Image Classification

Motivation

- Data-driven building, averaging, and freezing a convolutional neural network or an image dataset from development to training
- Start a deeper understanding of feature maps and filters by visualizing some from a pre-trained network.

Setup and Resources

You must solve using the provided template notebook.

Instructions for setting up a Google Cloud Platform account for Question 2.1 can be provided document or Minerva about setting up a Google Cloud Platform account for colab notebooks (GPU).

Pretrained models for questions using **torchvision.models**, and links to the notebooks and exercises provided.

The coursework will use a subset of Imagenet-Mini. This ImageNet, which is a subset of the ImageNet dataset (https://linear.mit.edu/). For details of The ImageNet contains 25 different categories, as will refer to it as "ImageNet25". The training set has 400 record per class, and the validation set has 100 records per class. ImageNet25 is hosted on Google Drive, and can be downloaded using a script from the Kaggle dataset.

- https://kaggle.com/datasets/gerasimfa/imagenet-mini-25

The ImageNet25 dataset is quite large, at 15GB, and it takes a few hours to download. Even if you have an existing Kaggle account, please carefully read the instructions to make sure you do not accidentally use your quota.

The class Kaggle competition also includes a blind test set, which will be used in Question 3 for evaluating your custom model's performance on a server. The competition website will unlocks the test set on access, as well as positive you create on the class leaderboard.

Imports

Import as you write the sections or needed.

You may need to download 3.8[8] using ppa on Ubuntu.

```
!pip install -q -U keras-cv
!pip install -q -U keras-nlp
!pip install -q -U keras-tuner
!pip install -q -U keras
```

```
!kaggle
```

```
!mkdir ~/.kaggle
!cp kaggle.json ~/.kaggle/
!ls ~/.kaggle
!chmod 600 /root/.kaggle/kaggle.json
!kaggle datasets list
!ls -l
```

```
import os
import sys
import glob
import shutil
import os, shutil, pathlib
import matplotlib as mpl
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import tensorflow as tf
import keras_cv
import keras_nlp
import keras
from keras import layers, Sequential
from keras import utils, losses, optimizers, metrics
from keras.utils.dataset_utils import split_dataset
from keras.utils.image_utils import load_img, list_pictures
from keras.utils import image_dataset_from_directory
import keras_tuner
import sklearn
import sklearn.model_selection
from PIL import Image, ImageOps
import PIL
import numpy as np
import os.path
import pathlib
```

QUESTION 1

Data wrangling or munging is deep learning model is to choose an architecture that can learn the features in the dataset without being distracted by irrelevant concepts. The focus will be the coursework involves building a DNN and training it on ImageNet25.

Overview:

- 1.0 Kaggle **Dataset** and **Metadata**[1]: *classes*
- 1.1 Single-batch training: *one batch, one image*
- 1.2 Training on a single batch
- 1.3 Training on complete dataset
- 1.4 Free-form model
- 1.5 Training on complete dataset
- 1.6 Setting a lower-bound on the loss

1.1 Single-batch training:

Start with a collection of datasetset called "single-batch training", or "overfitting a single batch", in which we ollate that our model and the training code is working properly and can overfit a single training batch (i.e., we can drive the training loss to zero). That we ensure we're training on the complete training set and debugging it by only overfitting batch that loss that models say shapeshifters.

1.1.1 Dataset class

Create a class that inherits from the appropriate torch.utils.data.Dataset which loads the ImageNet25 dataset and **Metadata**[1] for training and validation.

```
!ls -l /kaggle/working/imagenet-mini/train/n01440764/n01440764_10026.JPEG
```

```
import os, shutil, pathlib
from keras.utils.image_utils import load_img, list_pictures
from keras.utils.dataset_utils import split_dataset
from keras import utils
from PIL import Image, ImageOps
import os.path
import PIL
import numpy as np
import matplotlib.pyplot as plt
import pathlib
import os
import keras_cv
import keras_nlp
import keras
import keras_tuner
import pandas as pd
import sklearn
import sklearn.model_selection
import glob, os.path
import sys
import os, shutil, pathlib
import zipfile, os.path, shutil, sys
import tarfile, os.path, shutil, sys
import os, os.path, shutil
import os, os.path, shutil, pathlib
import os, os.path, shutil, pathlib, zipfile
import os, os.path, shutil, pathlib, tarfile
import os, os.path, shutil, pathlib, zipfile, tarfile
import os, os.path, shutil, pathlib, zipfile, tarfile, sklearn
import os, os.path, shutil, pathlib, zipfile, tarfile, sklearn, keras
import os, os.path, shutil, pathlib, zipfile, tarfile, sklearn, keras, keras_cv
import os, os.path, shutil, pathlib, zipfile, tarfile, sklearn, keras, keras_cv, keras_nlp
```

```
!ls -l /kaggle/working/
```

```
!ls -l /kaggle/working/imagenet-mini/
```

```
!ls -l /kaggle/working/imagenet-mini/train/
```

```
!ls -l /kaggle/working/imagenet-mini/train/n01440764/
```

```
!ls -l /kaggle/working/imagenet-mini/train/n01440764/n01440764_10026.JPEG
```

```
# Logistic Regression
# Binary classification
# Data: https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database
# Author: Ivan Gerasimov
# Date: 2023-03-15

# Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, cross_validation
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, roc_auc_score
from sklearn.linear_model import LogisticRegression

# Load data
df = pd.read_csv('diabetes.csv')
print(df.shape)
print(df.head())
print(df.info())
print(df.describe())

# Check for missing values
print(df.isnull().sum())

# Check for duplicates
print(df.duplicated().sum())

# Check for outliers
sns.boxplot(data=df, orient='h')
plt.show()

# Check for correlation
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
plt.show()

# Check for class imbalance
sns.countplot(x='Outcome', data=df)
plt.show()

# Check for feature distributions
df.hist(bins=50, figsize=(20,15))
plt.show()

# Check for feature distributions by class
sns.pairplot(df, hue='Outcome')
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Glucose', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='BloodPressure', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='SkinThickness', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Insulin', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='BMI', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='DiabetesPedigreeFunction', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Age', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Pregnancies', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Glucose', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='BloodPressure', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='SkinThickness', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Insulin', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='BMI', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='DiabetesPedigreeFunction', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Age', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Pregnancies', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Glucose', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='BloodPressure', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='SkinThickness', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Insulin', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='BMI', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='DiabetesPedigreeFunction', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Age', data=df)
plt.show
```

```
# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Pregnancies', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Glucose', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='BloodPressure', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='SkinThickness', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Insulin', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='BMI', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='DiabetesPedigreeFunction', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Age', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Pregnancies', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Glucose', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='BloodPressure', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='SkinThickness', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Insulin', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='BMI', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='DiabetesPedigreeFunction', data=df)
plt.show()

# Check for feature distributions by class
sns.boxplot(x='Outcome', y='Age', data=df)
plt.show
```

```
```

1.3 Testing on test data

1.3.1 Dataset and generating predictions

Create a PyTorch `DataLoader` for the combined test data in this set, and folder of the buggy competition and generate predictions using your first model.

```
# to cuda
# to cpu
# to numpy
from torchvision.transforms.functional import to_pil_image, pad, resize

def collate_fn(batch):
    ...
    return torch.stack(pixel_values).to(device)
test_ds = ConcatDataset([test_ds, buggy_ds])
test_loader = DataLoader(test_ds, batch_size=32, collate_fn=collate_fn)

model.eval()
preds = []
with torch.no_grad():
    for batch in test_loader:
        ...
        preds.append(logits.cpu().numpy())
preds = np.concatenate(preds, axis=0)
print(f"Predictions shape: {preds.shape}")
```

```
Device: cuda
Test samples: 1000
Buggy samples: 1000
Total test samples: 2000
Predictions shape: (2000, 10)
```

1.3.2 F20 the and test set accuracy

Save all test predictions in a file that will submit it to the private leaderboard. **Please save your best CSV file submission using your student username (the one with letters, 0, 1e3@ab, and the @ with only numbers)**. For example, 1e3@ab.csv. This will help us to identify your submission.

The F20 the test accuracy, top-5 accuracy, and per-Category accuracies (top-5) are shown below:

```
Top-5 accuracy: 0.9900
Per-Category accuracy (top-5):
  Category                 acc
0    Animal  0.9875000000000000
1     Bottle  0.9925000000000000
2       Car  0.9975000000000000
3      Chair  0.9925000000000000
4       Cup  0.9975000000000000
5     Flower  0.9975000000000000
6      Human  0.9975000000000000
7      Plane  0.9975000000000000
8      Table  0.9975000000000000
9      Train  0.9975000000000000
```

Overview:

- 0.9 Correct Rate (top-5acc: 90/20_TITANXP_DAC_NAME1)
- 0.9 Correct Rate
- 0.8 Correct Rate(top-5: grad-cam_90/20_TITANXP_DAC_NAME1)
- 0.8 Correct Rate (top-5: 90/20_TITANXP_DAC_NAME2)
- 0.8 Correct Rate (100 Hamster: 90/20_TITANXP_DAC_NAME1, 10ep2, 0ep2)
- 0.8 Correct Rate (top-5: 90/20_TITANXP_DAC_NAME2)
- 0.8 Correct Rate (Cup: 90/20_TITANXP_DAC_NAME1, 10ep2, 0ep2)

Loading a pre-trained model

Run the cell below to load an Inception model with pre-trained weights.

```
model = torch.hub.load('pytorch/vision:v0.9.0', 'inception', pretrained=True)
model.eval()
```

```
Using cache found in /root/.cache/torch/hub/pytorch_vision_v0.9.0
```

```
Sequential(
  (0): BasicConv2d(
    (conv): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), bias=False)
    (bn): BatchNorm2d(32, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (1): BasicConv2d(
    (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(32, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (2): BasicConv2d(
    (conv): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn): BatchNorm2d(64, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (3): BasicConv2d(
    (conv): Conv2d(64, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(80, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (4): BasicConv2d(
    (conv): Conv2d(80, 192, kernel_size=(3, 3), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (5): BasicConv2d(
    (conv): Conv2d(192, 288, kernel_size=(3, 3), stride=(2, 2), bias=False)
    (bn): BatchNorm2d(288, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (6): InceptionA(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch5x5_1): BasicConv2d(
      (conv): Conv2d(288, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(48, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch5x5_2): BasicConv2d(
      (conv): Conv2d(48, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(64, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_3): BasicConv2d(
      (conv): Conv2d(96, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(288, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(32, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (7): InceptionA(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch5x5_1): BasicConv2d(
      (conv): Conv2d(288, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(48, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch5x5_2): BasicConv2d(
      (conv): Conv2d(48, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(64, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_3): BasicConv2d(
      (conv): Conv2d(96, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (8): InceptionA(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch5x5_1): BasicConv2d(
      (conv): Conv2d(288, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(48, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch5x5_2): BasicConv2d(
      (conv): Conv2d(48, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(64, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_3): BasicConv2d(
      (conv): Conv2d(96, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (9): InceptionB(
    (branch3x3): BasicConv2d(
      (conv): Conv2d(288, 384, kernel_size=(3, 3), stride=(2, 2), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(288, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(64, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(64, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_3): BasicConv2d(
      (conv): Conv2d(96, 96, kernel_size=(3, 3), stride=(2, 2), bias=False)
      (bn): BatchNorm2d(96, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (10): InceptionC(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7_1): BasicConv2d(
      (conv): Conv2d(768, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(128, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7_2): BasicConv2d(
      (conv): Conv2d(128, 128, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
      (bn): BatchNorm2d(128, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7_3): BasicConv2d(
      (conv): Conv2d(128, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7dbl_1): BasicConv2d(
      (conv): Conv2d(768, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(128, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7dbl_2): BasicConv2d(
      (conv): Conv2d(128, 128, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
      (bn): BatchNorm2d(128, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7dbl_3): BasicConv2d(
      (conv): Conv2d(128, 128, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
      (bn): BatchNorm2d(128, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7dbl_4): BasicConv2d(
      (conv): Conv2d(128, 128, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
      (bn): BatchNorm2d(128, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7dbl_5): BasicConv2d(
      (conv): Conv2d(128, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (11): InceptionC(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7_1): BasicConv2d(
      (conv): Conv2d(768, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7_2): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7_3): BasicConv2d(
      (conv): Conv2d(160, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7dbl_1): BasicConv2d(
      (conv): Conv2d(768, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7dbl_2): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7dbl_3): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7dbl_4): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7dbl_5): BasicConv2d(
      (conv): Conv2d(160, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (12): InceptionC(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7_1): BasicConv2d(
      (conv): Conv2d(768, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7_2): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7_3): BasicConv2d(
      (conv): Conv2d(160, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7dbl_1): BasicConv2d(
      (conv): Conv2d(768, 160, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7dbl_2): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7dbl_3): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7dbl_4): BasicConv2d(
      (conv): Conv2d(160, 160, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
      (bn): BatchNorm2d(160, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7dbl_5): BasicConv2d(
      (conv): Conv2d(160, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (13): InceptionD(
    (branch3x3_1): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3_2): BasicConv2d(
      (conv): Conv2d(192, 320, kernel_size=(3, 3), stride=(2, 2), bias=False)
      (bn): BatchNorm2d(320, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7x3_1): BasicConv2d(
      (conv): Conv2d(768, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7x3_2): BasicConv2d(
      (conv): Conv2d(192, 192, kernel_size=(1, 7), stride=(1, 1), padding=(0, 3), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7x3_3): BasicConv2d(
      (conv): Conv2d(192, 192, kernel_size=(7, 1), stride=(1, 1), padding=(3, 0), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch7x7x3_4): BasicConv2d(
      (conv): Conv2d(192, 192, kernel_size=(3, 3), stride=(2, 2), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (14): InceptionE(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(1280, 320, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(320, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3_1): BasicConv2d(
      (conv): Conv2d(1280, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3_2a): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3_2b): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(1280, 448, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(448, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(448, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_3a): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_3b): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(1280, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (15): InceptionE(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(2048, 320, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(320, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3_1): BasicConv2d(
      (conv): Conv2d(2048, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3_2a): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3_2b): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(2048, 448, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(448, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(448, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_3a): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch3x3dbl_3b): BasicConv2d(
      (conv): Conv2d(384, 384, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(2048, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.9997, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
)
```

2.1 Extract and visualize the filters

In this section you will extract and visualize the filters from the pre-trained GoogleNet.

2.1.1 Extract filters

Complete the following function `RGB_TENSOR()`, to return all of the filters from the convolutional layers at the given index. In `model[0][0].conv` we provide access to convolution.

- We will not test the definition of your function using hidden tests.

```
def RGB_TENSOR(model, idx):
    ...
    return ...
model = model.cpu()
for idx, conv in enumerate(model[0].conv):
    if isinstance(conv, nn.Conv2d):
        print(f"Layer {idx}: {conv.weight.shape}")
W = RGB_TENSOR(model, 0)
print(f"\nLoaded filters: {W.shape}")
# Expect: torch.Size([64, 3, 7, 7])
assert len(W.shape) == 4, "Should be 4D tensor"
assert W.shape[1] == 3, "Should have 3 input channels (RGB)"
```

```
Layer 0: torch.Size([64, 3, 7, 7])
Layer 3: torch.Size([192, 64, 3, 3])
Layer 5: torch.Size([288, 192, 3, 3])
Layer 6: torch.Size([64, 288, 1, 1])
Layer 7: torch.Size([96, 64, 3, 3])
Layer 8: torch.Size([96, 96, 3, 3])
Layer 9: torch.Size([32, 288, 1, 1])
Layer 10: torch.Size([64, 288, 1, 1])
Layer 11: torch.Size([96, 64, 3, 3])
Layer 12: torch.Size([96, 96, 3, 3])
Layer 13: torch.Size([64, 288, 1, 1])
Layer 14: torch.Size([384, 288, 3, 3])
Layer 15: torch.Size([64, 768, 1, 1])
Layer 16: torch.Size([128, 768, 1, 1])
Layer 17: torch.Size([128, 128, 1, 7])
Layer 18: torch.Size([192, 128, 7, 1])
Layer 19: torch.Size([128, 768, 1, 1])
Layer 20: torch.Size([128, 128, 7, 1])
Layer 21: torch.Size([128, 128, 1, 7])
Layer 22: torch.Size([128, 128, 7, 1])
Layer 23: torch.Size([192, 128, 1, 7])
Layer 24: torch.Size([192, 768, 1, 1])
Layer 25: torch.Size([160, 768, 1, 1])
Layer 26: torch.Size([160, 160, 1, 7])
Layer 27: torch.Size([192, 160, 7, 1])
Layer 28: torch.Size([160, 768, 1, 1])
Layer 29: torch.Size([160, 160, 7, 1])
Layer 30: torch.Size([160, 160, 1, 7])
Layer 31: torch.Size([160, 160, 7, 1])
Layer 32: torch.Size([192, 160, 1, 7])
Layer 33: torch.Size([192, 768, 1, 1])
Layer 34: torch.Size([192, 768, 1, 1])
Layer 35: torch.Size([320, 192, 3, 3])
Layer 36: torch.Size([192, 768, 1, 1])
Layer 37: torch.Size([192, 192, 1, 7])
Layer 38: torch.Size([192, 192, 7, 1])
Layer 39: torch.Size([192, 192, 3, 3])
Layer 40: torch.Size([320, 2048, 1, 1])
Layer 41: torch.Size([384, 2048, 1, 1])
Layer 42: torch.Size([384, 384, 1, 3])
Layer 43: torch.Size([384, 384, 3, 1])
Layer 44: torch.Size([448, 2048, 1, 1])
Layer 45: torch.Size([384, 448, 3, 3])
Layer 46: torch.Size([384, 384, 1, 3])
Layer 47: torch.Size([384, 384, 3, 1])
Layer 48: torch.Size([192, 2048, 1, 1])
Layer 49: torch.Size([320, 2048, 1, 1])
Layer 50: torch.Size([384, 2048, 1, 1])
Layer 51: torch.Size([384, 384, 1, 3])
Layer 52: torch.Size([384, 384, 3, 1])
Layer 53: torch.Size([448, 2048, 1, 1])
Layer 54: torch.Size([384, 448, 3, 3])
Layer 55: torch.Size([384, 384, 1, 3])
Layer 56: torch.Size([384, 384, 3, 1])
Layer 57: torch.Size([192, 2048, 1, 1])

Loaded filters: torch.Size([64, 3, 7, 7])
```

2.1.2 Visualize filters

Run the cells with `model` to apply the image transformation expected by the model.

```
# Transform: resize to 299x299, normalize to [-1, 1]
transform = transforms.Compose([
    transforms.Resize((299, 299)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])

def denormalize(tensor):
    # Denormalize from [-1, 1] to [0, 1]
    return (tensor + 1) / 2

def visualize_filters(W, ncols=8):
    ...
    plt.suptitle(f'{W.shape[0]} filters', fontsize=14)
    plt.tight_segment(rect=[0, 0.03, 1, 0.95])
    plt.show()

visualize_filters(W)
```

2.2.1 Load test image

```
# Load test image
import urllib.request
urllib.request.urlretrieve('https://upload.wikimedia.org/wikipedia/commons/thumb/3/3a/Cat03.jpg/1200px-Cat03.jpg', 'test_cat.jpg')
print("Downloaded test_cat.jpg")
```

```
Downloaded test_cat.jpg
```

Run the cells with `model` to apply the image transformation expected by the model.

2.2.2 Extract feature maps

Complete the following function `FEATURE_MAPS()`, to return all of the feature maps for the convolutional layers at the given index.

- We will not test the definition of your function using hidden tests.

```
def FEATURE_MAPS(model, conv_idx, image):
    activations = {}
    def hook(module, input, output):
        activations['conv'] = output.cpu()
    ...
    return activations['conv']
model = model.cpu()
for idx, conv in enumerate(model[0].conv):
    if isinstance(conv, nn.Conv2d):
        print(f"Layer {idx}: {conv.weight.shape}")
test_image = transform(Image.open('test_cat.jpg'))
fmaps = FEATURE_MAPS(model, 0, test_image)
print(f"\nFeature maps shape: {fmaps.shape}")
# Expect: torch.Size([1, 64, 150, 150])
assert len(fmaps.shape) == 4, "Should be 4D tensor"
assert fmaps.shape[0] == 1, "Batch size should be 1"
```

```
Layer 0: torch.Size([64, 3, 7, 7])
Layer 3: torch.Size([192, 64, 3, 3])
Layer 5: torch.Size([288, 192, 3, 3])
Layer 6: torch.Size([64, 288, 1, 1])
Layer 7: torch.Size([96, 64, 3, 3])
Layer 8: torch.Size([96, 96, 3, 3])
Layer 9: torch.Size([32, 288, 1, 1])
Layer 10: torch.Size([64, 288, 1, 1])
Layer 11: torch.Size([96, 64, 3, 3])
Layer 12: torch.Size([96, 96, 3, 3])
Layer 13: torch.Size([64, 288, 1, 1])
Layer 14: torch.Size([384, 288, 3, 3])
Layer 15: torch.Size([64, 768, 1, 1])
Layer 16: torch.Size([128, 768, 1, 1])
Layer 17: torch.Size([128, 128, 1, 7])
Layer 18: torch.Size([192, 128, 7, 1])
Layer 19: torch.Size([128, 768, 1, 1])
Layer 20: torch.Size([128, 128, 7, 1])
Layer 21: torch.Size([128, 128, 1, 7])
Layer 22: torch.Size([128, 128, 7, 1])
Layer 23: torch.Size([192, 128, 1, 7])
Layer 24: torch.Size([192, 768, 1, 1])
Layer 25: torch.Size([160, 768, 1, 1])
Layer 26: torch.Size([160, 160, 1, 7])
Layer 27: torch.Size([192, 160, 7, 1])
Layer 28: torch.Size([160, 768, 1, 1])
Layer 29: torch.Size([160, 160, 7, 1])
Layer 30: torch.Size([160
```

```
Saving images
```

```
Saved images.zip
```

2.4.2 Display heatmaps

Finally, **GradCAM**, as a class-discriminating super-imposed onto the original image, To get results as shown in the paper, we recommend the following steps:

1. Resize the heatmap to 14x14 (the size of the image)
2. Normalize the heatmap to [0,1] (for better image viewing)
3. Apply a colormap to the heatmap using `cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)`.
4. Scale the colorized heatmap by a factor of 255 (for better image viewing)
5. Superimpose the heatmap onto the original image (Note: where pixel=(0,0,0) (addition - addition of the (r,0,0) images, not overlap addition, like in tensorflow) `uint8(255*colorMap(heatmap)[:,:,:3]+0.5*imread(infile))`(Remember to convert to uint8 for replication.)
6. Resize the image back to 224x224
7. Display the resulting image

```
# 2.4 Visualize
# Load the saved Grad-CAM heatmaps (replacing the extension of .jpg with .npy)
image_id = 'ILSVRC2012_val_00000001'
heatmap = np.load('/content/drive/MyDrive/Colab Notebooks/2_2/heatmaps/'+image_id+'.npy')
print('Loaded heatmap shape:', heatmap.shape)
# Load the original image
inpath = '/content/drive/MyDrive/Colab Notebooks/2_2/imagenet-val-500/'+image_id+'.jpg'
orig_image = cv2.imread(inpath)
print('Original image shape:', orig_image.shape)
```

```
Loaded heatmap shape: (14, 14)
Original image shape: (375, 500, 3)
```

```
# Resize heatmap to 14x14
heatmap = cv2.resize(heatmap, (14, 14))
print('Resized heatmap shape:', heatmap.shape)
```

```
Resized heatmap shape: (14, 14)
```

```
# Normalize heatmap to [0,1]
heatmap = (heatmap - np.min(heatmap)) / (np.max(heatmap) - np.min(heatmap))
print('Normalized heatmap range:', np.min(heatmap), np.max(heatmap))
```

```
Normalized heatmap range: 0.0 1.0
```

```
# Apply colormap to heatmap
colorMap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)
print('Colorized heatmap shape:', colorMap.shape)
```

```
Colorized heatmap shape: (14, 14, 3)
```

```
# Scale by 255
colorMap = 255 * colorMap
print('Scaled colorMap range:', np.min(colorMap), np.max(colorMap))
```

```
Scaled colorMap range: 0.0 255.0
```

```
# Superimpose heatmap onto original image
superimposed = np.uint8(255*colorMap[:,:,:3] + 0.5*orig_image)
print('Superimposed image shape:', superimposed.shape)
```

```
Superimposed image shape: (375, 500, 3)
```

```
# Display the resulting image
plt.imshow(cv2.cvtColor(superimposed, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.title('Superimposed image')
plt.show()
```

Show the heatmap for class "(n04285008, sleep)", (n04285008, sleep-sleep) (`ILSVRC_2012_val_00000001`), super-imposed onto the original image.

```
# 2.4 Visualize
# Load the saved Grad-CAM heatmaps (replacing the extension of .jpg with .npy)
image_id = 'ILSVRC2012_val_00000001'
heatmap = np.load('/content/drive/MyDrive/Colab Notebooks/2_2/heatmaps/'+image_id+'.npy')
print('Loaded heatmap shape:', heatmap.shape)
# Load the original image
inpath = '/content/drive/MyDrive/Colab Notebooks/2_2/imagenet-val-500/'+image_id+'.jpg'
orig_image = cv2.imread(inpath)
print('Original image shape:', orig_image.shape)
# Resize heatmap to 14x14
heatmap = cv2.resize(heatmap, (14, 14))
print('Resized heatmap shape:', heatmap.shape)
# Normalize heatmap to [0,1]
heatmap = (heatmap - np.min(heatmap)) / (np.max(heatmap) - np.min(heatmap))
print('Normalized heatmap range:', np.min(heatmap), np.max(heatmap))
# Apply colormap to heatmap
colorMap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)
print('Colorized heatmap shape:', colorMap.shape)
# Scale by 255
colorMap = 255 * colorMap
print('Scaled colorMap range:', np.min(colorMap), np.max(colorMap))
# Superimpose heatmap onto original image
superimposed = np.uint8(255*colorMap[:,:,:3] + 0.5*orig_image)
print('Superimposed image shape:', superimposed.shape)
# Display the resulting image
plt.imshow(cv2.cvtColor(superimposed, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.title('Superimposed image')
plt.show()
```


2.4.3 Failure analysis using Grad-CAM

Find an image similar to that magpie or another element which identifier classifies incorrectly. Display the image name, and show the model's predictions/ clues. Then, generate the Grad-CAM heatmap and display it super-imposed onto the image.

```
# 2.4 Visualize
# Load the saved Grad-CAM heatmaps (replacing the extension of .jpg with .npy)
image_id = 'ILSVRC2012_val_00000001'
heatmap = np.load('/content/drive/MyDrive/Colab Notebooks/2_2/heatmaps/'+image_id+'.npy')
print('Loaded heatmap shape:', heatmap.shape)
```

```
# 2.4 Visualize
image_id = 'ILSVRC2012_val_00000001'
heatmap = np.load('/content/drive/MyDrive/Colab Notebooks/2_2/heatmaps/'+image_id+'.npy')
print('Loaded heatmap shape:', heatmap.shape)
# Load the original image
inpath = '/content/drive/MyDrive/Colab Notebooks/2_2/imagenet-val-500/'+image_id+'.jpg'
orig_image = cv2.imread(inpath)
print('Original image shape:', orig_image.shape)
# Resize heatmap to 14x14
heatmap = cv2.resize(heatmap, (14, 14))
print('Resized heatmap shape:', heatmap.shape)
# Normalize heatmap to [0,1]
heatmap = (heatmap - np.min(heatmap)) / (np.max(heatmap) - np.min(heatmap))
print('Normalized heatmap range:', np.min(heatmap), np.max(heatmap))
# Apply colormap to heatmap
colorMap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)
print('Colorized heatmap shape:', colorMap.shape)
# Scale by 255
colorMap = 255 * colorMap
print('Scaled colorMap range:', np.min(colorMap), np.max(colorMap))
# Superimpose heatmap onto original image
superimposed = np.uint8(255*colorMap[:,:,:3] + 0.5*orig_image)
print('Superimposed image shape:', superimposed.shape)
# Display the resulting image
plt.imshow(cv2.cvtColor(superimposed, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.title('Superimposed image')
plt.show()
```


Briefly describe what explanation this Grad-CAM heatmap provides about why the model has failed to correctly classify your test image.

- Double click to expand here

Because golden retriever can similar to cocker so, the model will recognize wrong. However, the cat is not as big as a dog, so that's why the model cannot realize that and cut off.