Deep Learning Image Captioning: A Comprehensive Code Walkthrough

Introduction and Architecture Overview

This code implements a state-of-the-art image captioning system using a **CNN-Transformer hybrid architecture**. The intuition is simple yet powerful: we need to "see" the image (CNN) and then "describe" what we see in natural language (Transformer).

The overall architecture follows the **encoder-decoder paradigm**:

- **CNN Encoder**: Extracts visual features from images
- **Transformer Encoder**: Processes and refines these visual features
- **Transformer Decoder**: Generates captions word by word, conditioned on the visual features

---

1. Setup and Dependencies

```python
import warnings
warnings.filterwarnings("ignore", message="You are using a softmax over axis 3...")
```

Why this warning filter? During training, Keras sometimes warns about softmax operations on tensors with size 1 in certain dimensions. This is typically harmless in our context, so we suppress it to keep the output clean.

```python
!pip install nltk
!git clone https://github.com/tylin/coco-caption.git
!cd coco-caption/pycocoevalcap && pip install pycocoevalcap pycocotools
```

Why these specific packages?

- **NLTK**: Natural Language Toolkit for text preprocessing
- **pycocoevalcap**: The gold standard for evaluating image captioning models - it implements BLEU, METEOR, ROUGE, CIDEr, and SPICE metrics
- **pycocotools**: Required dependency for handling COCO dataset format

python

```python
import nltk
nltk.download('punkt')  # For sentence tokenization
nltk.download('wordnet')  # For semantic similarity in METEOR metric
```

Professor's Note: NLTK downloads are essential because some evaluation metrics (like METEOR) need linguistic resources to compute semantic similarity between words.

---

2. Model Hyperparameters

python

```python
IMAGE_SIZE = (299, 299)  # Input image dimensions
VOCAB_SIZE = 10000       # Maximum vocabulary size
SEQ_LENGTH = 25          # Maximum caption length
EMBED_DIM = 512          # Embedding dimension
FF_DIM = 512            # Feed-forward network dimension
BATCH_SIZE = 64
EPOCHS = 30
```

Design Rationale:

- **299x299**: Chosen to match EfficientNet's expected input size
- **VOCAB_SIZE = 10000**: Large enough to capture rich vocabulary, small enough to be computationally manageable
- **SEQ_LENGTH = 25**: Long enough for detailed captions, short enough to avoid vanishing gradients
- **EMBED_DIM = 512**: Standard dimension that balances expressiveness with computational efficiency

---

3. Evaluation Metrics Function

python

```python
def evaluate_metrics(generated_captions, reference_captions_dict):
```

This function is crucial for **scientific validation**. It implements the standard evaluation protocol used in major image captioning papers.

Key Steps:

1. **Format Conversion**: Converts our data to COCO evaluation format
2. **Temporary Files**: Creates JSON files that the evaluation tools expect

3. **Metric Calculation**: Computes BLEU, METEOR, ROUGE, CIDEr, and SPICE scores

Why COCO format? The computer vision community standardized on COCO's evaluation protocol. Using it ensures our results are comparable to published research.

Professor's Insight: This is how you make your research reproducible and comparable. Always use standard evaluation protocols!

---

4. Dataset Loading and Preprocessing

python

```python
def load_captions_data(filename):
    """Loads captions and maps them to images"""
```

Critical Design Decisions:

python

```python
if len(tokens) < 5 or len(tokens) > SEQ_LENGTH:
    images_to_skip.add(img_name)
    continue
```

Why filter by length?

- **Too short** (< 5): Likely not descriptive enough
- **Too long** (> SEQ_LENGTH): Would be truncated anyway, might lose important information

python

```python
caption = "<start> " + caption.strip() + " <end>"
```

Why special tokens?

- <start>: Tells the decoder when to begin generation
- <end>: Tells the decoder when to stop generation
- These are **essential** for sequence-to-sequence models to learn proper boundaries

---

5. Text Vectorization

```python
def custom_standardization(input_string):
    lowercase = tf.strings.lower(input_string)
    return tf.strings.regex_replace(lowercase, "[%s]" % re.escape(strip_chars), "")

strip_chars = "!\"#$%&'()*+,-./:;<=>?@[\]^_`{|}~"
strip_chars = strip_chars.replace("<", "")  # Keep < for <start>
strip_chars = strip_chars.replace(">", "")  # Keep > for <end>
```

Preprocessing Philosophy:

- **Lowercase**: Reduces vocabulary size ("Cat" and "cat" become the same)
- **Remove punctuation**: Except our special tokens
- **Why keep < and >?** We need them for our `<start>` and `<end>` tokens

Professor's Note: Text preprocessing is an art. Too aggressive and you lose meaning; too lenient and you get a sparse vocabulary.

---

6. Data Pipeline

```python
def make_dataset(images, captions):
    dataset = tf.data.Dataset.from_tensor_slices((images, captions))
    dataset = dataset.shuffle(BATCH_SIZE * 8)  # Shuffle buffer
    dataset = dataset.map(process_input, num_parallel_calls=AUTOTUNE)
    dataset = dataset.batch(BATCH_SIZE).prefetch(AUTOTUNE)
```

Why this specific pipeline?

- **Shuffle buffer = BATCH_SIZE * 8**: Good balance between randomness and memory usage
- **num_parallel_calls=AUTOTUNE**: Let TensorFlow optimize parallelization
- **prefetch(AUTOTUNE)**: Overlap data loading with model training for efficiency

Performance Insight: This pipeline can be the difference between GPU utilization of 60% vs 95%!

---

7. CNN Feature Extractor

python

```python
def get_cnn_model():
    base_model = efficientnet_v2.EfficientNetV2S(
        input_shape=(*IMAGE_SIZE, 3),
        include_top=False,  # Remove classification layer
        weights="imagenet", # Pre-trained weights
    )
```

Why EfficientNet?

- **State-of-the-art accuracy** with reasonable computational cost
- **include_top=False**: We don't want ImageNet classification; we want feature extraction
- **Pre-trained weights**: Transfer learning from ImageNet gives us a huge head start

python

```python
base_model.trainable = False
for layer in base_model.layers[-20:]:
    if not isinstance(layer, layers.BatchNormalization):
        layer.trainable = True
```

Fine-tuning Strategy:

- **Initial freeze**: Prevent catastrophic forgetting of pre-trained features
- **Unfreeze last 20 layers**: Allow adaptation to our specific task
- **Keep BatchNorm frozen**: BatchNorm statistics are dataset-specific; freezing prevents instability

python

```python
base_model_out = layers.GlobalAveragePooling2D()(base_model_out)
base_model_out = layers.Reshape((1, -1))(base_model_out)
```

Why GlobalAveragePooling2D?

- Reduces spatial dimensions while preserving feature information
- **Reshape to (1, features)**: Creates a "sequence" of length 1 for the Transformer encoder

---

8. Transformer Encoder

python

```python
class TransformerEncoderBlock(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        self.attention_1 = layers.MultiHeadAttention(num_heads=num_heads, key_dim=embe
```

Architecture Insights:

- **Multi-head attention**: Allows the model to attend to different aspects of the visual features
- **num_heads=2**: Fewer heads than decoder because we're processing a single image feature vector
- **dropout=0.1**: Regularization to prevent overfitting

python

```python
def call(self, inputs, training, mask=None):
    projected_inputs = self.input_projection(inputs)
    normalized_inputs = self.layernorm_1(projected_inputs)
    attention_output = self.attention_1(query=normalized_inputs, value=normalized_inpu
```

Self-Attention Logic:

- **Query = Key = Value**: This is self-attention
- **Pre-layer normalization**: Modern Transformer architecture (more stable than post-norm)
- **Residual connections**: Help with gradient flow during training

---

9. Positional Embedding

python

```python
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, vocab_size, embed_dim, **kwargs):
        self.token_embeddings = layers.Embedding(input_dim=vocab_size, output_dim=embe
        self.position_embeddings = layers.Embedding(input_dim=sequence_length, output_
        self.embed_scale = tf.math.sqrt(tf.cast(embed_dim, tf.float32))
```

Why positional embeddings?

- **Transformers have no inherent sense of order** (unlike RNNs/LSTMs)
- **Position embeddings** tell the model where each word is in the sequence
- **embed_scale**: Scaling factor from the original Transformer paper to prevent embeddings from being too small relative to positional encodings

---

10. Transformer Decoder

python

```python
class TransformerDecoderBlock(layers.Layer):
    def __init__(self, embed_dim, ff_dim, num_heads, **kwargs):
        self.attention_1 = layers.MultiHeadAttention(num_heads=num_heads, key_dim=embe
        self.attention_2 = layers.MultiHeadAttention(num_heads=num_heads, key_dim=embe
```

Two Types of Attention:

1. **Self-attention (attention_1)**: Words in the caption attend to each other
2. **Cross-attention (attention_2)**: Words in the caption attend to image features

python

```python
def call(self, inputs, encoder_outputs, training, mask=None):
    causal_mask = self.get_causal_attention_mask(inputs)
```

Causal Masking: Prevents the model from "cheating" by looking at future words during training. Word at position `i` can only see words at positions `0` to `i-1`.

python

```python
def get_causal_attention_mask(self, inputs):
    i = tf.range(sequence_length)[:, tf.newaxis]
    j = tf.range(sequence_length)
    mask = tf.cast(i >= j, dtype="int32")
```

Mask Logic: Creates a lower triangular matrix where `mask[i,j] = 1` if `i >= j`, else `0`.

---

11. Complete Model Architecture

python

```python
class ImageCaptioningModel(keras.Model):
    def __init__(self, cnn_model, encoder, decoder, num_captions_per_image=5, image_au
```

Why 5 captions per image?

- Flickr8K provides 5 reference captions per image
- During training, we use all 5 to maximize learning signal

python

```
def train_step(self, batch_data):
    for i in range(self.num_captions_per_image):
        with tf.GradientTape() as tape:
            loss, acc = self._compute_caption_loss_and_acc(img_embed, batch_seq[:, i,
        grads = tape.gradient(loss, train_vars)
        self.optimizer.apply_gradients(zip(grads, train_vars))
```

Training Strategy:

- **Separate gradient tape for each caption**: Allows for more stable gradients
- **Only train encoder and decoder**: CNN is mostly frozen (except last 20 layers)

---

12. Learning Rate Schedule

python

```
class LRSchedule(keras.optimizers.schedules.LearningRateSchedule):
    def __call__(self, step):
        warmup_progress = global_step / warmup_steps
        warmup_learning_rate = self.post_warmup_learning_rate * warmup_progress
        return tf.cond(global_step < warmup_steps, lambda: warmup_learning_rate, lambd
```

Why warmup?

- **Large models can be unstable** with high initial learning rates
- **Gradual increase** allows the model to "settle" into a good optimization landscape
- **Standard practice** in modern deep learning, especially for Transformers

---

13. Inference and Evaluation

python

```
def run_full_evaluation(model, val_dataset, vectorization, max_decoded_sentence_length
    for t in range(max_decoded_sentence_length - 1):
        tokenized_caption = vectorization([decoded_caption])[:, :-1]
        predictions = model.decoder(tokenized_caption, encoder_outputs, training=False
        sampled_token_index = np.argmax(predictions[0, t, :])
```

Greedy Decoding Process:

1. Start with `<start>` token
2. At each step, predict the most likely next word

3. Add predicted word to the sequence
4. Repeat until `<end>` token or max length

Why greedy and not beam search?

- **Simplicity**: Easier to implement and debug
- **Speed**: Much faster than beam search
- **Good enough**: For educational purposes, greedy often works well

---

Key Design Principles and Intuitions

1. Modular Architecture

Each component (CNN, Transformer Encoder, Transformer Decoder) has a specific responsibility. This makes the code maintainable and allows for easy experimentation.

2. Transfer Learning

We leverage pre-trained EfficientNet for visual feature extraction. This is crucial because training a CNN from scratch would require much more data and compute.

3. Attention Mechanisms

- **Self-attention** helps the model understand relationships between words
- **Cross-attention** helps the model align words with visual features
- **Causal masking** ensures proper autoregressive generation

4. Regularization

Multiple techniques prevent overfitting:

- Dropout in attention layers
- Layer normalization
- Early stopping
- Frozen BatchNorm layers

5. Evaluation Rigor

Using standard metrics (BLEU, CIDEr, etc.) ensures results are comparable to published research.

Common Pitfalls and Why This Code Avoids Them

1. **Vocabulary Explosion**: Limited to 10K words with proper text cleaning
2. **Gradient Instability**: Learning rate warmup and gradient clipping

3. **Overfitting**: Multiple regularization techniques
4. **Inefficient Data Loading**: Optimized tf.data pipeline with prefetching
5. **Evaluation Inconsistency**: Standard COCO evaluation protocol

---

Extension Ideas

1. **Beam Search**: Replace greedy decoding for better caption quality
2. **Attention Visualization**: Show which parts of the image the model focuses on
3. **Different Architectures**: Try Vision Transformer instead of CNN
4. **Advanced Techniques**: Implement techniques like self-critical training

This implementation represents a solid foundation for image captioning research and demonstrates best practices in modern deep learning!