COURSE CODE: DJS22DSL501 **DATE:**

COURSE NAME: Machine Learning - II **CLASS: AY 2024-25**

LAB EXPERIMENT NO. 10

AIM :

Transfer Learning with Pre-trained CNN model as a Feature Extractor for Image Classification With a data availability constraint

THEORY:

What is Transfer Learning?

Transfer learning is a machine learning technique where a model developed for a particular task is reused as the starting point for a model on a second, related task. This is particularly useful in deep learning, where training models from scratch often requires vast amounts of data and computational resources.

Convolutional Neural Networks (CNNs)

CNNs are a class of deep learning models specifically designed for processing structured grid data, such as images. They consist of multiple layers, including convolutional layers, pooling layers, and fully connected layers, which allow them to automatically learn and extract features from images.

Pre-trained Models

Pre-trained models are CNNs that have been previously trained on large datasets, such as ImageNet, which contains millions of images across thousands of categories. These models capture a wide range of features, from simple edges to complex textures and patterns. Other than this many different models can also be used like

Data Availability Constraint

One common challenge in deep learning is the availability of labeled data, particularly for specific applications or domains where data collection is costly or time-consuming. Here’s how transfer learning helps in such scenarios:

1. **Reduced Data Requirement**:
    - By leveraging a pre-trained model, the need for a large amount of labeled data in the new domain is significantly reduced. You may only require a small amount of labeled data to fine-tune the classifier.
2. **Data Augmentation**:
    - Techniques such as data augmentation can further enhance the effectiveness of transfer learning. By artificially increasing the size of the dataset through transformations (e.g., rotation, flipping, scaling), you can improve the model's robustness.
3. **Domain Adaptation**:
    - In cases where the new dataset has different characteristics (domain shift), techniques like domain adaptation can be applied to align the feature spaces of the pre-trained model and the target dataset.
4. **Overcoming Class Imbalance**:
    - Transfer learning can help mitigate issues with class imbalance by focusing on feature extraction from the more frequent classes and using them to inform the classifier about less frequent classes.

Advantages of Using Pre-trained CNNs for Feature Extraction

- **Efficiency**: Training from scratch can be computationally expensive; using a pre-trained model accelerates the process.
- **Performance**: Pre-trained models often achieve better performance with less data, as they benefit from features learned on large datasets.
- **Flexibility**: This approach allows for quick adaptation to new tasks with minimal adjustments.

Tasks to be performed:

1. **Choose a dataset suitable for image classification. (Eg. CIFAR-10, Dogs vs Cats)**
2. **Resize images to the required input size of the chosen pre-trained model.**
3. **Load Pre-trained Model (LeNet-5, AlexNet, VGG-16, Inception-v1, ResNet-50)**
4. **Compare the performances of all the models and visualize**
5. **Write down your observations and conclusions**

Content

- Introduction
- Load packages and set parameters
- Read the data
- Data exploration
  - Class distribution
  - Images samples
- Model
  - Prepare the model
  - Train the model
  - Validation accuracy and loss
  - Validation accuracy per class
- Prepare submission
- Conclusions
- References

Introduction

Dataset

The **train** folder contains **25,000** images of **dogs** and **cats**. Each image in this folder has the label as part of the filename. The **test** folder contains **12,500** images, named according to a numeric id.
For each image in the test set, you should predict a probability that the image is a dog (**1** = **dog**, **0** = **cat**).

Method

For the solution of this problem we will use a pre-trained model, ResNet-50, replacing only the last layer.

Load packages

```
import os, cv2, random
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from tqdm import tqdm
from random import shuffle
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot
from keras.utils import plot_model
from tensorflow.python.keras.applications import ResNet50
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense, Flatten,
GlobalAveragePooling2D
%matplotlib inline
```

```
Using TensorFlow backend.
```

Parameters

Here we set few parameters used in the model. The image size is **224**.
The images are stored in two folders, **train** and **test**.
There are two image classes: **Dog** and **Cat**.
We will use a subset of the training data set (**20,000** images). From the training set, **50%** will be used for training, **50%** for validation.
A pre-trained model from **ResNet-50** will be used.
A number of **10** epochs will be used for training.

```
TEST_SIZE = 0.5
RANDOM_STATE = 2018
BATCH_SIZE = 64
NO_EPOCHS = 20
NUM_CLASSES = 2
SAMPLE_SIZE = 20000
PATH = '/kaggle/input/dogs-vs-cats-redux-kernels-edition/'
TRAIN_FOLDER = './train/'
TEST_FOLDER =  './test/'
IMG_SIZE = 224
RESNET_WEIGHTS_PATH =
'/kaggle/input/resnet50/resnet50_weights_tf_dim_ordering_tf_kernels_no
top.h5'
```

Read the data

We set the train image list.
Setting the **SAMPLE_SIZE** value we can reduce/enlarge the size of the training set.
Currently **SAMPLE_SIZE** is set to **20,000**.

```
train_image_path = os.path.join(PATH, "train.zip")
test_image_path = os.path.join(PATH, "test.zip")

import zipfile
with zipfile.ZipFile(train_image_path,"r") as z:
    z.extractall(".")

with zipfile.ZipFile(test_image_path,"r") as z:
    z.extractall(".")

train_image_list = os.listdir("./train/")[0:SAMPLE_SIZE]
test_image_list = os.listdir("./test/")
```

We set a function for parsing the image names to extract the first 3 letters from the image names, which gives the label of the image. It will be either a cat or a dog. We are using one hot encoder, storing [1,0] for **cat** and [0,1] for **dog**.

```
def label_pet_image_one_hot_encoder(img):
    pet = img.split('.')[-3]
    if pet == 'cat': return [1,0]
    elif pet == 'dog': return [0,1]
```

We are defining as well a function to process the data (both train and test set).

```
def process_data(data_image_list, DATA_FOLDER, isTrain=True):
    data_df = []
    for img in tqdm(data_image_list):
        path = os.path.join(DATA_FOLDER,img)
        if(isTrain):
            label = label_pet_image_one_hot_encoder(img)
        else:
            label = img.split('.')[0]
        img = cv2.imread(path,cv2.IMREAD_COLOR)
        img = cv2.resize(img, (IMG_SIZE,IMG_SIZE))
        data_df.append([np.array(img),np.array(label)])
    shuffle(data_df)
    return data_df
```

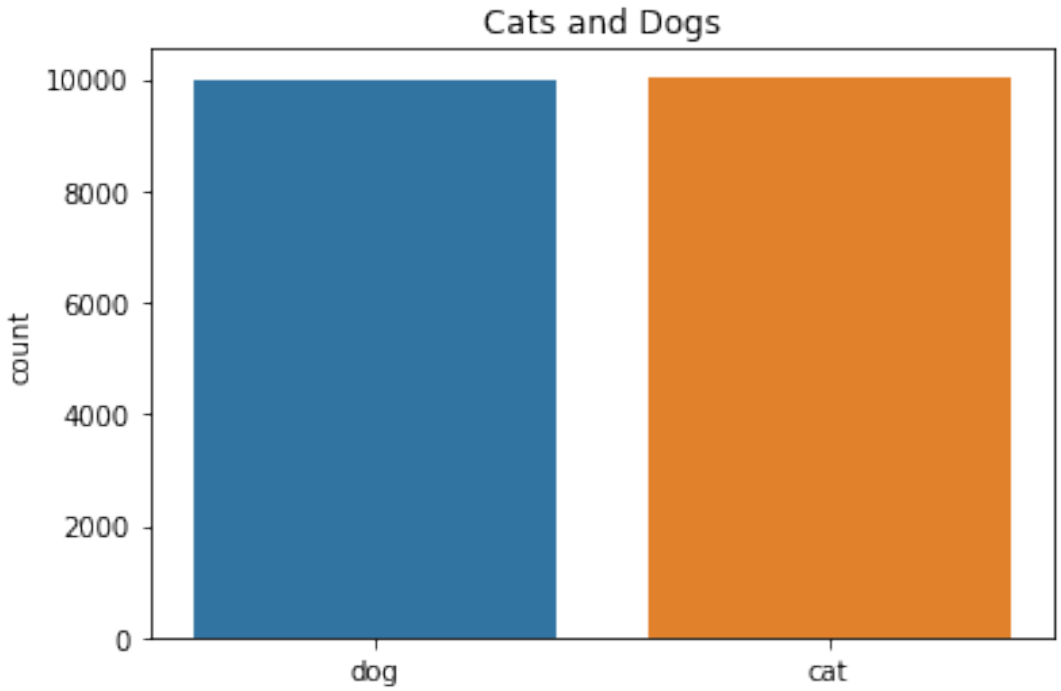
Data exploration

Class distribution

Let's inspect the train data to check the **cat**/**dog** distribution. We show first the split in the reduced train data.

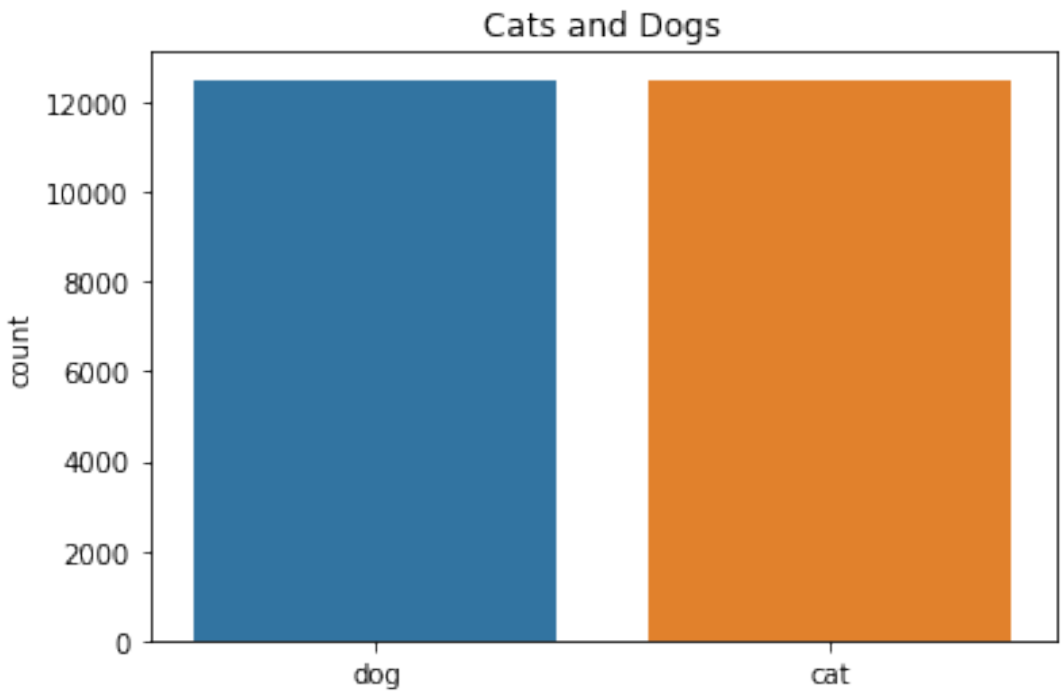
```
def plot_image_list_count(data_image_list):
    labels = []
    for img in data_image_list:
        labels.append(img.split('.')[-3])
    sns.countplot(labels)
    plt.title('Cats and Dogs')

plot_image_list_count(train_image_list)
```



Let's show also the class distribution in the full train data set.

```
plot_image_list_count(os.listdir(TRAIN_FOLDER))
```



Images samples

Let's represet some of the images. We start with a selection from the train set. We will show the first 25 images from the train set.

First, we process the train data, reading the images and creating a table with images and labels. If the data is trom train set, the label is the one calculated with one hot encoding; if the data is from test set, the label will be the image number.

```
train = process_data(train_image_list, TRAIN_FOLDER)
```

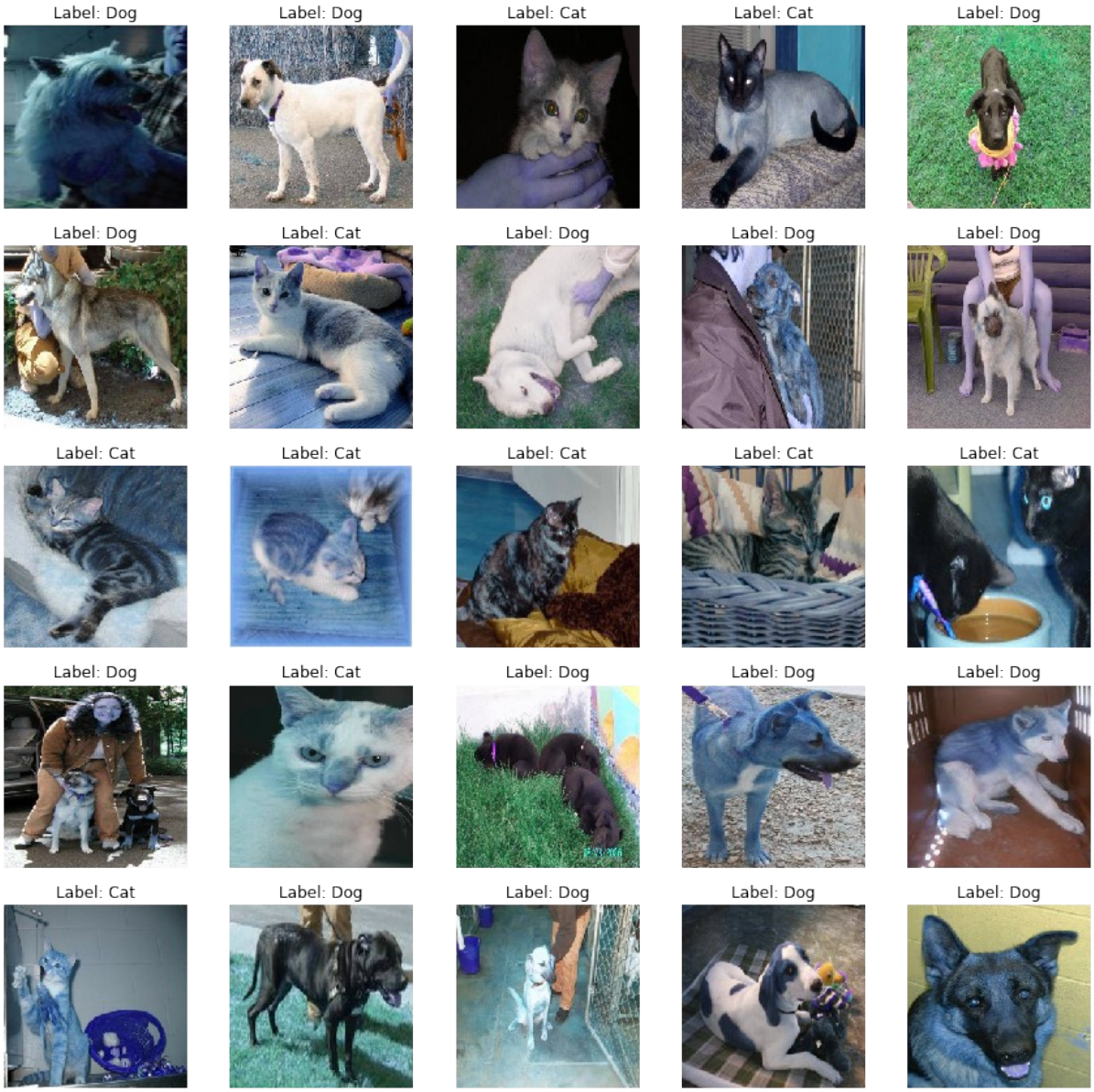
```
100%|██████████| 20000/20000 [00:43<00:00, 457.45it/s]
```

Then, we plot the image selection.

```
def show_images(data, isTest=False):
    f, ax = plt.subplots(5,5, figsize=(15,15))
    for i,data in enumerate(data[:25]):
        img_num = data[1]
        img_data = data[0]
        label = np.argmax(img_num)
        if label  == 1:
            str_label='Dog'
        elif label == 0:
            str_label='Cat'
        if(isTest):
            str_label="None"
        ax[i//5, i%5].imshow(img_data)
```

```
        ax[i//5, i%5].axis('off')
        ax[i//5, i%5].set_title("Label: {}".format(str_label))
    plt.show()

show_images(train)
```



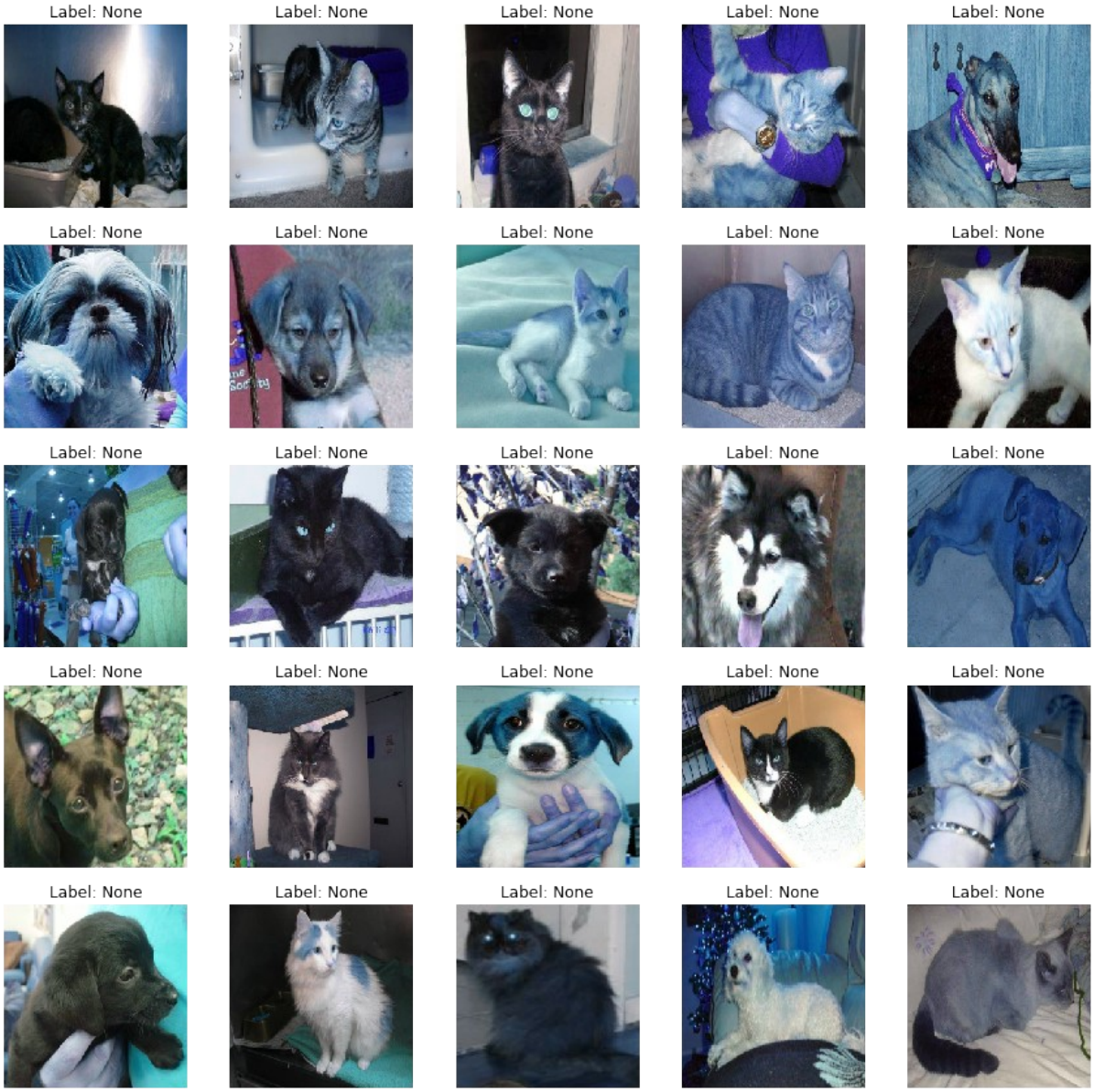
Let's also show a selection of the train set. We prepare the test set.

```
test = process_data(test_image_list, TEST_FOLDER, False)
```

```
100%|██████████| 12500/12500 [00:26<00:00, 476.31it/s]
```

Then, we show a selection of the test set.

```
show_images(test,True)
```



Model

Prepare the model

Let's start by preparing the model.

Prepare the train data

```
X = np.array([i[0] for i in train]).reshape(-1,IMG_SIZE,IMG_SIZE,3)
y = np.array([i[1] for i in train])
```

Prepare the model

We initialize the **ResNet-50** model, adding an additional last layer of type **Dense**, with **softmax** activation function.

We also set the first layer of the model to be not trainable, becaise **ResNet-50** model was already trained.

```
model = Sequential()
model.add(ResNet50(include_top=False, pooling='max',
weights=RESNET_WEIGHTS_PATH))
model.add(Dense(NUM_CLASSES, activation='softmax'))
# ResNet-50 model is already trained, should not be trained
model.layers[0].trainable = True
```

Compile the model

We compile the model, using a **sigmoid** optimized, the loss function as **categorical crossentropy** and the metric **accuracy**.

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

Model summary

We plot the model description. We can see that the **ResNet-50** model represent the 1st layer of our model, of type **Model**.

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
resnet50 (Model)             (None, 2048)              23587712
_________________________________________________________________
dense (Dense)                (None, 2)                 4098
=================================================================
Total params: 23,591,810
Trainable params: 23,538,690
Non-trainable params: 53,120
_________________________________________________________________
```

Let's also show the model graphical representation using **plot_model**.

```
plot_model(model, to_file='model.png')
SVG(model_to_dot(model).create(prog='dot', format='svg'))
```

Split the train data in train and validation

We split the train data in two parts. One will be reserved for train set, the second for validation set. Only the train subset of the data will be used for training the model; the validation set will be used for validation, during training.

```
X_train, X_val, y_train, y_val = train_test_split(X, y,
test_size=TEST_SIZE, random_state=RANDOM_STATE)
```

Train the model

We are now ready to train our model.

```
train_model = model.fit(X_train, y_train,
                  batch_size=BATCH_SIZE,
                  epochs=NO_EPOCHS,
                  verbose=1,
                  validation_data=(X_val, y_val))

Train on 10000 samples, validate on 10000 samples
Epoch 1/20
10000/10000 [==============================] - 105s 11ms/step - loss:
0.7823 - acc: 0.9403 - val_loss: 5.7698 - val_acc: 0.6090
Epoch 2/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.5717 - acc: 0.9551 - val_loss: 2.2545 - val_acc: 0.8356
Epoch 3/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.4005 - acc: 0.9667 - val_loss: 0.6695 - val_acc: 0.9375
Epoch 4/20
```

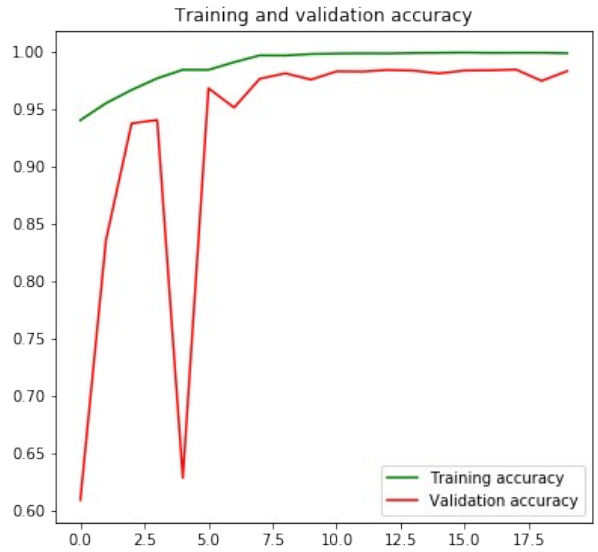
```
10000/10000 [==============================] - 93s 9ms/step - loss:
0.2591 - acc: 0.9768 - val_loss: 0.6475 - val_acc: 0.9405
Epoch 5/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.1732 - acc: 0.9843 - val_loss: 3.6218 - val_acc: 0.6283
Epoch 6/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.1600 - acc: 0.9842 - val_loss: 0.2799 - val_acc: 0.9682
Epoch 7/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.0777 - acc: 0.9909 - val_loss: 0.3348 - val_acc: 0.9513
Epoch 8/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.0303 - acc: 0.9968 - val_loss: 0.1962 - val_acc: 0.9764
Epoch 9/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.0293 - acc: 0.9967 - val_loss: 0.1631 - val_acc: 0.9812
Epoch 10/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.0193 - acc: 0.9980 - val_loss: 0.1875 - val_acc: 0.9757
Epoch 11/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.0164 - acc: 0.9984 - val_loss: 0.1460 - val_acc: 0.9829
Epoch 12/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.0162 - acc: 0.9986 - val_loss: 0.1484 - val_acc: 0.9827
Epoch 13/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.0156 - acc: 0.9985 - val_loss: 0.1414 - val_acc: 0.9841
Epoch 14/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.0134 - acc: 0.9989 - val_loss: 0.1373 - val_acc: 0.9835
Epoch 15/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.0126 - acc: 0.9991 - val_loss: 0.1516 - val_acc: 0.9811
Epoch 16/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.0114 - acc: 0.9993 - val_loss: 0.1382 - val_acc: 0.9836
Epoch 17/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.0135 - acc: 0.9990 - val_loss: 0.1368 - val_acc: 0.9838
Epoch 18/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.0146 - acc: 0.9991 - val_loss: 0.1345 - val_acc: 0.9845
Epoch 19/20
10000/10000 [==============================] - 93s 9ms/step - loss:
0.0136 - acc: 0.9991 - val_loss: 0.2399 - val_acc: 0.9746
Epoch 20/20
```

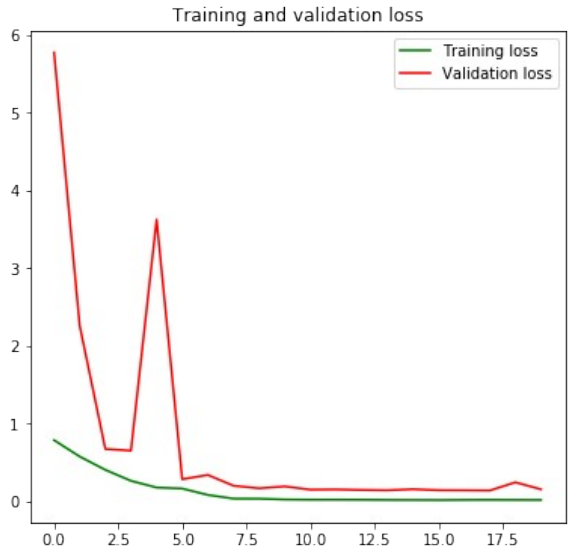
```
10000/10000 [==============================] - 93s 9ms/step - loss:
0.0126 - acc: 0.9987 - val_loss: 0.1502 - val_acc: 0.9832
```

Validation accuracy and loss

Let's show the train and validation accuracy on the same plot. As well, we will represent the train and validation loss on the same graph.

```
def plot_accuracy_and_loss(train_model):
    hist = train_model.history
    acc = hist['acc']
    val_acc = hist['val_acc']
    loss = hist['loss']
    val_loss = hist['val_loss']
    epochs = range(len(acc))
    f, ax = plt.subplots(1,2, figsize=(14,6))
    ax[0].plot(epochs, acc, 'g', label='Training accuracy')
    ax[0].plot(epochs, val_acc, 'r', label='Validation accuracy')
    ax[0].set_title('Training and validation accuracy')
    ax[0].legend()
    ax[1].plot(epochs, loss, 'g', label='Training loss')
    ax[1].plot(epochs, val_loss, 'r', label='Validation loss')
    ax[1].set_title('Training and validation loss')
    ax[1].legend()
    plt.show()
plot_accuracy_and_loss(train_model)
```





Let's also show the numeric validation accuracy and loss.

```
score = model.evaluate(X_val, y_val, verbose=0)
print('Validation loss:', score[0])
print('Validation accuracy:', score[1])
```

```
Validation loss: 0.15023201193418587
Validation accuracy: 0.9832
```

Validation accuracy per class

Let's show the validation accuracy per each class.

We start by predicting the labels for the validation set.

```
#get the predictions for the test data
predicted_classes = model.predict_classes(X_val)
#get the indices to be plotted
y_true = np.argmax(y_val,axis=1)
```

We create two indices, **correct** and **incorrect**, for the images in the validation set with class predicted correctly and incorrectly, respectively.

```
correct = np.nonzero(predicted_classes==y_true)[0]
incorrect = np.nonzero(predicted_classes!=y_true)[0]
```

We saw what is the number of correctly vs. incorrectly predicted values in the validation set.

We show here the classification report for the validation set, with the accuracy per class and overall.

```
target_names = ["Class {}:".format(i) for i in range(NUM_CLASSES)]
print(classification_report(y_true, predicted_classes,
target_names=target_names))
```

```
              precision    recall  f1-score   support

    Class 0:       0.98      0.99      0.98      5061
    Class 1:       0.99      0.98      0.98      4939

   micro avg       0.98      0.98      0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

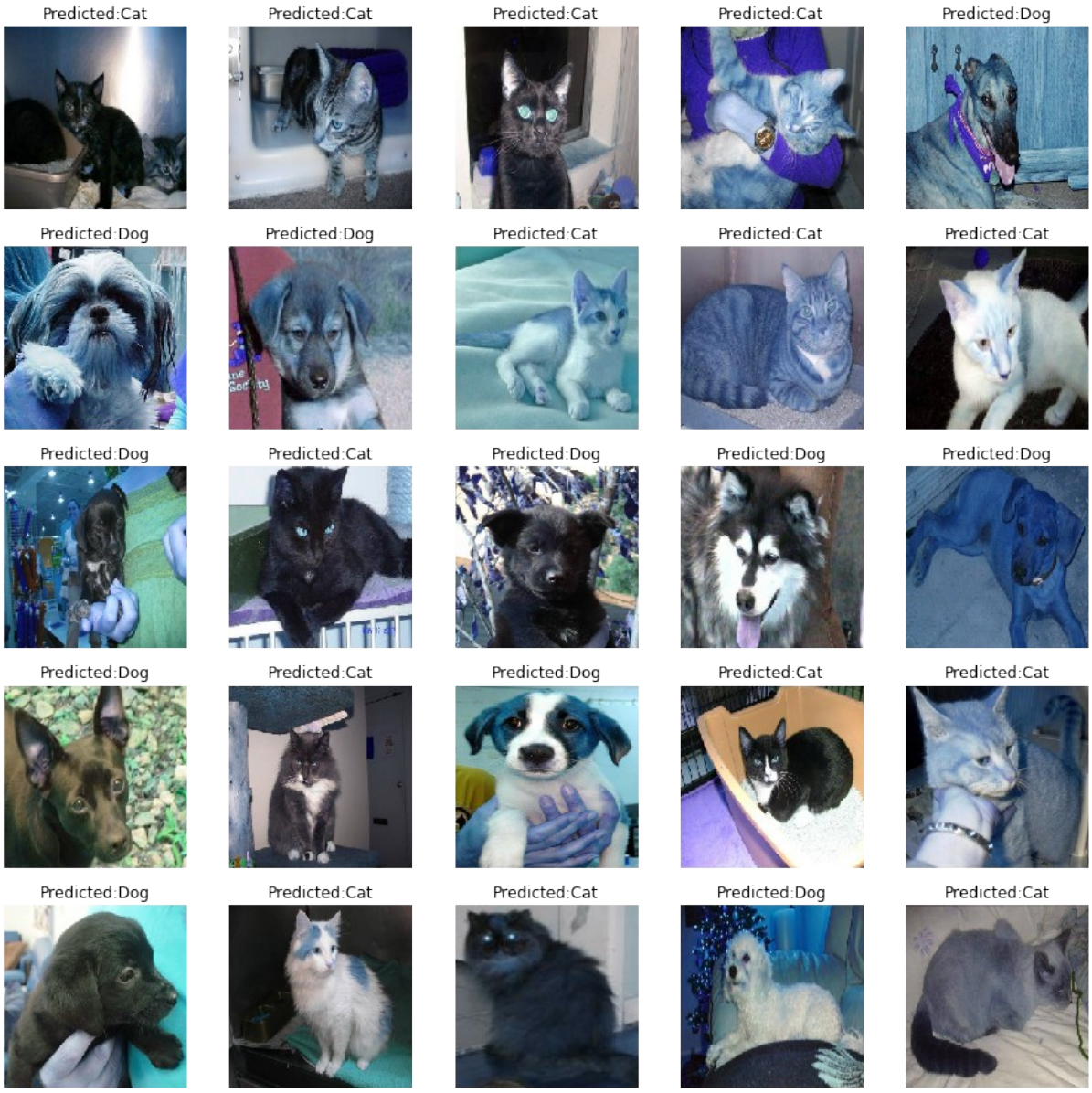
Prepare the submission

Show test images with predicted class

Let's show few of the test images with the predicted class. For this, we will have to predict the class.

```
f, ax = plt.subplots(5,5, figsize=(15,15))
for i,data in enumerate(test[:25]):
    img_num = data[1]
    img_data = data[0]
    orig = img_data
    data = img_data.reshape(-1,IMG_SIZE,IMG_SIZE,3)
    model_out = model.predict([data])[0]

    if np.argmax(model_out) == 1:
        str_predicted='Dog'
    else:
        str_predicted='Cat'
    ax[i//5, i%5].imshow(orig)
    ax[i//5, i%5].axis('off')
    ax[i//5, i%5].set_title("Predicted:{}".format(str_predicted))
plt.show()
```



Test data prediction

```
pred_list = []
img_list = []
for img in tqdm(test):
    img_data = img[0]
    img_idx = img[1]
    data = img_data.reshape(-1,IMG_SIZE,IMG_SIZE,3)
    predicted = model.predict([data])[0]
    img_list.append(img_idx)
    pred_list.append(predicted[1])
```

```
100%|██████████| 12500/12500 [02:10<00:00, 95.56it/s]
```

Submission file

Let's prepare now the submission file.

```
submission = pd.DataFrame({'id':img_list , 'label':pred_list})
submission.head()
submission.to_csv("submission.csv", index=False)
```

Conclusions

Using a pretrained model for Keras, ResNet-50, with a Dense model with softmax activation added on top and training with a reduced set of we were able to obtain quite good model in terms of validation accuracy.
The model was used to predict the classes of the images from the independent test set and results were submitted to test the accuracy of the prediction with fresh data.

References

[1] Dogs vs. Cats Redux: Kernels Edition, https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition
[2] ResNet pretrained models for Keras, https://www.kaggle.com/keras/resnet50