SWS3009 Summer Workshop

Lab 4 – Transfer Learning

1. INTRODUCTION

In this lab we will be looking a very important topic called transfer learning, where we take a neural network that has already been pre-trained on data that is related, but not exactly the same as our own, and retrain it for our own data.

2. SUBMISSION

SUBMISSION DEADLINE: SUNDAY 9 JULY 2023 11.59 PM

There are a total of four questions for you to answer in this lab. Please answer these questions in the enclosed answer book called sws3009lab3ansbk.docx, and upload Canvas. This lab is worth 3 marks:

0 marks – No submission or empty submission
1 mark – Poor submission
2 marks – Acceptable submission
3 marks – Good submission

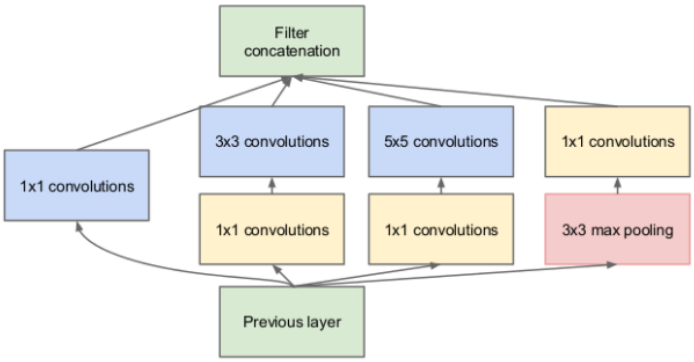
You are to work in TEAMS OF 2. You only need to upload ONE copy of the answer book per team, but ensure that both names are shown there. All submissions must be in English. Any submission not in English will not be marked.

3. TRANSFER LEARNING

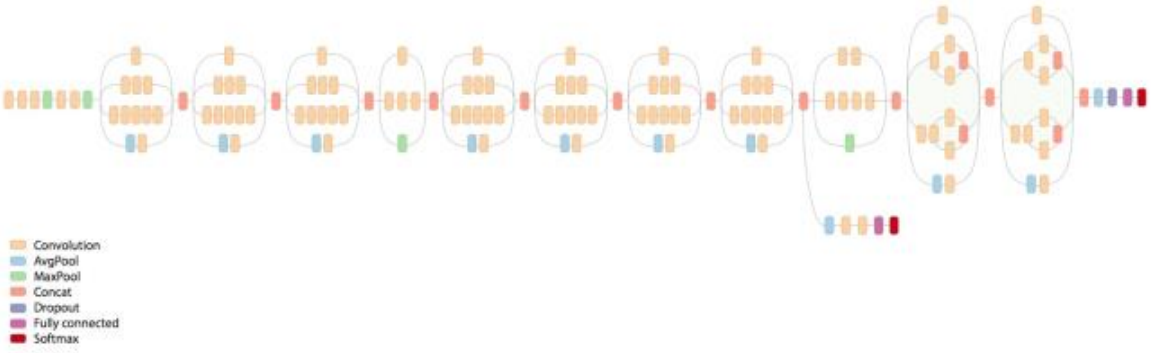
In our previous lab when we created dense and CNN networks for learning the MNIST and CIFAR-10 datasets, Keras (and TensorFlow) programs can take very, very long to run. If you were to train a CNN on a few million images on a GPU, it can take up to 2 to 3 weeks to complete training.

For this exercise we will use an very deep CNN called InceptionV3. InceptionV3 has many layers, and tens of millions of parameters to train. Here we will use a version of InceptionV3 that has been pre-trained on over 3 million images in over 4,000 categories. If we wanted to train our InceptionV3 model from scratch it would take several weeks even on a machine with several good GPUs.

InceptionV3 consists of "modules" that try out various filter and pooling configurations and chooses the best ones.



Many of these modules are chained together to produce very deep neural networks:



The greatest beauty about InceptionV3 is that someone has already taken the time to train the hundreds of layers and tens of millions of parameters. All we have to do is to adapt the top layers. This is called "Transfer Learning".

a. Getting Training Images

You need to obtain the training images for this exercise. To do so, at the LINUX or MacOS prompt type:

```
curl http://download.tensorflow.org/example_images/flower_photos.tgz | tar -xz
```

This will create a new directory called "flower_photos" with thousands of pictures of flowers sorted into 5 directories. Downloading this dataset may take some time as it is very large.

Change to the flower_photos directory and you will see that it has five directories:

```
(tensorflow) ctank@adminsocs-MacBook-Pro Lab 3 - Transfer Learning % cd flower_photos
(tensorflow) ctank@adminsocs-MacBook-Pro flower_photos % ls
LICENSE.txt     dandelion       sunflowers
daisy           roses           tulips
(tensorflow) ctank@adminsocs-MacBook-Pro flower_photos %
```

If you look in each directory you will see hundreds of images of flowers of each category. The Keras Data Generator, which we will see shortly, requires training data to be sorted like this into directories reflecting the various categories.

Now create a file called train.py and follow the instructions below.

b. Imports

We need to import the InceptionV3 model from keras.applications. We also need to import in Pooling, Dense and Dropout layers that we will need to adapt the InceptionV3 model.

Create a new file called "train.py" and key in the following code.

```
from tensorflow.keras.applications.inception_v3 import InceptionV3
from tensorflow.keras.preprocessing import image
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import SGD
import os.path

MODEL_FILE = "flowers.hd5"
```

c. Defining your Model

In this part we will do something special; we will ask TensorFlow (through Keras) to download the InceptionV3 model that has been trained on ImageNet. We will then add in new layers.

KEY IN THE FOLLOWING INTO train.py

```
def create_model(num_hidden, num_classes):

    # We get the base model using InceptionV3 and the imagenet
    # weights that was trained on tens of thousands of images.
    base_model = InceptionV3(include_top = False, weights = 'imagenet')

    # Get the output layer, then does an average pooling of this
    # output, and feeds it to a final Dense layer that we
    # will train

    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(num_hidden, activation='relu')(x)
    predictions = Dense(num_classes, activation='softmax')(x)

    # Set base model layers to be non trainable so we focus
    # our training only in the Dense layer. This lets us
    # adapt much faster and doesn't corrupt the weights that
    # were already trained on imagenet.

    for layer in base_model.layers:
        layer.trainable = False

    # Create a Functional Model (as opposed to the usual
    # Sequential Model that we create

    model = Model(inputs=base_model.input, outputs = predictions)

    return model
```

Explanation of what you keyed in (DO NOT KEY IN ANYTHING HERE)

In this code the basic InceptionV3 model is put into "base_model". In the InceptionV3 function call we specify "include_top" to be False, which allows us to specify our own Dense layer. This is important because we will retrain the Dense layer to recognize new objects. We also specify "weights='imagenet'" because we want our InceptionV3 model to be pre-trained on the millions of ImageNet images.

Let's see how this code works. The following few lines stack new layers onto the InceptionV3 model: the first is a pooling layer that takes an average of the entire layer below (to reduce dimensionality). This is followed by a dense layer, using ReLu activation, followed by an output layer "predictions" using softmax.

```
# Get the output layer, then does an average pooling of this
# output, and feeds it to a final Dense layer that we
# will train

x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(num_hidden, activation='relu')(x)
predictions = Dense(num_classes, activation='softmax')(x)
```

We only want to train the top layers, so we freeze all the layers in the base model:

```
# Set base model layers to be non trainable so we focus
# our training only in the Dense layer. This lets us
# adapt much faster and doesn't corrupt the weights that
# were already trained on imagenet.

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

Then we create a new model based on what we have loaded and modified and return that:

```
# Create a Functional Model (as opposed to the usual
# Sequential Model that we create

model = Model(inputs=base_model.input, outputs = predictions)

return model
```

Question 1.

We normally create our neural network models using "Sequential" instead of "Model". What is the difference between the two? When do we use "Sequential" and when do we use "Model"?

d. Loading an Existing Model

We will create a new function called load_existing that will load any existing model that we may have previously saved. This uses the standard load_model function that we've used in our MNSTI CNN model, but adds code that freezes the layers in the original model.

KEY IN THE FOLLOWING INTO train.py

```
# Loads an existing model file, then sets only the last
# 3 layers (which we added) to trainable.

def load_existing(model_file):

    # Load the model
    model = load_model(model_file)

    # Set only last 3 layers as trainable
    numlayers = len(model.layers)

    for layer in model.layers[:numlayers-3]:
        layer.trainable = False

    # Set remaining layers to be trainable.
    for layer in model.layers[numlayers-3:]:
        layer.trainable = True

    return model
```

Recall that we added 3 layers of our own (pooling, dense and output), so we set layers up to numlayers-3 to be non-trainable, and set layers from numlayers-3 onwards to be trainable.

e. Training the Model

This part gets complicated. We first show you the entire train function then explain each part.

KEY IN THE FOLLOWING INTO train.py

```
# Trains a model. Creats a new model if it doesn't already exist.

def train(model_file, train_path, validation_path, num_hidden=200, num_classes=5,  steps=32, num_epochs=20):

    # If an existing model exists, we load it. Otherwise we create a
    # new model from scratch

    if os.path.exists(model_file):
        print("\n*** Existing model found at %s. Loading.***\n\n" % model_file)
        model = load_existing(model_file)
    else:
        print("\n*** Creating new model ***\n\n")
        model = create_model(num_hidden, num_classes)


    # Since we have multiple categories and a softmax output
    model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

    # Create a checkpoint to save the model after every epoch
    checkpoint = ModelCheckpoint(model_file)

    # Now we create a generator. This will take our image data, rescale it,
    # shear it, zoom in and out to create additional images for training.
    train_datagen = ImageDataGenerator(\
    rescale=1./255,\
    shear_range=0.2,\
    zoom_range=0.2,\
    horizontal_flip = True)

    # Image generator for test data
    test_datagen = ImageDataGenerator(rescale=1./255)
```

```
# Now we tell the generator where to get the images from.
# We also scale the images to 249x249 pixels.

train_generator = train_datagen.flow_from_directory(\
train_path,\
target_size=(249, 249),
batch_size=5,
class_mode = "categorical")

# We do the same for the validation set.

validation_generator = test_datagen.flow_from_directory(\
validation_path,\
target_size=(249,249),
batch_size=5,
class_mode='categorical')

# Finally we train the neural network
model.fit(\
train_generator,\
steps_per_epoch = steps,\
epochs = num_epochs,\
callbacks = [checkpoint],\
validation_data = validation_generator,\
validation_steps = 50)


# Train last two layers

# Now we twek the training by freezing almost all the layers and
# just train the topmost layer
for layer in model.layers[:249]:
    layer.trainable = False

for layer in model.layers[249:]:
    layer.trainable = True

model.compile(optimizer=SGD(learning_rate=0.00001, momentum=0.9), loss='categorical_crossentropy')

model.fit(\
train_generator,\
steps_per_epoch=steps,\
epochs = num_epochs,\
callbacks = [checkpoint],\
validation_data = validation_generator,\
validation_steps=50)
```

Explanation of what you've just keyed in

Let's now look at each part (**DO NOT KEY IN ANYTHING HERE):**

We begin first by checking if the model exists, and load it using our load_existing function if it does. Otherwise we create a new model calling create_model, and compile it with a categorical cross entropy loss, and using the RMSProp optimizer. We also create a checkpoint to save the weights after each epoch.

```
# If an existing model exists, we load it. Otherwise we create a
# new model from scratch

if os.path.exists(model_file):
    print("\n*** Existing model found at %s. Loading.***\n\n" % model_file)
    model = load_existing(model_file)
else:
    print("\n*** Creating new model ***\n\n")
    model = create_model(num_hidden, num_classes)


# Since we have multiple categories and a softmax output
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

# Create a checkpoint to save the model after every epoch
checkpoint = ModelCheckpoint(model_file)
```

We then create an ImageDataGenerator for training data and one more for testing data:

```
# Now we create a generator. This will take our image data, rescale it,
# shear it, zoom in and out to create additional images for training.
train_datagen = ImageDataGenerator(\
rescale=1./255,\
shear_range=0.2,\
zoom_range=0.2,\
horizontal_flip = True)

# Image generator for test data
test_datagen = ImageDataGenerator(rescale=1./255)
```

QUESTION 2.

ImageDataGenerator produces new image data with transformations applied, so that we can produce more data for training that we actually have. Explain why we need to do this.

Also find at least two other transformations that you can do with ImageDataGenerator and explain them.

Our "train_datagen" object will produce additional images from the images we provide it by shearing the image, zooming in or out, or by flipping the image horizontally. This is to artificially create new data to reduce overfitting. It also divides all values by 255 to scale them to [0,1].

Our "test_datagen" object is simpler; it just simply scales the data to [0,1].

We now use the "flow_from_directory" method in ImageDataGenerator to tell Keras to take all the images from the specified directory. IMPORTANT: the images must be sorted into directories corresponding to each category. The directories should also be named after each category.

Now we actually create the generators:

```
# Now we tell the generator where to get the images from.
# We also scale the images to 249x249 pixels.

train_generator = train_datagen.flow_from_directory(\
train_path,\
target_size=(249, 249),
batch_size=5,
class_mode = "categorical")

# We do the same for the validation set.

validation_generator = test_datagen.flow_from_directory(\
validation_path,\
target_size=(249,249),
batch_size=5,
class_mode='categorical')
```

Here we specify the path in "train_path", and we tell ImageDataGenerator to scale every image to 249 x 249 pixels (the size used to train InceptionV3). We tell it to present images in batches of 32, and to use "categorical" class mode. I.e. training labels will be presented as one-hot vectors instead of integers.

We do the same for “validation_generator”. We then call model.fit to train the neural network:

```
# Finally we train the neural network
model.fit(\
train_generator,\
steps_per_epoch = steps,\
epochs = num_epochs,\
callbacks = [checkpoint],\
validation_data = validation_generator,\
validation_steps = 50)
```

After training of the top layers is complete, we train some of the layers in the original InceptionV3 CNN to fine tune them. So we set all layers up to layer 249 to be non-trainable, and all layers from 249 onwards to be trainable:

```
# Train last two layers

# Now we twek the training by freezing almost all the layers and
# just train the topmost layer
for layer in model.layers[:249]:
    layer.trainable = False

for layer in model.layers[249:]:
    layer.trainable = True
```

We then compile with an SGD optimizer instead of rmprop that we used earlier, and we train again with model.fit.

```
model.compile(optimizer=SGD(lr=0.00001, momentum=0.9), loss='categorical_crossentropy')

model.fit(\
train_generator,\
steps_per_epoch=steps,\
epochs = num_epochs,\
callbacks = [checkpoint],\
validation_data = validation_generator,\
validation_steps=50)
```

f. Putting It Together

KEY IN THE FOLLOWING INTO train.py

```
def main():
    train(MODEL_FILE, train_path="flower_photos",
    validation_path="flower_photos", steps=120, num_epochs=10)

if __name__ == "__main__":
    main()
```

g. Starting the Training

Save your file, and type “python train.py” to start the training. If this is the first time you are running this program, it will download the ImageNet weights file for the InceptionV3 deep learning network. This can take a while as the network is very large

h. Predicting

We will now see how to use the model we have trained to predict based on images we give it. (Note: Pillow must be installed. If your system complains that the PIL module does not exist, install Pillow with “pip3 install pillow” without the quotes).

Create a file called “predict.py” and key in the following:

```
from tensorflow.keras.models import load_model
import tensorflow as tf
from tensorflow.python.keras.backend import set_session
import numpy as np
from PIL import Image
from os import listdir
from os.path import join

MODEL_NAME='flowers.hd5'

# Our samples directory
SAMPLE_PATH = './samples'

dict={0:'daisy', 1:'dandelion', 2:'roses',  3:'sunflowers', 4:'tulips'}

# Takes in a loaded model, an image in numpy matrix format,
# And a label dictionary

session = tf.compat.v1.Session(graph = tf.compat.v1.Graph())

def classify(model, image):

    with session.graph.as_default():
        set_session(session)
        result = model.predict(image)
        themax = np.argmax(result)

    return (dict[themax], result[0][themax], themax)

# Load image
def load_image(image_fname):
    img = Image.open(image_fname)
    img = img.resize((249, 249))
    imgarray = np.array(img)/255.0
    final = np.expand_dims(imgarray, axis=0)
    return final
```

```
# Test main
def main():
    with session.graph.as_default():
        set_session(session)

        print("Loading model from ", MODEL_NAME)
        model = load_model(MODEL_NAME)
        print("Done")


        print("Now classifying files in ", SAMPLE_PATH)

        sample_files = listdir(SAMPLE_PATH)

        for filename in sample_files:
            filename = join(SAMPLE_PATH, filename)
            img = load_image(filename)
            label,prob,_ = classify(model, img)

            print("We think with certainty %3.2f that image %s is %s."%(prob, filename, label))

if __name__=="__main__":
    main()
```

Explanation of What You Keyed In (DO NOT KEY IN ANYTHING IN THIS SECTION)

We go through each part to see what it is doing. We start first by importing what we need.

```
from tensorflow.keras.models import load_model
import tensorflow as tf
from tensorflow.python.keras.backend import set_session
import numpy as np
from PIL import Image
from os import listdir
from os.path import join

MODEL_NAME='flowers.hd5'

# Our samples directory
SAMPLE_PATH = './samples'
```

Most of this is very familiar to you except maybe PIL. PIL is Pillow, the Python Image Library and it lets us load and manipulate images. In addition to solve some threading problems in the Keras library, we also import “set_session”, which allows Tensorflow to preserve its states correctly in a multithreaded environment (see later).

Following this we create a dictionary that lets us map predicted indexes back to class names. Sadly Keras has no way of doing this neatly so we use our own dictionary.

```
dict={0:'daisy', 1:'dandelion', 2:'roses',  3:'sunflowers', 4:'tulips'}
```

Next, we create a new TensorFlow session, to preserve its states correctly in a multithreaded environment, which we will encounter when we integrate with libraries like MQTT (Message Queuing Telemetry Transport):

```
session = tf.compat.v1.Session(graph = tf.compat.v1.Graph())
```

Next, we write the classification function. It is relatively straightforward and just uses the model.predict method. The only thing special is the use of **session.graph.as_default**. This is used here to fix a multithreading bug in Keras that causes it to lose graphs:

```
def classify(model, image):

    with session.graph.as_default():
        set_session(session)
        result = model.predict(image)
        themax = np.argmax(result)

    return (dict[themax], result[0][themax], themax)
```

Here model.predict returns a vector of probabilities in “result”.

Question 3.

The model.predict call returns an array that looks like this:

[[0.231701 0.6328776 0.01869423 0.0764939 0.04023323]]

Explain what these numbers mean.

Question 4.

What does “np.argmax” do? Why do we call it here? Also explain what the last line returns:

return (dict[themax], result[0][themax], themax)

We now define a function called load_image:

```
# Load image
def load_image(image_fname):
    img = Image.open(image_fname)
    img = img.resize((249, 249))
    imgarray = np.array(img)/255.0
    final = np.expand_dims(imgarray, axis=0)
    return final
```

This uses the Open method from Image in PIL to load the JPG image, then calls the resize method to turn the image into a 249x249 image. We then turn the picture into a Numpy array, and add in one extra dimension to the image for the batch size (which model.classify needs).

Finally we have main() which loads the files in the “samples” directory and classifies the files it finds in there. We again use **session.graph.as_default** to preserve graphs:

```python
# Test main
def main():
    with session.graph.as_default():
        set_session(session)

        print("Loading model from ", MODEL_NAME)
        model = load_model(MODEL_NAME)
        print("Done")


        print("Now classifying files in ", SAMPLE_PATH)

        sample_files = listdir(SAMPLE_PATH)

        for filename in sample_files:
            filename = join(SAMPLE_PATH, filename)
            img = load_image(filename)
            label,prob,_ = classify(model, img)

            print("We think with certainty %3.2f that image %s is %s."%(prob, filename, label))

if __name__=="__main__":
    main()
```

Assuming that you have tulip2.jpg (or download any picture of a tulip and name it “tulip2.jpg”) in your directory, when you run this program it will output:

```
(tensorflow) ctank@adminsocs-MacBook-Pro Lab 3 - Transfer Learning % python predict.py
Loading model from  flowers.hd5
2021-07-07 20:51:56.277554: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:196] None of the MLIR optimization passes are enabled (registered 0 passes)
2021-07-07 20:51:56.363050: W tensorflow/core/platform/profile_utils/cpu_utils.cc:126] Failed to get CPU frequency: 0 Hz
Done
Now classifying files in  ./samples
/Users/ctank/miniforge3/envs/tensorflow/lib/python3.8/site-packages/tensorflow/python/keras/engine/training.py:2325: UserWarning: `Model.state_updates` will be removed in a future v
ersion. This property should not be used in TensorFlow 2.0, as `updates` are applied automatically.
  warnings.warn('`Model.state_updates` will be removed in a future version. '
We think with certainty 0.63 that image ./samples/dandelion1.jpg is dandelion.
We think with certainty 0.98 that image ./samples/rose1.jpg is dandelion.
We think with certainty 0.82 that image ./samples/sunflower1.jpeg is dandelion.
We think with certainty 0.80 that image ./samples/rose3.jpg is dandelion.
We think with certainty 0.44 that image ./samples/dandelion3.jpg is tulips.
We think with certainty 0.94 that image ./samples/dandelion2.jpeg is dandelion.
We think with certainty 0.62 that image ./samples/rose2.jpeg is tulips.
We think with certainty 0.57 that image ./samples/tulip.jpg is tulips.
We think with certainty 0.82 that image ./samples/tulip1.jpg is dandelion.
We think with certainty 0.31 that image ./samples/sunflower3.jpeg is tulips.
We think with certainty 0.43 that image ./samples/daisy3.jpeg is tulips.
We think with certainty 0.55 that image ./samples/tulip2.jpg is tulips.
We think with certainty 0.55 that image ./samples/tulips2.jpg is tulips.
We think with certainty 0.37 that image ./samples/tulip3.jpg is tulips.
We think with certainty 0.29 that image ./samples/daisy2.jpeg is daisy.
We think with certainty 0.52 that image ./samples/sunflower2.jpeg is dandelion.
(tensorflow) ctank@adminsocs-MacBook-Pro Lab 3 - Transfer Learning %
```

(Note: The sample output was taken after just ONE epoch of training. It isn’t perfect but already quite good!)

TIP: As you can observe from running this program, it takes a long time to load the models. To speed up execution load up the model ONCE before performing the classification over many images.