# Deep Learning Assignment: Image Captioning with CNN-RNN Architecture

Dr. Mahdi Eftekhari
Department of Computer Engineering

Spring 2025

## Overview

This assignment focuses on building an end-to-end image captioning system using a combination of Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). Image captioning lies at the intersection of computer vision and natural language processing, requiring models to understand visual content and generate appropriate textual descriptions.

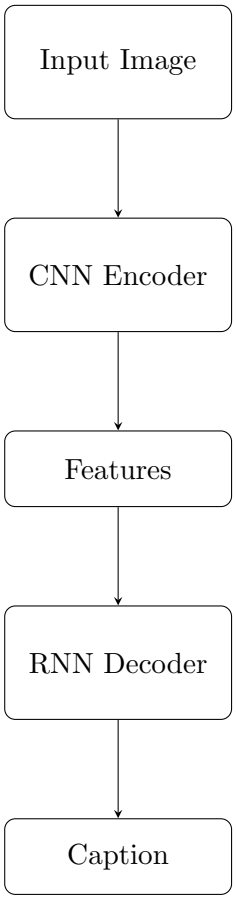


Figure 1: Image Captioning Architecture

## Learning Objectives

Through this assignment, you will:

- Implement an encoder-decoder architecture for multimodal learning
- Build and train sequence models (LSTMs) for text generation
- Process and prepare image and text data for deep learning
- Evaluate natural language generation using appropriate metrics

## 1 Background

Image captioning combines computer vision and natural language processing to automatically generate textual descriptions of images. This task requires:

1. Understanding the visual content of images (objects, scenes, actions, attributes)
2. Generating grammatically correct and contextually relevant sentences

The standard architecture consists of:

- **Encoder**: A CNN (typically pre-trained on ImageNet) that extracts visual features
- **Decoder**: An RNN/LSTM that generates captions word-by-word based on image features

## 2 Dataset

For this assignment, you will work with the Flickr8k dataset, which contains:

- Approximately 8,000 images from Flickr
- 5 different captions for each image (40,000 captions total)
- A diverse range of scenes, objects, and actions

The dataset is suitable for academic projects and offers a good balance between size and training time requirements.

## 3 Project Structure

Your implementation should follow this directory structure:

```
image_captioning_assignment/
|-- data/
|   |-- download_flickr.py
|-- models/
|   |-- encoder.py
|   |-- decoder.py
|   |-- caption_model.py
|-- utils/
|   |-- dataset.py
```

```
|   |-- vocabulary.py
|   |-- trainer.py
|   |-- metrics.py
|-- notebooks/
|   |-- 1_Data_Exploration.ipynb
|   |-- 2_Feature_Extraction.ipynb
|   |-- 3_Model_Training.ipynb
|   |-- 4_Evaluation_Visualization.ipynb
|-- requirements.txt
|-- README.md
```

# 4 Detailed Tasks

## 4.1 Task 1: Data Preparation (15%)

- Implement the data download and preprocessing functionality
- Create a PyTorch dataset class for the Flickr8k dataset
- Implement image transformation pipeline
- Preprocess captions (tokenization, normalization)
- Build a vocabulary from the captions
- Split the data into training, validation, and test sets

## 4.2 Task 2: Model Implementation (30%)

- **Encoder**: Implement a CNN-based encoder using a pre-trained model
  - Choose from ResNet18, ResNet50, MobileNetV2, or InceptionV3
  - Extract features from the penultimate layer
  - Add projection layers if needed
- **Decoder**: Implement an LSTM-based caption generator
  - Word embedding layer
  - LSTM/GRU layer(s)
  - Linear projection to vocabulary size
- **Combined Model**: Integrate the encoder and decoder into a unified model

## 4.3 Task 3: Training Pipeline (20%)

- Implement the training loop with teacher forcing
- Define appropriate loss function (typically cross-entropy)
- Implement validation during training
- Add early stopping and learning rate scheduling
- Save and load model checkpoints
- Plot training and validation curves

### 4.4 Task 4: Inference and Evaluation (20%)

- Implement caption generation for new images
- Evaluate using standard metrics:
  - BLEU score
  - METEOR (optional)
  - CIDEr (optional)
- Create a function to display images with generated captions

### 4.5 Task 5: Extensions and Analysis (Optional, for Extra Credit)

Choose at least one extension to implement:

**Note**: Optional task gives up to 15% extra credit to your base grade.

- Attention mechanism between encoder and decoder
- Beam search for better caption generation
- Ensemble of models with different CNN backbones
- Fine-tuning the CNN backbone
- Data augmentation techniques specific to this task

Provide a detailed analysis of the following:

- Model performance on different types of images
- Common error patterns in generated captions
- Impact of hyperparameter choices
- Effects of vocabulary size and frequency threshold
- Model performance on different types of images
- Common error patterns in generated captions
- Impact of hyperparameter choices
- Effects of vocabulary size and frequency threshold

## 5 Deliverables

Submit the following:

1. Complete code repository following the provided structure
2. Four completed notebooks:
   - Data exploration and preprocessing
   - Feature extraction
   - Model training
   - Evaluation and visualization

3. Project report (PDF, 5-7 pages) including:
    - Problem description
    - Model architecture details
    - Experimental setup
    - Quantitative and qualitative results
    - Analysis and discussion
    - Conclusion and potential improvements
4. Pre-trained model weights(Saved models)
5. README with instructions for running your code

## 6 Evaluation Criteria

Your assignment will be evaluated based on:

| Criterion | Weight |
|---|---|
| Correctness of implementation | 30% |
| Quality of generated captions | 25% |
| Code organization and documentation | 15% |
| Analysis and report quality | 15% |
| Extensions and creativity | 15% |

## 7 Recommended Resources

- PyTorch documentation: `https://pytorch.org/docs/stable/index.html`
- Original Show and Tell paper: `https://arxiv.org/abs/1411.4555`
- Show, Attend and Tell paper: `https://arxiv.org/abs/1502.03044`
- PyTorch tutorials on sequence models: `https://pytorch.org/tutorials/`

## 8 Timeline

- **Week 1**: Data preparation, exploration, and model architecture design
- **Week 2**: Model implementation and training
- **Week 3**: Evaluation, analysis, and final report preparation

## 9 Submission Instructions

1. Create a private GitHub repository with the provided structure
2. Ensure all code runs without errors
3. Submit the GitHub repository link through the course management system
4. Include a requirements.txt file with all necessary dependencies
5. Submit the project report as a separate PDF file

# 10 FAQ and Common Questions

1. **How much GPU memory will I need for this assignment?**
   The basic implementation with ResNet18 or MobileNetV2 backbones should work with 2-4GB GPU memory. If you encounter memory issues, try reducing the batch size or image dimensions.

2. **Can I use pre-trained word embeddings like GloVe or Word2Vec?**
   Yes, you can incorporate pre-trained embeddings, but make sure to explain your approach and any modifications in your report.

3. **Is it necessary to implement the attention mechanism?**
   No, a basic encoder-decoder model is sufficient for the main requirements. Attention is part of the optional Task 5 for extra credit.

4. **How should I handle rare words in captions?**
   Consider implementing a minimum word frequency threshold when building your vocabulary, replacing rare words with an ¡UNK¿ token.

5. **What if my captions don't make grammatical sense?**
   This is common in early iterations. Try increasing the model capacity, training longer, or implementing beam search instead of greedy decoding.

# 11 Questions for Student Experiments

As you work through this assignment, consider exploring these questions in your report:

1. How does the choice of CNN backbone (ResNet18 vs. MobileNetV2 vs. others) affect both training time and caption quality?

2. What impact does the embedding dimension size have on model performance? Try at least two different sizes and analyze the results.

3. Compare the results of greedy decoding vs. beam search (if implemented). Which produces more natural-sounding captions?

4. How does the vocabulary size threshold affect your model's ability to generate accurate captions?

5. What types of images does your model perform well on? What types of images cause difficulty?

6. Analyze the most common errors in your generated captions. Are they primarily semantic errors, grammatical errors, or missing key objects?

# A Technical Concepts

## A.1 Tokenization

Tokenization is the process of converting text into tokens (words, subwords, or characters) that can be processed by neural networks. For image captioning:

- **Word-level tokenization**: Each word becomes a token, requiring a vocabulary of all possible words.

- **Implementation**: Use NLTK, spaCy, or simple string splitting with regex cleaning.
- **Special tokens**: Include ¡START¿, ¡END¿, ¡PAD¿, and ¡UNK¿ tokens to handle sequence beginnings, endings, padding, and unknown words.

Example code:

```
def tokenize_caption(caption):
    # Convert to lowercase and remove punctuation
    caption = caption.lower().replace("[^A-Za-z0-9 ]", "")
    # Add start and end tokens
    caption = "startseq " + caption + " endseq"
    return caption.split()
```

### A.2 Word Embedding Layer

The embedding layer transforms token indices into dense vectors:

- **Purpose**: Converts sparse one-hot word representations into dense, semantically meaningful vectors.
- **Implementation**: PyTorch's nn.Embedding layer maps vocabulary indices to vectors of specified dimension.
- **Dimensions**: Common embedding sizes range from 128 to 512 dimensions, depending on vocabulary size.

Example code:

```
# In model definition
self.embedding = nn.Embedding(vocab_size, embedding_dim)

# In forward pass
embedded = self.embedding(caption_indices)
```

### A.3 Linear Projection to Vocabulary Size

The final layer projects LSTM outputs to vocabulary space:

- **Purpose**: Transforms the LSTM hidden state (typically 256-1024 dimensions) to vocabulary size dimensions.
- **Implementation**: A linear layer followed by softmax activation.
- **Output**: Probability distribution over all words in the vocabulary.

Example code:

```
# In model definition
self.output_projection = nn.Linear(hidden_size, vocab_size)

# In forward pass
scores = self.output_projection(lstm_output)
probabilities = F.softmax(scores, dim=1)
```

### A.4 Vocabulary Extraction

Building a vocabulary from caption data:

- **Process**: Collect all unique words from the training captions.
- **Frequency threshold**: Filter out rare words (appear less than N times).
- **Mapping**: Create mappings from words to indices and vice versa.

Example code:

```
from collections import Counter

def build_vocabulary(captions, threshold=5):
    word_counts = Counter()

    # Count word frequencies
    for caption in captions:
        word_counts.update(caption.split())

    # Filter by threshold and create mappings
    vocab = {'<PAD>': 0, '<START>': 1, '<END>': 2, '<UNK>': 3}
    idx = 4

    for word, count in word_counts.items():
        if count >= threshold:
            vocab[word] = idx
            idx += 1

    return vocab
```

### A.5 Teacher Forcing

Teacher forcing is a training technique for sequence generation models:

- **Concept**: During training, use the ground truth previous word as input at each decoder step, rather than the model's own prediction.
- **Advantage**: Faster convergence and stability during training.
- **Implementation**: Use ground truth captions during training, but generated words during inference.

Example code:

```
# Training (with teacher forcing)
for t in range(max_length - 1):
    output, hidden = decoder(inputs[:, t], hidden)
    loss += criterion(output, targets[:, t])

# Inference (without teacher forcing)
input_word = torch.ones(batch_size, 1) * word_to_idx['<START>']
for t in range(max_length):
    output, hidden = decoder(input_word, hidden)
    predicted = output.argmax(dim=1, keepdim=True)
    input_word = predicted  # Use model's own prediction
```

### A.6 Attention Mechanism in RNNs/LSTMs

Attention allows the decoder to focus on different parts of the input image:

- **Concept**: Instead of using a single context vector, attention computes a weighted sum of all encoder outputs at each decoding step.
- **Implementation**: Calculate similarity scores between the decoder state and encoder outputs, normalize with softmax, and use as weights.
- **Benefits**: Improved caption quality, interpretability via attention maps.

Basic attention equation: $\text{attention}(query, keys) = \text{softmax}(query \cdot keys^T / \sqrt{d_k})$

### A.7 BLEU Score

BLEU (Bilingual Evaluation Understudy) is a metric for evaluating generated text:

- **Concept**: Measures overlap of n-grams between generated captions and reference captions.
- **Calculation**: Precision of n-grams, modified with brevity penalty.
- **Implementation**: Use NLTK's BLEU score implementation.
- **Interpretation**: Higher is better; scores typically range from 0 to 1.

Example code:

```
from nltk.translate.bleu_score import sentence_bleu

reference = [['a', 'cat', 'sitting', 'on', 'a', 'mat']]
candidate = ['a', 'cat', 'on', 'a', 'mat']
score = sentence_bleu(reference, candidate)
```

### A.8 Greedy Search vs. Beam Search

Caption generation requires decoding strategies to convert model outputs (probability distributions) into word sequences:

- **Greedy Search**: At each step, select the single highest probability word.
- **Beam Search**: Maintain k most likely sequences and expand each, providing better caption quality at the cost of computation.
- **Trade-off**: Greedy is faster but often produces suboptimal captions; beam search offers better results but requires more computation.

Example code for greedy search:

```
def greedy_search(model, image_features, max_length, start_token):
    # Initialize with start token
    current_token = start_token
    generated_caption = [current_token]

    # Generate one word at a time
    for _ in range(max_length):
```

```
        # Get model predictions
        outputs = model(image_features, generated_caption)
        # Select highest probability token
        current_token = outputs.argmax(dim=1)
        # Add to generated sequence
        generated_caption.append(current_token)
        # Stop if end token is generated
        if current_token == end_token:
            break

    return generated_caption
```

Example code for beam search:

```
def beam_search(model, image_features, max_length, start_token, beam_size=3):
    # Initialize with start token
    sequences = [([start_token], 0.0)]

    # Generate sequence iteratively
    for _ in range(max_length):
        all_candidates = []

        # Expand each current sequence
        for seq, score in sequences:
            if seq[-1] == end_token:
                # Keep completed sequences
                all_candidates.append((seq, score))
                continue

            # Get predictions for next token
            outputs = model(image_features, seq)
            probs = F.log_softmax(outputs, dim=1)

            # Get top k candidates for each sequence
            topk_probs, topk_indices = probs.topk(beam_size)

            for i in range(beam_size):
                # Create new candidate with additional token
                candidate = seq + [topk_indices[i].item()]
                candidate_score = score + topk_probs[i].item()
                all_candidates.append((candidate, candidate_score))

        # Select top k candidates overall
        sequences = sorted(all_candidates, key=lambda x: x[1], reverse=True)[:beam_size]

        # Early stopping if all sequences end with EOS
        if all(s[-1] == end_token for s, _ in sequences):
            break

    return sequences[0][0]  # Return highest scoring sequence
```

### A.9 Temperature Sampling

Temperature controls randomness in text generation:

- **Mathematical formulation**: $P(w_i) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$ where $T$ is temperature parameter
- **Low temperature** ($T < 1.0$): More deterministic outputs, higher confidence predictions are strongly favored
- **High temperature** ($T > 1.0$): More diverse and random outputs, probability distribution becomes more uniform
- $T = 1.0$: Standard softmax behavior, no adjustment to the probability distribution

Example code:

```
def generate_with_temperature(model, image_features, max_length, start_token, temperature=1
    # Initialize with start token
    current_token = start_token
    generated_caption = [current_token]

    for _ in range(max_length):
        # Get model predictions
        outputs = model(image_features, generated_caption)

        # Apply temperature scaling
        scaled_outputs = outputs / temperature

        # Convert to probabilities
        probs = F.softmax(scaled_outputs, dim=1)

        # Sample from the distribution
        current_token = torch.multinomial(probs, 1).item()

        # Add to generated sequence
        generated_caption.append(current_token)

        # Stop if end token is generated
        if current_token == end_token:
            break

    return generated_caption
```

Temperature applications:

- $T = 0.7$: Often provides a good balance between quality and diversity
- $T = 0.5$: More focused and conservative captions
- $T = 1.2$: More creative but potentially less accurate captions

*Best of luck with your implementation!*
♡