Blood cell type Prediction using MobileNet-v2:

Réalisé par:

Hiba El Idrissi Elabkary

Contents:

Introduction

As part of my project, which falls under the "Deep Learning" module, I have chosen Project B. Specifically, I have decided to work on predicting the type of blood cells using the MobileNet-V2 model. The following report will present the different steps and tools used in my project.

Around Transfer Learning

Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task.

A teacher has years of experience in the particular topic he/she teaches. With all this accumulated information, the lectures that students get is a concise and brief overview of the topic. So it can be seen as a “transfer” of information from the learned to a novice.

Keeping in mind this analogy, we compare this to neural network. A neural network is trained on a data. This network gains knowledge from this data, which is compiled as “weights” of the network. These weights can be extracted and then transferred to any other neural network. Instead of training the other neural network from scratch, we **“transfer”** the learned features.

Fine-Tuning Pre-trained models:

Fine-tuning is the process of taking a model previously trained on a dataset, and adapting it to a more specialized dataset / task. Typically the original dataset is very large and very general (for example: a crawl of a large portion of the public Internet), and consequently the models are very large in order to reason about all this information (billions of parameters or more).

One of the main ways to fine-tune a model:

1. **Feature extraction** – We can use a pre-trained model as a feature extraction mechanism. What we can do is that we can remove the output layer(the one which gives the probabilities for being in each of the 1000 classes) and then use the entire network as a fixed feature extractor for the new data set.
2. **Use the Architecture of the pre-trained model –** What we can do is that we use architecture of the model while we initialize all the weights randomly and train the model according to our dataset again.
3. **Train some layers while freeze others** – Another way to use a pre-trained model is to train is partially. What we can do is we keep the weights of initial layers of the model

frozen while we retrain only the higher layers. We can try and test as to how many layers to be frozen and how many to be trained.

Scenario 1 – Size of the Data set is small while the Data similarity is very high – In this case, since the data similarity is very high, we do not need to retrain the model. All we need to do is to customize and modify the output layers according to our problem statement. We use the pre-trained model as a feature extractor. Suppose we decide to use models trained on ImageNet to identify if the new set of images have cats or dogs. Here the images we need to identify would be similar to ImageNet, however we just need two categories as my output – cats or dogs. In this case all we do is just modify the dense layers and the final softmax layer to output 2 categories instead of a 1000.

Scenario 2 – Size of the data is small as well as data similarity is very low – In this case we can freeze the initial (let's say k) layers of the pre-trained model and train just the remaining (n-k) layers again. The top layers would then be customized to the new data set. Since the new data set has low similarity it is significant to retrain and customize the higher layers according to the new dataset. The small size of the data set is compensated by the fact that the initial layers are kept pre-trained (which have been trained on a large dataset previously) and the weights for those layers are frozen.

Scenario 3 – Size of the data set is large however the Data similarity is very low – In this case, since we have a large dataset, our neural network training would be effective. However, since the data we have is very different as compared to the data used for training our pre-trained models. The predictions made using pre-trained models would not be effective. Hence, it's best to train the neural network from scratch according to your data.

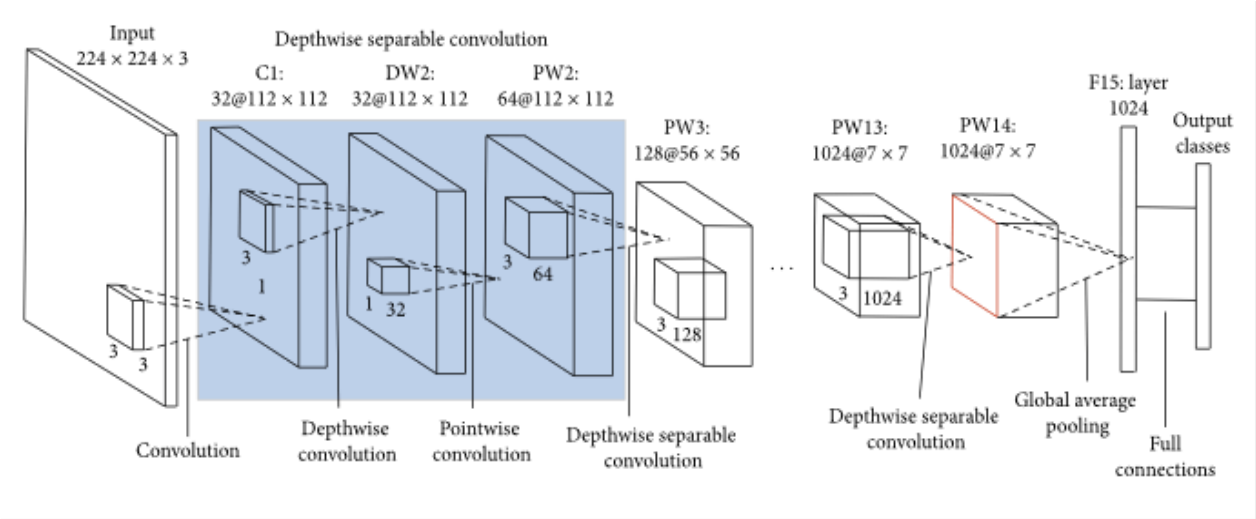
Scenario 4 – Size of the data is large as well as there is high data similarity – This is the ideal situation. In this case the pre-trained model should be most effective. The best way to use the model is to retain the architecture of the model and the initial weights of the model. Then we can retrain this model using the weights as initialized in the pre-trained model.

Around MobileNet-v2

MobileNet-v2 is a convolutional neural network that is 53 layers deep. You can load a pre-trained version of the network trained on more than a million images from the ImageNet database. The pre-trained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the network has learned rich feature representations for a wide range of images. The network has an image input size of 224-by-224.

The MobileNet-v2 model is based on an inverted residual structure where the residual connections are between the bottleneck layers. The intermediate expansion layer uses lightweight depthwise convolutions to filter features as a source of non-linearity. As a whole,

the architecture of MobileNetV2 contains the initial fully convolution layer with 32 filters, followed by 19 residual bottleneck layers.



Blood Cell Images Dataset

The Dataset used for Blood Cell Type Prediction is the kaggle Dataset “Blood Cell Images”. This dataset contains 12,500 augmented images of blood cells (JPEG) with accompanying cell type labels (CSV). There are approximately 3,000 images for each of 4 different cell types grouped into 4 different folders (according to cell type). The cell types are Eosinophil, Lymphocyte, Monocyte, and Neutrophil.

Project Steps

Libraries Used

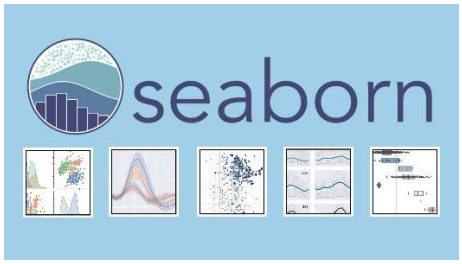
Numpy

NumPy is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices.

Matplotlib

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

Seaborn



Seaborn is a Python data visualization library based on Matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

Tensorflow

TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries, and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML-powered applications.

TensorFlow was originally developed by researchers and engineers working within the Machine Intelligence team at Google Brain to conduct research in machine learning and neural networks. However, the framework is versatile enough to be used in other areas as well.

TensorFlow provides stable Python and C++ APIs, as well as a non-guaranteed backward compatible API for other languages.

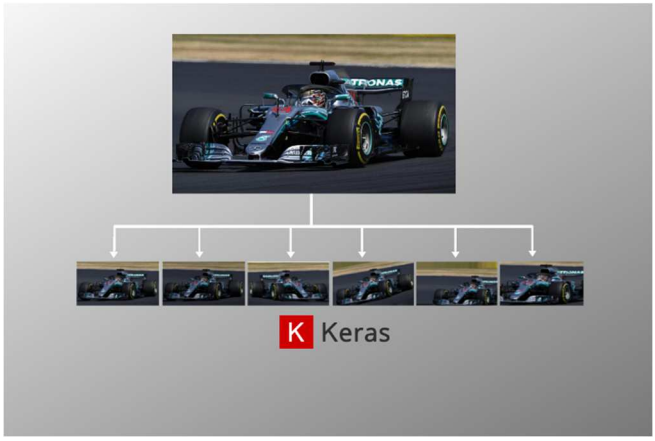
Scikit-learn:

Scikit-learn is a Python module for machine learning built on top of SciPy and is distributed under the 3-Clause BSD license.

Data Preprocessing

Image pre-processing with generators for image augmentation:

Image augmentation is a technique of applying different transformations to original images which results in multiple transformed copies of the same image. Each copy, however, is different from the other in certain aspects depending on the augmentation techniques you apply like shifting, rotating, flipping, etc.
Applying these small amounts of variations on the original image does not change its target class but only provides a new perspective of capturing the object in real life. And so, we use it is quite often for building deep learning models.
These image augmentation techniques not only expand the size of my dataset but also incorporate a level of variation in the dataset which allows my model to generalize better on unseen data. Also, the model becomes more robust when it is trained on new, slightly altered images.



Keras ImageDataGenerator class provides a quick and easy way to augment my images. It provides a host of different augmentation techniques like standardization, rotation, shifts, flips, brightness change, and many more.

ImageDataGenerator class ensures that the model receives new variations of the images at each epoch. But it only returns the transformed images and does not add it to the original

corpus of images. If it was, in fact, the case, then the model would be seeing the original images multiple times which would definitely overfit our model.

```
[6] train_gen = tf.keras.preprocessing.image.ImageDataGenerator(preprocessing_function=tf.keras.applications.mobilenet_v2.preprocess_input, validation_split=0.2)
    test_gen = tf.keras.preprocessing.image.ImageDataGenerator(preprocessing_function=tf.keras.applications.mobilenet_v2.preprocess_input)
```

```
train_images=train_gen.flow_from_directory(
    directory=train_dir,
    target_size=(224, 224),
    color_mode='rgb',
    class_mode='categorical',
    batch_size=32,
    shuffle=True,
    seed=42,
    subset='training'
)

val_images=train_gen.flow_from_directory(
    directory=train_dir,
    target_size=(224, 224),
    color_mode='rgb',
    class_mode='categorical',
    batch_size=32,
    shuffle=True,
    seed=42,
    subset='validation'
)

test_images=test_gen.flow_from_directory(
    directory=test_dir,
    target_size=(224, 224),
    color_mode='rgb',
    class_mode='categorical',
    batch_size=32,
    shuffle=False,
    seed=42
)
```

```
Found 7968 images belonging to 4 classes.
Found 1989 images belonging to 4 classes.
Found 2487 images belonging to 4 classes.
```

Pre-trained Model Building:

```
pretrained_model = tf.keras.applications.MobileNetV2(
    input_shape=(224, 224, 3),
    include_top=False,
    weights='imagenet',
    pooling='avg'
)

pretrained_model.trainable = False
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_224_no_top.h5
9406464/9406464 [==============================] - 0s 0us/step
```

As mentioned before, the model to pre-train for this project is the MobileNet-V2. In the code snippet above, I created an instance of the MobileNetV2 model by calling the **tf.keras.applications.MobileNetV2** function. The parameters I passed to the function are:

input_shape=(224, 224, 3): Specifies the shape of the input images that the model will expect. In this case, it's a 224x224 pixel RGB image (3 channels).

include_top=False: This parameter determines whether to include the fully connected layer (top) of the model. Setting it to **False** means that the final classification layer of the MobileNetV2 model will be excluded, which is useful when you want to use the model for feature extraction rather than classification.

weights='imagenet': Specifies the source of pre-trained weights for the model. In this case, you are using the weights trained on the ImageNet dataset. These weights are useful for transfer learning.

pooling='avg': Specifies the type of global pooling to be applied to the feature maps before the classification layer. Setting it to 'avg' means that global average pooling will be used.

Classification Model Building:

```
inputs=pretrained_model.input
x=tf.keras.layers.Dense(128, activation='relu')(pretrained_model.output)
outputs=tf.keras.layers.Dense(4, activation='softmax')(x)
model = tf.keras.Model(inputs=inputs, outputs=outputs)

model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
print(model.summary())
```

The code snippet above continues building a model on top of the “pre-trained model” using **TensorFlow’s Keras** API.

I started defining the input of the new model as the input of the pre-trained model (which is the placeholder for the input images). Then, I passed the output of the pre-trained-model through a **Dense layer** with 128 units and **ReLU** activation. This adds a new trainable layer on top of the pre-trained layers. Finally, I passed the output of the previous layer through another **Dense layer** with 4 units (assuming you have 4 classes) and a **softmax activation** function, which is commonly used for multi-class classification tasks.

Then, I compiled the model using the compile method. I specified the optimizer as **adam**, which is a popular optimization algorithm. The loss function is set to **categorical_crossentropy**, which is commonly used for multi-class classification problems. Lastly, I defined the metric to be tracked during training as **accuracy**.

Training:

```
history=model.fit(
    train_images,
    validation_data=val_images,
    epochs=100,
    callbacks=[tf.keras.callbacks.EarlyStopping(
            monitor='val_loss',
            patience=4,
            restore_best_weights=True)])
```

After building the pre-trained model and the classification model to be used, I needed to train the whole model. The code snippet above shows the training model I used.

train_images is the input data for training the model. It should be a NumPy array or a **TensorFlow tf.data.Dataset** containing the training images.

validation_data=val_images specifies the validation data for evaluating the model's performance during training. **val_images** is usually a separate set of images used to monitor the model's generalization ability.

epochs=100 indicates the number of times the entire training dataset will be passed through the model during training.

callbacks is a list of callbacks to be applied during training. In this case, the code includes the tf.keras.callbacks.EarlyStopping callback.

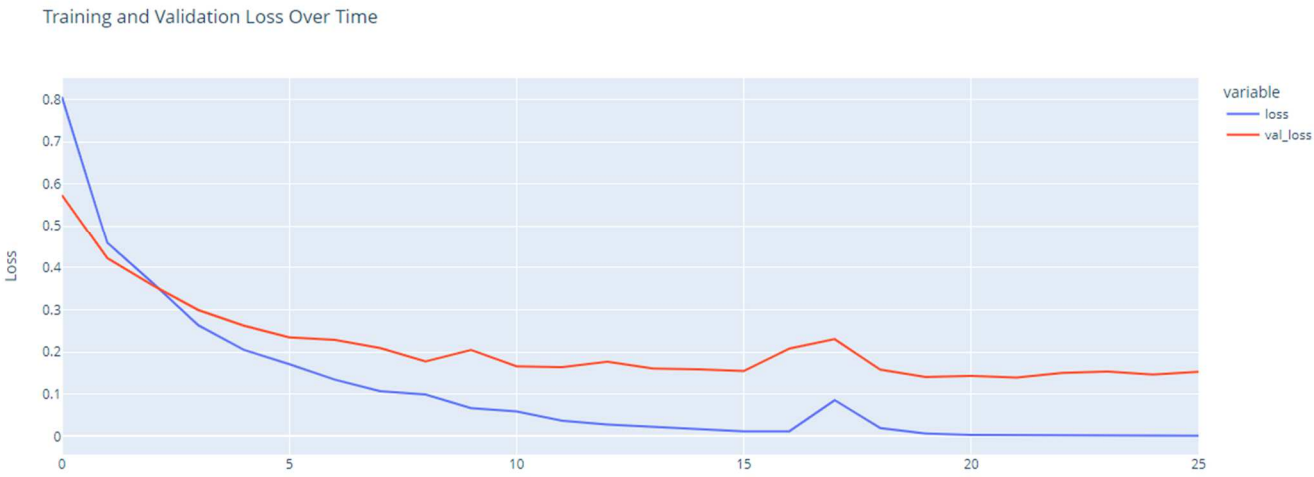
monitor='val_loss' specifies that the early stopping callback should monitor the validation loss. It will stop training if the validation loss does not improve.

patience=4 sets the number of epochs with no improvement in the monitored quantity (validation loss) after which training will be stopped.

restore_best_weights=True ensures that the model's weights are restored to the best observed values based on the validation loss when training is stopped.

Results

Training and **Validation Loss** variation over time.



Test accuracy:

```
predictions = np.argmax(model.predict(test_images), axis=1)

acc = accuracy_score(test_images.labels, predictions)
cm = tf.math.confusion_matrix(test_images.labels, predictions)
clr = classification_report(test_images.labels, predictions, target_names=CLASS_NAMES)

print("Test Accuracy: {:.3f}%".format(acc * 100))
```

The code above enables us to get the test accuracy of the model which is equal to: 52.795%

Classification report after testing the model on the "test images":

```
Classification Report:
----------------------
              precision    recall  f1-score   support

  EOSINOPHIL       0.46      0.54      0.50       623
  LYMPHOCYTE       0.78      0.48      0.60       620
    MONOCYTE       0.79      0.33      0.47       620
  NEUTROPHIL       0.42      0.76      0.54       624

    accuracy                           0.53      2487
   macro avg       0.61      0.53      0.53      2487
weighted avg       0.61      0.53      0.53      2487
```

Classification report after testing the model on the "validation images":

```
Classification Report:
----------------------
              precision    recall  f1-score   support

  EOSINOPHIL       0.94      0.91      0.93       499
  LYMPHOCYTE       0.98      0.98      0.98       496
    MONOCYTE       0.98      0.98      0.98       495
  NEUTROPHIL       0.92      0.95      0.94       499

    accuracy                           0.96      1989
   macro avg       0.96      0.96      0.96      1989
weighted avg       0.96      0.96      0.96      1989
```

Model Deployment with Gradio

Gradio is a free and open-source Python library that allows you to develop an easy-to-use customizable component demo for machine learning models that anyone can use anywhere. Gradio integrates with the most popular Python libraries, including Scikit-learn, PyTorch, NumPy, seaborn, pandas, Tensor Flow, and others.
One of its advantages is that it allows you to interact with the web app you are currently developing in your Jupyter or Colab notebook. It has a lot of unique features that can help you construct a web app that users can interact with.

Deployment Results

After deploying my machine learning model on Gradio and testing with an image, I tested with a random image of a Lymphocyte Blood cell I got from the internet:

As it is shown in the image above, the blood cell type was correctly predicted.

Resources:

https://www.analyticsvidhya.com/blog/2017/06/transfer-learning-the-art-of-fine-tuning-a-pre-trained-model/

https://machinelearningmastery.com/transfer-learning-for-deep-learning/

https://ludwig.ai/latest/user_guide/distributed_training/finetuning/

https://paperswithcode.com/method/mobilenetv2

https://www.mathworks.com/help/deeplearning/ref/mobilenetv2.html

https://www.kaggle.com/datasets/paultimothymooney/blood-cells

https://www.analyticsvidhya.com/blog/2020/08/image-augmentation-on-the-fly-using-keras-imagedatagenerator/

https://www.freecodecamp.org/news/how-to-deploy-your-machine-learning-model-as-a-web-app-using-gradio/

https://www.kaggle.com/code/gcdatkin/blood-cell-type-prediction/notebook