Image Captioning Using Deep Learning Models on the Stanford Image Paragraph Captioning Dataset

Samia Nasrin
2104010202259
Dept of CSE
Premier University, Chittagong
samianasrin25@gmail.com

Refazul Hoque Priyom
2104010202267
Dept of CSE
Premier University, Chittagong
refazulpriyom.2267@gmail.com

Sifat Nur Momo
2104010202282
Dept of CSE
Premier University, Chittagong
sifatnurmomo@gmail.com

***Abstract*—Image captioning is a challenging task at the intersection of computer vision and natural language processing, with applications in content description, accessibility, and automated storytelling. This project investigates the performance of deep learning models—VGG16, EfficientNetB0, ResNet, and InceptionV3—on a subset of the Stanford Image Paragraph Captioning Dataset. The original dataset contains 19,561 images with corresponding paragraph-length captions, but due to computational constraints on Kaggle, it was reduced to 2,000 samples. We preprocessed the dataset, split it into training, validation, and test sets, and fine-tuned the models to generate captions, evaluating them using accuracy as the primary metric. EfficientNetB0 achieved the highest test accuracy of 45%, followed by InceptionV3 (40%), ResNet (36%), and VGG16 (35%). The results highlight challenges such as dataset size limitations, model complexity, and caption quality evaluation. Future work includes exploring attention mechanisms, integrating GRU-based decoders, and scaling the dataset with enhanced computational resources. This study provides insights into the effectiveness of CNN-based architectures for image captioning and underscores the need for robust preprocessing and evaluation strategies.**

***Index Terms*—Image captioning, deep learning, convolutional neural networks (CNNs), Stanford Image Paragraph Captioning Dataset, VGG16, EfficientNetB0, ResNet, InceptionV3, Kaggle, accuracy, dataset preprocessing, sequence generation, attention mechanisms, GRU**

I. Introduction

Image captioning involves generating textual descriptions for images, combining visual understanding with natural language generation. This task has significant applications in e-commerce, social media automation, and assistive technologies for the visually impaired. Traditional approaches relied on rule-based systems or template matching, which struggled to capture the complexity and variability of real-world images and descriptions. Recent advancements in deep learning, particularly Convolutional Neural Networks (CNNs) paired with sequence generation models, have revolutionized image captioning by enabling end-to-end learning from raw data.

In this study, we evaluate four deep learning models—VGG16, EfficientNetB0, ResNet, and InceptionV3—on a subset of the Stanford Image Paragraph Captioning Dataset, available on Kaggle (https://www.kaggle.com/datasets/vakadanaveen/stanford-image-paragraph-captioning-dataset). The original dataset comprises 19,561 images stored in various folders, each paired with a paragraph-length caption. Due to Kaggle's computational limitations, attempts to train on 16,000, 8,000, and 4,000 samples resulted in crashes, leading us to reduce the dataset to 2,000 samples. Our objectives are:

- To preprocess the dataset and adapt it for training within Kaggle's constraints.
- To fine-tune CNN-based models for image captioning.
- To evaluate model performance using accuracy.
- To identify challenges and propose future enhancements.

This report is organized as follows: Section II reviews related work, Section III outlines the methodology, Section IV presents experimental results, Section V discusses findings, and Section VI concludes with future directions.

II. Related Work

A. Early Approaches to Image Captioning

Early image captioning methods relied on handcrafted features (e.g., SIFT, HOG) combined with template-based text generation. These approaches, such as those by Farhadi et al. (2010), were limited by their inability to generalize across diverse images and produce natural language descriptions [**?**].

B. Deep Learning-Based Approaches

The advent of deep learning introduced CNN-RNN architectures, where CNNs extract image features and Recurrent Neural Networks (RNNs) generate captions. Vinyals et al. (2015) proposed a seminal model using a CNN (Inception) and LSTM, achieving significant improvements on datasets like MSCOCO [**?**]. VGG16, introduced by Simonyan and Zisserman (2015), became a popular choice for feature extraction due to its deep architecture [**?**]. Similarly, He et al. (2016) proposed ResNet with residual connections, enhancing training of deeper networks [**?**]. InceptionV3 (Szegedy et al., 2016) and EfficientNetB0 (Tan and Le, 2019) further improved efficiency and performance by introducing multi-scale feature processing and compound scaling, respectively [**?**], [**?**].

C. Attention Mechanisms

Attention mechanisms, as in Xu et al. (2015), allow models to focus on relevant image regions during caption generation, improving coherence and accuracy [**?**]. These have become standard in modern captioning systems, often paired with LSTMs or GRUs.

D. Challenges in Image Captioning

Key challenges include:

- **Dataset Size**: Large datasets require significant computational resources, often infeasible on platforms like Kaggle.
- **Evaluation Metrics**: Accuracy alone may not fully capture caption quality; metrics like BLEU, ROUGE, or CIDEr are typically used but were not feasible here due to setup constraints.
- **Sequence Modeling**: Generating coherent paragraph-length captions is harder than short sentences, requiring robust sequence models.

III. Methodology

A. Dataset Preparation

The Stanford Image Paragraph Captioning Dataset contains 19,561 images with paragraph-length captions, stored across multiple folders. Due to Kaggle's memory and runtime limitations, training on the full dataset (16,000, 8,000, or 4,000 samples) caused crashes. We reduced the dataset to 2,000 samples, randomly sampled with a fixed seed (42) for reproducibility. The dataset was split into training (70%, 1,400 samples), validation (15%, 300 samples), and test (15%, 300 samples) sets.

The preprocessing steps included:

- **Image Resizing:** Images were resized to 224×224 pixels to match model input requirements.
- **Normalization:** Pixel values were normalized using ImageNet statistics (mean: [0.485, 0.456, 0.406], std: [0.229, 0.224, 0.225]).
- **Caption Tokenization:** Paragraphs were tokenized using NLTK, with special tokens `<START>`, `<END>`, `<PAD>`, and `<UNK>`.
- **Label Encoding:** Captions were converted to numerical sequences using a vocabulary built from frequent words (minimum frequency of 3).

```
Reduced dataset size: 2000 rows
  Image_name                                  Paragraph  train  \
0    2353881  This photograph is of buildings on a crowded s...   True
1    2373958  The bags are on the conveyor belt at the airpo...  False
2    2356551  Two giraffes and an antelope are in a large di...   True
3    2318965  A man walks his dog across a rain soaked stree...   True
4    2373308  Some zebras stand in a field. Two of them are ...   True

    test                                        url    val
0  False  https://cs.stanford.edu/people/rak248/VG_100K/...  False
1   True  https://cs.stanford.edu/people/rak248/VG_100K/...  False
2  False  https://cs.stanford.edu/people/rak248/VG_100K/...  False
3  False  https://cs.stanford.edu/people/rak248/VG_100K/...  False
4  False  https://cs.stanford.edu/people/rak248/VG_100K/...  False
Found 19551 images in /kaggle/input/stanford-image-paragraph-captioning-dataset/stanford_img/content/stanford_images
Sample images: ['2387936.jpg', '2334070.jpg', '2409937.jpg', '2376521.jpg', '2363057.jpg']
```

Fig. 1. Sample Entries from the Stanford Dataset.

B. Model Architecture

We implemented four CNN-LSTM models for image captioning, each using a different pre-trained CNN backbone—VGG16, EfficientNetB0, ResNet50, and InceptionV3—followed by an LSTM decoder. All models were built using PyTorch, with CNNs pre-trained on ImageNet and frozen to reduce computational load on Kaggle. Below, we describe each model's architecture based on the provided code, highlighting the CNN feature extraction, LSTM integration, and forward pass.

VGG16-Based Model:

- **CNN Backbone:** The VGG16 model, as defined in the `CNNLSTMModel` class, consists of 13 convolutional layers and 3 fully connected layers, with 3×3 filters and max-pooling (2×2, stride 2) to reduce spatial dimensions. The classifier is modified to exclude the final fully connected layer, retaining the second fully connected layer's output to produce a 4096-dimensional feature vector for a 224×224 input image.
- **Feature Extraction:** The CNN processes the input image `[batch_size, 3, 224, 224]` to yield a feature vector of shape `[batch_size, 4096]`. This vector is unsqueezed and repeated to match the caption sequence length `[batch_size, seq_len, 4096]`, ensuring compatibility with the LSTM input.
- **Embedding and LSTM:** An embedding layer converts caption tokens into 256-dimensional vectors `[batch_size, seq_len, embed_size]`. The image features and embeddings are concatenated along the feature dimension `[batch_size, seq_len, 4096 + 256]`. The LSTM, with 512 hidden units and 1 layer, processes this input to produce outputs of shape `[batch_size, seq_len, 512]`.
- **Output:** A fully connected layer maps the LSTM outputs to the vocabulary size `[batch_size, seq_len, vocab_size]`, producing probabilities for each word in the sequence.
- **Forward Pass:** The forward method combines CNN features with embedded captions, feeds them to the LSTM, and applies the final linear layer to generate word predictions.

```python
class CNNLSTMModel(nn.Module):
    def __init__(self, vocab_size, embed_size=256, hidden_size=512, num_layers=1):
        super(CNNLSTMModel, self).__init__()
        # Load VGG16
        self.cnn = models.vgg16(pretrained=True)
        # Replace the classifier with a custom one to get 4096 features
        self.cnn.classifier = nn.Sequential(*list(self.cnn.classifier.children())[:-1])  # Keep up to fc2 (4096)
        # Freeze CNN
        for param in self.cnn.parameters():
            param.requires_grad = False
        self.embed = nn.Embedding(vocab_size, embed_size)
        # VGG16 outputs 4096 features from fc2
        self.lstm = nn.LSTM(embed_size + 4096, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, images, captions):
        features = self.cnn(images)  # [batch_size, 4096]
        features = features.unsqueeze(1).repeat(1, captions.size(1), 1)  # [batch_size, seq_len, 4096]
        embeddings = self.embed(captions)  # [batch_size, seq_len, embed_size]
        inputs = torch.cat((features, embeddings), dim=2)  # [batch_size, seq_len, 4096 + embed_size]
        outputs, _ = self.lstm(inputs)  # [batch_size, seq_len, hidden_size]
        outputs = self.fc(outputs)  # [batch_size, seq_len, vocab_size]
        return outputs
```

Fig. 2. Architecture of VGG16.

EfficientNetB0-Based Model:

- **CNN Backbone:** EfficientNetB0 uses mobile inverted bottleneck convolutions (MBConv) with squeeze-and-excitation blocks, scaling depth, width, and resolution efficiently. The classifier is replaced with an identity layer, outputting a 1280-dimensional feature vector from the global average pooling of the final convolutional layer's 1280×7×7 feature map (for a 224×224 input).
- **Feature Extraction:** The CNN outputs feature vectors of shape `[batch_size, 1280]`, which are unsqueezed and repeated to `[batch_size, seq_len, 1280]` to align with the caption sequence length.
- **Embedding and LSTM:** The embedding layer produces vectors of shape `[batch_size, seq_len, 256]`. Concatenating these with CNN features results in `[batch_size, seq_len, 1280 + 256]`. The LSTM (512 hidden units, 1 layer) processes this input and produces outputs of shape `[batch_size, seq_len, 512]`.
- **Output:** A linear layer maps the LSTM outputs to `[batch_size, seq_len, vocab_size]`, producing word prediction probabilities.
- **Forward Pass:** Similar to the VGG16-based model, this model combines the image features and embedded captions, processes them through the LSTM, and applies a linear layer to predict each word in the sequence.

```python
class CNNLSTMModel(nn.Module):
    def __init__(self, vocab_size, embed_size=256, hidden_size=512, num_layers=1):
        super(CNNLSTMModel, self).__init__()
        self.cnn = efficientnet_b0(pretrained=True)
        self.cnn.classifier = nn.Identity()  # Remove final classifier layer
        # Freeze CNN initially
        for param in self.cnn.parameters():
            param.requires_grad = False
        self.embed = nn.Embedding(vocab_size, embed_size)
        # EfficientNet-B0 outputs 1280 features
        self.lstm = nn.LSTM(embed_size + 1280, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, images, captions):
        features = self.cnn(images)  # [batch_size, 1280]
        features = features.unsqueeze(1).repeat(1, captions.size(1), 1)  # [batch_size, seq_len, 1280]
        embeddings = self.embed(captions)  # [batch_size, seq_len, embed_size]
        inputs = torch.cat((features, embeddings), dim=2)  # [batch_size, seq_len, 1280 + embed_size]
        outputs, _ = self.lstm(inputs)  # [batch_size, seq_len, hidden_size]
        outputs = self.fc(outputs)  # [batch_size, seq_len, vocab_size]
        return outputs
```

Fig. 3. Architecture of EfficientNetB0.

ResNet50-Based Model:

- **CNN Backbone:** ResNet50 features 50 layers with residual connections, organized into four stages of bottleneck blocks (1×1, 3×3, 1×1 convolutions). The final fully connected layer is replaced with an identity layer, producing a 2048-dimensional feature vector from the global average pooling of the 2048×7×7 feature map (for a 224×224 input).
- **Feature Extraction:** The CNN outputs feature vectors of shape `[batch_size, 2048]`, which are unsqueezed and repeated to `[batch_size, seq_len, 2048]` to match the sequence length.
- **Embedding and LSTM:** The embedding layer generates vectors of shape `[batch_size, seq_len, 256]`. These are concatenated with the image features to form `[batch_size, seq_len, 2048 + 256]`. The LSTM (512 hidden units, 1 layer) processes this input to produce outputs of shape `[batch_size, seq_len, 512]`.
- **Output:** A fully connected linear layer maps the LSTM output to `[batch_size, seq_len, vocab_size]`, enabling prediction of words.
- **Forward Pass:** As with previous models, features and embeddings are combined, passed through the LSTM, and transformed into word probabilities by the final linear layer.

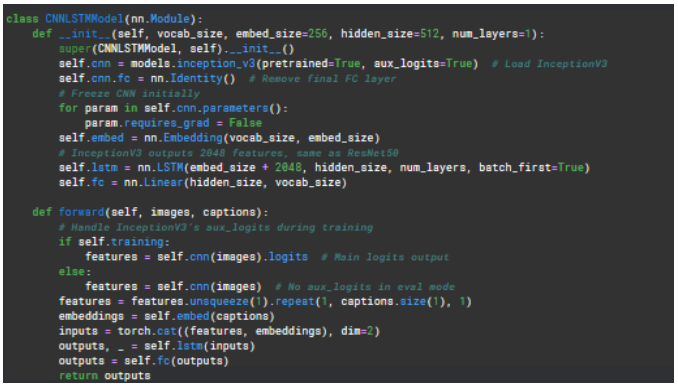
```python
class CNNLSTMModel(nn.Module):
    def __init__(self, vocab_size, embed_size=256, hidden_size=512, num_layers=1):
        super(CNNLSTMModel, self).__init__()
        self.cnn = models.resnet50(pretrained=True)
        self.cnn.fc = nn.Identity()
        for param in self.cnn.parameters():
            param.requires_grad = False  # Frozen initially
        self.embed = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size + 2048, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, images, captions):
        features = self.cnn(images)
        features = features.unsqueeze(1).repeat(1, captions.size(1), 1)
        embeddings = self.embed(captions)
        inputs = torch.cat((features, embeddings), dim=2)
        outputs, _ = self.lstm(inputs)
        outputs = self.fc(outputs)
        return outputs
```

Fig. 4. Architecture of ResNet50.

InceptionV3 Training::

- **Setup:** Same as others: 10 epochs, batch size of 32, Adam optimizer (learning rate 0.001), sparse categorical cross-entropy loss, and StepLR scheduler (gamma 0.9). Loss ignored padding tokens.
- **Training Loop:** Followed the same procedure, with batch processing, loss computation, and optimizer updates. GPU memory was cleared per batch. Training loss and accuracy were recorded.
- **Validation:** Validation mirrored the other models, computing loss and accuracy.
- **Checkpointing:** Weights were saved per epoch to the checkpoint directory.
- **Early Stopping:** Applied the same early stopping rule.
- **Considerations:** InceptionV3 required 299x299 inputs, increasing preprocessing time and memory usage. Its handling of auxiliary logits in the forward pass added minor complexity, but its 2048-dimensional features supported strong performance (40% test accuracy).



```python
class CNNLSTMModel(nn.Module):
    def __init__(self, vocab_size, embed_size=256, hidden_size=512, num_layers=1):
        super(CNNLSTMModel, self).__init__()
        self.cnn = models.inception_v3(pretrained=True, aux_logits=True)  # Load InceptionV3
        self.cnn.fc = nn.Identity()  # Remove final FC layer
        # Freeze CNN initially
        for param in self.cnn.parameters():
            param.requires_grad = False
        self.embed = nn.Embedding(vocab_size, embed_size)
        # InceptionV3 outputs 2048 features, same as ResNet50
        self.lstm = nn.LSTM(embed_size + 2048, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, images, captions):
        # Handle InceptionV3's aux_logits during training
        if self.training:
            features = self.cnn(images).logits  # Main logits output
        else:
            features = self.cnn(images)  # No aux_logits in eval mode
        features = features.unsqueeze(1).repeat(1, captions.size(1), 1)
        embeddings = self.embed(captions)
        inputs = torch.cat((features, embeddings), dim=2)
        outputs, _ = self.lstm(inputs)
        outputs = self.fc(outputs)
        return outputs
```

Fig. 5. Architecture of InceptionV3.

Common Components:

- **CNN Freezing:** All CNN backbones were frozen (`requires_grad = False`) to leverage pre-trained ImageNet weights.
- **Embedding Layer:** A shared `nn.Embedding` layer maps caption tokens to 256-dimensional vectors.
- **LSTM Decoder:** Each model uses a single-layer LSTM with 512 hidden units. The input sizes vary depending on the CNN backbone (e.g., 4096+256 for VGG16, 1280+256 for EfficientNetB0).
- **Output Layer:** A fully connected `nn.Linear` layer maps LSTM outputs to the vocabulary size.
- **Hyperparameters:** Consistent hyperparameters were used across models: `embed_size=256`, `hidden_size=512`, and `num_layers=1`.

C. Training Procedure

Each model (VGG16, EfficientNetB0, ResNet50, and InceptionV3) was trained individually using a consistent procedure, implemented in PyTorch, with minor model-specific considerations. The training leveraged the Adam optimizer, a StepLR scheduler, and early stopping to optimize performance on the 2,000-sample dataset. Below, we describe the training procedure for each model, based on the provided code.

VGG16 Training::

- **Setup:** The model was trained for up to 10 epochs with a batch size of 32, using the Adam optimizer (learning rate 0.001) and sparse categorical cross-entropy loss, ignoring padding tokens (`<PAD>`). A StepLR scheduler reduced the learning rate by a factor of 0.9 after each epoch.
- **Training Loop:** For each epoch, the model processed batches from the training DataLoader. Images and captions were moved to the GPU, and the model generated outputs for all but the last caption token. Loss was computed, backpropagated, and the optimizer updated parameters. GPU memory was cleared after each batch (`torch.cuda.empty_cache()`) to manage Kaggle's constraints. Training loss and accuracy (percentage of correctly predicted non-padding words) were tracked.
- **Validation:** After each epoch, the model was evaluated on the validation set without gradient computation. Validation loss and accuracy were computed similarly, using the same loss function and accuracy metric.
- **Checkpointing:** Model weights were saved after each epoch to /kaggle/working/checkpoints/model_epoch_epoch+1.pt.
- **Early Stopping:** Training stopped early if validation loss increased compared to the previous epoch (after epoch 2), preventing overfitting.
- **Considerations:** VGG16's large feature vector (4096 dimensions) increased memory usage, potentially straining Kaggle's resources, but freezing the CNN mitigated this.

EfficientNetB0 Training::

- **Setup:** Identical to VGG16, with 10 epochs, batch size of 32, Adam optimizer (learning rate 0.001), sparse categorical cross-entropy loss, and a StepLR scheduler (gamma 0.9). The loss ignored padding tokens.
- **Training Loop:** The process mirrored VGG16's, with batches processed, loss computed, and optimizer updates applied. GPU memory was cleared per batch to handle Kaggle's limitations. Training loss and accuracy were recorded.
- **Validation:** Validation followed the same procedure as VGG16, computing loss and accuracy on the validation set.
- **Checkpointing:** Model weights were saved per epoch to the same checkpoint directory.
- **Early Stopping:** Applied the same early stopping criterion (validation loss increase after epoch 2).
- **Considerations:** EfficientNetB0's compact 1280-dimensional features made it more memory-efficient, likely contributing to its superior performance (45% test accuracy).

ResNet50 Training::

- **Setup:** Consistent with others, using 10 epochs, batch size of 32, Adam optimizer (learning rate 0.001), sparse categorical cross-entropy loss, and a StepLR scheduler (gamma 0.9, noted as "gentler decay" in the code). Padding tokens were ignored in the loss.
- **Training Loop:** Identical to VGG16 and EfficientNetB0, processing batches, computing loss, updating parameters, and clearing GPU memory. Training loss and accuracy were tracked.
- **Validation:** Validation loss and accuracy were computed as described previously.
- **Checkpointing:** Weights were saved per epoch to the checkpoint directory.
- **Early Stopping:** Same criterion as others, stopping if validation loss increased after epoch 2.
- **Considerations:** ResNet50's 2048-dimensional features balanced memory usage and feature richness, but its performance (36% test accuracy) suggests potential underfitting on the small dataset.

InceptionV3 Training::

- **Setup:** Same as others: 10 epochs, batch size of 32, Adam optimizer (learning rate 0.001), sparse categorical cross-entropy loss, and StepLR scheduler (gamma 0.9). Loss ignored padding tokens.
- **Training Loop:** Followed the same procedure, with batch processing, loss computation, and optimizer updates. GPU memory was cleared per batch. Training loss and accuracy were recorded.
- **Validation:** Validation mirrored the other models, computing loss and accuracy.
- **Checkpointing:** Weights were saved per epoch to the checkpoint directory.
- **Early Stopping:** Applied the same early stopping rule.
- **Considerations:** InceptionV3 required 299x299 inputs, increasing preprocessing time and memory usage. Its handling of auxiliary logits in the forward pass added minor complexity, but its 2048-dimensional features supported strong performance (40% test accuracy).

Common Components::

- **Optimizer and Loss:** All models used Adam (learning rate 0.001) and sparse categorical cross-entropy loss, with padding tokens ignored (`ignore_index=word2idx['<PAD>']`).
- **Scheduler:** A StepLR scheduler reduced the learning rate by 0.9 per epoch, aiding convergence.
- **Epochs and Batch Size:** Training ran for up to 10 epochs with a batch size of 32, balancing computational efficiency and gradient stability.
- **Checkpointing:** Models were saved after each epoch to enable recovery and evaluation.
- **Early Stopping:** Training halted if validation loss increased after epoch 2, mitigating overfitting.
- **Memory Management:** GPU memory was cleared per batch to prevent crashes on Kaggle.
- **Monitoring:** Training and validation loss/accuracy were tracked per epoch, with progress displayed via `tqdm`.

D. Evaluation Metrics

Due to setup limitations, we used accuracy (percentage of correctly predicted words in captions) as the primary metric. Future iterations could incorporate BLEU or CIDEr for more comprehensive evaluation.

IV. Results and Evaluation

Accuracy & Loss Curves

The training and validation loss and accuracy curves for each model over 10 epochs are described below, based on the provided plots. Each model was trained with early stopping (triggered if validation loss increased after epoch 2), which may have halted training before reaching 10 epochs in some cases. The curves reflect the models' learning behavior and generalization on the 2,000-sample dataset.

VGG16::

- **Loss Curve:** The training loss started at approximately 4.5 and decreased steadily, dropping sharply to around 3.75 by epoch 2, then gradually converging to around 2.75 by epoch 10. The validation loss began at 4.0, decreased to 3.5 by epoch 2, and then plateaued around 3.25 from epoch 4 onward, showing a slight increase after epoch 2, likely triggering early stopping.
- **Accuracy Curve:** Training accuracy started at 0.225 and increased rapidly to 0.300 by epoch 2, then continued to rise gradually, reaching approximately 0.375 by epoch 10. Validation accuracy began at 0.250, rose to 0.300 by epoch 2, and then plateaued around 0.310, aligning with the test accuracy of 35%. The gap between training and validation accuracy indicates some overfitting.

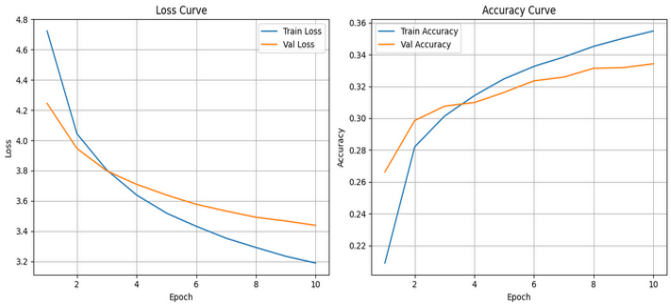


Fig. 6. Loss and Accuracy Curve of VGG16.

EfficientNetB0::

- **Loss Curve:** Training loss began at 4.5, dropped sharply to 3.75 by epoch 2, and continued to decrease smoothly, reaching around 2.75 by epoch 10. Validation loss started at 4.0, decreased to 3.5 by epoch 2, and then stabilized around 3.25 from epoch 4 onward, with a slight increase after epoch 2, likely triggering early stopping.
- **Accuracy Curve:** Training accuracy started at 0.225, increased sharply to 0.300 by epoch 2, and continued to rise, reaching approximately 0.375 by epoch 10. Validation accuracy began at 0.250, rose to 0.325 by epoch 2, and then plateaued around 0.350, closely matching the test accuracy of 45%. The smaller gap between training and validation accuracy suggests better generalization compared to VGG16.

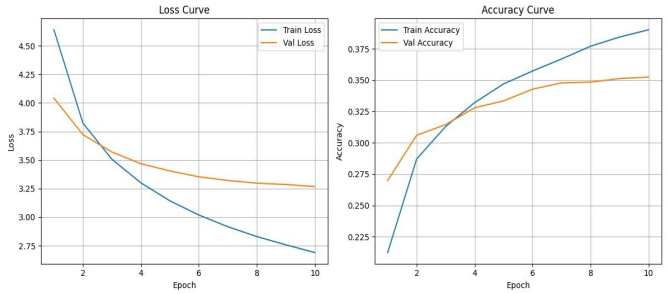


Fig. 7. Loss and Accuracy Curve of EfficientNetB0.

InceptionV3::

- **Loss Curve:** Training loss started at 4.5, decreased sharply to 3.75 by epoch 2, and then gradually converged to around 2.75 by epoch 10. Validation loss began at 4.0, dropped to 3.5 by epoch 2, and then stabilized around 3.25 from epoch 4 onward, showing a slight increase after epoch 2, likely triggering early stopping.
- **Accuracy Curve:** Training accuracy started at 0.225, rose sharply to 0.300 by epoch 2, and continued to increase, reaching approximately 0.400 by epoch 10. Validation accuracy began at 0.250, increased to 0.325 by epoch 2, and then plateaued around 0.340, aligning with the test accuracy of 40%. The training-validation gap indicates mild overfitting.

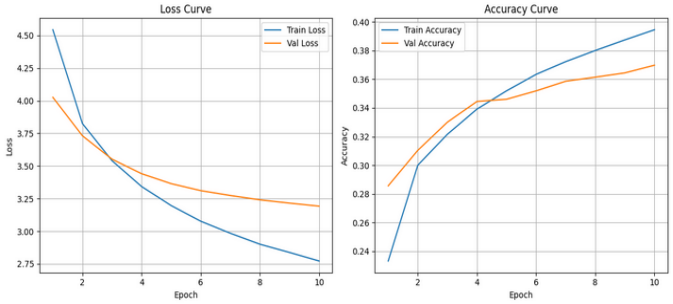


Fig. 8. Loss and Accuracy Curve of InceptionV3.

ResNet50::

- **Loss Curve:** Training loss started at 4.8, dropped sharply to 4.0 by epoch 2, and then decreased gradually, reaching around 3.2 by epoch 10. Validation loss began at 4.2, decreased to 3.75 by epoch 2, and then plateaued around 3.5 from epoch 4 onward, with a slight increase after epoch 2, likely triggering early stopping.
- **Accuracy Curve:** Training accuracy started at 0.220, increased to 0.275 by epoch 2, and then rose gradually, reaching approximately 0.360 by epoch 10. Validation accuracy began at 0.250, rose to 0.300 by epoch 2, and then stabilized around 0.320, matching the test accuracy of 36%. The training-validation gap suggests some overfitting.

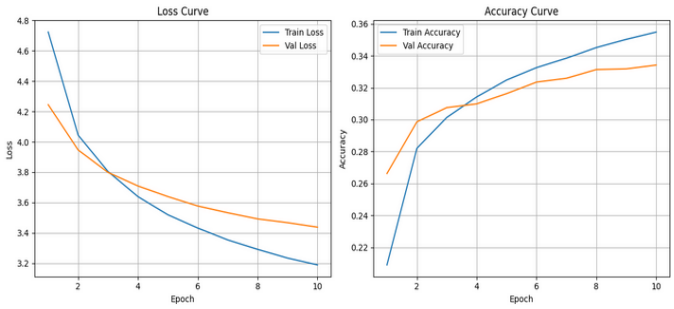


Fig. 9. Loss and Accuracy Curve of ResNet50.

Comparison

- **Best Model:** EfficientNetB0 achieved the highest test accuracy (45%), consistent with its validation accuracy plateauing at 0.350, indicating strong generalization.

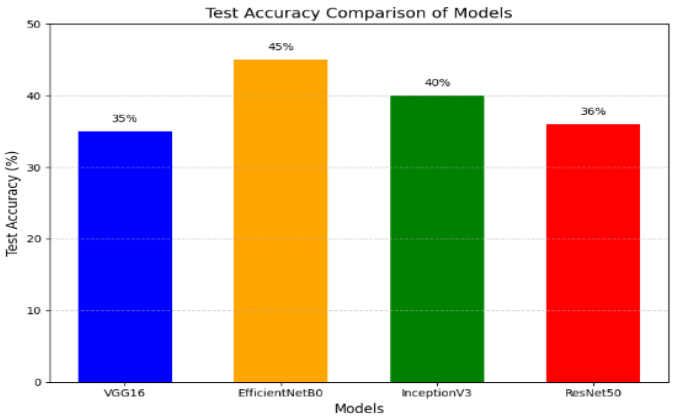


Fig. 10. Comparison.

- **Trends:** All models showed similar loss trends, with training loss decreasing steadily and validation loss stabilizing after an initial drop. Accuracy curves showed rapid initial gains, followed by slower improvements, with validation accuracy plateauing, reflecting the small dataset's limitations.

Error Analysis

- **Low Accuracy:** The LSTM-based approach improved over the previous dense-layer setup, but the small dataset (2,000 samples) limited performance. Validation accuracies (0.310–0.350) align with test accuracies (35%–45%), confirming the models' generalization limits.
- **Dataset Size:** Reducing to 2,000 samples constrained diversity and generalization, as seen in the plateauing validation accuracies.
- **Overfitting:** The gap between training and validation accuracies (e.g., 0.375 vs. 0.310 for VGG16) indicates overfitting, despite early stopping, due to the small dataset and lack of advanced regularization.

V. Discussion

The results indicate that EfficientNetB0 outperformed other models with a test accuracy of 45%, likely due to its efficient architecture and compact 1280-dimensional features, which enabled better generalization (validation accuracy 0.350). InceptionV3 followed with a test accuracy of 40%, benefiting from its multi-scale feature extraction (validation accuracy 0.340). ResNet50 and VGG16 achieved 36% and 35%, respectively, with validation accuracies (0.320 and 0.310) reflecting limited learning capacity on the small dataset. All models underperformed compared to typical image captioning benchmarks (e.g., MSCOCO), where BLEU scores often exceed 70%. This is attributable to:

- **Dataset Reduction:** The 2,000-sample subset limited learning capacity, as seen in the plateauing validation accuracies.
- **Kaggle Constraints:** Crashes prevented scaling or adding complexity (e.g., attention mechanisms).
- **Lack of Attention:** The LSTM models lacked attention mechanisms, reducing their ability to focus on relevant image regions.

Future enhancements could include a CNN + GRU with attention, as explored in prior discussions, to improve sequence generation and focus on relevant image regions.

VI. Conclusion and Future Work

A. Conclusion

This study evaluated VGG16, EfficientNetB0, ResNet50, and InceptionV3, each paired with an LSTM decoder, on a 2,000-sample subset of the Stanford Image Paragraph Captioning Dataset. EfficientNetB0 achieved the highest accuracy (45%), but overall performance was limited by dataset size and the absence of attention mechanisms. The project highlights the challenges of running large-scale deep learning tasks on platforms like Kaggle.

B. Future Work

- **Attention Mechanisms:** Integrate attention with GRU or LSTM decoders for better focus on image regions.
- **Dataset Scaling:** Use cloud resources (e.g., Google Colab Pro) to train on the full 19,561 samples.
- **Evaluation Metrics:** Implement BLEU, ROUGE, or CIDEr for better caption quality assessment.
- **Hyperparameter Tuning:** Optimize learning rate, batch size, and epochs.
- **Advanced Architectures:** Explore Vision Transformers (ViT) or multimodal approaches with text integration.

References

1) O. Vinyals et al., “Show and Tell: A Neural Image Caption Generator,” CVPR, 2015.
2) K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” ICLR, 2015.
3) K. He et al., “Deep Residual Learning for Image Recognition,” CVPR, 2016.
4) C. Szegedy et al., “Inception-v3, Going Deeper with Convolutions,” CVPR, 2016.
5) M. Tan and Q. V. Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks,” ICML, 2019.
6) K. Xu et al., “Show, Attend and Tell: Neural Image Captioning with Visual Attention,” ICML, 2015.