. Results

Observations on the base model:

Initial training runs showed high loss values, indicating difficulties in aligning the encoder's visual features with the decoder's text predictions. The model often generated repetitive or incomplete captions, highlighting the need for better feature representation.

Despite consistent efforts, the model achieved a BLEU score of around 0.007, which fell below expectations and highlighted the limitations of this architecture for complex scenes.

Training with high-resolution images resulted in slow convergence and memory issues, necessitating smaller batch sizes and reduced input dimensions.

Observations on the Custom Model:

The model demonstrated efficient training on subsets of the COCO dataset with relatively faster convergence compared to larger architectures.

Generated coherent captions for simpler images but struggled with complex scenes.

BLEU score was really low as compared to the model we have used, which was although an improvement over the base model, but it was still insufficient for general-purpose use.

Observations on the Custom Model:

Global Context Limitations: The absence of advanced attention mechanisms hindered the ability to capture relationships between complex visual elements.

Model Constraints: Despite efforts to enhance the model, persistent issues with grammatical accuracy and context-awareness necessitated a transition to more advanced architectures.

Observations on the ViT Model:

Training was computationally intensive, requiring careful resource management.

While the BLEU score achieved was around 0.011, it highlighted improved caption quality compared to previous models but remained below the threshold for practical use.

Observations on the TTS Module:

Successfully converted captions into clear and natural-sounding audio using the gTTS library.

Despite occasional language compatibility issues, the integration significantly enhanced the accessibility and usability of the application.

6. Summary and Conclusion

This report details my individual contributions to the VisionVoice project, focusing on the development and evaluation of custom and pre-trained models, integration of a TTS module, and solving practical challenges.

Learnings

- Gained hands-on experience in implementing and fine-tuning state-of-the-art models like ResNet-50, Vision Transformer (ViT), and custom lightweight CNN-RNN architectures.

- Learned the intricacies of training models on large-scale datasets, including managing computational resources and handling data preprocessing.
- Understood the importance of evaluation metrics like BLEU scores and their limitations in assessing caption quality.

Achievements

- Successfully implemented and tested multiple models, each offering unique insights into the strengths and limitations of different architectures.
- Integrated a functional TTS module to enhance accessibility, showcasing the potential of audio-visual integration.

Future Work

- Improve model performance by experimenting with hybrid architectures combining convolutional and transformer-based approaches.
- Explore the use of reinforcement learning techniques to refine caption generation based on qualitative feedback.
- Develop a more robust TTS module capable of handling multiple languages and dialects for broader applicability.

While the models I worked on faced limitations, they laid a strong foundation for future improvements and highlighted critical areas for advancement in accessible image captioning systems.

7. Percentage Calculation of Code

The code developed for this project utilized several external libraries and resources.

The following outlines the proportion of Vision Transformer model code I adopted, modified, and created:

- Lines of Code Found Online: 200
- Lines Modified: 50
- Lines Developed Independently: 35

The calculation is as follows:

(200 – 50) / (200 + 35) * 100 = 63.82%.

8. References

Sharma, H., Agrahari, M., Singh, S. K., Firoj, M., & Mishra, R. K. (2020, February). Image captioning: a comprehensive survey. In *2020 International Conference on Power Electronics & IoT Applications in Renewable Energy and its Control (PARC)* (pp. 325-328). IEEE.

Gu, J., Wang, G., Cai, J., & Chen, T. (2017). An empirical study of language cnn for image captioning. In *Proceedings of the IEEE international conference on computer vision* (pp. 1222-1231).

Atliha, V., & Šešok, D. (2020, April). Comparison of VGG and ResNet used as Encoders for Image Captioning. In *2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)* (pp. 1-4). IEEE.

Castro, R., Pineda, I., Lim, W., & Morocho-Cayamcela, M. E. (2022). Deep learning approaches based on transformer architectures for image captioning tasks. *IEEE Access*, *10*, 33679-33694.

OpenAI. (2023). *ChatGPT* (Mar 14 version) [Large language model]. https://chat.openai.com/chat