Image Caption Generation

Presented by: Shivangi Bharat Modi (20093)
Shruti Dilip Kavishwar (20022)

Content

- Introduction
- Approach
  - Get Dataset
  - Prepare Image Data
  - Prepare Text Data
  - Load Data
  - Encode Text Data
  - Define Model
  - Fit Model
  - Evaluate Model
  - Generate Captions
- Conclusions

Introduction

- **What is Image Caption Generation?**

- The process of automatically generating descriptive textual captions from images.
- Combines the power of computer vision (understanding images) and natural language processing (NLP) (generating text).

- **Why is Image Caption Generation Important?**

- Assistive technology for visually impaired users
- Content moderation - Automatically detects and describes image content on social media.
- Enhancing social media platforms

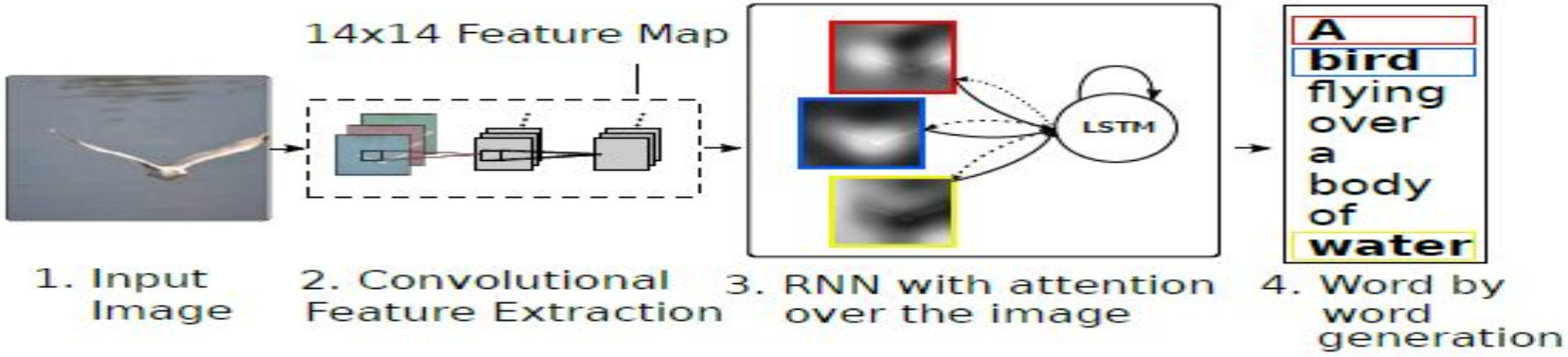
- **How It Works:**

- Input: An image.
- Process: Extract image features (using CNN) → Understand context (using Transformer) → Generate text.
- Output: A coherent and relevant caption.

Related Work

- **Early methods**: Implement a caption generation model using a CNN to condition a LSTM language model

  - CNN + LSTM architectures (Paper - Show and Tell paper)

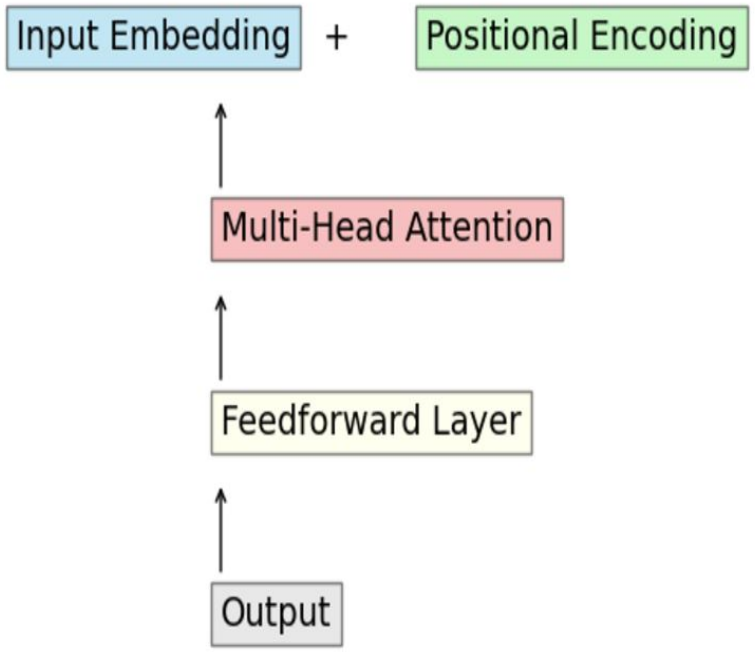


- **Limitations:**

- Long-term Dependencies: LSTM struggles with maintaining information over long sequences.
- Training Complexity: Sequential processing in LSTM makes training slower.
- Fixed Context: Lacks the ability to focus dynamically on different parts of the image during caption generation.

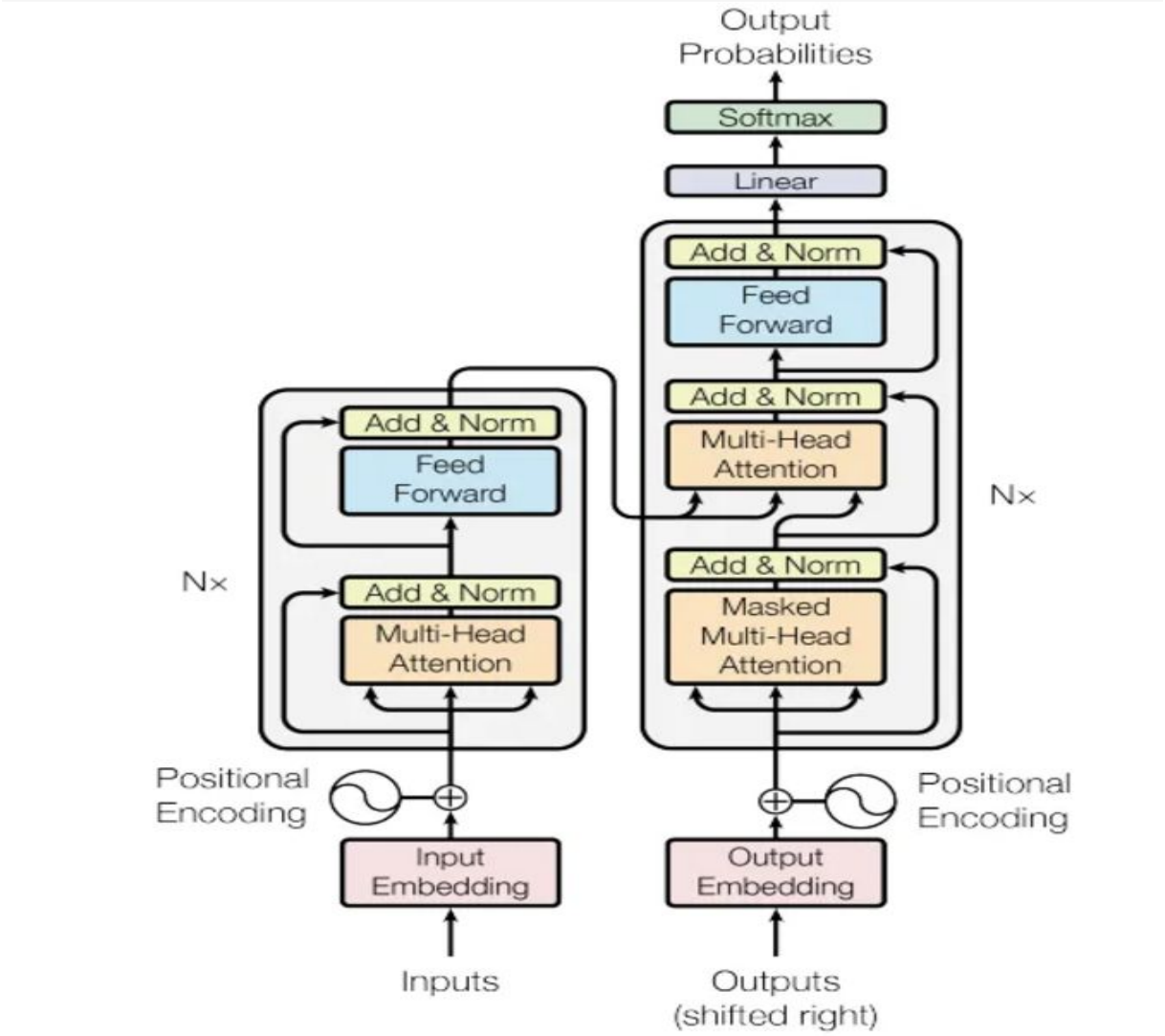
Related Work

- **Early Method:** Attention Is All You Need
- **Approach**:
  - Use ResNet50 to extract visual features.
  - Introduced the Transformer architecture, which relies entirely on self-attention mechanisms.
  - Eliminates the need for recurrence (LSTMs) by using multi-head attention and positional encoding to capture relationships between words.



Related Work

- **Transformer-model architecture**



Related Work (Cont..)

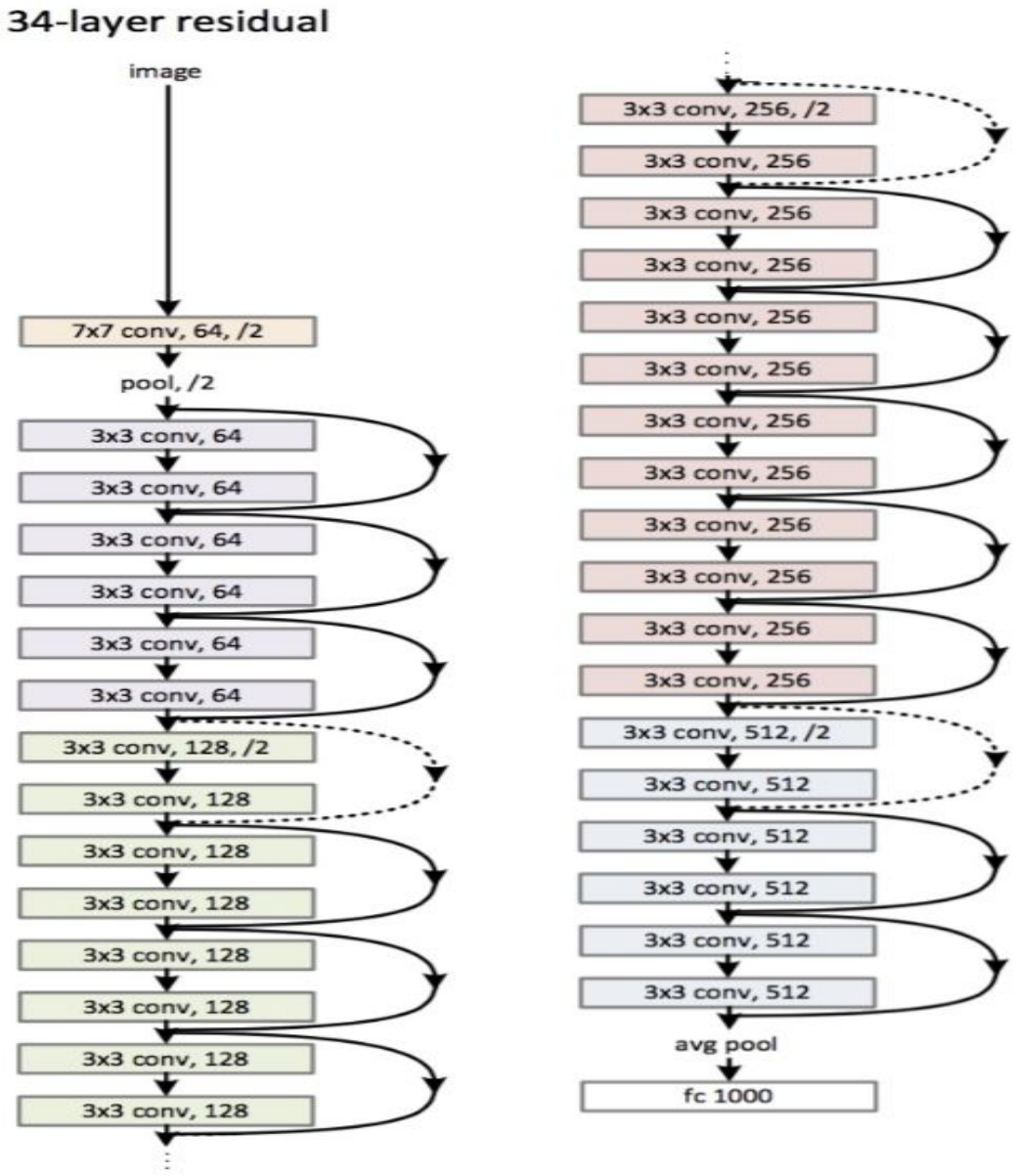
- **Paper2:** Attention Is All You Need
- **Innovations/Advantages**:

- Parallel Processing: Faster training compared to LSTM-based models due to parallelization.
- Attention Mechanism: Enables the model to dynamically focus on different parts of the input (image features or words) during caption generation.
- Handling Long Sequences: Effectively captures long-term dependencies in text data.

- **Challenges**:

- Computational Complexity: Requires significant memory and computation, especially for large datasets.
- Data Requirements: Transformers need a large amount of data to generalize effectively.

ResNet-50 Architecture



MobileNet Architecture

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Dataset

Flickr8 Dataset:

- 8000 images
- 5 captions for each image

Larger datasets:

- Flickr30k
- MSCOCO

Caption1: closeup of white dog that is laying its head on its paws
Caption 2: large white dog lying on the floor
Caption 3: white dog has its head on the ground
Caption 4: white dog is resting its head on tiled floor with its eyes open
Caption 5: white dog rests its head on the patio bricks

Caption 1: little tan dog with large ears running through the grass
Caption 2: playful dog is running through the grass
Caption 3: small dogs ears stick up as it runs in the grass
Caption 4: the small dog is running across the lawn
Caption 5: this is small beige dog running through grassy field

Implementation (CNN+LSTM)

Step -1: Data Loading

- Dataset: The Flickr8k dataset is loaded, which contains images with corresponding captions.

```
[ ] def get_data(path = 'flickr8k'):
        path = pathlib.Path('flickr8k')

        tf.keras.utils.get_file(origin = 'https://github.com/jbrownlee/Datasets/releases/download/Flickr8k/Flickr8k_
                                cache_dir = '.',
                                cache_subdir = path,
                                extract = True)

        tf.keras.utils.get_file(origin = 'https://github.com/jbrownlee/Datasets/releases/download/Flickr8k/Flickr8k_
                                cache_dir = '.',
                                cache_subdir = path,
                                extract = True)
```

```
[ ] def get_dataset(path = 'flickr8k'):
        path = pathlib.Path('flickr8k')
        captions = (path/'Flickr8k.token.txt').read_text().splitlines()
        captions = [cap.split('\t') for cap in captions]
        captions = [(img_path.split('#')[0], cap) for (img_path, cap) in captions]
        cap_dict = collections.defaultdict(list)
        for img_path, cap in captions:
            cap_dict[img_path].append(cap)
```

Implementation (CNN+LSTM)

Step-2: Preprocessing image with augmentation to ensure diverse variations of same image

- Images are resized to (224,224,3) to ensure all images have same dimension
- Randomly flip the images horizontally
- Adjust the brightness of image by random factor 0.2
- Rotate the image by random angle between -0.2 to 0.2 radians
- Normalization by dividing pixel values by 255 to scale in range [0,1]

```python
# Step 2: Preprocess images with augmentation
def preprocess_image(img_path, augment=False):
    img = tf.io.read_file(img_path)
    img = tf.io.decode_jpeg(img, channels=3)
    img = tf.image.resize(img, IMG_SHAPE[:2])
    if augment:
        img = tf.image.random_flip_left_right(img)
        img = tf.image.random_brightness(img, max_delta=0.2)

        # Custom random rotation
        angle = tf.random.uniform([], -0.2, 0.2)  # Random angle in radians
        img = tfa_custom_rotate(img, angle)
    return img / 255.0
```

Implementation (CNN+LSTM)

Step-2: Preprocessing image with augmentation to ensure diverse variations of same image

- This function is particularly useful for **data augmentation**, where images are randomly rotated to make a machine learning model more robust to variations.

```python
def tfa_custom_rotate(image, angle):
    # Compute the rotation matrix
    rotation_matrix = tf.convert_to_tensor([
        [tf.cos(angle), -tf.sin(angle), 0],
        [tf.sin(angle), tf.cos(angle), 0],
    ])
    # Flatten the matrix to [1 x 8] with additional zeros for affine transform
    flat_matrix = tf.reshape(rotation_matrix, [-1])
    flat_matrix = tf.concat([flat_matrix, tf.zeros([2])], axis=0)  # Add padding to

    # Apply the rotation
    image = tf.expand_dims(image, axis=0)  # Add batch dimension
    rotated_image = tf.raw_ops.ImageProjectiveTransformV2(
        images=image,
        transforms=tf.expand_dims(flat_matrix, axis=0),
        output_shape=tf.shape(image)[1:3],
        interpolation="BILINEAR"
    )
    return tf.squeeze(rotated_image, axis=0)  # Remove batch dimension
```

Implementation (CNN+LSTM)

- **Step-3: Standardize and Tokenize captions**

 Standardize function ensure all captions are in the same format

 It implements :

  - Lowercasing
  - Removing punctuations
  - Adding [START] and [END] tokens to mark each captions start and end.

```
# Step 3: Preprocess captions
def standardize(text):
    text = tf.strings.lower(text)
    text = tf.strings.regex_replace(text, f"[{re.escape(string.punctuation)}]", "")
    text = tf.strings.join(["[START]", text, "[END]"], separator=" ")
    return text
```

Implementation (CNN+LSTM)

- **Step-3: Standardize and Tokenize captions**

- The tokenize_captions function converts text captions into tokenized sequences using TensorFlow's TextVectorization layer.
- Process:
  - Initialize TextVectorization layer has max_tokens=5000 and the sequence length is fixed, padded and truncated.
  - Maps each word to unique index using ‘adapt’

```
def tokenize_captions(captions):
    vectorizer = tf.keras.layers.TextVectorization(
        max_tokens=VOCAB_SIZE,
        output_sequence_length=MAX_SEQ_LEN,
        standardize=standardize
    )
    vectorizer.adapt(captions)
    return vectorizer
```

Implementation (CNN+LSTM)

- **Step-4: Encoder Block**

 Extract meaningful features from input images

Input image	• MobileNetV3Small • Global Ave Pooling spatial features to single feature vector	Dense Layer: Lower dimension embeddings	Feature embeddings

```python
# Step 5: Create Encoder-Decoder Model
class Encoder(tf.keras.layers.Layer):
    def __init__(self, embedding_dim):
        super(Encoder, self).__init__()
        self.feature_extractor = tf.keras.applications.MobileNetV3Small(
            input_shape=IMG_SHAPE, include_top=False, pooling="avg"
        )
        self.dense = tf.keras.layers.Dense(embedding_dim, activation="relu")

    def call(self, images):
        features = self.feature_extractor(images)
        # print(f"Encoder Output Shape: {features.shape}")
        return self.dense(features)
```

Implementation (CNN+LSTM)

- **Step-5: Decoder Block**

Initialization (__init__)

- **Embedding Layer**:
  - Converts input tokens (e.g., words) into dense vector embeddings of size embedding_dim.
  - This helps the model understand word relationships.
- **LSTM Layer**:
  - A recurrent layer with lstm_units neurons to process sequential data (e.g., captions).
  - **Configuration**:
    - return_sequences=True: Outputs the sequence of hidden states.
    - return_state=True: Also returns the final hidden and cell states for use in subsequent steps.
    - dropout=0.2: Adds dropout for regularization.
- **Dense Layer**:
  - Fully connected layer to map the LSTM output to vocabulary size (vocab_size), producing logits for the next token prediction.

Implementation (CNN+LSTM)

- **Step-5: Decoder Block**

Forward Pass (call)

➢ **Inputs**:
  - captions: Sequence of tokenized words.
  - features: Encoded features from the image (output of the encoder).
  - hidden_state: Initial hidden and cell states for the LSTM.

➢ **Steps**:
  - **Embed Captions**: Converts captions into embeddings using the Embedding layer.
  - **Add Features**: Concatenates the image features as the first timestep in the sequence.
  - **LSTM Processing**: Feeds the sequence into the LSTM, producing the output sequence and updated hidden states.
  - **Token Prediction**:
    - Uses the Dense layer to compute logits for the last token in the sequence.
    - Logits represent the probabilities of each word in the vocabulary.

Implementation (CNN+LSTM)

- **Step-5: Decoder Block**

- **Sequential Data Processing**: The LSTM allows the model to learn dependencies between words in a sequence.
- **Image-Text Integration**: Combines visual features with text to generate meaningful sequences.
- **Prediction Capability**: Outputs logits that can be converted into words using techniques like beam search or greedy decoding.

```python
class Decoder(tf.keras.Model):
    def __init__(self, embedding_dim, lstm_units, vocab_size):
        super(Decoder, self).__init__()
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
        self.lstm = tf.keras.layers.LSTM(lstm_units, return_sequences=True, return_state=True, dropout=0.2)
        self.dense = tf.keras.layers.Dense(vocab_size)

    def call(self, captions, features, hidden_state):
        embeddings = self.embedding(captions)
        # Only pass the features during the first time step
        embeddings = tf.concat([tf.expand_dims(features, 1), embeddings], axis=1)
        lstm_output, *hidden_state = self.lstm(embeddings, initial_state=hidden_state)
        logits = self.dense(lstm_output[:, -1, :])  # Predict for the last token
        return logits, hidden_state
```

Implementation (CNN+LSTM)

- **Sample output**

 Prints number of images, Number of captions and sample vectorization

```
# Sanity checks
print(f"Number of images: {len(image_paths)}")
print(f"Number of captions: {len(captions_list)}")
print(f"Sample caption after vectorization: {vectorizer(captions_list[:1]).numpy()}")
```

```
Number of images: 8091
Number of captions: 8091
Sample caption after vectorization: [[  9   4  63  87   3   4  94  14  63  18  35   3  29 338 133  94  14  63
   35   3   2 133  94  14  63  87   3   4  25  13]]
```

Implementation (CNN+LSTM)

- **Sample output**

 Prints number of images, Number of captions and sample vectorization

```
# Sanity checks
print(f"Number of images: {len(image_paths)}")
print(f"Number of captions: {len(captions_list)}")
print(f"Sample caption after vectorization: {vectorizer(captions_list[:1]).numpy()}")
```

```
Number of images: 8091
Number of captions: 8091
Sample caption after vectorization: [[  9   4  63  87   3   4  94  14  63  18  35   3  29 338 133  94  14  63
   35   3   2 133  94  14  63  87   3   4  25  13]]
```

Implementation (CNN+LSTM)

- **Sample output**

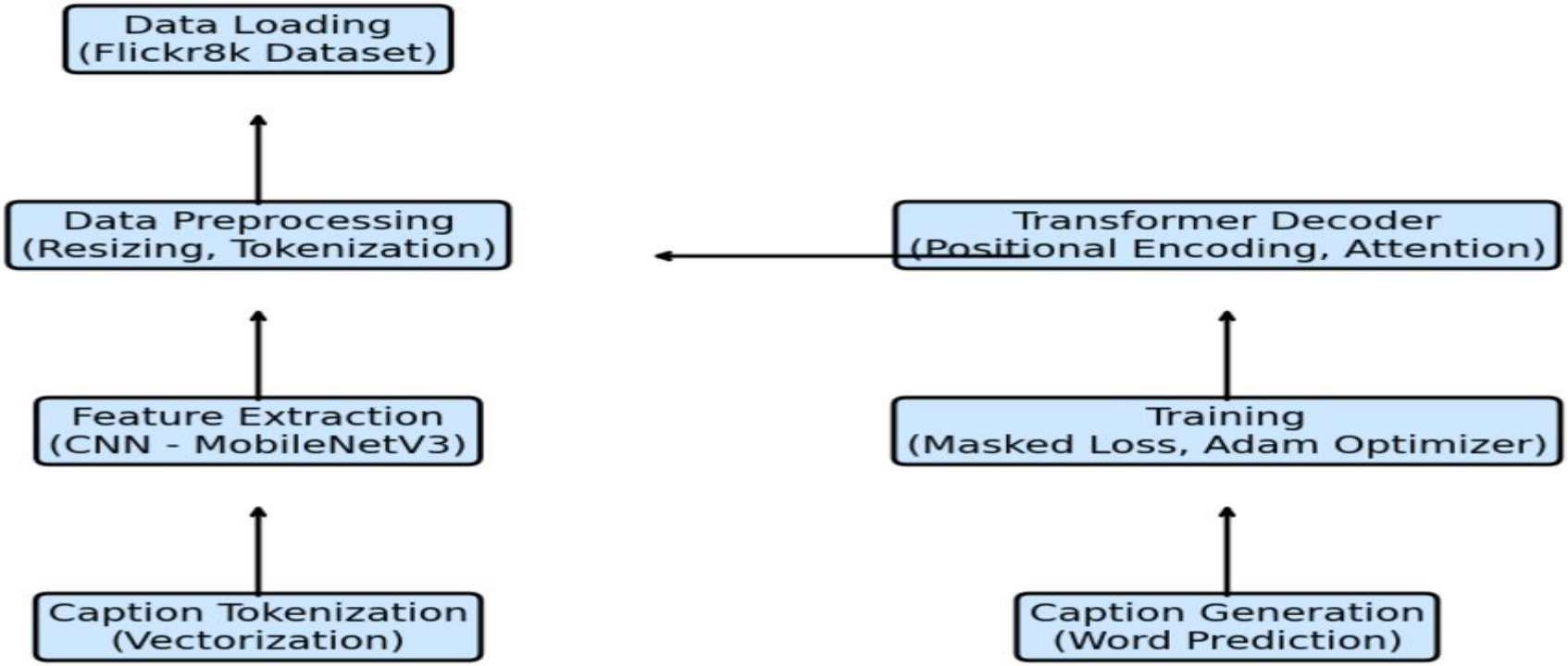
- Generated caption is not that accurate but is quite generic.

```
Predicted token ID: 2, Word: a
```

a black dog is jumping in

Implementation Work

- Develop an improved image caption generation model using a Transformer-based architecture that surpasses existing accuracy benchmarks.
- Hybrid Architecture: Combine CNN and Transformer with residual connections for better feature extraction and text generation.
- Multi-Modal Fusion: Integrate external datasets or pre-trained language models to enrich semantic understanding.



Implementation Work

- **Data Preparation:**
- Dataset: Use an image caption dataset Flickr8k.
- Image Preprocessing: Apply a pre-trained CNN (MobileNet-v3) to extract feature embeddings.
- Normalize and resize images to a consistent format.
- **Model Architecture:**
- Feature Extraction (Encoder): Use a CNN to convert images into a sequence of high-dimensional feature vectors (Mobilenet-v3).
- Transformer Decoder: Process caption tokens with positional encoding to retain sequence order (TextVectorization + Positional Embedding).
- Implement multi-head attention to dynamically focus on different parts of the image while generating text.

Implementation Steps and Code Explanation

- **Step-1: Data Loading and Preprocessing**
- **Step-2: Feature Extraction:**

 MobileNetV3: Used to extract features from images, generating a lower-dimensional representation.

```
[ ] image_shape = (224, 224, 3)
    feature_extractor = tf.keras.applications.MobileNetV3Small(input_shape = image_shape, include_preprocessing = Tr
    feature_extractor.trainable = False
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v3/weights_mobilenet_v3_small_224_1.0_float_no_top_v
4334752/4334752 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

- **Step-3: Caption Tokenization and Vectorization:**

 Text Vectorization: Converts captions to sequences of integers.

```
vocab_size = 5000
vectorizer = tf.keras.layers.TextVectorization(max_tokens = vocab_size,
                                               standardize = standardize,
                                               ragged = True)

vectorizer.adapt(train_raw.map(lambda img_path, cap: cap).unbatch().batch(1024))
```

```
[ ] print(vectorizer.get_vocabulary()[:10])
```

```
['', '[UNK]', 'a', '[START]', '[END]', 'in', 'the', 'on', 'is', 'and']
```

Implementation Steps and Code Explanation

- **Step-3: Caption Tokenization and Vectorization:**

- The vocab also includes other reserved tokens like
- **pad token** : For padding the sequence so that the batch of input sentences that we input to the model is of fixed shape.
- **unk token** : This token is for the unknown words that are not present in the vocab.
- **[START] token** : This token indicates the start of a sentence.
- **[END] token:** This indicates the end of a sentence.
- Using TextVectorization layer in Tensorflow that does the same for us.

Implementation Steps and Code Explanation

- **Step-4: Model Architecture**

 Positional Encoding: Adds information about the position of tokens.

```
class PositionalEmbedding(tf.keras.layers.Layer):
    def __init__(self, vocab_size, d_model):
        super().__init__()
        self.d_model = d_model
        self.embedding = tf.keras.layers.Embedding(vocab_size, d_model, mask_zero = True)
        self.pos_enc = positional_encoding(length = 2048, depth = d_model)

    def compute_mask(self, *args, **kwargs):
        return self.embedding.compute_mask(*args, **kwargs)

    def call(self, x):
        length = tf.shape(x)[1]
        x_emb = self.embedding(x)
        x_pos_enc = self.pos_enc[:, :length, :]

        x_emb *= tf.cast(self.d_model, tf.float32)
        x_emb += x_pos_enc

        return x_emb
```

- **Step-5: Attention Mechanisms**

 Causal Attention: Ensures each token only attends to previous tokens.

 Cross-Attention: Allows the decoder to attend to image features.

Implementation Steps and Code Explanation

- **What is attention ?**

- In simple words, Attention is the probability of a context vector (word in case of sentence , segment of image in case of images) associated with a word.
- Attention vector can be calculated in two ways: hard attention and soft attention.
  - Hard Attention just picks up the context vector that has the maximum attention value
  - Soft attention takes the weighted sum of all the context vectors with their corresponding attention values.

$$e_{ti} = f_{\text{att}}(\mathbf{a}_i, \mathbf{h}_{t-1})$$

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{k=1}^{L} \exp(e_{tk})}.$$

- Use scaled-dot product attention which is basically involves matrix multiplications and is quite faster compared to previous one.

Implementation Steps and Code Explanation

- **Scaled Dot Product Attention**

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Here Q is query vector (corresponding to the words that we want to produce), K is key vector and V is value vector (K, V are corresponding to the context vector)

- Cross Attention: When Q is from a different set of vectors than K and V. For example, in our case, Q is vectors from image after passing through mobilenet model whereas K, V are vectors corresponding to input text to decoder.
- Self Attention: When Q, K, V are from same vectors.
- Causal Self Attention: When Q, K, V are from same vectors but while calculating attention scores, we mask the words that appear after the word for which we are finding attention values. This is done whenever we are generating text so it is in transformer decoder part.

Implementation Steps and Code Explanation

- **Step-6: Decoder and Caption Generation:**

 The decoder generates captions one word at a time, using Transformer layers.

```
class DecoderLayer(tf.keras.layers.Layer):
    def __init__(self, d_model, dff, num_heads, dropout_rate = 0.1):
        super().__init__()
        self.causal_attention = CausalAttention(num_heads = num_heads,
                                                key_dim = d_model,
                                                dropout = dropout_rate)
        self.cross_attention = CrossAttention(num_heads = num_heads,
                                              key_dim = d_model,
                                              dropout = dropout_rate)
        self.ffn = FeedForward(d_model = d_model,
                               dff = dff,
                               dropout_rate = dropout_rate)
        self.last_attention_scores = None

    def call(self, context, x):
        x = self.causal_attention(x)
        x = self.cross_attention(context = context, x = x)
        x = self.ffn(x)
        self.last_attention_scores = self.cross_attention.last_attention_scores

        return x
```

- **Step-7: Training Process**

```
hist = captioner_model.fit(
    train_ds.repeat(),
    steps_per_epoch = 100,
    validation_data = test_ds.repeat(),
    validation_steps = 20,
    epochs = 150,
    callbacks = callbacks
)
```

Implementation Steps and Code Explanation

- For training, keeping stack of 2 decoder layers, embedding dimension and model dimension as 128.
- Training model for 150 epochs.
- **Step-8: Loss and Accuracy Score**

 For calculating loss and accuracy function, we use a mask that masks all the padded tokens so that it does not impact the loss value.

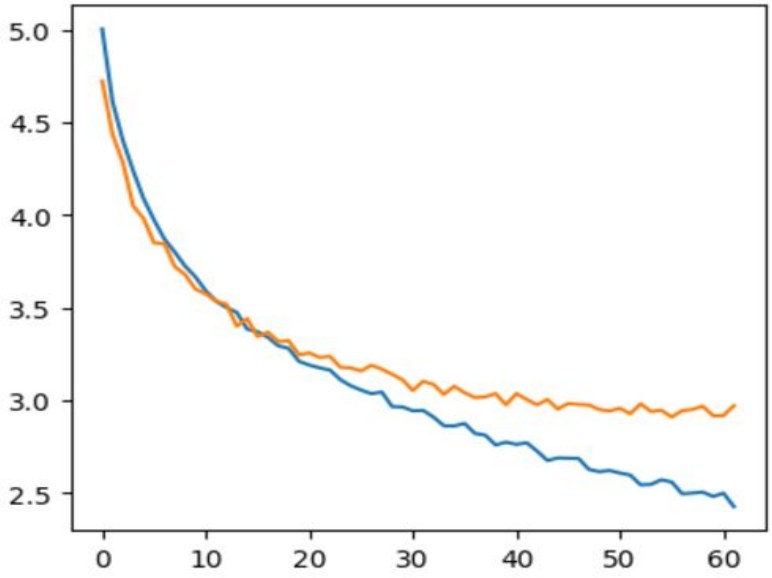
```
def masked_loss(labels, preds):
    loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits = True, reduction = 'none')
    loss = tf.cast(loss_fn(labels, preds), tf.float32)
    mask = ((labels != 0) & (loss < 1e8))
    mask = tf.cast(mask, tf.float32)
    loss *= mask
    return tf.math.reduce_sum(loss) / tf.math.reduce_sum(mask)

def masked_accuracy(labels, preds):
    preds = tf.cast(tf.argmax(preds, axis = -1), tf.float32)
    labels = tf.cast(labels, tf.float32)
    mask = tf.cast(labels != 0, tf.float32)
    acc = tf.cast(preds == labels, tf.float32)
    acc *= mask
    return tf.math.reduce_sum(acc) / tf.math.reduce_sum(mask)
```

Model Results

- Caption Generation:
  - The trained model predicts word sequences to generate captions for new images.

```
[ ] img = load_img(img_path[1].numpy().decode('utf-8'))
    plt.imshow(img/255.0)
    plt.show()
    print(f'Generated Caption: {captioner_model.generate_text(img[tf.newaxis, ...])}')
```

- Evaluation Metrics:
  - BLEU Score: Measures caption quality.
  - BLEU Score: 0.75 (average across test set)
  - Loss and accuracy monitored during training.



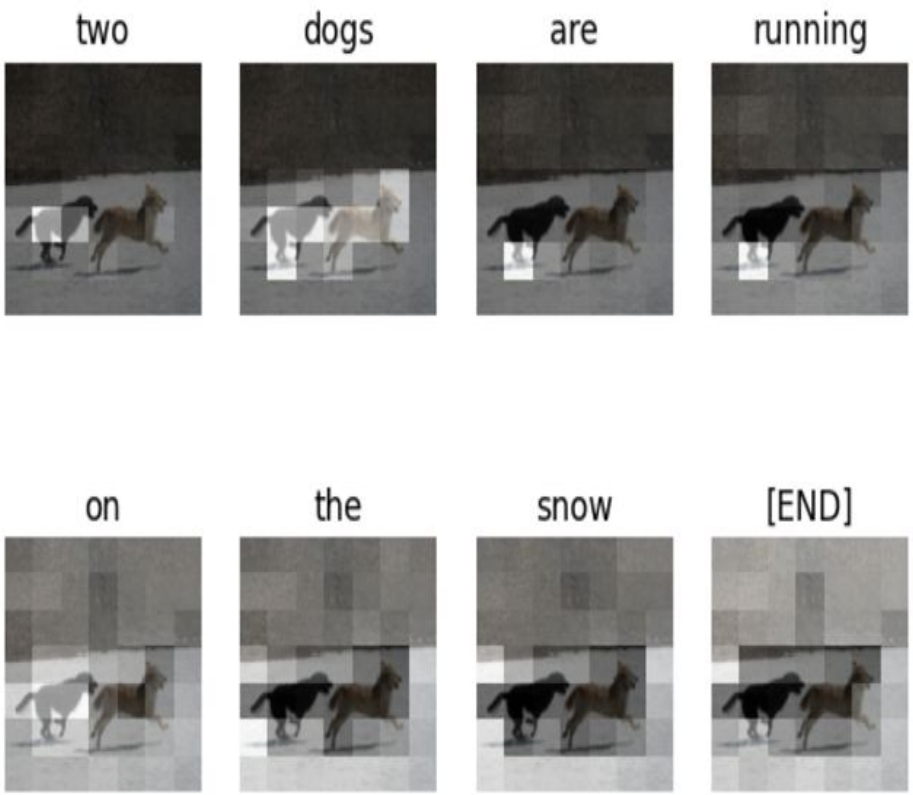
Model Results

- **Results:**

two dogs are running on the snow

Visualizing Attention

```
captioner_model.show_attention(img)
```



Conclusion and Future Work

- **Conclusion**

- Using Pre-trained Mobilenet Architecture to convert images to vectors that can be fed to the cross attention layer in the transformer decoder architecture.
- TextVectorization use for the process of conversion of text into tokens and further into embeddings.
- Transformers significantly improve image captioning through attention mechanisms.
- Use Cross Attention and Casual Attention Mechanism.
- Creating the entire model architecture and comparing it previous architectures
- Visualising the attention of each word generated on different parts of the image.

- **Future Directions:**

- Use larger datasets.
- Fine-tune models with more diverse images.

References

1. Attention is all you need paper: link
2. Show Attend and Tell paper: link

Q&A

- Questions and Answers
- Thank you for your attention!