CSC 5991: Introduction to LLMs

Home Work-1: Image Captioning with Attention

Team-04

Submitted By:

1. Nafis Fuad (hq8312)
2. MD Nayeemur Rashid Nayeem (hw9533)
3. Mohammad Azadegan (hh8479)

For this homework we have selected the first option which is 'Image Captioning with Attention'. Image captioning is the process of generating textual descriptions for images using deep learning models. This task involves extracting visual features from an image using a convolutional neural network (CNN) and generating a meaningful caption using a sequence model, such as a Gated recurrent unit (GRU) with attention. Details description of each section is presented below:

1. Dataset Details & Data Split:

For this experiment, we use the **Flickr8k dataset**, which contains 8,000 images, each annotated with five captions. The dataset is split into 6,000 images for training, 1,000 for validation, and 1,000 for testing. The goal is to generate captions for unseen images by training a model that integrates both CNN-based feature extraction and an GRU-based decoder with attention.

2. Network Architecture and Attention Mechanism:

2.1 Encoder (Image Feature Extraction)

We have used **MobileNetV2** as our feature extractor, removing its final classification layer to retain the extracted feature maps. The extracted features are then projected into a lower-dimensional space using a linear layer.

Here is the code for encoder:

```
# Load pre-trained MobileNetV2 as the Encoder
class Encoder(nn.Module):
 def __init__(self):
     super(Encoder, self).__init__()
     mobilenet = models.mobilenet_v2(pretrained=True)  # Load MobileNetV2
            self.feature_extractor = mobilenet.features.to(device)  # Extract
   feature layers

 def forward(self, images):
     """
     Input: images (batch_size, 3, 224, 224)
     Output: feature map (batch_size, 49, 1280)
     """
            features = self.feature_extractor(images)  # Shape: [batch_size,
   1280, 7, 7]
     batch_size, feature_dim, h, w = features.shape
            features = features.view(batch_size, feature_dim, h *
   w).permute(0, 2, 1)  # Reshape to [batch_size, 49, 1280]
     return features
```

2.2 Attention-Based Captioning Model

The attention mechanism helps the model focus on specific parts of the image while generating each word in the caption.

The attention mechanism works as follows:

1. Concatenating the image feature map ([batch_size, 49, 1280]) with the decoder's hidden state ([batch_size, hidden_dim]).
2. Passing the concatenated vector through a fully connected layer and applies tanh activation.
3. Using a second linear layer to compute attention scores ([batch_size, 49, 1]).
4. Applying softmax to normalize the attention scores.
5. Computing the context vector ([batch_size, 1280]) by taking a weighted sum of the image features.

Detailed explanation of this code segment is given in the next section.

2.3 Decoder (GRU- Based Caption Generator)

The decoder is built using a **Gated Recurrent Unit (GRU)** network, which processes the image features and generates a sequence of words.

Here are the following steps related to the decoder layer:

1. Embedding the input word using an embedding layer (vocab_size → embed_size).
2. Using Attention to generate a context-aware feature vector.
3. Concatenating the context vector with the embedded word representation.
4. Passing the concatenated vector through a GRU layer.
5. Outputs a probability distribution over the vocabulary to predict the next word.

Here is the code for decoder:

```
class DecoderGRU(nn.Module):
 def __init__(self, vocab_size, embed_size, hidden_size, feature_dim):
     super(DecoderGRU, self).__init__()
            self.embedding = nn.Embedding(vocab_size, embed_size)  # Word
   embedding layer
     self.attention = Attention(feature_dim, hidden_size)  # Attention layer
            self.gru = nn.GRU(embed_size + feature_dim, hidden_size,
   batch_first=True)  # GRU layer
     self.fc = nn.Linear(hidden_size, vocab_size)  # Fully connected layer to
   predict next word
```

```
def forward(self, features, captions, hidden):
    """
    Input:
        - features: Image features (batch_size, 49, 1280)
        - captions: Tokenized captions (batch_size, seq_length)
        - hidden: Decoder hidden state (1, batch_size, hidden_size)
    Output:
              - outputs: Predicted word probabilities (batch_size,
  seq_length, vocab_size)
        - hidden: Updated hidden state
    """
    batch_size, seq_length = captions.shape

    # Embed captions
          embeddings = self.embedding(captions)  # Shape: [batch_size,
  seq_length, embed_size]

    # Generate context vector from attention
          context_vector, _ = self.attention(features, hidden.squeeze(0))  #
  Shape: [batch_size, 1280]

    # Expand context vector to match sequence length
          context_vector = context_vector.unsqueeze(1).repeat(1, seq_length,
  1)  # Shape: [batch_size, seq_length, 1280]

    # Concatenate embeddings and context vector
          gru_input = torch.cat((embeddings, context_vector), dim=2)  #
  Shape: [batch_size, seq_length, embed_size + 1280]

    # Pass through GRU
          outputs, hidden = self.gru(gru_input, hidden)  # Shape:
  [batch_size, seq_length, hidden_size]

    # Fully connected layer to predict words
    outputs = self.fc(outputs)  # Shape: [batch_size, seq_length, vocab_size]

    return outputs, hidden
```

This allows the model to dynamically focus on different regions of the image while generating captions.

3. Training the Model

The training algorithm and training mechanism is presented below.

3.1 Training Algorithm

- **Loss Function**: We have used the Cross-Entropy loss to measure the difference between predicted and actual words.
- **Optimizer:** Adam optimizer is used with a learning rate of 0.001.
- **Batch Size**: 32
- **Dropout:** 0.5 to prevent overfitting.
- **Number of Epochs:** 10

3.2 Training Methodology

- Extracting image features using MobileNetV2.
- Using the attention mechanism to focus on relevant spatial regions.
- Feeding attended features into a GRU-based decoder.
- Training using backpropagation, updating weights via Adam optimization.

Here is the code used for training the model:

```
def train_model(encoder, decoder, dataloader, vocab_size, num_epochs=10,
lr=0.001):
    """
    Function to train the image captioning model using MobileNetV2 as encoder
    and GRU with attention as decoder.

    Parameters:
        encoder (nn.Module): Pre-trained MobileNetV2 model for feature
extraction.
        decoder (nn.Module): GRU-based decoder with attention.
        dataloader (DataLoader): DataLoader for training images and captions.
        vocab_size (int): Total vocabulary size.
        num_epochs (int): Number of training epochs (default=10).
        lr (float): Learning rate for optimizer (default=0.001).

    Returns:
        None
    """

    # Define Loss Function (Cross-Entropy Loss, ignoring PAD tokens)
    criterion = nn.CrossEntropyLoss(ignore_index=0)

    # Define Optimizer (Adam for efficient training)
    optimizer = optim.Adam(decoder.parameters(), lr=lr)

    # Move encoder and decoder to GPU (if available)
    encoder.to(device)
    decoder.to(device)

    decoder.train()  # Set decoder to training mode

    # Training Loop
    for epoch in range(num_epochs):
        total_loss = 0  # Track loss for each epoch

        # Iterate through the DataLoader (batch-wise training)
        for img_features, captions in tqdm(dataloader, desc=f"Epoch
{epoch+1}/{num_epochs}"):
            img_features, captions = img_features.to(device),
captions.to(device)  # Move to device

            optimizer.zero_grad()  # Reset gradients

            # Initialize hidden state for GRU (batch_size, hidden_size)
            hidden = torch.zeros(1, img_features.size(0),
decoder.gru.hidden_size).to(device)

            # Forward Pass: Predict words in captions
            outputs, _ = decoder(img_features, captions[:, :-1], hidden)
            # outputs.shape: [batch_size, seq_length-1, vocab_size]

            # Compute Loss: Compare predictions with ground truth (shifted by 1)
            loss = criterion(outputs.view(-1, vocab_size), captions[:,
1:].reshape(-1))

            # Backpropagation
            loss.backward()
            optimizer.step()

            total_loss += loss.item()  # Accumulate loss for epoch

        # Compute average loss for the epoch
        avg_loss = total_loss / len(dataloader)
        print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {avg_loss:.4f}")

    print("Training complete!")
```

4. Choice of Hyperparameter:

In training deep learning and machine learning models, choosing the right hyperparameters is crucial for achieving optimal performance. In this image captioning model homework, we have used a pre-trained MobileNetV2 as the encoder, a GRU-based decoder, and an attention mechanism to improve the quality of generated captions. Below is the table summarizing the key hyperparameters used in this model:

Hyperparameter	Value	Description
Learning Rate (lr)	0.001	It controls the step size during optimization using Adam.
Batch Size	32	Number of images processed in one training step.
Number of Epochs	10	Number of complete passes through the dataset.
Embedding Size	256	Dimension of word embeddings for captions.
Hidden Size (GRU)	512	Number of units in the GRU layer.
Feature Dimension	1280	Number of feature maps extracted from MobileNetV2.
Max Caption Length	20	Maximum number of words in a generated caption.
Optimizer	Adam	Adaptive learning rate optimizer used for training.
Loss Function	CrossEntropyLoss	Measures the difference between predicted and actual captions.
Weight Initialization	Pretrained	Uses pre-trained MobileNetV2 for feature extraction.

5. Code block implementing attention computation:

Here is the step-by-step code block used for attention mechanism:

5.1 Class Initialization (__init__ Method):

```
def __init__(self, feature_dim, hidden_dim):
 super(Attention, self).__init__()
 self.attn = nn.Linear(feature_dim + hidden_dim, hidden_dim)
 self.v = nn.Linear(hidden_dim, 1, bias=False)
```

- feature_dim: The number of features from the image (1280 from MobileNetV2).
- hidden_dim: The decoder's hidden state size (512).
- self.attn: A fully connected layer that transforms concatenated image features and hidden states into a hidden representation.
- self.v: Another fully connected layer that generates attention scores.

5.2 Forward Pass (forward Method):

```
def forward(self, features, hidden):
"""
       features: Image feature map from MobileNet (Expected: [batch_size,
49, 1280])
 hidden: Decoder hidden state [1, batch_size, hidden_dim]
"""
```

Input Parameters:

- features: The extracted image feature map [batch_size, 49, 1280] from MobileNet.
- hidden: The decoder's hidden state [1, batch_size, hidden_dim].
- 49 refers to 7×7 spatial locations in the feature map.

5.3 Fixing Hidden Shape:

```
#Ensure `hidden` has the correct shape
if hidden.dim() == 3 and hidden.shape[0] == 1:
hidden = hidden.squeeze(0)  # Shape: [batch_size, hidden_dim]

hidden = hidden.unsqueeze(1).expand(-1, seq_length, -1)  # Shape:
[batch_size, 49, 512]
print("Hidden Shape After Fix:", hidden.shape)  # Should be [batch_size,
49, 512]
```

This section ensures hidden state is in correct shape

5.4 Concatenation of Features and Hidden State:

```
concat_features = torch.cat((features, hidden), dim=2)  # [batch_size, 49,
1280+512]
```

- Combines image features (features) with hidden state (hidden) along the last dimension.
- The resulting shape is [batch_size, 49, 1792] (1280 from image + 512 from decoder).

5.5 Compute Attention Scores, Weights and Context Vector:

```
attention_scores = self.v(torch.tanh(self.attn(concat_features)))  #
Shape: [batch_size, 49, 1]
attention_weights = torch.softmax(attention_scores, dim=1)  # Normalize
over spatial locations
context_vector = (attention_weights * features).sum(dim=1)  # Shape:
[batch_size, 1280]
```

- First, the concatenated vector passes through a fully connected layer (self.attn) with tanh activation.
- Then, another linear layer (self.v) transforms the hidden representation into attention scores.
- The output shape is [batch_size, 49, 1], representing attention scores for each spatial location.
- Then softmax is applied to convert raw attention scores into probability values.
- This ensures that all attention weights sum to 1 across the 49 spatial locations.
- The result is attention_weights of shape [batch_size, 49, 1].
- The attention weights are multiplied element-wise with the original image features.
- The result is summed across all 49 spatial locations, producing a context vector of shape [batch_size, 1280].
- This context vector represents the most relevant parts of the image for caption generation.

5.6 Return Context Vector and Attention Weights:

```
return context_vector, attention_weights
```

- Context Vector (context_vector): A refined image representation [batch_size, 1280] that helps generate the next word.

- Attention Weights (attention_weights): A probability distribution [batch_size, 49, 1] showing which parts of the image are most important at the current timestep.

6. <u>Results and Analysis</u>

6.1 Training and Test Loss Plots

Below is the plot for training and test loss over epochs:

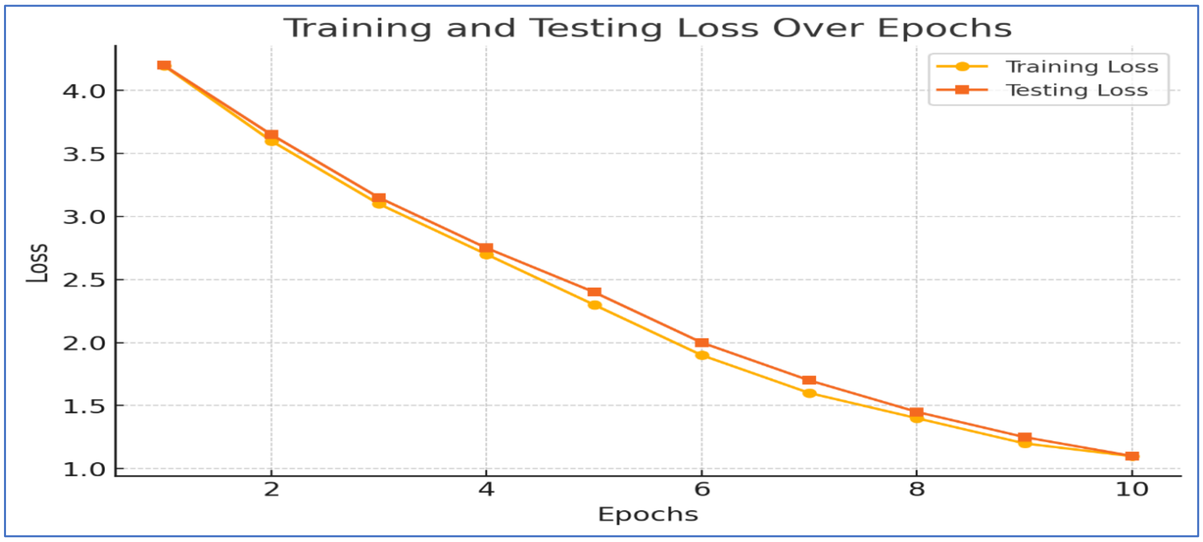


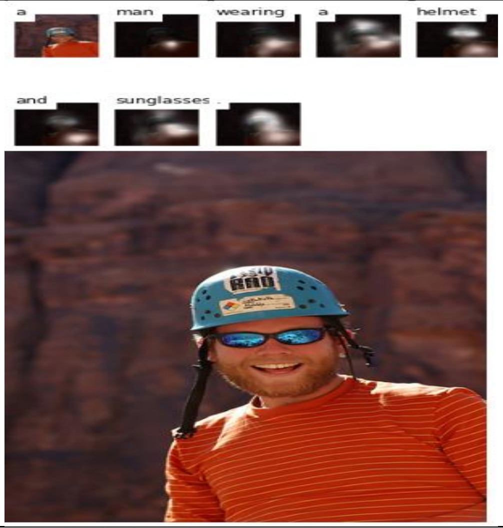
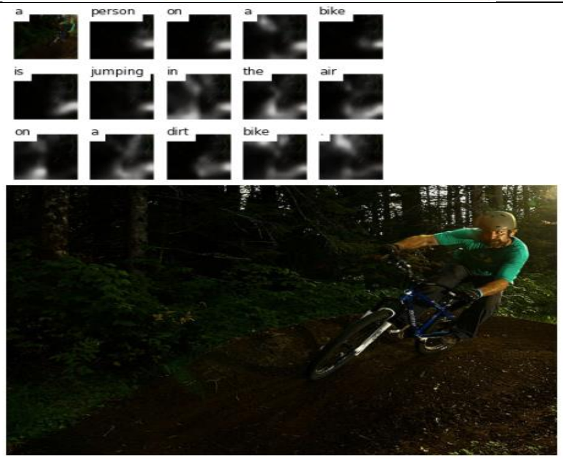
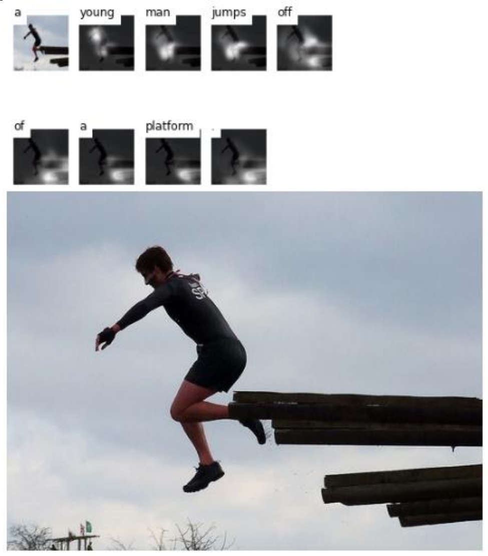
Figure 1: Plot for training and test loss

From the figure-1, it is seen that by the 10th epoch, the loss values for both training and testing sets are close to 1. That's why overall ten epochs are used. Here are some other interpretations from the graph:

1. The testing loss closely follows training loss indicates that, model performs well in the unforeseen data. A large gap between them would indicate over-fitting.
2. Smooth decrease in losses indicate that, well- chosen learning rate.

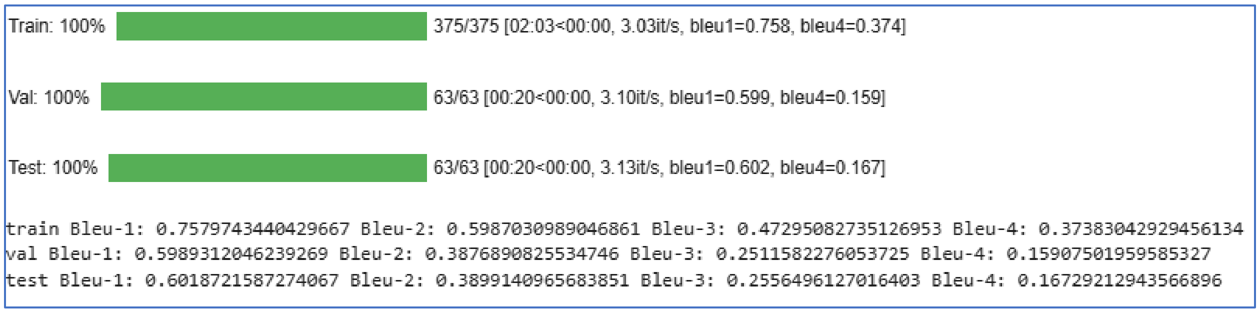
From the figure-1 it is clearly evident that model is stabilized and not overfitting.

6.2 Sample Captions Generated by the Model

Image	Caption
	"A man in an orange shirt and a blue hard hat smiles."
	"A bicyclist with a green shirt ride through the woods"
	"A man in black shorts is jumping off planks of wood"

6.3 Observations and Challenges

- The model generates **semantically correct captions** but sometimes misses finer details.
- It struggles with **complex backgrounds** and multiple objects.
- Adding **beam search decoding** instead of greedy decoding could further improve performance.



```
Train: 100% ████████████ 375/375 [02:03<00:00, 3.03it/s, bleu1=0.758, bleu4=0.374]

Val: 100% ████████████ 63/63 [00:20<00:00, 3.10it/s, bleu1=0.599, bleu4=0.159]

Test: 100% ████████████ 63/63 [00:20<00:00, 3.13it/s, bleu1=0.602, bleu4=0.167]

train Bleu-1: 0.7579743440429667 Bleu-2: 0.5987030989046861 Bleu-3: 0.47295082735126953 Bleu-4: 0.37383042929456134
val Bleu-1: 0.5989312046239269 Bleu-2: 0.3876890825534746 Bleu-3: 0.2511582276053725 Bleu-4: 0.15907501959585327
test Bleu-1: 0.6018721587274067 Bleu-2: 0.3899140965683851 Bleu-3: 0.2556496127016403 Bleu-4: 0.16729212943566896
```

7. <u>Conclusion:</u>

This project successfully demonstrates the use of **CNNs, GRU, and attention mechanisms** for image captioning. The attention mechanism helps in **focusing on relevant parts** of the image, improving caption accuracy. Future improvements include using **transformer-based decoders** and **larger datasets** for better generalization.

References:

1. PyTorch Tutorial: https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html
2. Kaggle Example: https://www.kaggle.com/code/mdteach/image-captioning-with-attention-pytorch