LAB 03 - Transfer Learning

The main purpose of this laboratory is to cover the essential tf.keras tools for transfer learning and iterate through a few experiments, additionally, you will also have the opportunity to revise what you've learned so far.

Clearly seeing the impact of transfer learning requiers GPU power, which the computers found here cannot provide, additionally, training a CNN on cifar10 for a single epoch on Google Colab takes about 7 minutes. Since we cannot fully train a base model for comparison and run multiple experiments within the time frame of this laboratory we will not be focusing on the performance of our models, instead, we will be training the models for few itarations and focus on going through all the examples in order the aquire the neccesary knowledge. You are encouraged to go through the Tensorflow GPU [1] setup at home, provided you have a resource of this kind available, run similar experiments fully and use those techniques in your project.

The experiments will go as follows:

1. Building a base model for classification
2. Loading the previously built classifier, replacing the layers following the convolutions and training it on a new task
3. Loading a tf.keras model pretrained on a large dataset as a base model
4. Freezing the layers from the base model and training only the newly added ones
5. Gradually unfreezing the layers from the base model

[1] `https://www.tensorflow.org/install/gpu`

Building a base model

Using the knowledge you have gathered during the first and second laboratories, build a classifier for the *tf_flowers* dataset.
The workflow should generally look like this:

1. Load the *tf_flowers* dataset using the tensorflow_datasets package. Go through the metadata in order to find out the number of classes.

2. Define a function that resize each image to (32, 32), casts it to float and downscales the pixel values by 255. Map this function over the dataset.

3. Define a convolutional neural network with the following architecture:

   - 3 convolutional layers with 5x5 kernels, same padding and relu activation
   - 2 fully connected layers, the first one having 128 neurons and relu activation while the latter should be used for classification and use softmax activation

4. Train the network for a few iterations and save the model.

Transfering to a new task

Start by loading the model you have previously trained and the cifar10 dataset. tf.keras allows you to access all model layers; in order to see them you can simply

```
print(model.layers)
```

You can iterate through the layers to see their properties and reuse them in any way you see fit. For our experiment, we will keep all the convolutional and max pooling layers from the base model. On top of them we will define two new dense layers and train the entire model on the new dataset. The following code illustrates the process.

```
# if you first model followed the specified architecture,
# the flattening layer applied after the convolutions
# should be model.layer[-3]
# to get its output we simply use model.layers[-3].output
conv_output = base_model.layers[-3].output

# we now build the new layers on top of it
fc_1 = layers.Dense(128, activation = 'relu')(conv_output)
fc_2 = layers.Dense(10, activation = 'softmax')(fc_1)

# and define a new model that outputs the results from our newly created layers
new_model = tf.keras.Model(inputs = base_model.inputs, outputs = fc_2)
new_model.compile(
    optimizer = tf.keras.optimizers.Adam(),
    loss = tf.keras.losses.sparse_categorical_crossentropy,
    metrics = ['accuracy']
)
```

You can now train the new model on the cifar10 dataset. Alternatively, if you have computing power, you can fully train a network on any dataset and then use this type of transfer to see if you can improve your accuracy, you should see a score improvement if the network from which you are transfering the knowledge was trained on a similar task or on a very large set of data.

Loading keras pretrained models

Some of the best results in transfer learning are obtained by starting from very deep models trained on a large scale dataset. This process is simplified in the case of well known models. Loading a ResNet50 pretrained on ImageNet (a dataset with 1000 classes and milions of traning samples) is as easy as:

```
model = tf.keras.applications.ResNet50(
    weights = 'imagenet',
```

```
    include_top = False
)
```

This model already has the fully connected layers removed and has the ImageNet weights preloaded. The entire tensorflow model zoo can be found here: `https://www.tensorflow.org/api_docs/python/tf/keras/applications`. Each model can have different parameters to be set but generally the *include_top* and *weights* parameters are all you need.

If you repeat the procedure from the previous section by resizing the input images to (224, 224), applying a global average pooling and a fully connected layer, you should probably observe that it takes a considerable amount of time to complete and single epoch. The reason is you are now training the entire model. In practice, some of the time you might want to train only the newly added layers and leave the convolutional features as they are. This technique is called layer **freezing**.

Freezing layers in tf.keras

You can specify whether the variables from a layer should be updated or not during the training process by setting the layers *trainable* attribute. For instance you can see that all the layers from the loaded model have this paremeter set to True if you iterate through them.

```
for layer in model.layers:
    print(layer.trainable)
```

If you want to freeze them (i.e. not update their weights during training) you can set the parameter to False.

```
for layer in model.layers:
    layer.trainable = False
```

Then you can add your layers on top of the frozen model. Should you do this, you should observe a considerable diffrence in training time.

Gradually unfreezing layers

For some tasks, the weights learned by the base network are not appropriate enough to constitute a final solution and you might have to update the entire network during training. To avoid training a network that has a highly refined first set of layers and a randomly initialized couple of last layers, a suggested approach is to fine tune the added layers and then gradually unfreeze the layers starting from the top as the training progresses.

As a last task you should implement this approach. You can use a smaller pretrained network such as VGG16 or set up a small amount of iterations between each un freezing.