



# Deep Learning

## Image Captioning in Arabic (Multilingual Seq2Seq)

الدرجة	السكنى	كود الطالب	الاسم
	2	22030190	ندى ماهر محمد الهايدي السيد
	2	22030196	هاجر خالد ابراهيم محمود كوتة
	2	22030184	منه الله السعيد عبد الفتاح
	2	22030200	هناه محمود احمد

# section1 : problem domain

Image captioning is a computer vision and natural language generation task whose objective is to automatically produce descriptive sentences that reflect the visual content of an image. This requires a deep understanding of objects, actions, spatial relationships, and scene context, followed by converting that understanding into coherent natural language.

This domain lies at the intersection of several advanced deep learning techniques, including:

- **Convolutional Neural Networks (CNNs):** used to extract rich visual features from images.
- **Sequence-to-Sequence (Seq2Seq) architectures:** used to generate text descriptions token by token.
- **Attention mechanisms:** used to help the model focus on specific regions of the image while generating each word, improving caption relevance and accuracy.

Traditional computer vision-based approaches struggle to manage this complexity because they rely on hand-crafted features, limited pattern recognition, and inflexible language models. In contrast, modern deep learning systems combine CNN encoders with language decoders to produce fluent and context-aware captions, even for challenging real-world images.

In multilingual settings, the challenge increases further. Translating generated captions from English to Arabic introduces linguistic and structural differences such as sentence order, morphology, gender agreement, and vocabulary ambiguity. This makes the task significantly more complex and less explored in research.

Therefore, this project aims to build a deep learning-based Arabic image captioning system by:

- Generating English captions from images using a CNN–Seq2Seq architecture.
- Translating these captions into Arabic using neural machine translation models.

The work bridges the gap between computer vision and Arabic natural language processing, enabling image understanding and caption generation for Arabic-speaking users.

## Section2 : Project Description (What Was Done)

This project was implemented using deep learning architectures as follows:

### Dataset Preparation :

- **The Flickr8k dataset :** [Flickr 8k Dataset](#)
  - collection for sentence-based image description and search, consisting of 8,000 images that are each paired with five different captions which provide clear descriptions of the salient entities and events.
- **The Arabic to English Translation Sentence:** [Arabic to English Translation Sentences](#)
  - This dataset contains parallel translation pairs, allowing the system to learn linguistic patterns and vocabulary required to translate captions from English into Arabic.

### Image Feature Extraction (Encoder Stage) :

A customized encoder was built using a pre-trained ResNet-50 CNN model to extract spatial visual features from every image in the dataset. The encoder accepts input images with a resolution of  $224 \times 224 \times 3$ , which represents standard RGB format input.

ResNet-50 was loaded without the fully connected classification layers (`include_top=False`), allowing the model to output high-dimensional feature maps rather than predicted categories.

Transfer learning was used to stabilize training: most ResNet layers were frozen while the top 20 layers remained trainable to allow fine-tuning on the image captioning task.

The final output of ResNet-50 is a spatial feature map of size  $7 \times 7 \times 2048$ , representing 49 spatial locations, each encoded with 2048 deep visual descriptors. This feature map was reshaped to  $(49, 2048)$  to match the attention-based decoder requirements, producing 49 feature vectors per image.

To further align the extracted features with the decoder hidden dimension, a fully connected layer reduced the feature size from 2048 to 512 units, followed by dropout regularization to prevent overfitting.

These processed feature embeddings form the encoder output, enabling the decoder to attend to different image regions while generating captions.

## Caption Generation Model (Decoder Stage):

A Sequence-to-Sequence (Seq2Seq) architecture was developed to generate captions from visual feature vectors. This includes:

- Word embedding layer
- LSTM decoder
- Attention mechanism

The model was trained to generate English captions one word at a time from extracted image features.

using an LSTM decoder with attention. The decoder receives a sequence of word tokens and generates caption text one word at a time.

First, pre-trained GloVe word embeddings (glove.6B.100d.txt) were used to represent each word in the vocabulary with a dense 100-dimensional vector. These embeddings were loaded into a matrix that initializes the decoder's embedding layer. The embedding layer was set to non-trainable to preserve the semantic knowledge learned from large-scale text corpora. This improves language fluency, reduces training time, and prevents overfitting.

Next, the decoder initializes its hidden states directly from the encoder output. The encoder feature vectors are averaged to produce the initial hidden state ( $h_0$ ) and cell state ( $c_0$ ) for the LSTM. This allows the decoder to begin generation based on the global visual understanding of the image.

The LSTM network then processes the embedded word sequence and generates output vectors for each timestep. To improve model accuracy and enable selective focus on different image regions, an attention mechanism was added. The decoder output vectors are aligned with the encoder's 49 spatial visual vectors through dot-product attention, producing attention weights via softmax normalization. These weights highlight which visual regions are most relevant at each word prediction step.

A context vector is computed from these attention weights and combined with the LSTM output to produce a richer decoder representation. This combined feature is passed into a dense softmax output layer to predict the next word in the caption sequence.

This architecture allows the model to:

- attend to different image regions dynamically,
- preserve linguistic structure through pretrained embeddings, and
- generate captions that closely match the visual content of the image.

## **Bootstrapping Language Image Pretraining model (BLIP) for English Caption Generation :**

To enhance caption quality and align the model with the Flickr8k dataset, the BLIP (Bootstrapping Language-Image Pretraining) image captioning model was fine-tuned using paired image–text examples. BLIP combines visual feature extraction and language generation in a single transformer-based architecture, making it highly effective for captioning tasks.

A custom PyTorch dataset class was created to load image–caption pairs from Flickr8k. Each entry contains an image and one of its associated ground-truth captions. Images were preprocessed and tokenized using the BlipProcessor, while captions were encoded into input IDs and attention masks. Padding tokens were masked during loss computation to improve training efficiency and accuracy.

During fine-tuning, pixel values, word IDs, and labels were passed through the BLIP model to compute the cross-entropy loss between predicted and ground-truth caption tokens. The model was trained seven epoch using the AdamW optimizer with a linear learning-rate scheduler over a predefined number of epochs. At each iteration, backpropagation updated the model weights to better align generated captions with actual image descriptions.

### **This fine-tuning process enabled the BLIP model to:**

- learn dataset-specific visual–semantic relationships
- generate captions with improved contextual relevance
- better describe objects and scenes in Flickr8k images

Once trained, this model was used to produce high-quality English captions, which were then passed to the Arabic translation pipeline.

# Multilingual Bidirectional Auto-Regressive Transformers 50 (mBART50) “English to Arabic Translation Model”:

After generating English captions from the decoder, a neural machine translation (NMT) pipeline was used to convert them into Arabic. This step enhances multilingual usability and enables caption generation for Arabic-speaking users. The translation process was implemented using the MBART50 multilingual sequence-to-sequence model, which supports translation between English and Arabic through transfer learning.

The translation model was fine-tuned on an Arabic–English parallel dataset to optimize translation quality and improve language accuracy. This step addressed linguistic challenges such as sentence structure variations, morphology, word agreement, and vocabulary alignment. By training on real bilingual text pairs, the model learned to produce fluent Arabic sentences that closely follow the semantic meaning of the original English captions.

## The translation process consists of two main stages:

1. **Model Initialization** : The MBART50 tokenizer and model were loaded with English set as the source language (en\_XX) and Arabic as the target language (ar\_AR). GPU acceleration was used to enable more efficient training and inference.
2. **Fine-Tuning** : The model was trained for five epochs on the bilingual dataset. Each training batch included tokenized inputs and translation labels. Loss-based optimization was performed to minimize prediction errors and enhance translation fluency.

This translation framework completes the multilingual pipeline by combining computer vision, natural language generation, and machine translation into an end-to-end Arabic captioning system.

# Training and Optimization :

The captioning model was trained over multiple epochs to progressively improve its ability to generate accurate image descriptions. Several training strategies were applied to stabilize learning, reduce error propagation, and optimize language generation accuracy.

- **From-Slack Encoder–Decoder (ResNet50 + LSTM + Attention) :**

- **Architecture:** CNN encoder + LSTM decoder with attention
- **Training:** Fully trained from scratch
- **Strengths:**
  - Good for learning the fundamentals
  - Attention maps are interpretable
- **Weaknesses:**
  - Captions often short and generic
  - Requires heavy training time
  - Struggles with uncommon objects
- **Typical Output Quality:**
  - Captions are meaningful but sometimes lack detail
  - Grammar can be inconsistent
- **Performance:**
  - The model was trained for 60 epochs, but Early Stopping halted training at epoch 29 because the validation loss stopped improving. During training, both loss and validation loss decreased steadily before reaching a plateau.
  - Final Training Results (Before Early Stopping)
  - From the logs:
    - Final training loss (epoch 29): 0.6489
    - Final validation loss (epoch 29): 0.9345
    - Early stopping triggered at: epoch 29
    - The **best model** was saved when **val\_loss = 0.9313** at epoch 24

```
Epoch 24/60
1011/1011 0s 329ms/step - loss: 0.6952
Epoch 24: val_loss improved from 0.93239 to 0.93133, saving model to best_model.weights.h5
1011/1011 360s 356ms/step - loss: 0.6952 - val_loss: 0.9313 - learning_rate: 1.0000e-04
Epoch 25/60
1011/1011 0s 328ms/step - loss: 0.6859
Epoch 25: val_loss did not improve from 0.93133
1011/1011 357s 353ms/step - loss: 0.6859 - val_loss: 0.9328 - learning_rate: 1.0000e-04
Epoch 26/60
1011/1011 0s 330ms/step - loss: 0.6782
Epoch 26: val_loss did not improve from 0.93133
1011/1011 359s 355ms/step - loss: 0.6782 - val_loss: 0.9367 - learning_rate: 1.0000e-04
Epoch 27/60
1011/1011 0s 330ms/step - loss: 0.6716
Epoch 27: ReduceLROnPlateau reducing learning rate to 1.999999494757503e-05.

Epoch 27: val_loss did not improve from 0.93133
1011/1011 359s 355ms/step - loss: 0.6716 - val_loss: 0.9322 - learning_rate: 1.0000e-04
Epoch 28/60
1011/1011 0s 331ms/step - loss: 0.6600
Epoch 28: val_loss did not improve from 0.93133
1011/1011 359s 356ms/step - loss: 0.6600 - val_loss: 0.9362 - learning_rate: 2.0000e-05
Epoch 29/60
1011/1011 0s 330ms/step - loss: 0.6489
Epoch 29: val_loss did not improve from 0.93133
1011/1011 359s 355ms/step - loss: 0.6489 - val_loss: 0.9345 - learning_rate: 2.0000e-05
```

- **Bootstrapping Language Image Pretraining model (BLIP) on Flickr8k :**

The BLIP image captioning model was fine-tuned on the Flickr8k dataset to align its language–vision understanding with the specific image–caption pairs in the dataset. Unlike the from-scratch LSTM model, BLIP required significantly fewer epochs due to its transformer-based architecture and strong pretrained initialization.

- **Architecture:** Transformer encoder–decoder (vision + language unified)
- **Training:** Fine-tuned for 7 epochs
- **Strengths:**
  - Learns dataset
  - Generates longer, richer, more accurate captions
  - Significant improvement over CNN+LSTM
- **Weaknesses:**
  - Requires GPU memory
- **Output Quality:**
  - Descriptions more natural and detailed
  - Better object–relationship understanding
- **Performance:**
  - During training, the model showed stable convergence, with the loss steadily decreasing across epochs.
  - Total training epochs: 7
  - **Final training loss: 0.3868**

```
Epoch 1/7: 100%|██████████| 5057/5057 [55:15<00:00, 1.53it/s]
```

```
Epoch 1/7 - Average Loss: 1.9890
```

```
Epoch 2/7: 100%|██████████| 5057/5057 [55:18<00:00, 1.52it/s]
```

```
Epoch 2/7 - Average Loss: 1.5720
```

```
Epoch 3/7: 100%|██████████| 5057/5057 [55:18<00:00, 1.52it/s]
```

```
Epoch 3/7 - Average Loss: 1.2803
```

```
Epoch 4/7: 100%|██████████| 5057/5057 [55:20<00:00, 1.52it/s]
```

```
Epoch 4/7 - Average Loss: 1.0073
```

```
Epoch 5/7: 100%|██████████| 5057/5057 [55:18<00:00, 1.52it/s]
```

```
Epoch 5/7 - Average Loss: 0.7538
```

```
Epoch 6/7: 100%|██████████| 5057/5057 [55:17<00:00, 1.52it/s]
```

```
Epoch 6/7 - Average Loss: 0.5420
```

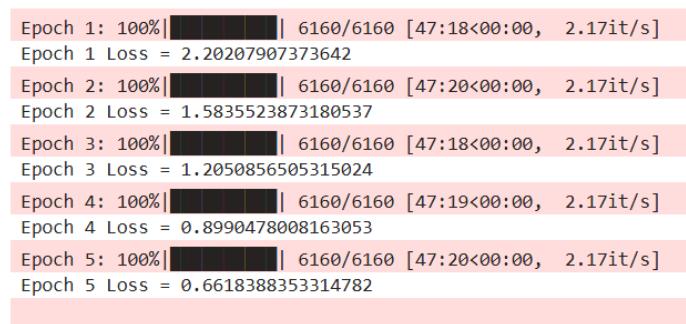
```
Epoch 7/7: 100%|██████████| 5057/5057 [55:20<00:00, 1.52it/s]
```

```
Epoch 7/7 - Average Loss: 0.3868
```

## • Pretrained Model mBART50 for English to Arabic Translation :

The Multilingual Bidirectional Auto regressive Transformers 50 (mBART50) model was fine-tuned to translate the English captions generated by the Bootstrapping Language-Image Pretraining model into fluent and semantically accurate Arabic. Because mBART50 is a pretrained multilingual sequence-to-sequence transformer, it required relatively few epochs to adapt to the caption-style bilingual dataset.

- **Architecture:** Transformer encoder–decoder for multilingual sequence-to-sequence translation
- **Training:** Fine-tuned for **5 epochs**
- **Strengths :**
  - Learns Arabic grammar, morphology, and sentence structure effectively
  - Maintains semantic meaning from English caption to Arabic output
- **Weaknesses :**
  - Requires GPU memory for training
  - Sometimes produces slightly formal phrasing depending on dataset style
- **Output Quality :**
  - Translations are coherent, fluent, and contextually accurate
  - Arabic sentences reflect natural structure (subject–verb agreement, proper word order)
  - Preserves the meaning and details of the original English captions
- **Performance :**
  - **Total training epochs:** 5
  - **Final training loss:** 0.6618



mBART50 successfully adapted to the caption translation task, producing high-quality Arabic descriptions suitable for real deployment.

# Inference for Image Captioning (From-Scratch ResNet + LSTM + Attention)

After training the encoder-decoder model with attention, the model can generate captions for unseen images. The inference process consists of two main steps: **feature extraction** using the encoder and **caption generation** using the decoder with attention and beam search.

- **Feature Extraction Using the Encoder**

The encoder in our model is a **pre-trained ResNet50**, which extracts meaningful visual features from an input image. Each image is first resized to  $224 \times 224$  pixels and preprocessed according to ResNet50 requirements. The output of the encoder is a feature map of shape (49,512), which corresponds to  $7 \times 7$  spatial locations with 512-dimensional feature vectors.

**Before passing to the decoder, we add a batch dimension to match the expected input shape.**

- **Decoder with Attention**

The decoder is a **single-layer LSTM** that predicts one word at a time. During inference, the decoder uses its **previous hidden and cell states** and the **encoder features** to generate the next word.

Our attention mechanism allows the decoder to focus on relevant regions of the image when predicting each word. The key steps are:

- a. Compute **attention scores** between the current decoder state and encoder outputs.
- b. Convert scores to **attention weights** via softmax.
- c. Generate a **context vector** as the weighted sum of encoder features.
- d. Concatenate the context vector with the decoder output and pass it through a dense layer to predict the next word.

- **Caption Generation Using Beam Search**

Beam search is used to maintain multiple candidate sequences at each time step, which improves caption quality compared to greedy decoding.

Algorithm steps:

- a. Initialize sequences with the start token, score 0, and initial LSTM states.
- b. For each time step up to max\_len:
  - Predict the probability of the next word for each sequence.
  - Select the top k most probable words (beam width).
  - Update sequences with new candidates and their scores.
  - Separate finished sequences that reached the end token.
- c. Continue until all sequences are finished or maximum length is reached.
- d. Select the sequence with the lowest negative log-likelihood as the final caption.

- **Summary**

This inference pipeline demonstrates how a from-scratch ResNet + LSTM + Attention model generates natural language descriptions for images. By combining:

- CNN-based feature extraction (ResNet50)
- Sequential modeling (LSTM)
- Attention mechanism
- Beam search decoding

the model produces interpretable and contextually accurate captions.

# **Output Generation (Using the Bootstrapping Language-Image Pretraining Model and mBART50 Translation Model) :**

During inference, the system generates **both English and Arabic captions** directly from the input image using a combination of the Bootstrapping Language-Image Pretraining (BLIP) model and the Multilingual Bidirectional Auto regressive from Transformers 50 (mBART50) translation model. This eliminates the need for manual feature extraction or a custom LSTM decoder. The complete process works as follows:

## **1. Input Image Acquisition :**

- The user uploads an image through the graphical interface.
- The image is automatically preprocessed (resized, normalized, and formatted) to meet the input requirements of the BLIP vision language transformer.

## **2. English Caption Generation (BLIP) :**

- The preprocessed image is passed into the BLIP model, which integrates visual feature extraction and language generation within a unified transformer-based architecture.
- BLIP produces a detailed and context-aware **English caption** directly from the image in a single step.

## **3. Arabic Caption Generation (mBART50) :**

- The generated English caption is then sent to the mBART50 sequence-to-sequence translation model.
- mBART50, fine-tuned for English to Arabic translation, converts the English caption into a fluent, grammatically correct **Arabic caption** while preserving meaning and context.

## **4. Bilingual Output :**

The system can produce:

- English caption only
- Arabic caption only

## **Final Result**

This approach enables a fully integrated bilingual captioning pipeline capable of producing accurate, natural, and contextually meaningful captions for any input image.

By relying on the pretrained BLIP model and the mBART50 translation model, the system simplifies the inference workflow, reduces preprocessing steps, and ensures high-quality caption generation without training a custom encoder–decoder architecture from scratch.

## **Deployment and GUI Inference :**

To make the bilingual image captioning system accessible and user-friendly, a Gradio based graphical user interface (GUI) was developed. This interface enables users to upload an image and automatically receive a descriptive caption in English or Arabic. The deployment integrates the full inference pipeline into a simple, interactive interface.

### **Model Loading**

During deployment, two pretrained models are loaded into memory for fast real-time inference:

- **Bootstrapping Language-Image Pretraining (BLIP) :**  
Used to generate the initial English caption directly from the input image.
- **Multilingual Bidirectional Auto Regressive from Transformers 50 (mBART50) :**  
Fine-tuned for English to Arabic translation, used to convert the BLIP-generated English caption into a fluent Arabic sentence.

### **Caption Generation Function**

The inference function processes user input in the following sequence:

#### **1. Image Input:**

The uploaded image is preprocessed using the BLIP processor to match the model's visual input format.

#### **2. English Caption Generation:**

The preprocessed image is fed into BLIP, which produces a descriptive English caption using its unified vision–language transformer architecture.

#### **3. Arabic Caption Generation (Optional):**

If the user selects Arabic output, the English caption is tokenized and passed to the mBART50 translation model.

mBART50 generates a fluent and grammatically correct Arabic caption while preserving the meaning of the original English description.

#### **4. Dynamic Language Selection:**

The function supports generating:

- English captions
- Arabic captions

## **Gradio Interface**

The graphical interface includes the following components:

- An image uploader for selecting input images
- A dropdown menu to choose the desired output language (English, Arabic, or both)
- A caption display area showing the generated result

The interface is intuitive and requires no technical knowledge, allowing any user to interact with the captioning system effortlessly.

## **Final Deployment**

This deployment approach provides a real-time, end-to-end bilingual image captioning system that integrates computer vision, natural language generation, and neural machine translation in one unified application.

It demonstrates the practical usability of BLIP and mBART50 models without requiring the user to interact directly with code, making the system suitable for demonstrations, research presentations, and real-world use.

# Comparison Between From Scratch and Pretrained Models

Aspect	From Scratch Model	Pretrained Model
<b>Learning Purpose</b>	Helps in understanding model internals and architecture design	Focuses on achieving high-quality results
<b>Model Knowledge</b>	Learns visual and linguistic patterns only from the given dataset	Leverages rich prior knowledge learned from large-scale data
<b>Caption Quality</b>	Basic, short, and sometimes generic captions	Fluent, detailed, and context-aware captions
<b>Visual Understanding</b>	Limited object and scene recognition	Strong understanding of objects, actions, and context
<b>Language Fluency</b>	Limited language modeling capability	Advanced language generation capability
<b>Generalization</b>	Weak generalization to unseen images	Strong generalization to new images
<b>Loss Behavior (Initial)</b>	Starts with high loss (2.28)	Starts with lower loss (1.98)
<b>Loss Convergence</b>	Slow and unstable convergence	Fast and stable convergence
<b>Final Loss Result</b>	<code>val_loss = 0.9313</code>	<code>loss = 0.3868</code>
<b>Practical Usability</b>	Mainly educational and experimental	Suitable for real-world applications

## Section3 : Source Code

### encoder-decoder from scratch for image caption in english :

```
# import libraries
```

```
import os
```

```
import numpy as np
```

```
import pandas as pd
```

```
import tensorflow as tf
```

```
import re
```

```
from collections import defaultdict
```

```
from tensorflow.keras.preprocessing.text import Tokenizer
```

```
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

```
from tensorflow.keras.preprocessing.image import load_img, img_to_array
```

```
from tensorflow.keras.applications import ResNet50
```

```
from tensorflow.keras.applications.resnet import preprocess_input
```

```
from tensorflow.keras.layers import Input, Dense, LSTM, Embedding, Concatenate, Activation, Reshape,Dropout
```

```
from tensorflow.keras.models import Model
```

```
from tensorflow.keras import layers
```

```
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau,ModelCheckpoint
```

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

```
from tensorflow.keras.layers import Lambda
```

```
from sklearn.model_selection import train_test_split
```

```
import matplotlib.pyplot as plt
```

```
from PIL import Image
```

```
#dataset path
```

```
images_path = "/kaggle/input/flickr8k/Images"
```

```
captions_file = "/kaggle/input/flickr8k/captions.txt"
```

```

# load images from the dataset

def load_flickr_captions_csv(filename):

    df = pd.read_csv(filename)

    captions_dict = defaultdict(list) #Creates an empty dictionary where each key starts with an empty list by default

    for _, row in df.iterrows():      # _ ignore row index , row access caption and image

        captions_dict[row["image"]].append(row["caption"]) #Appends caption to the matching image

    return captions_dict

captions_dict = load_flickr_captions_csv(captions_file)

print("Total images:", len(captions_dict))

    Total images: 8091

# preprocessing on text

def clean_caption(caption):

    caption = caption.lower().strip()

    return caption

# prepare for decoder input

for img in captions_dict:

    captions_dict[img] = [
        "<STARTSEQ> " + clean_caption(c) + " <ENDSEQ>" for c in captions_dict[img]
    ]

example_img = list(captions_dict.keys())[0]

print(example_img) # image name

print(captions_dict[example_img]) # image captions

```

1000268201\_693b08cb0e.jpg  
['<STARTSEQ> a child in a pink dress is climbing up a set of stairs in an entry way . <ENDSEQ>', '<STARTSEQ> a girl going into a wooden building . <ENDSEQ>', '<STARTSEQ> a little girl climbing into a wooden playhouse . <ENDSEQ>', '<STARTSEQ> a little girl climbing the stairs to her playhouse . <ENDSEQ>', '<STARTSEQ> a little girl in a pink dress going into a wooden cabin . <ENDSEQ>']

```
all_captions = []

for caps in captions_dict.values():
    all_captions.extend(caps)

tokenizer = Tokenizer()
tokenizer.fit_on_texts(all_captions)

vocab_size = len(tokenizer.word_index) + 1
max_len = max(len(c.split()) for c in all_captions)

print("Vocab size:", vocab_size)
print("Max caption length:", max_len)
```

**Vocab size: 8496**  
**Max caption length: 40**

```
# data augmentation

datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.1, # Applies shearing (slanted stretching)
    zoom_range=0.1,
    horizontal_flip=True,
    fill_mode='nearest'
)
```

```

def load_image(img_path):
    img = load_img(img_path, target_size=(224, 224))

    img = img_to_array(img) # convert to machine readable

    img = preprocess_input(img)

    return img

X_images = [] # encoder input

X_captions_input = [] # decoder input

y_captions_output = [] # decoder output

for img_name, captions in captions_dict.items():

    img_path = os.path.join(images_path, img_name)

    image = load_image(img_path)

    for caption in captions:

        seq = tokenizer.texts_to_sequences([caption])[0]

        in_seq = seq[:] #all

        out_seq = seq[1:] # shifted

        in_seq = pad_sequences([in_seq], maxlen=max_len, padding="post")[0] # [0] to access [[]]

        out_seq = pad_sequences([out_seq], maxlen=max_len, padding="post")[0]

        X_images.append(image)

        X_captions_input.append(in_seq)

        y_captions_output.append(out_seq)

X_images = np.array(X_images, dtype=np.float16)

X_captions_input = np.array(X_captions_input, dtype=np.int16)

y_captions_output = np.array(y_captions_output, dtype=np.int16)

print("X_images:", X_images.shape) #encoder input

print("X_captions_input:", X_captions_input.shape) #decoder output

print("y_captions_output:", y_captions_output.shape) #decoder input

X_images: (40455, 224, 224, 3)
X_captions_input: (40455, 40)
y_captions_output: (40455, 40)

```

```

X_train_imgs, X_val_imgs, X_train_inp, X_val_inp, y_train, y_val = train_test_split(
    X_images, X_captions_input, y_captions_output, test_size=0.2, random_state=42)

print("Training images:", X_train_imgs.shape)

print("Validation images:", X_val_imgs.shape)

print("Training decoder input:", X_train_inp.shape)

print("Validation decoder input:", X_val_inp.shape)

print("Training decoder output:", y_train.shape)

print("Validation decoder output:", y_val.shape)

    Training images: (32364, 224, 224, 3)
    Validation images: (8091, 224, 224, 3)
    Training decoder input: (32364, 40)
    Validation decoder input: (8091, 40)
    Training decoder output: (32364, 40)
    Validation decoder output: (8091, 40)

del X_images, X_captions_input,y_captions_output

# encoder input

encoder_input = Input(shape=(224, 224, 3), name="encoder_input")

# load resnet50 base model spatial features, no pooling

resnet_base = ResNet50(include_top=False, # without the top classification layer

                      weights="imagenet",

                      input_tensor=encoder_input)

for layer in resnet_base.layers[:-20]:

    layer.trainable = False

for layer in resnet_base.layers[-20:]:

    layer.trainable = True

# output shape from renet50

x = resnet_base.output # (batch,7,7,2048) 7*7 spatial dimention ,2048 channels or features

x = Reshape((49, 2048))(x) # (batch,49,2048) # Prepare features for attention to help decoder focus on specific regions

encoder_output = Dense(512, activation="relu")(x) # Reduce dimensionality for decoder hidden size 2048 to 512

encoder_output = Dropout(0.3)(encoder_output)

encoder_model = Model(inputs=encoder_input, outputs=encoder_output) #Input= raw image, Output= feature vector

```

```
glove_path = "/kaggle/input/text-data/glove.6B.100d.txt"
```

```
embedding_dim = 100
```

### # Load GloVe embeddings into a dictionary

```
embeddings_index = {}
```

```
with open(glove_path, 'r', encoding='utf8') as f:
```

```
    for line in f:
```

```
        values = line.split()
```

```
        word = values[0]
```

```
        coefs = np.asarray(values[1:], dtype='float32')
```

```
        embeddings_index[word] = coefs
```

```
print(f"Loaded {len(embeddings_index)} word vectors from GloVe.")
```

```
embedding_matrix = np.zeros((vocab_size, embedding_dim))
```

```
for word, i in tokenizer.word_index.items():
```

```
    if i >= vocab_size:
```

```
        continue
```

```
    embedding_vector = embeddings_index.get(word)
```

```
    if embedding_vector is not None:
```

```
        embedding_matrix[i] = embedding_vector
```

```
# decoder

decoder_input = Input(shape=(max_len,), name="decoder_input")

# Embedding layer

embedding = Embedding(
    vocab_size,
    embedding_dim,
    weights=[embedding_matrix],
    mask_zero=True,
    trainable=False
)(decoder_input)

embedding = Dropout(0.2)(embedding)

# initial states from encoder

# Compute mean across the 49 spatial vectors

h0 = Lambda(lambda x: tf.reduce_mean(x, axis=1), name="init_h")(encoder_output)

c0 = Lambda(lambda x: tf.reduce_mean(x, axis=1), name="init_c")(encoder_output)

# decoder lstm

decoder_lstm = LSTM(
    512,
    return_sequences=True,
    return_state=True,
    dropout=0.3,
    recurrent_dropout=0.3,
    name="decoder_lstm"
)

decoder_seq, state_h, state_c = decoder_lstm(
    embedding, initial_state=[h0, c0]
)
```

## # attention

#dot product along feature dimension to compute similarity between each decoder timestep and each encoder spatial vector.

```
score = layers.Dot(axes=[2,2], name="attention_score")([decoder_seq, encoder_output]) #score= (batch size,max len,49)
```

# softmax attention matrix

```
attention_weights = layers.Activation('softmax', name="attention_weights")(score)
```

# context = weighted sum of encoder output

```
context = layers.Dot(axes=[2,1], name="context_vector")([attention_weights, encoder_output])
```

## # concatenate context + decoder timestep output

```
decoder_combined = Concatenate(axis=-1, name="decoder_context_concat")([decoder_seq, context])
```

```
decoder_concat = Dropout(0.3)(decoder_combined)
```

```
decoder_dense = Dense(vocab_size, activation="softmax", name="output_dense")
```

```
output = decoder_dense(decoder_combined)
```

```
model = Model(inputs=[encoder_input, decoder_input], outputs=output)
```

```
model.compile(
```

```
    loss="sparse_categorical_crossentropy",
```

```
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4)
```

```
)
```

```
model.summary()
```

```

def array_data_generator(X_imgs, X_inp, y_out, batch_size=32, augment=False):

    num_samples = len(X_imgs)

    while True:

        for i in range(0, num_samples, batch_size):

            batch_imgs = X_imgs[i:i+batch_size].copy().astype(np.float32)

            batch_inp = X_inp[i:i+batch_size]

            batch_out = y_out[i:i+batch_size]

            if augment:

                for j in range(len(batch_imgs)):

                    batch_imgs[j] = datagen.random_transform(batch_imgs[j]) # apply augmentation

            yield (batch_imgs, batch_inp), batch_out


train_gen = array_data_generator(X_train_imgs, X_train_inp, y_train, batch_size=32, augment=True)

val_gen = array_data_generator(X_val_imgs, X_val_inp, y_val, batch_size=32, augment=False)

steps_per_epoch = len(X_train_imgs) // 32

val_steps = len(X_val_imgs) // 32

early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

reduce_lr = ReduceLROnPlateau( monitor='val_loss', factor=0.2, patience=3, verbose=1, min_lr=1e-7)

checkpoint = ModelCheckpoint( filepath='best_model.weights.h5', monitor='val_loss', verbose=1,
                             save_best_only=True, save_weights_only=True)

```

```
model.fit( train_gen,
           steps_per_epoch=steps_per_epoch,
           epochs=60,
           validation_data=val_gen,
           validation_steps=val_steps,
           callbacks=[early_stop, reduce_lr, checkpoint]
)

Epoch 22/60
1011/1011 0s 327ms/step - loss: 0.7113
Epoch 22: val_loss improved from 0.93504 to 0.93239, saving model to best_model.weights.h5
1011/1011 357s 353ms/step - loss: 0.7113 - val_loss: 0.9324 - learning_rate: 1.0000e-04
Epoch 23/60
1011/1011 0s 329ms/step - loss: 0.7019
Epoch 23: val_loss did not improve from 0.93239
1011/1011 358s 354ms/step - loss: 0.7019 - val_loss: 0.9369 - learning_rate: 1.0000e-04
Epoch 24/60
1011/1011 0s 329ms/step - loss: 0.6952
Epoch 24: val_loss improved from 0.93239 to 0.93133, saving model to best_model.weights.h5
1011/1011 360s 356ms/step - loss: 0.6952 - val_loss: 0.9313 - learning_rate: 1.0000e-04
Epoch 25/60
1011/1011 0s 328ms/step - loss: 0.6859
Epoch 25: val_loss did not improve from 0.93133
1011/1011 357s 353ms/step - loss: 0.6859 - val_loss: 0.9328 - learning_rate: 1.0000e-04
Epoch 26/60
1011/1011 0s 330ms/step - loss: 0.6782
Epoch 26: val_loss did not improve from 0.93133
1011/1011 359s 355ms/step - loss: 0.6782 - val_loss: 0.9367 - learning_rate: 1.0000e-04
Epoch 27/60
1011/1011 0s 330ms/step - loss: 0.6716
Epoch 27: ReduceLROnPlateau reducing learning rate to 1.999999494757503e-05.

Epoch 27: val_loss did not improve from 0.93133
1011/1011 359s 355ms/step - loss: 0.6716 - val_loss: 0.9322 - learning_rate: 1.0000e-04
Epoch 28/60
1011/1011 0s 331ms/step - loss: 0.6600
Epoch 28: val_loss did not improve from 0.93133
1011/1011 359s 356ms/step - loss: 0.6600 - val_loss: 0.9362 - learning_rate: 2.0000e-05
Epoch 29/60
1011/1011 0s 330ms/step - loss: 0.6489
Epoch 29: val_loss did not improve from 0.93133
1011/1011 359s 355ms/step - loss: 0.6489 - val_loss: 0.9345 - learning_rate: 2.0000e-05

model.save("english_caption_60epoch.keras")
```

## # Encoder input (already built)

```
encoder_inputs = encoder_input # shape=(224,224,3)
```

## # Decoder inputs

```
decoder_inputs_inf = Input(shape=(1,), name="decoder_input_inf") # one word at a time
```

```
state_h_input = Input(shape=(512,), name="state_h_input")
```

```
state_c_input = Input(shape=(512,), name="state_c_input")
```

```
encoder_out_input = Input(shape=(49, 512), name="encoder_out_input")
```

```
embedding_inf = model.get_layer("embedding")(decoder_inputs_inf)
```

```
decoder_lstm_layer = model.get_layer("decoder_lstm")
```

```
decoder_seq_inf, state_h_out, state_c_out = decoder_lstm_layer(
```

```
embedding_inf, initial_state=[state_h_input, state_c_input])
```

## # Attention layers from training model

```
attention_score_layer = model.get_layer("attention_score")
```

```
attention_weights_layer = model.get_layer("attention_weights")
```

```
context_layer = model.get_layer("context_vector")
```

```
concat_layer = model.get_layer("decoder_context_concat")
```

```
dense_layer = model.get_layer("output_dense")
```

```
score_inf = attention_score_layer([decoder_seq_inf, encoder_out_input])
```

```
attention_weights_inf = attention_weights_layer(score_inf)
```

```
context_inf = context_layer([attention_weights_inf, encoder_out_input])
```

```
decoder_combined_inf = concat_layer([decoder_seq_inf, context_inf])
```

```
output_inf = dense_layer(decoder_combined_inf) # predicted word probabilities
```

```
inference_model = Model(
```

```
inputs=[decoder_inputs_inf, state_h_input, state_c_input, encoder_out_input],
```

```
outputs=[output_inf, state_h_out, state_c_out])
```

```

def generate_caption_beam_fixed(encoder_feature, tokenizer, max_len=35, beam_width=3):

    start_token = tokenizer.word_index['startseq']

    end_token = tokenizer.word_index['endseq']

    sequences = [[ [start_token], 0.0, np.zeros((1,512)), np.zeros((1,512)) ]]

    for _ in range(max_len):

        all_candidates = []

        finished_sequences = []

        for seq, score, state_h, state_c in sequences:

            if seq[-1] == end_token:

                finished_sequences.append([seq, score])

                continue

            seq_input = np.array([seq[-1]]).reshape(1,1)

            yhat, state_h_new, state_c_new = inference_model.predict([seq_input, state_h, state_c, encoder_feature],
verbose=0)

            top_indices = np.argsort(yhat[0,0])[::-1][:beam_width]

            for idx in top_indices:

                candidate_seq = seq + [idx]

                candidate_score = score - np.log(yhat[0,0,idx]+1e-10)

                all_candidates.append([candidate_seq, candidate_score, state_h_new, state_c_new])

        ordered = sorted(all_candidates, key=lambda tup: tup[1])

        sequences = ordered[:beam_width]

        if len(finished_sequences) >= beam_width:

            break

    if finished_sequences:

        sequences = sorted(finished_sequences, key=lambda tup: tup[1])

        best_seq = sequences[0][0]

    else: best_seq = sequences[0][0]

    reverse_word_index = {v:k for k,v in tokenizer.word_index.items()}

```

```
caption_words = [ reverse_word_index[int(i)] for i in best_seq if i not in [start_token, end_token] and int(i) in reverse_word_index]

return ''.join(caption_words)

def extract_resnet_features(image_path, encoder_model):

# Load image and resize to 224x224

img = load_img(image_path, target_size=(224, 224))

img = img_to_array(img)

img = np.expand_dims(img, axis=0)

# Preprocess for ResNet

img = preprocess_input(img)

# Extract features

features = encoder_model.predict(img, verbose=0) # shape (1, 49, 512)

return features[0] # return shape (49, 512)

image_path = r"/kaggle/input/flickr8k/Images/102351840_323e3de834.jpg"

img_feature = extract_resnet_features(image_path, encoder_model)

img_feature = np.expand_dims(img_feature, axis=0) # add batch dim

caption = generate_caption_beam_fixed(img_feature, tokenizer)

print("Predicted caption:", caption)

def show_image(img_path):

    img = Image.open(img_path)

    plt.imshow(img)

    plt.axis('off')

    plt.show()

img_path = "/kaggle/input/flickr8k/Images/102351840_323e3de834.jpg"

show_image(img_path)
```

```
def load_and_process_image(img_path):
    img = load_img(img_path, target_size=(224, 224))
    img = img_to_array(img)
    img = preprocess_input(img)
    img = np.expand_dims(img, axis=0)
    return img

def generate_caption(model, tokenizer, image_path, max_len):
    # load & preprocess image
    img = load_and_process_image(image_path) # shape (1,224,224,3)
    start_token = tokenizer.word_index['startseq']
    end_token = tokenizer.word_index['endseq']
    caption_seq = [start_token]
    for _ in range(max_len):
        seq = pad_sequences([caption_seq], maxlen=max_len, padding='post')
        preds = model.predict([img, seq], verbose=0)
        next_id = np.argmax(preds[0, len(caption_seq)-1])
        if next_id == 0:
            break
        caption_seq.append(next_id)
        if next_id == end_token:
            break
    caption = [tokenizer.index_word[i] for i in caption_seq]
    if i not in [start_token, end_token, 0]:
        return " ".join(caption)

test_image = "/kaggle/input/sora-mn-barra/Screenshot 2025-12-19 013622.png"
print(generate_caption(model, tokenizer, test_image, max_len))
```

## **pretrained model for image captioning “BLIP ” :**

### **#import libraries**

```
from transformers import MBartForConditionalGeneration, MBart50TokenizerFast  
from torch.utils.data import Dataset, DataLoader  
import torch  
from tqdm import tqdm  
import pandas as pd  
from transformers import BlipProcessor, BlipForConditionalGeneration, get_scheduler  
from torch.optim import AdamW  
from PIL import Image  
import os
```

### **for image captioning in english :**

```
image_folder = "/kaggle/input/flickr8k/Images"  
captions_file = "/kaggle/input/flickr8k/captions.txt" # CSV: image,caption  
batch_size = 8  
epochs = 7  
lr = 5e-5  
max_length = 64  
device = "cuda" if torch.cuda.is_available() else "cpu"  
print("Using device:", device)  
processor = BlipProcessor.from_pretrained("Salesforce/blip-image-captioning-base")  
model = BlipForConditionalGeneration.from_pretrained(  
    "Salesforce/blip-image-captioning-base"  
).to(device)
```

```
class Flickr8kDataset(Dataset):

    def __init__(self, captions_file, image_folder, processor):
        self.data = []
        self.image_folder = image_folder
        self.processor = processor

        with open(captions_file, "r", encoding="utf-8") as f:
            lines = f.readlines()

            for line in lines[1:]: # skip header
                line = line.strip()

                if not line:
                    continue

                parts = line.split(",", 1) # split commas one time because captions can has commas

                if len(parts) != 2:
                    continue

                img_name, caption = parts

                img_path = os.path.join(self.image_folder, img_name)

                if not os.path.exists(img_path):
                    continue

                self.data.append((img_name, caption)) # Stores valid samples in memory

            print("Total samples:", len(self.data))

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        img_name, caption = self.data[idx]

        img_path = os.path.join(self.image_folder, img_name)

        image = Image.open(img_path).convert("RGB")
```

```
encoding = self.processor( # img encoder , caption decoder from blib
    images=image,
    text=caption,
    padding="max_length",
    truncation=True,
    max_length=max_length,
    return_tensors="pt")

input_ids = encoding["input_ids"].squeeze(0) # remove batch dimension

attention_mask = encoding["attention_mask"].squeeze(0)

pixel_values = encoding["pixel_values"].squeeze(0)

labels = input_ids.clone()

labels[labels == self.processor.tokenizer.pad_token_id] = -100

return {
    "pixel_values": pixel_values,
    "input_ids": input_ids,
    "attention_mask": attention_mask,
    "labels": labels
}
```

---

Total samples: 40455  
Number of batches: 5057

```
optimizer = AdamW(model.parameters(), lr=lr)

num_training_steps = epochs * len(dataloader)

scheduler = get_scheduler( "linear",
    optimizer=optimizer,
    num_warmup_steps=0, # start with initial value without increase
    num_training_steps=num_training_steps)

print("Training steps:", num_training_steps)
```

Training steps: 35399

```
model.train() # dropout and gradient active
```

```
for epoch in range(epochs):
    loop = tqdm(dataloader, desc=f'Epoch {epoch+1}/{epochs}')
    total_loss = 0.0 # to accumulate batch losses

    for batch in loop:
        pixel_values = batch["pixel_values"].to(device)
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)
        labels = batch["labels"].to(device)

        outputs = model(pixel_values=pixel_values,
                        input_ids=input_ids,
                        attention_mask=attention_mask,
                        labels=labels)

        loss = outputs.loss
        total_loss += loss.item() # accumulate

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    avg_loss = total_loss / len(dataloader) # average over batches
    print(f'Epoch {epoch+1}/{epochs} - Average Loss: {avg_loss:.4f}')

model.save_pretrained("./blip-finetuned-flickr8k")
processor.save_pretrained("./blip-finetuned-flickr8k")

# load model

device = "cuda" if torch.cuda.is_available() else "cpu"
print("Using device:", device)

model_path = "./blip-finetuned-flickr8k"
processor = BlipProcessor.from_pretrained(model_path)
model = BlipForConditionalGeneration.from_pretrained(model_path).to(device)
```

## pretrained translation model “mBART50”:

# load data

```
df = pd.read_csv("/kaggle/input/arabic-to-english-translation-sentences/ara_eng.txt", sep="\t", names=["en","ar"])
```

```
df = df.dropna()
```

```
print(df.shape)
```

```
df.head()
```

(24638, 2)

	en	ar
0	Hi.	مرحبا.
1	Run!	اركض!
2	Help!	النجدة!
3	Jump!	قفز!
4	Stop!	قف!

# load pretrained model

```
model_name = "facebook/mbart-large-50-many-to-many-mmt"
```

```
tokenizer = MBart50TokenizerFast.from_pretrained(model_name, src_lang="en_XX", tgt_lang="ar_AR")
```

```
model = MBartForConditionalGeneration.from_pretrained(model_name).cuda() # load translation weights
```

```
class TranslationDataset(Dataset):

    def __init__(self, df):
        self.en = df["en"].tolist()
        self.ar = df["ar"].tolist()

    def __len__(self):
        return len(self.en)

    def __getitem__(self, idx):
        src = tokenizer(
            self.en[idx],
            padding="max_length",
            truncation=True,
            max_length=64,
            return_tensors="pt"
        )

        tgt = tokenizer(
            self.ar[idx],
            padding="max_length",
            truncation=True,
            max_length=64,
            return_tensors="pt"
        )

        labels = tgt["input_ids"].squeeze()
        labels[labels == tokenizer.pad_token_id] = -100

        return {
            "input_ids": src["input_ids"].squeeze(),
            "attention_mask": src["attention_mask"].squeeze(),
            "labels": labels
        }
```

```
dataset = TranslationDataset(df)

train_loader = DataLoader(dataset, batch_size=4, shuffle=True)

optimizer = torch.optim.AdamW(model.parameters(), lr=2e-5)

# start training

epochs = 5

model.train()

for epoch in range(epochs):

    total_loss = 0

    for batch in tqdm(train_loader, desc=f"Epoch {epoch+1}"):
        batch = {k: v.cuda() for k, v in batch.items()} # Moves all tensors in the batch to GPU

        outputs = model(
            input_ids=batch["input_ids"],
            attention_mask=batch["attention_mask"],
            labels=batch["labels"]
        )

        loss = outputs.loss

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        total_loss += loss.item()

    print(f"Epoch {epoch+1} Loss = {total_loss/len(train_loader)}")
```

```
# save model
```

```
model.save_pretrained("mbart_en_ar_model")
```

```
tokenizer.save_pretrained("mbart_en_ar_model")
```

```
#load model
```

```
model_dir = "/kaggle/working/mbart_en_ar_model"
```

```
tokenizer = MBart50TokenizerFast.from_pretrained(model_dir)
```

```
model = MBartForConditionalGeneration.from_pretrained(model_dir, device_map="auto")
```

## Deployment on Gradio :

```
import gradio as gr
from PIL import Image
import torch
from transformers import BlipProcessor, BlipForConditionalGeneration
from transformers import MBartForConditionalGeneration, MBart50TokenizerFast
```

### # Load models

```
device = "cuda" if torch.cuda.is_available() else "cpu"
print("Using device:", device)
```

### # BLIP (English caption generator)

```
blip_processor = BlipProcessor.from_pretrained("Salesforce/blip-image-captioning-base")
blip_model = BlipForConditionalGeneration.from_pretrained(
    "Salesforce/blip-image-captioning-base"
).to(device)
```

### # mBART (English -> Arabic translator)

```
mbart_model_path = "mbart_en_ar_model" # path to your fine-tuned model
mbart_tokenizer = MBart50TokenizerFast.from_pretrained(mbart_model_path, src_lang="en_XX", tgt_lang="ar_AR")
mbart_model = MBartForConditionalGeneration.from_pretrained(mbart_model_path).to(device)
```

## # Caption + Translation function

```
def generate_caption(image, language="en"):

    # Step 1: English caption

    inputs = blip_processor(images=image, return_tensors="pt").to(device)

    out = blip_model.generate(**inputs, max_length=64)

    english_caption = blip_processor.decode(out[0], skip_special_tokens=True)

    if language == "en":

        return english_caption

    else:

        # Step 2: Translate to Arabic

        inputs = mbart_tokenizer(english_caption, return_tensors="pt").to(device)

        translated_ids = mbart_model.generate(**inputs, max_length=64)

        arabic_caption = mbart_tokenizer.decode(translated_ids[0], skip_special_tokens=True)

        return arabic_caption
```

## # Gradio interface

```
iface = gr.Interface(

    fn=generate_caption,

    inputs=[

        gr.Image(type="pil", label="Input Image"),

        gr.Dropdown(["en", "ar"], label="Language")

    ],

    outputs=gr.Textbox(label="Generated Caption"),

    title="Image Captioning + Translation",

    description="Upload an image and get a caption in English or Arabic using BLIP and mBART."

)

iface.launch()
```

# Section4 : Program Visualization Results

This section presents snapshots and explanations of the key outputs from the bilingual image captioning system.

## 1. Upload Image

**Description:** User uploads an image to the Gradio interface.

## 2. English Caption Generation

**Description:** BLIP generates a descriptive caption in English.

## 3. Arabic Caption Translation

**Description:** mBART translates the English caption into Arabic.

## 4. Full Interface

**Description:** The Gradio interface showing the uploaded image and generated captions.

The image contains two screenshots of the Gradio web application interface, demonstrating the bilingual image captioning process.

**Screenshot 1 (Top):** This screenshot shows the interface for generating an English caption. On the left, there is an "Input Image" field containing a white horse running through a field of flowers. Below it is a "Language" dropdown set to "en". At the bottom are "Clear" and "Submit" buttons. On the right, under "Generated Caption", the text "a white horse running through a field of flowers" is displayed. A "Flag" button is located below the caption area.

**Screenshot 2 (Bottom):** This screenshot shows the interface for generating an Arabic caption. On the left, there is an "Input Image" field containing an orange and white cat playing with a soccer ball on grass. Below it is a "Language" dropdown set to "ar". At the bottom are "Clear" and "Submit" buttons. On the right, under "Generated Caption", the text "قطة تلعب كرة القدم على العشب" is displayed. A "Flag" button is located below the caption area.