TensorFlow VGG16-Based Image Classification Model for Human Detection

AI Assistant

February 3, 2025

1 Overview

This document details the implementation of a binary image classification model using TensorFlow and Keras, with a focus on human detection. The model employs transfer learning, utilizing a pre-trained VGG16 model as a feature extractor. The document covers the dataset used, the rationale behind transfer learning, code structure, model training, and inference process.

2 Dataset

The dataset used for training was sourced from Kaggle: Human Detection Dataset.

3 Rationale for Using a Pre-Trained VGG16 Model

When working with limited datasets, it is often more effective to use a pre-trained model such as VGG16, as opposed to training a model from scratch. Below are the key reasons:

3.1 1. Transfer Learning

Pre-trained models like VGG16 are trained on large and diverse datasets, such as ImageNet. These models learn generic features like edges, textures, and shapes that are relevant across various image types. Transfer learning leverages this knowledge by reusing the learned features for a different task, such as our human detection. This allows our model to achieve high performance with relatively little data.

3.2 2. Overfitting Avoidance

Small datasets are susceptible to overfitting, where the model memorizes the training data rather than generalizing to new data. By using VGG16, which

has been trained on a large dataset, the risk of overfitting is reduced as its lower layers already contain useful and general features.

3.3 3. Reduced Training Time and Resource Requirements

Training a deep neural network from scratch requires extensive computational resources and time. Pre-trained models significantly reduce this requirement as the majority of the learning has already been done. Fine-tuning a pre-trained model reduces both training time and the required computational resources.

3.4 4. Improved Performance with Limited Data

Models trained from scratch on small datasets often perform poorly because of an insufficient amount of data. Fine-tuning a pre-trained model allows us to achieve better performance with limited data.

4 Conclusion

Using a pre-trained model like VGG16 with transfer learning techniques is an effective way to train image classifiers when a small dataset is available. This methodology offers better performance, reduces the risk of overfitting, and requires less training time and computational resources, making it an ideal choice for our project.

5 Dependencies

The following dependencies are required to run the training and inference scripts:

- **TensorFlow**: Provides the core framework for machine learning.
- **Keras**: High-level API for building and training models.
- **NumPy**: Used for numerical computations.
- **OpenCV (cv2)**: For image processing.
- **Python 3.7 or 3.8**: The versions most compatible with tensorflow 2.6.0
- A directory containing images (`data`) organized into folders.
- A model file named `human_detector.h5` (result of training).

6 Project Structure

The project structure is as follows:

- `main.py`: The training script.
- `use.py`: The inference script.
- `data`: Directory for training images (subfolders for human/no-human).
- `evaluate`: Directory for new images that the trained model will classify.
- `human_detector.h5`: Trained model weights.

7 Features

- GPU acceleration, with dynamic memory management.
- Image loading and preprocessing with resizing and normalization.
- Splitting of training/validation sets, ensuring optimal data loading with prefetching.
- Pre-trained VGG16 base model with layers frozen for transfer learning.
- Custom classification layers, including dropout to prevent overfitting.
- Model training for a set number of epochs (default 10), with model saving at the end.
- Separate inference script (`use.py`) for easy classification of new images.

8 Code Breakdown: Training Script (`main.py`)

8.1 1. Environment Configuration

Listing 1: Environment Configuration

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
```

This snippet suppresses less important TensorFlow logs, improving clarity.

8.2 2. Importing Required Libraries

Listing 2: Importing Required Libraries

```
import tensorflow as tf
from tensorflow import keras
from keras import layers, models, regularizers
from tensorflow.keras.applications.vgg16 import VGG16
```

Imports core TensorFlow and Keras modules for model building and training.

8.3 3. GPU Configuration

Listing 3: GPU Configuration

```
physical_devices = tf.config.experimental.list_physical_devices('GPU')
tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

This configures TensorFlow to use GPU memory efficiently by dynamically growing as needed.

8.4 4. Image Preprocessing Function

Listing 4: Image Preprocessing Function

```
def preprocess_image(image, label):
    image = tf.image.resize_with_pad(image, 256, 256)
    image = image / 255.0  # Normalize pixel values
    return image, label
```

This function resizes images to a uniform 256x256 pixels and normalizes pixel values.

8.5 5. Loading and Preprocessing Dataset

Listing 5: Loading and Preprocessing Dataset

```
data = tf.keras.utils.image_dataset_from_directory('data', shuffle=True)
data = data.map(preprocess_image)
```

Loads images from the `data` directory, shuffling them and applying preprocessing.

8.6 6. Splitting the Dataset

Listing 6: Splitting the Dataset

```
train_size = int(len(data) * .8)
val_size = int(len(data) * .2)

train = data.take(train_size)
val = data.skip(train_size).take(val_size)
```

Splits the dataset into an 80/20 training and validation split.

8.7 7. Optimizing Data Loading

Listing 7: Optimizing Data Loading

```
AUTOTUNE = tf.data.AUTOTUNE
train = train.prefetch(buffer_size=AUTOTUNE)
val = val.prefetch(buffer_size=AUTOTUNE)
```

Optimizes loading data with prefetching.

8.8 8. Defining the Model

Listing 8: Defining the Model

```
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(256, 256, 3))
base_model.trainable = False
```

Loads the pre-trained VGG16 model, excluding the top layer, and freezes its weights.

8.9 9. Adding Custom Classification Layers

Listing 9: Custom Classification Layers

```
model = models.Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dense(128, activation='relu'),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.25),
    layers.Dense(1, activation='sigmoid')
])
```

Adds custom classification layers on top of VGG16.

8.10 10. Compiling and Training the Model

Listing 10: Compiling and Training

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(train, epochs=10)
model.save('human_detector.h5')
```

Compiles the model using the Adam optimizer, binary cross-entropy loss, trains it for 10 epochs, and saves it to `human_detector.h5`.

9 Code Breakdown: Inference Script (`use.py`)

The `use.py` script is designed for testing the trained model on new images.

9.1 1. Usage

To run the script, execute the following command in your terminal:

```
python use.py
```

The script will process images from the "evaluate" folder, displaying classification results.

9.2 2. Code

Listing 11: Inference Code

```
import os
import tensorflow as tf
from tensorflow import keras
import cv2
import numpy as np

print(tf.keras.__version__)

def load_model(model_path):
    try:
        model = tf.keras.models.load_model(model_path)
        return model
    except Exception as e:
        print(f"Error: {e}")
        return None

def preprocess_image(image_path):
    try:
        img = cv2.imread(image_path)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (256, 256))
        img = img / 255.0
        img = np.expand_dims(img, axis=0)
        return img
    except Exception as e:
        print(f"Error: {e}")
        return None


def predict_image(model, image_path):
    try:
        img = preprocess_image(image_path)
        if img is None:
            return None
        prediction = model.predict(img)[0][0]
        return prediction
    except Exception as e:
        print(f"Error: {e}")
        return None


def detect_humans(model_path, image_folder):
    model = load_model(model_path)
    if model is None:
        return

    for filename in os.listdir(image_folder):
        if filename.lower().endswith(('.png', '.jpg', '.jpeg')):
            image_path = os.path.join(image_folder, filename)
            prediction = predict_image(model, image_path)

            if prediction is None:
                continue
            if prediction > 0.5:
```

```
                print(f"Image␣{filename}:␣Human␣detected")
            else:
                print(f"Image␣{filename}:␣Human␣not␣detected")


if __name__ == "__main__":
    model_path = 'human_detector.h5'
    image_folder = 'evaluate'
    detect_humans(model_path, image_folder)
```

This script loads the trained model, preprocesses images, and outputs a classification result for each image in the specified folder, indicating if humans were detected.

10 Conclusion

This document describes the steps, code, and design decisions behind building an image classifier using a pre-trained VGG16 model. Using transfer learning greatly reduces the resources and time required to build a robust image classification model, while also providing a high degree of accuracy.