Project 3 - Visual Search Engine

Authors

1. Julia Lorenz, 156066
2. Marcel Rojewski, 156059

Github repo: https://github.com/marcelrojo/SearchEngine-CV.git

1. Dataset Overview

The dataset used is a subset of the **Caltech 256 Image Dataset**, originally sourced from Kaggle. While the full dataset includes 257 object categories and a total of 30,607 images, for this project, we have selected 100 categories from the Caltech 256 dataset and added 6 custom object categories. Each category contains approximately 100 images. And in total we have 11,520 images.

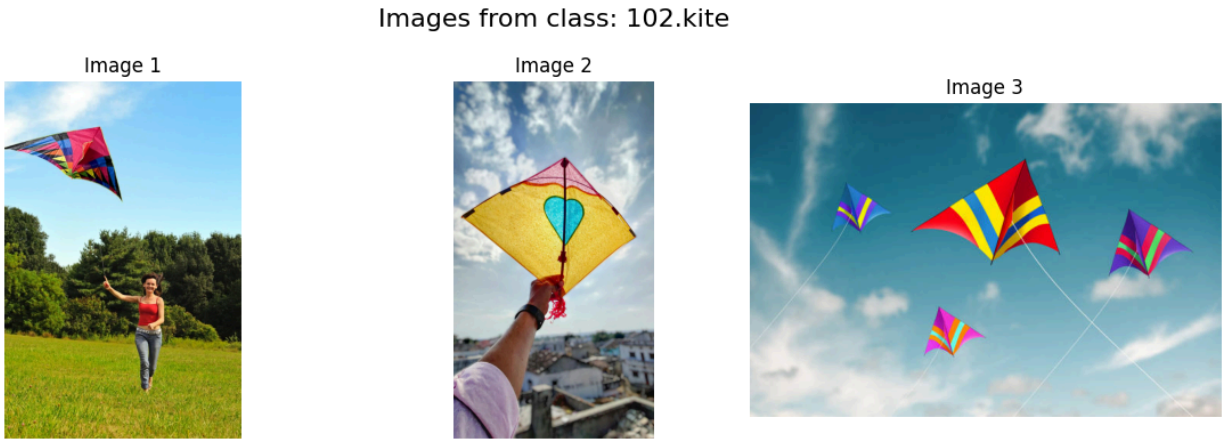
The dataset features a broad range of categories, including everyday objects and more unusual items, such as backpacks, bulldozers, rubber ducks, fried eggs, and diamond rings. These objects vary in size and appearance, providing a rich and diverse collection for training and evaluating models.

Below are some examples of the object categories included in the dataset:

- **Fireworks**

- **Baseball Bats**

- **Kites**



2. Problem

The task is to build a visual search engine that, given an image query, retrieves the top `k` most similar images from a dataset. This involves training a model to extract meaningful features (embeddings) from images, which can then be used to compare and rank other images in the dataset based on their similarity to the query.

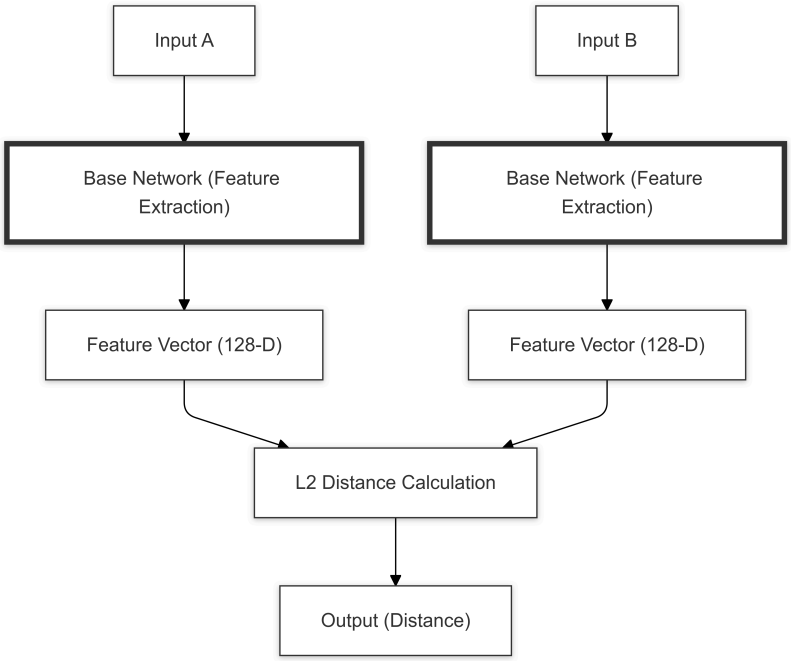
3. Architectures

3.1 Architectures - Experimentation

Siamese Neural Network

A **Siamese Neural Network** is a special type of neural network architecture designed to compare pairs of images. It learns to identify the similarities or differences between the inputs by training on labeled pairs of images.

Our idea was to create a Siamese network that learns to classify pairs of images as belonging to the same class (label 1.0) or from different classes (label 0.0). This process involved the following:



1. **Siamese Model Architecture** consisted of two identical base networks that share weights. The two input images were processed independently through these base networks, and the output embeddings were compared using an L2 distance metric. The model predicted the distance between the embeddings, which is then used to determine whether the images are similar or not.
2. **Base Network for Feature Extraction** was designed to extract relevant features from images and generate a 128-dimensional feature vector for each image. After training the Siamese network, we extracted the base model and used its output to create embeddings for new images.
3. **Contrastive Loss Function** was implemented to train the Siamese network, that aimed to encourage the network to output samller distances for similar pairs.
4. **Custom Accuracy Metric**, since the network outputs distance values, we introduced a custom accuracy function. This function assumes a threshold and converts the output distances into binary values (0 or 1) based on whether they are smaller or larger than the threshold, respectively. This allows the model's predictions to be evaluated in terms of accuracy.
5. **Comparison and Similarity**. After training the model, we extrected the base model to obtain embeddings for the database images and predict embeddings for the query image. After obtaining the embeddings, the cosine distance between the two feature vectors was computed to measure their similarity. The top k most similar images were selected based on this distance.

Although the model showed promising results, the model's performance was unstable and did not meet our expectations. Possibly due to inadequate data and model overfitting.

Below are presented some examples produced by this approach:

Model Analysis

Total params: 34,064,128 (129.94 MB)

Trainable params: 34,064,128 (129.94 MB)

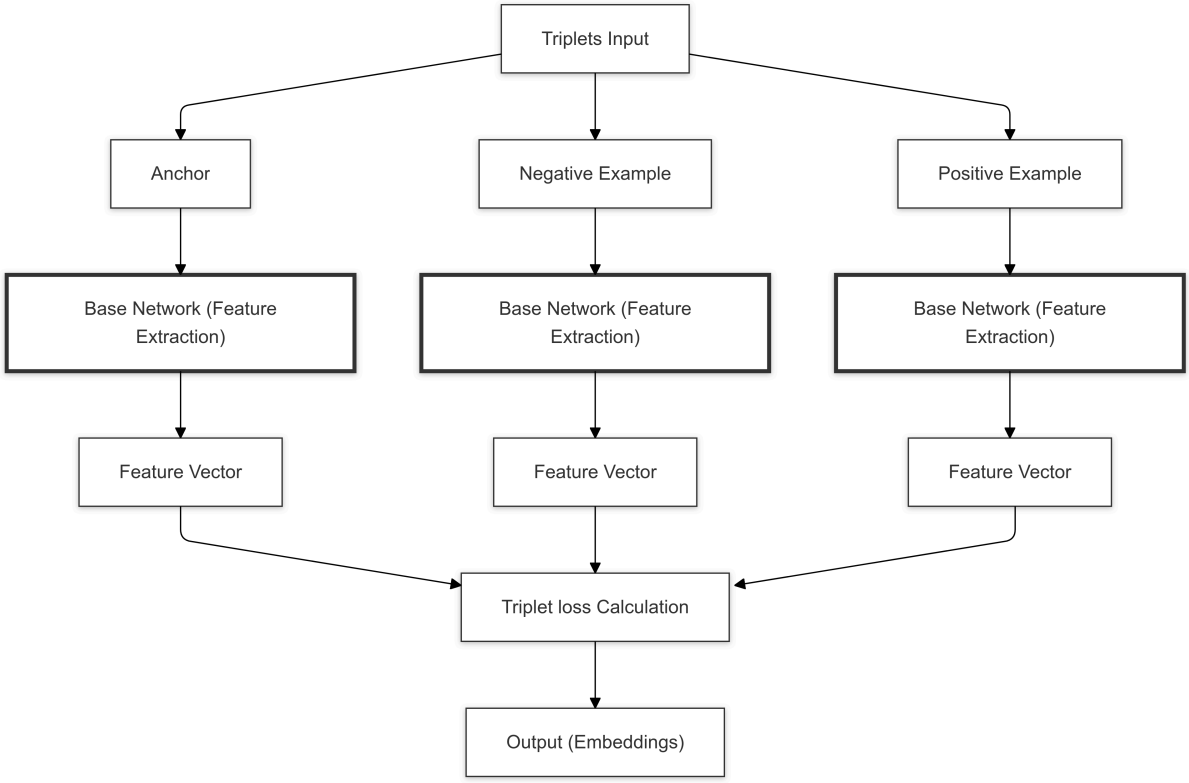
Non-trainable params: 0 (0.00 B)

In [ ]:
```python
# base model that will be used to extract features from the images, creates a feature vector
def build_base_network(input_shape):
    inputs = Input(shape=input_shape)
    x = Conv2D(64, (7, 7), activation="relu", padding="same")(inputs)
    x = MaxPooling2D(pool_size=(2, 2))(x)
    x = Conv2D(128, (5, 5), activation="relu", padding="same")(x)
    x = MaxPooling2D(pool_size=(2, 2))(x)
    x = Conv2D(256, (3, 3), activation="relu", padding="same")(x)
    x = MaxPooling2D(pool_size=(2, 2))(x)
    x = Flatten()(x)
    outputs = Dense(128, activation="relu")(x) # 128-dimensional feature vector
    return Model(inputs, outputs)

# full siamese network that takes two images as input and outputs the L2 distance between the
def build_siamese_network(input_shape):
    base_network = build_base_network(input_shape)

    input_a = Input(shape=input_shape)
    input_b = Input(shape=input_shape)

    encoded_a = base_network(input_a)
    encoded_b = base_network(input_b)

    # compute the L2 distance between the encoded vectors
    l2_distance = Lambda(lambda tensors: K.sqrt(K.sum(K.square(tensors[0] - tensors[1]), axis

    return Model(inputs=[input_a, input_b], outputs=l2_distance)
```

Triplet Loss Contrastive Model

A **Triplet Loss**-based Contrastive Neural Network is an architecture designed to minimize the distance between similar images while maximizing the distance between different ones. We hypothesized that contrasting negative examples from different categories with positive examples from the same category

would help optimize the embedding space, ensuring similar images cluster together. This process followed the architecture outlined below:



1. **Triplet Loss Based Contrastive Model Architecture**: The model takes triplets of images as input—an Anchor, to which distances are optimized; a Positive Example, which should be close in the embedding space; and a Negative Example, which should be distant from the anchor. A chosen Feature Extractor then generates an embedding vector for each image. The model outputs these embeddings while minimizing triplet loss.
2. **Base Netwrok for Feature Extraction**: The triplet loss-based contrastive model alone is not capable of creating a well-structured embedding space; rather, it is better suited for refining embeddings obtained from a feature extractor. Because of this, we experimented with various base networks. We tested optimizing only the embeddings of ResNet50, retraining the entire ResNet model on our dataset with a small learning rate, fine-tuning a custom classification model, and even training a custom feature extractor from scratch.
3. **Triplet Loss Function**: This function was implemented to train and optimize the network, aiming to refine the normalized embedding space.
4. **Comparison and Similarity**: The trained feature extraction model was saved and later used to generate embeddings for new images. The similarity between images was then measured based on their embedding distances.

While the approach seemed promising, the model did not perform as expected. It tended to cluster all embeddings too closely together. Although in some cases the results improved, in most scenarios, this led to a high number of false positives. Additionally it seemed to focus to hardly on the color space of the images, leading to some questionable choices for images with white background

Below are examples produced by this model using a custom embedding network trained from scratch solely with the contrastive loss function.

Model Analysis

Total params: 1,562,432 (5.96 MB)

Trainable params: 1,559,680 (5.95 MB)

Non-trainable params: 2,752 (10.75 KB)

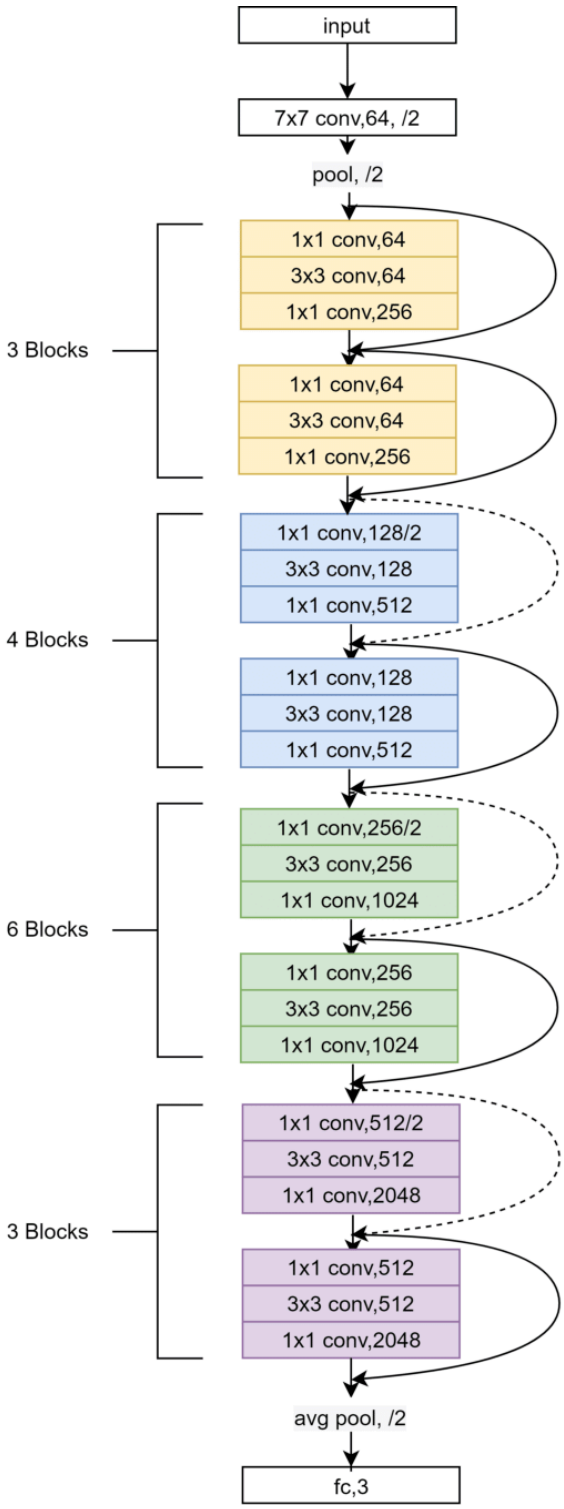
In [ ]:
```python
class L2Normalization(Layer):
    def call(self, inputs):
        return tf.math.l2_normalize(inputs, axis=1)

def build_embedding_network(input_shape=(256, 256, 3), embedding_dim=512):
    inputs = layers.Input(shape=input_shape)

    # Feature extraction block
    x = layers.Conv2D(32, (7, 7), strides=2, padding='same', activation='relu')(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling2D((3, 3), strides=2, padding='same')(x)

    # Second block
    x = layers.Conv2D(64, (5, 5), strides=2, padding='same', activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling2D((3, 3), strides=2, padding='same')(x)

    # Third block
    x = layers.Conv2D(128, (3, 3), strides=1, padding='same', activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(128, (3, 3), strides=1, padding='same', activation='relu')(x)
    x = layers.BatchNormalization()(x)
```

```
    x = layers.MaxPooling2D((2, 2), strides=2, padding='same')(x)

    # Feature extraction block 4
    x = layers.Conv2D(256, (3, 3), strides=1, padding='same', activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(256, (3, 3), strides=1, padding='same', activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling2D((2, 2), strides=2, padding='same')(x)

    # Bottleneck and global pooling
    x = layers.Conv2D(512, (1, 1), activation='relu', kernel_regularizer=regularizers.l2(0.01
    x = layers.BatchNormalization()(x)
    x = layers.GlobalAveragePooling2D()(x)

    # Dense layers for embedding
    x = layers.Dense(embedding_dim, activation='relu')(x)
    x = L2Normalization()(x)
    x = layers.Dropout(0.5)(x)

    model = models.Model(inputs, x, name="EmbeddingNetwork")
    return model


embedding_model = build_embedding_network(input_shape=(256, 256, 3), embedding_dim=512)
embedding_model.summary()

def build_siamese_model(embedding_model, input_shape=(256, 256, 3)):
    # Inputs for anchor, positive, and negative images
    anchor_input = layers.Input(name="anchor", shape=input_shape)
    positive_input = layers.Input(name="positive", shape=input_shape)
    negative_input = layers.Input(name="negative", shape=input_shape)

    # Pass each input through the embedding network
    anchor_embedding = embedding_model(anchor_input)
    positive_embedding = embedding_model(positive_input)
    negative_embedding = embedding_model(negative_input)

    embeddings = layers.Lambda(lambda x: tf.concat(x, axis=1))(
        [anchor_embedding, positive_embedding, negative_embedding]
    )

    # Combine embeddings into a Siamese model
    siamese_model = models.Model(
        inputs=[anchor_input, positive_input, negative_input],
        outputs=embeddings
    )

    return siamese_model

#Use the embedding model of your choice, initially trained on classificaiton embedding models
siamese_model = build_siamese_model(embedding_model)
siamese_model.summary()

def triplet_loss(y_true, y_pred, margin=0.4):
    # Split y_pred into anchor, positive, and negative
    anchor, positive, negative = tf.split(y_pred, num_or_size_splits=3, axis=1)

    # # Compute distances
    pos_similarity = tf.reduce_sum(anchor * positive, axis=1)  # Dot product
    neg_similarity = tf.reduce_sum(anchor * negative, axis=1)

    # Convert similarity to distance
```

```
    pos_dist = 1 - pos_similarity
    neg_dist = 1 - neg_similarity


    # # Compute triplet loss
    loss = tf.maximum(pos_dist - neg_dist + margin, 0.0)
    return tf.reduce_mean(loss)
```

ResNet50 model

ResNet50 is a state-of-the-art image classification residual network. It is 50 layers deep, utilizing convolutional and residual blocks that enable efficient learning even at such depth. Its architecture is presented below:



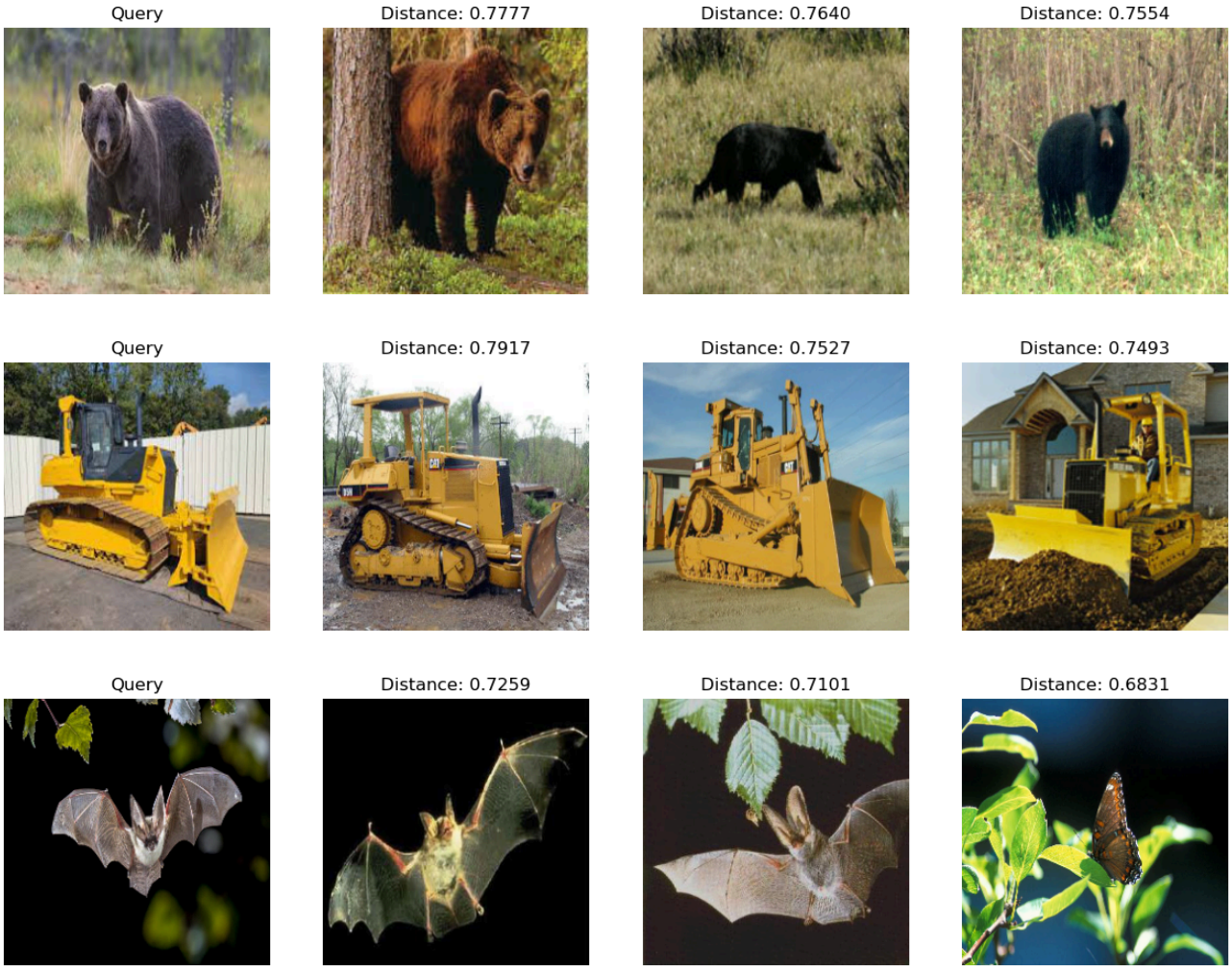
Source: https://www.researchgate.net/figure/An-illustration-of-ResNet-50-layers-architecture_fig1_350421671

Trained on the large-scale ImageNet dataset, ResNet50 achieves excellent results in classification tasks. To make predictions, ResNet first generates image embeddings that capture key features. We hypothesized that these embeddings could serve as an effective vector space for calculating image similarity.

During our experiments, we treated ResNet embeddings as a feature extractor. We explored three approaches:

- Using raw ResNet embeddings for image similarity calculations.
- Fine-tuning ResNet's weights with a small learning rate using triplet loss.
- Training a single dense layer to compress ResNet's 2048-dimensional embedding vector into a 512-dimensional one while optimizing triplet loss.

Unfortunately, due to the high intra-class variability in our dataset, all attempts to optimize ResNet's embedding space performed worse than using raw ResNet embeddings. Below are example predictions based on ResNet's extracted embedding space.



Model Analysis

To extract only the embedding extraction part of a resnet we used the following code snippet:

In [ ]:
```
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(256, 256, 3))

base_model.trainable = False #Turn to True if you wnat to retrain whole model
```

```
x = base_model.output
x = GlobalAveragePooling2D()(x)

embedding_model = Model(inputs=base_model.input, outputs=x)
```

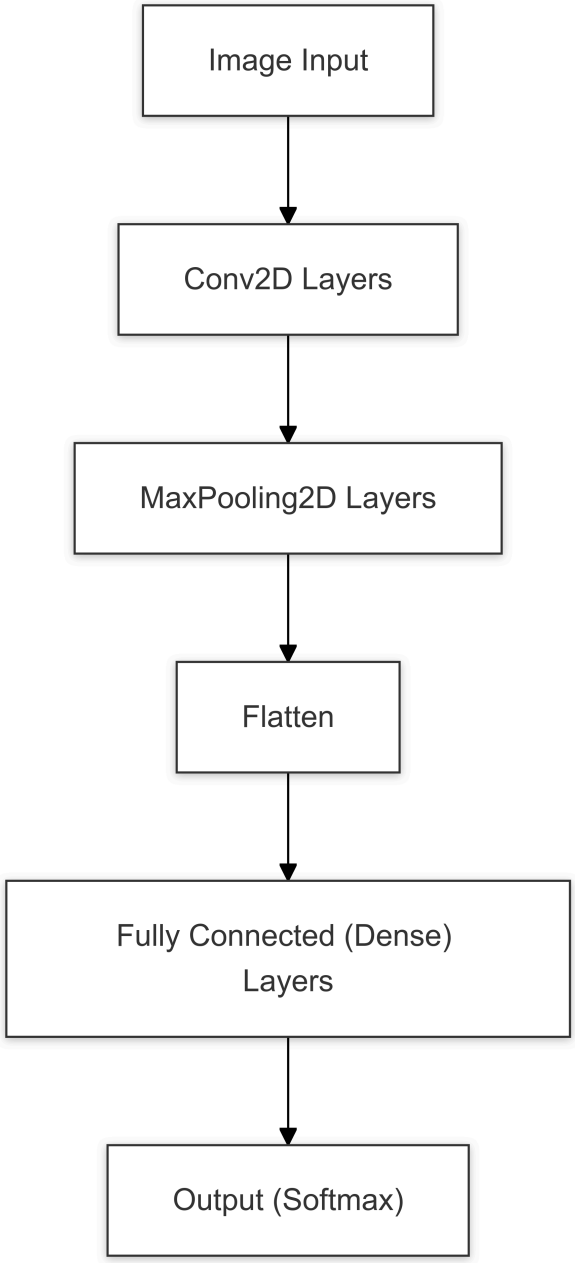
```
Total params: 23,587,712 (89.98 MB)

Trainable params: 0 (0.00 B)

Non-trainable params: 23,587,712 (89.98 MB)
```

After adding a normalization layer, the ResNet50 feature extractor could then be used in experiments with the contrastive network to attempt fine-tuning the weights.

3.2 Architectures - Final

Classification Convolutional Neural Network

We developed a **convolutional neural network (CNN)** designed specifically to classify images into 106 distinct categories.



This approach consisted of steps:

1. **Train and Validation Split** The dataset was divided into training and validation sets to facilitate the training process. Since the test data will be provided separately for querying the application, we excluded a test split during this stage. For each class (corresponding to a folder), 90% of the images were used for training, and 10% were allocated for validation. At this point each image was assigned a label derived from the folder name (e.g., `001`, `002`, etc.).
2. **Data Generator**, a custom data generator was implemented to preprocess images during training. This generator handled the following tasks:
   - **Loading and Normalization**: Images were read and normalized to have pixel values in the desired range.
   - **Resizing**: Images were resized by cropping instead of stretching to preserve the original aspect ratio.
   - **Augmentation**: Gaussian blur was applied to enhance robustness to noise.
   - **Batch Generation**: The generator returned batches of preprocessed images and their corresponding one-hot encoded labels. Shuffling was applied at the end of each epoch to ensure randomness and reduce overfitting.

3. **Classification CNN Model**, the CNN model was designed to extract features and classify images into their respective categories. The architecture consisted of convolutional and pooling layers, followed by fully connected layers for classification. The final output layer used a **softmax activation** to produce a probability distribution over the 106 categories.
4. **Embedding Model Extraction**, to facilitate similarity-based querying, we extracted the second-to-last layer of the trained classification model. This layer produced a **512-dimensional feature vector** (embedding) for each image.
5. **Similarity Computation**, once embeddings were generated, we used them for similarity-based image retrieval.
    - For a given query image, its embedding was predicted using the embedding model.
    - **Cosine similarity** was computed between the query embedding and all database embeddings.
    - The top-`k` most similar images were identified and returned based on the similarity scores.

This approach achieved satisfactory performance, with the CNN producing accurate classifications in most cases. When used for similarity-based image retrieval, the results were generally very reliable. The retrieved images often belonged to the same category as the query, and even in cases where they did not, the retrieved images were visually similar.

Here are a few example outputs demonstrating the results:

In [ ]:
```python
def build_classification_network(input_shape, num_classes):
    inputs = Input(shape=input_shape)

    x = Conv2D(128, (3, 3), activation='relu', padding='same')(inputs)
    x = MaxPooling2D(pool_size=(2, 2))(x)
    x = BatchNormalization()(x)

    x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D(pool_size=(2, 2))(x)
    x = BatchNormalization()(x)

    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
```

```
    x = MaxPooling2D(pool_size=(2, 2))(x)
    x = BatchNormalization()(x)

    x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D(pool_size=(2, 2))(x)
    x = BatchNormalization()(x)

    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D(pool_size=(2, 2))(x)

    x = Flatten()(x)
    x = Dense(1024, activation='relu')(x)
    x = Dropout(0.3)(x)
    x = Dense(512, activation='relu')(x)
    x = Dropout(0.3)(x)

    outputs = Dense(num_classes, activation='softmax')(x)

    model = Model(inputs, outputs)
    model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy',
                                                                               Precision(name=
                                                                               Recall(name='re

    return model
```

Model Analysis

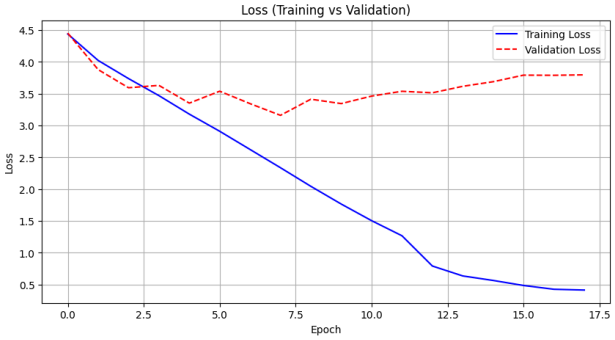
- **Memory Size**: The model occupies approximately 7.16 MB of memory.
- **Total Parameters**: The model has a total of 1,878,234 parameters.
- **Trainable Parameters**: Out of the total, 1,877,530 parameters are trainable.
- **Non-Trainable Parameters**: There are 704 non-trainable parameters, which are responsible for fixed operations such as batch normalization statistics.

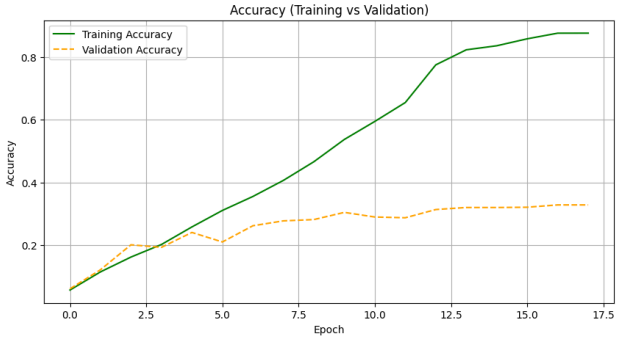
```
Total params: 1,878,234 (7.16 MB)

Trainable params: 1,877,530 (7.16 MB)

Non-trainable params: 704 (2.75 KB)
```

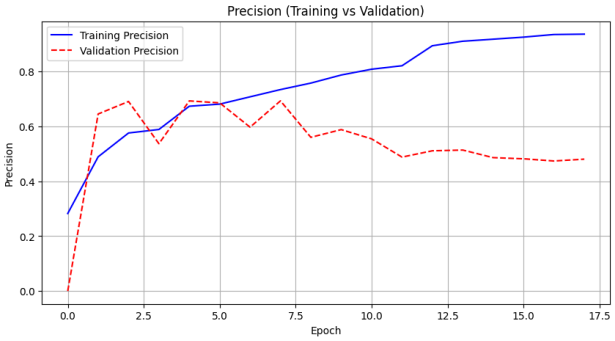
Training Analysis

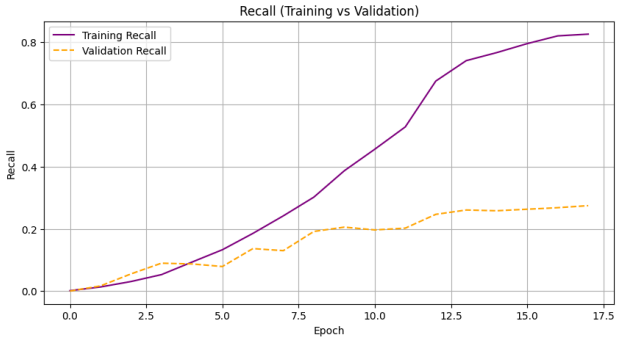
Epoch 12 - The learning rate was adjusted by the scheduler.

A callback function was used to monitor the validation loss and restore the model's best weights.









Metrics

- **Categorical Crossentropy** - used as the loss function for multi-class classification.
- **Accuracy** - measures the percentage of correct predictions.
- **Precision** - evaluates the proportion of true positive predictions among all positive predictions.
- **Recall** - assesses the proportion of true positives identified from all actual positives.

Hyperperameters

- **Optimizer Hyperparameters**
  - **Learning Rate** - statred with the default value of 0.001, and a shceduler was introduced, to reduce the learning rate by factor of 0.2 on plateu.
- **Training Hyperparameters**
  - **Batch Size** - a batch size of 32 was used, as it is commonly used, due to the fact that larger batches may increase the memory usage, and size 32 shown to be sufficient, smaller batches may introduce more noise.
  - **Number of Epochs** - the number of epochs was set to 50, but due to early stopping monitoring the validation loss, the training process halted eariler.
- **Callbacks**
  - **EarlyStopping** - stops the training and restores best weights.
  - **ReduceLROnPlateau** - reduces the learning rate.

Inference Time - 2.384185791015625e-07 s (predict embedding for a query)

Training Time - 75s * 18 epochs = approx 22.5 minutes

Runtime Environment The model was trained in a Kaggle runtime environment utilizing dual NVIDIA Tesla T4 GPUs.

4. GUI

For better result visualization, we created a Streamlit-based application. It allows users to select a trained model that we deemed successful and upload any image from their storage to see its closest similarity matches in our dataset.

Completed Items

Item	Points
Problem (Search Engine)	2
Model Own Architecture (Classification CNN)	2
Solving Additional Problem (Classification)	1
Model Own Architecture (Siamese NN)	2
Model Non-Trivial Solution (Contrastive Learning - Siamese NN)	1
Model Transfer Learning (ResNet)	1
Dataset (10,000+ photos)	1
Dataset (Own part of dataset 500>)	1
Architecture Tuning (3 architectures)	1
Tools (GUI)	1
Total	13

Bibliography

- Dataset:

  https://www.kaggle.com/datasets/jessicali9530/caltech256/data

  Griffin, G., Holub, A., & Perona, P. (2022). Caltech 256 (1.0) [Data set]. CaltechDATA. https://doi.org/10.22002/D1.20087

- Siamese Network Guide

  https://builtin.com/machine-learning/siamese-network

  https://medium.com/@rinkinag24/a-comprehensive-guide-to-siamese-neural-networks-3358658c0513

- Contrastive Network with Triplet Loss Guides

  https://encord.com/blog/guide-to-contrastive-learning/

  https://www.v7labs.com/blog/contrastive-learning-guide

- ResNet50 explained

  https://blog.roboflow.com/what-is-resnet-50/

  https://wandb.ai/mostafaibrahim17/ml-articles/reports/The-Basics-of-ResNet50---Vmlldzo2NDkwNDE2