DAT340 - Assignment 5

May 20, 2022

Group PA 47 - Author: Stefano Ribes, ribes@chalmers.se

This Notebook can be viewed online at this link: https://colab.research.google.com/drive/1710kq0kidvPSyd3qjTW

1 Programming Assignment 5: Image Classification

1.1 Setup

[1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[2]:
```
import os

ASSIGNMENT_ID = 'assignment_5'

data_dir = os.path.join(os.path.abspath(''), 'drive', 'MyDrive')
data_dir = os.path.join(data_dir, 'Colab Notebooks', 'dat340', ASSIGNMENT_ID)
data_dir = os.path.join(data_dir, 'data')
if os.path.exists(data_dir):
    print(f'Directory "{data_dir}" exists')
else:
    print(f'WARNING! Directory "{data_dir}" does not exist!')
```

```
Directory "/content/drive/MyDrive/Colab Notebooks/dat340/assignment_5/data"
exists
```

[3]:
```
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('pdf', 'svg')
```

1.2 Introduction: Loading images from a directory

The very first step is to unpack the images and split them into train and validation sets.

[4]:
```
%%capture
import re

images_zip = re.escape(os.path.join(data_dir, 'a5_images.zip'))
images_path = '/tmp/'
!unzip $images_zip -d $images_path
```

In order to obtain the datasets, an `ImageDataGenerator` class is used to manage the images. The constructor can be configured in various way, we start from applying a rescaling factor. In particular, it allows us to normalize between 0 and 1 all the pixels of a given image.

[5]:
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

data_gen = ImageDataGenerator(rescale=1.0 / 255)
```

Once instanstiated, the `ImageDataGenerator` can be exploited to load images (and their respective labels) from a given directory and preprocess them according to some configuration flags. This is done via the `flow_from_directory()` method, which returns a Python generator.

In our case, we resize the images to be $64 \times 64 \times 3$ and belonging to a binary class, either a *car* or *other*, which is represented as either 0 or 1. The `ImageDataGenerator` also allows us to "batchify" the images into a stacked volume of dimensions $n \times 64 \times 64 \times 3$, where n represents the desired batch size. Ideally, the larger the batch size the better the optimizer is at minimizing the loss function, since it "sees" more training data. Moreover, having large batch sizes improves throughput and can lead to shorter training time. However, a larger batch size has a higher memory requirements which depends on the available hardware onto which the training algorithm is running.

We start from the training images and labels.

[6]:
```
imgdir = os.path.join(images_path, 'a5_images')
img_size = 64
batch_size = 128

train_generator = data_gen.flow_from_directory(os.path.join(imgdir, 'train'),
                                               target_size=(img_size, img_size),
                                               batch_size=batch_size,
                                               class_mode='binary',
                                               classes=['other', 'car'],
                                               seed=12345,
                                               shuffle=True)
# NOTE: The `class_indices` attribute returns a dictionary of class labels and
# their associated value
num_classes = len(train_generator.class_indices.keys())
```

```
Found 1600 images belonging to 2 classes.
```

We then continue with the validation images and their labels.

[7]:
```
validation_generator = data_gen.flow_from_directory(
    os.path.join(imgdir, 'validation'),
    target_size=(img_size, img_size),
    batch_size=batch_size,
    class_mode='binary',
    classes=['other', 'car'],
    seed=12345,
    shuffle=True)
```

```
Found 576 images belonging to 2 classes.
```

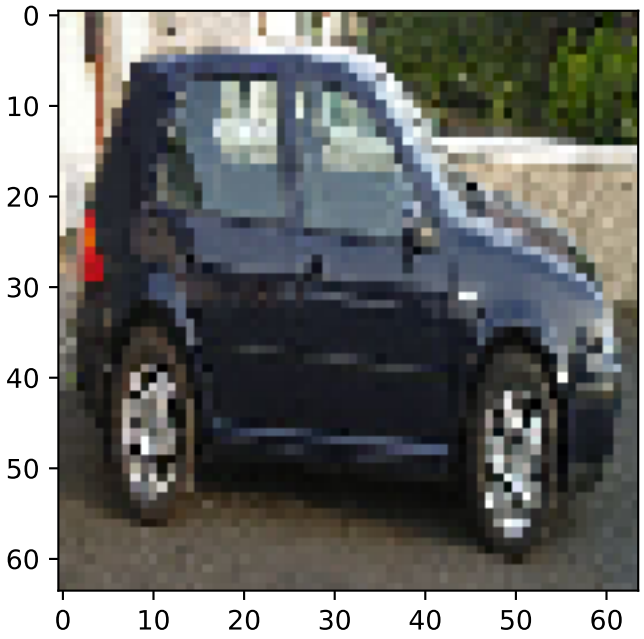
The method `flow_from_directory()` returns a generator which yields the X and Y batches of the corresponding dataset. We can inspect one training batch as follows.

[8]:
```
Xbatch, Ybatch = train_generator.next()
print(f'- X Batch dimensions: {Xbatch.shape}')
print(f'- X Batch size: {Xbatch.shape[0]}')
print(f'- Image dimensions: {Xbatch.shape[1:]}')
print(f'- Classes: {[c for c in validation_generator.class_indices.keys()]}')
print(f'- Y Batch dimensions: {Ybatch.shape}')
```

```
- X Batch dimensions: (128, 64, 64, 3)
- X Batch size: 128
- Image dimensions: (64, 64, 3)
- Classes: ['other', 'car']
- Y Batch dimensions: (128,)
```

As expected, each of the X batch elements has dimensions $n \times 64 \times 64 \times 3$, with $n = 128$ in our case. Each of the n elements is an image, that we can display as follows.

[9]:
```
from matplotlib import pyplot as plt

plt.imshow(Xbatch[4])
plt.show()
```



One can also check its corresponding label, which is 1, since it's an image of a car.

[10]:
```
Ybatch[4]
```

[10]:
```
1.0
```

Finally, before starting designing and training various classifiers, I setup a global dictionary to keep track of all the models and their performance.

[11]:
```
models = {}
```

Also, in order to have reproducible results, let's fix any random seed beforehand.

[12]:
```
import tensorflow as tf
import numpy as np

seed = 123456
tf.random.set_seed(seed)
np.random.seed(seed)
tf.keras.utils.set_random_seed(seed)
```

1.3 Part 1: Training a convolutional neural network

1.3.1 Saving and Loading Models

Before defining the model, let's define two helper functions to store and retrive trained Keras models from disk.

[13]:
```
def save_model(model, model_name):
    keras_dir = os.path.join(data_dir, model_name)
    keras_h5 = os.path.join(data_dir, model_name + '.h5')
    model.save(keras_dir)
    # model.save_weights(keras_h5) # Deprecated?
    model.save(keras_h5)
    print(f'Model saved at: {keras_h5}')

def load_model(model_name):
    keras_dir = os.path.join(data_dir, model_name)
    keras_h5 = os.path.join(data_dir, model_name + '.h5')
    print(f'keras_dir: {keras_dir}')
    print(f'keras_h5: {keras_h5}')
    if os.path.isfile(keras_h5):
        model = tf.keras.models.load_model(keras_h5)
        print(f'Model "{model_name}" loaded with weights.')
        return model
    if os.path.isdir(keras_dir):
        model = tf.keras.models.load_model(keras_dir)
        return model
    else:
        print(f'Model "{model_name}" not found in: {data_dir}')
    return None
```

1.3.2 Defining the Model

In designing the Convolutional Neural Network (CNN) model for the binary classification task at hand, I followed the following notes:

- The last dense layer shouldn't have an activation function thesholding the neurons output, *i.e.* it needs to be `linear`
- The stride parameter is usually set in the convolutional layers (CONV). If instead set in the pooling layers, one might end up going outside the output feature map of the CONV layer and not doing pooling at all!
- Since we are dealing with a binary classification task, the last layer should have an activation function whose image space is $[0, 1]$. A perfect candidate is the sigmoid function.

[14]:
```
from __future__ import print_function
import tensorflow as tf
import keras
```

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
from keras import Input

def make_convnet(model_name='baseline'):
    model = load_model(model_name)
    if model is not None:
        print(f'Model successfully loaded.')
    else:
        l2 = tf.keras.regularizers.l2(0.001)
        # NOTE: We are treating a binary classification problem, therefore we
        # shouldn't use a Softmax in the end.
        model = keras.Sequential(
            [
                Conv2D(32, kernel_size=5, strides=(3, 3), activation='relu',␣
↪input_shape=(img_size, img_size, 3), kernel_regularizer=l2),
                MaxPooling2D(pool_size=(2, 2)),
                Conv2D(64, kernel_size=3, strides=(2, 2), activation='relu',␣
↪kernel_regularizer=l2),
                MaxPooling2D(pool_size=(2, 2)),
                Flatten(),
                Dense(128, activation='linear', kernel_regularizer=l2),
                Dense(1, activation='sigmoid')
            ]
        )
        # NOTE: Again, it's a binary classification problem, we cannot use the
        # categorical_crossentropy loss function.
        model.compile(loss='binary_crossentropy',
                      optimizer=tf.keras.optimizers.Adam(learning_rate=0.01),
                      metrics=['accuracy'],)
    return model
```

1.3.3 Training

[15]:
```
# NOTE: The batch size is already defined in the data generator.
epochs = 15

model = make_convnet(model_name='baseline')
fit_info = model.fit(train_generator,
                     epochs=epochs,
                     verbose=1,
                     validation_data=validation_generator)
models['Baseline'] = {'model': model, 'history': fit_info.history}
```

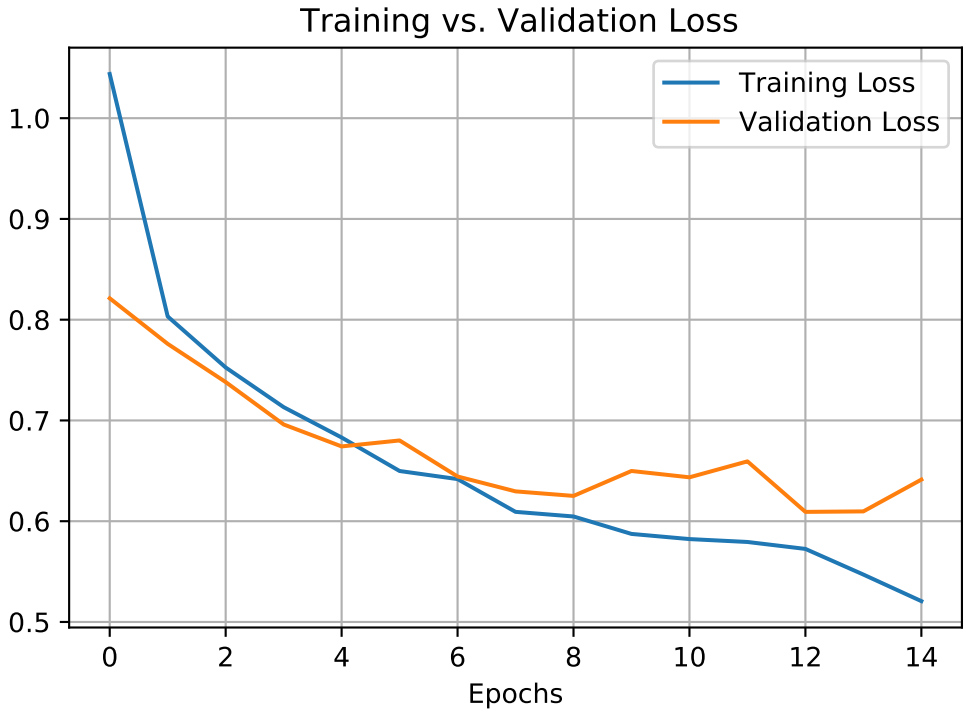
```
keras_dir: /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data/baseline
```

```
keras_h5: /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data/baseline.h5
Model "baseline" not found in: /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data
Epoch 1/15
13/13 [==============================] - 4s 260ms/step - loss: 1.0438 -
accuracy: 0.4950 - val_loss: 0.8212 - val_accuracy: 0.5069
Epoch 2/15
13/13 [==============================] - 3s 245ms/step - loss: 0.8032 -
accuracy: 0.5644 - val_loss: 0.7760 - val_accuracy: 0.5972
Epoch 3/15
13/13 [==============================] - 3s 253ms/step - loss: 0.7526 -
accuracy: 0.6394 - val_loss: 0.7381 - val_accuracy: 0.6111
Epoch 4/15
13/13 [==============================] - 3s 240ms/step - loss: 0.7132 -
accuracy: 0.6631 - val_loss: 0.6960 - val_accuracy: 0.6719
Epoch 5/15
13/13 [==============================] - 4s 297ms/step - loss: 0.6831 -
accuracy: 0.6731 - val_loss: 0.6742 - val_accuracy: 0.6667
Epoch 6/15
13/13 [==============================] - 3s 251ms/step - loss: 0.6498 -
accuracy: 0.6988 - val_loss: 0.6801 - val_accuracy: 0.6753
Epoch 7/15
13/13 [==============================] - 3s 261ms/step - loss: 0.6418 -
accuracy: 0.7081 - val_loss: 0.6444 - val_accuracy: 0.6997
Epoch 8/15
13/13 [==============================] - 3s 239ms/step - loss: 0.6093 -
accuracy: 0.7163 - val_loss: 0.6296 - val_accuracy: 0.7101
Epoch 9/15
13/13 [==============================] - 3s 246ms/step - loss: 0.6047 -
accuracy: 0.7212 - val_loss: 0.6251 - val_accuracy: 0.7083
Epoch 10/15
13/13 [==============================] - 3s 243ms/step - loss: 0.5874 -
accuracy: 0.7344 - val_loss: 0.6498 - val_accuracy: 0.6892
Epoch 11/15
13/13 [==============================] - 3s 269ms/step - loss: 0.5823 -
accuracy: 0.7406 - val_loss: 0.6436 - val_accuracy: 0.6875
Epoch 12/15
13/13 [==============================] - 3s 245ms/step - loss: 0.5794 -
accuracy: 0.7244 - val_loss: 0.6594 - val_accuracy: 0.6649
Epoch 13/15
13/13 [==============================] - 3s 250ms/step - loss: 0.5725 -
accuracy: 0.7350 - val_loss: 0.6093 - val_accuracy: 0.7066
Epoch 14/15
13/13 [==============================] - 3s 243ms/step - loss: 0.5470 -
accuracy: 0.7481 - val_loss: 0.6097 - val_accuracy: 0.7240
Epoch 15/15
13/13 [==============================] - 3s 246ms/step - loss: 0.5207 -
```

```
accuracy: 0.7719 - val_loss: 0.6413 - val_accuracy: 0.6944
```

Let's now plot the training and validation loss over the training epochs.

[16]:
```
plt.plot(models['Baseline']['history']['loss'], label='Training Loss')
plt.plot(models['Baseline']['history']['val_loss'], label='Validation Loss')
plt.grid('both')
plt.xlabel('Epochs')
plt.title('Training vs. Validation Loss')
plt.legend()
plt.show()
```



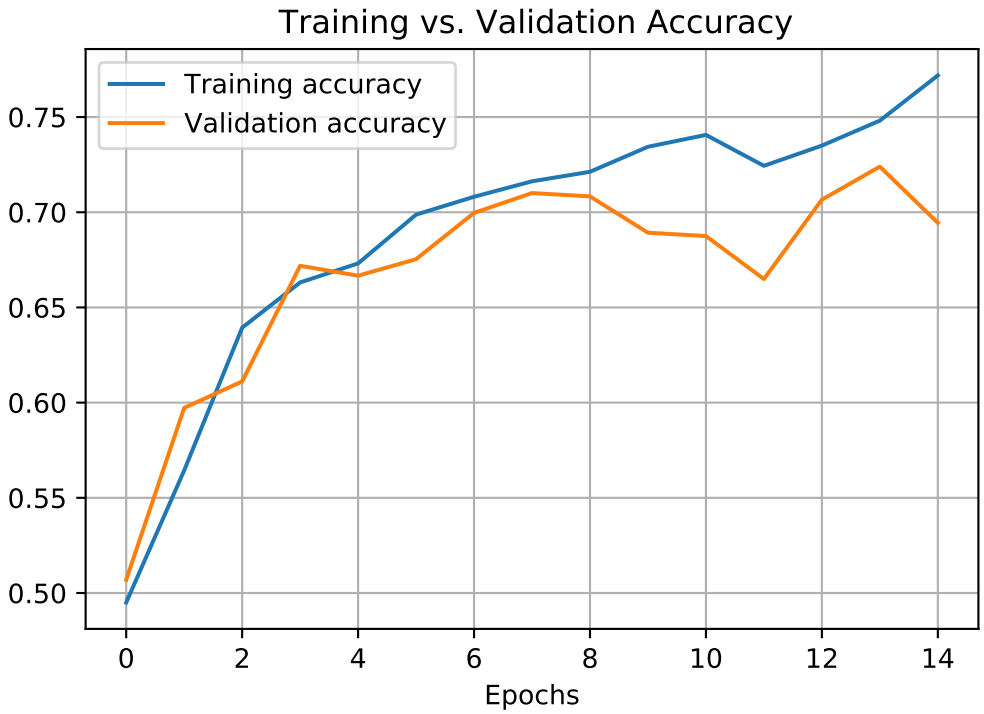
As we can see, both the training and validation loss curves have a downward trend, meaning that the model is not overfitting the training data.

In fact, if it was overfitting, we would have seen the validation loss first decreasing for a certain amount of epochs, then, at some point, diverge and increase, while the training loss would still go down.

This is why a technique called "early stopping" is sometimes effective against overfitting. As the name suggests, in early stopping the training algotihm keeps track of the model state until the validation loss starts increasing. At that point, the training stops and the model weights are restored to the values before the divergence.

We can now analyze and compare the training and validation accuracy.

[17]:
```
plt.plot(models['Baseline']['history']['accuracy'], label='Training accuracy')
plt.plot(models['Baseline']['history']['val_accuracy'], label='Validation␣
↪accuracy')
plt.grid('both')
plt.xlabel('Epochs')
plt.title('Training vs. Validation Accuracy')
plt.legend()
plt.show()
```



Despite "oscillating", the validation accuracy seems to have an upward trend, suggesting that there is no underfitting either.

[18]:
```
save_model(models['Baseline']['model'], 'baseline')
```

```
INFO:tensorflow:Assets written to: /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data/baseline/assets
Model saved at: /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data/baseline.h5
```

1.3.4 Evaluation

Since we are using the same set for both validation and testing, we see the same accuracy and loss values in the evaluation phase as in the last training epoch.

[19]:
```
loss, accuracy = models['Baseline']['model'].evaluate(validation_generator,␣
↪verbose=0)
models['Baseline']['accuracy'] = accuracy
models['Baseline']['loss'] = loss
print(f'Test loss: {loss:.4f}\nTest accuracy: {accuracy:.4f}')
```

```
Test loss: 0.6413
Test accuracy: 0.6944
```

1.4 Part 2: Data augmentation

1.4.1 Tuning the ImageDataGenerator

I tried experimenting with different configuration parameters. In the end, only applying an horizontal flip and having a rotation angle range of 20 degrees made the classifier achieve slightly better accuracy than the baseline.

Intuitively, since cars are "horizontally symmetrical", flipping the image horizontally is like seeing the same car from a different point of view. On the other hand, flipping vertically would not make much sense since features like the pixels of the "wheels" are typically at the bottom of the image.

[22]:
```
datagen_augmented = ImageDataGenerator(rescale=1.0 / 255,
                                       horizontal_flip=True,
                                    #    width_shift_range=0.2,
                                    #    height_shift_range=0.2,
                                       rotation_range=10)
train_generator_augmented = datagen_augmented.flow_from_directory(
    os.path.join(imgdir, 'train'),
    target_size=(img_size, img_size),
    batch_size=batch_size,
    class_mode='binary',
    classes=['other', 'car'],
    seed=12345,
    shuffle=True)
```

```
Found 1600 images belonging to 2 classes.
```

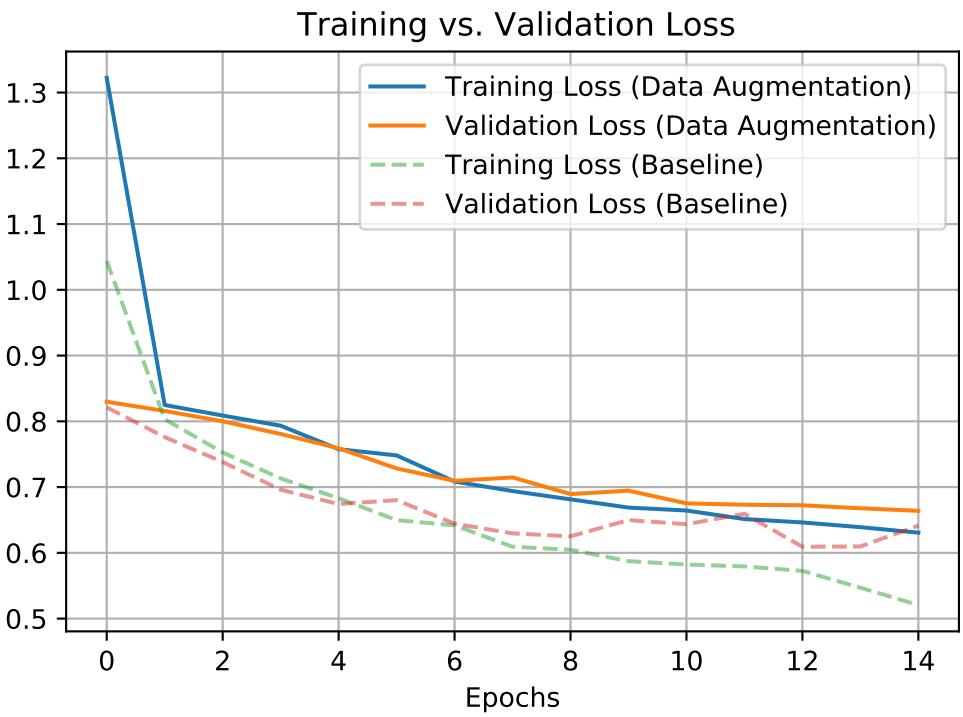
1.4.2 Training and Evaluation

[23]:
```
model = make_convnet(model_name='data_augmentation')
fit_info = model.fit(train_generator_augmented,
                     epochs=epochs,
                     verbose=1,
                     validation_data=validation_generator)
models['DataAugmentation'] = {'model': model, 'history': fit_info.history}
```

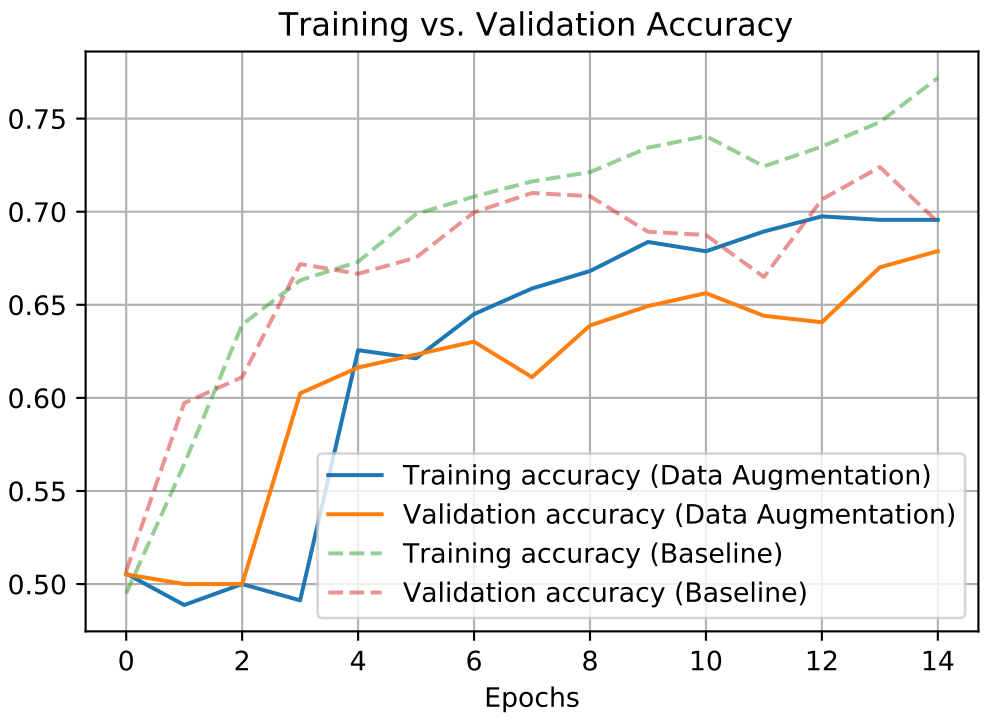
```
keras_dir: /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data/data_augmentation
keras_h5: /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data/data_augmentation.h5
Model "data_augmentation" not found in: /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data
Epoch 1/15
13/13 [==============================] - 5s 377ms/step - loss: 1.3223 -
accuracy: 0.5056 - val_loss: 0.8298 - val_accuracy: 0.5052
Epoch 2/15
13/13 [==============================] - 5s 348ms/step - loss: 0.8248 -
accuracy: 0.4888 - val_loss: 0.8158 - val_accuracy: 0.5000
Epoch 3/15
13/13 [==============================] - 4s 346ms/step - loss: 0.8088 -
accuracy: 0.5000 - val_loss: 0.8001 - val_accuracy: 0.5000
Epoch 4/15
13/13 [==============================] - 4s 344ms/step - loss: 0.7933 -
accuracy: 0.4913 - val_loss: 0.7808 - val_accuracy: 0.6024
Epoch 5/15
13/13 [==============================] - 5s 370ms/step - loss: 0.7576 -
accuracy: 0.6256 - val_loss: 0.7590 - val_accuracy: 0.6163
Epoch 6/15
13/13 [==============================] - 5s 371ms/step - loss: 0.7482 -
accuracy: 0.6212 - val_loss: 0.7284 - val_accuracy: 0.6233
Epoch 7/15
13/13 [==============================] - 5s 356ms/step - loss: 0.7084 -
accuracy: 0.6450 - val_loss: 0.7094 - val_accuracy: 0.6302
Epoch 8/15
13/13 [==============================] - 5s 359ms/step - loss: 0.6940 -
accuracy: 0.6587 - val_loss: 0.7147 - val_accuracy: 0.6111
Epoch 9/15
13/13 [==============================] - 5s 345ms/step - loss: 0.6813 -
accuracy: 0.6681 - val_loss: 0.6894 - val_accuracy: 0.6389
Epoch 10/15
13/13 [==============================] - 4s 342ms/step - loss: 0.6688 -
accuracy: 0.6837 - val_loss: 0.6946 - val_accuracy: 0.6493
Epoch 11/15
13/13 [==============================] - 4s 360ms/step - loss: 0.6644 -
```

```
accuracy: 0.6787 - val_loss: 0.6752 - val_accuracy: 0.6562
Epoch 12/15
13/13 [==============================] - 5s 348ms/step - loss: 0.6514 -
accuracy: 0.6894 - val_loss: 0.6732 - val_accuracy: 0.6441
Epoch 13/15
13/13 [==============================] - 4s 349ms/step - loss: 0.6462 -
accuracy: 0.6975 - val_loss: 0.6723 - val_accuracy: 0.6406
Epoch 14/15
13/13 [==============================] - 4s 346ms/step - loss: 0.6391 -
accuracy: 0.6956 - val_loss: 0.6678 - val_accuracy: 0.6701
Epoch 15/15
13/13 [==============================] - 5s 351ms/step - loss: 0.6307 -
accuracy: 0.6956 - val_loss: 0.6639 - val_accuracy: 0.6788
```

[24]:
```
plt.plot(models['DataAugmentation']['history']['loss'], '-', label='Training␣
 ↪Loss (Data Augmentation)')
plt.plot(models['DataAugmentation']['history']['val_loss'], '-',␣
 ↪label='Validation Loss (Data Augmentation)')
plt.plot(models['Baseline']['history']['loss'], '--', label='Training Loss␣
 ↪(Baseline)', alpha=0.5)
plt.plot(models['Baseline']['history']['val_loss'], '--', label='Validation␣
 ↪Loss (Baseline)', alpha=0.5)
plt.grid('both')
plt.xlabel('Epochs')
plt.title('Training vs. Validation Loss')
plt.legend()
plt.show()
```



[26]:
```
plt.plot(models['DataAugmentation']['history']['accuracy'], label='Training␣
 ↪accuracy (Data Augmentation)')
plt.plot(models['DataAugmentation']['history']['val_accuracy'],␣
 ↪label='Validation accuracy (Data Augmentation)')
plt.plot(models['Baseline']['history']['accuracy'], '--', label='Training␣
 ↪accuracy (Baseline)', alpha=0.5)
plt.plot(models['Baseline']['history']['val_accuracy'], '--', label='Validation␣
 ↪accuracy (Baseline)', alpha=0.5)
plt.grid('both')
plt.xlabel('Epochs')
plt.title('Training vs. Validation Accuracy')
plt.legend()
plt.show()
```



[27]:
```
save_model(models['DataAugmentation']['model'], 'data_augmentation')
```

```
INFO:tensorflow:Assets written to: /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data/data_augmentation/assets
Model saved at: /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data/data_augmentation.h5
```

[28]:
```
loss, accuracy = models['DataAugmentation']['model'].
 ↪evaluate(validation_generator, verbose=0)
models['DataAugmentation']['accuracy'] = accuracy
models['DataAugmentation']['loss'] = loss
print(f'Test loss: {loss:.4f}\nTest accuracy: {accuracy:.4f}')
```

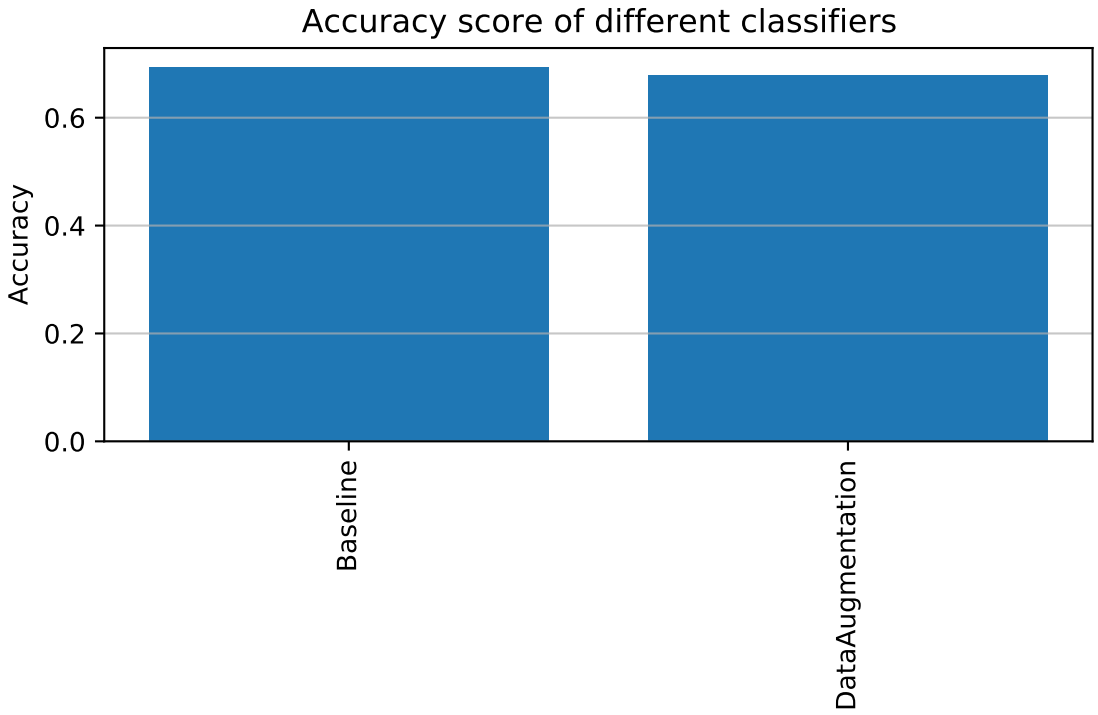
```
Test loss: 0.6639
Test accuracy: 0.6788
```

1.4.3 Baseline Comparison

[29]:
```
def plot_accuracy(model_focus=''):
    scores = []
    models_keys = []
    for model_type in models.keys():
        if model_focus in model_type:
```

```
            models_keys.append(model_type)
            print(f'{model_type} accuracy: {models[model_type]["accuracy"]:.
↪3f}s')
            scores.append(models[model_type]['accuracy'])
    # Alphabetically sort scores according to the model types
    scores = [x for _, x in sorted(zip(models_keys, scores))]
    linspace = [x for x in range(len(models_keys))]
    plt.bar(linspace, scores)
    plt.xticks(linspace, [f'{m}' for m in sorted(models_keys)], rotation=90)
    plt.grid(which='both', axis='y', alpha=0.7, zorder=1)
    plt.ylabel('Accuracy')
    plt.title('Accuracy score of different classifiers')
    plt.tight_layout()
    # plt.savefig(os.path.join(data_dir, f'accuracy.pdf'))
    plt.show()

plot_accuracy()
```

```
Baseline accuracy: 0.694s
DataAugmentation accuracy: 0.679s
```



By comparing the two loss curves and accuracy performance, I don't see any significant difference in performance.

1.5 Interlude: Applying a pre-trained convolutional neural network

[30]:
```
from tensorflow.keras import applications
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.applications.vgg16 import decode_predictions,␣
↪preprocess_input

vggmodel = applications.VGG16(weights='imagenet', include_top=True)
vggmodel.summary()
```

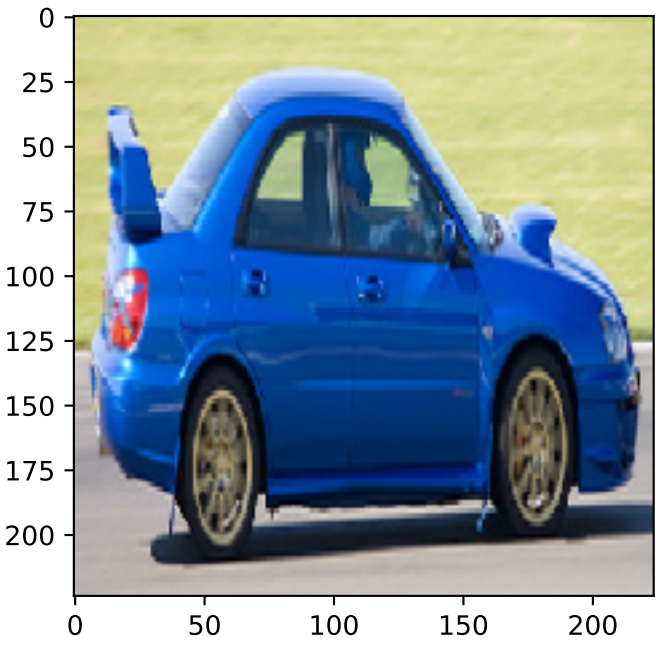
```
Downloading data from https://storage.googleapis.com/tensorflow/keras-
applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels.h5
553467904/553467096 [==============================] - 7s 0us/step
553476096/553467096 [==============================] - 7s 0us/step
Model: "vgg16"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 input_1 (InputLayer)         [(None, 224, 224, 3)]     0

 block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792

 block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928

 block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0

 block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856

 block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584

 block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0

 block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168

 block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080

 block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080

 block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0

 block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160

 block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
```

```
 block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

 flatten (Flatten)           (None, 25088)             0

 fc1 (Dense)                 (None, 4096)              102764544

 fc2 (Dense)                 (None, 4096)              16781312

 predictions (Dense)         (None, 1000)              4097000

=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
_________________________________________________________________
```

As one can see, the VGG16 network is quite deep and includes a series of convolutional blocks ending with two fully connected layers.

Let's visualize the performance of VGG16 on a specific example.

[31]:
```
test_img = load_img('/tmp/a5_images/train/car/0014.jpg', target_size=(224, 224))
plt.imshow(test_img)
plt.show()
```



The test image needs to be preprocessed accordingly before being fed to the network.

[32]:
```
test_img = img_to_array(test_img)
test_img = preprocess_input(test_img).reshape(1, 224, 224, 3)
decode_predictions(vggmodel.predict(test_img))
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/dat
a/imagenet_class_index.json
40960/35363 [==================================] - 0s 0us/step
49152/35363 [==========================================] - 0s 0us/step
```

[32]:
```
[[('n03930630', 'pickup', 0.8703232),
  ('n04037443', 'racer', 0.06258986),
  ('n02974003', 'car_wheel', 0.022384971),
  ('n04461696', 'tow_truck', 0.015090675),
  ('n04285008', 'sports_car', 0.013408977)]]
```

As we can seen, the network returns the top-5 most confident classes. In our case, the network is very confident that the image represents a *pickup* (with a confidence/probability of 0.87), even though I believe the image rather depicts a *racer* car (0.06) or a *sports_car* (0.01).

Both my classifications are included in the top-5 predictions, so overall I believe that the network result is quite impressive. I still have some doubts on why there is such a gap in confidence between the top-1 and the second one.

1.6 Part 3: Using VGG-16 as a feature extractor

1.6.1 Extract Features

I now instantiate a new VGG16 model, this time with an option for resizing the expected input image to the one of our original problem at hand, *i.e.* into a shape of $64 \times 64 \times 3$.

[33]:
```
feature_extractor = applications.VGG16(include_top=False, weights='imagenet',
                                       input_shape=(img_size, img_size, 3))
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-
applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58892288/58889256 [==============================] - 0s 0us/step
58900480/58889256 [==============================] - 0s 0us/step
```

Our classifier will be "attached" after the VGG16 network. In order to emulate that, we precompute the output of the VGG16 networks, *i.e.* its final output feature map.

[34]:
```
import numpy as np

def create_vgg16_features(datadir='train'):
    features_dataset = os.path.join(data_dir, datadir + '_features.npy')
    if os.path.isfile(features_dataset):
        print(f'Extracted features from {datadir} dataset into␣
 ↪{features_dataset} file.')
        return
    vgg_data_gen = ImageDataGenerator(preprocessing_function=preprocess_input)
    img_generator = vgg_data_gen.flow_from_directory(
        os.path.join(imgdir, datadir),
        target_size=(img_size, img_size),
        batch_size=batch_size,
        class_mode='binary',
        classes=['other', 'car'],
        seed=12345,
        shuffle=False)
    cnn_features = feature_extractor.predict(img_generator)
    with open(features_dataset, 'wb') as f:
        np.save(f, cnn_features)
    print(f'Extracted features from {datadir} dataset into {features_dataset}␣
 ↪file.')
```

[35]:
```
create_vgg16_features('train')
create_vgg16_features('validation')
```

```
Extracted features from train dataset into /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data/train_features.npy file.
Extracted features from validation dataset into /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data/validation_features.npy file.
```

1.6.2 Train the Classifier on the Extracted Features

As the VGG16 is quite potent, I thought that the backend classifier should not be overcomplicated nor too deep. Hence, I decided to implement it as two rather small fully-connected layers.

[36]:
```
def make_backend(model_name='backend'):
    model = load_model(model_name)
    if model is not None:
        print(f'Model successfully loaded.')
    else:
        l2 = tf.keras.regularizers.l2(0.001)
        # NOTE: We are treating a binary classification problem, therefore we
        # shouldn't use a Softmax in the end.
        model = keras.Sequential(
            [
                Flatten(),
                Dense(256, activation='linear', kernel_regularizer=l2),
                Dense(1, activation='sigmoid')
            ]
        )
        # NOTE: Again, it's a binary classification problem, we cannot use the
        # categorical_crossentropy loss function.
        model.compile(loss='binary_crossentropy',
                      optimizer=tf.keras.optimizers.Adam(learning_rate=0.01),
                      metrics=['accuracy'],)
    return model
```

I can now define a function that loads the precomputed features, builds the backend model and finally trains it.

[37]:
```
def get_labels(n):
    return np.array([0] * (n // 2) + [1] * (n // 2))

def train_on_cnnfeatures(epochs=10):
    # Load the precomputed features from files
    train_features = os.path.join(data_dir, 'train_features.npy')
    validation_features = os.path.join(data_dir, 'validation_features.npy')
    with open(train_features, 'rb') as f:
        x_train = np.load(f)
    with open(validation_features, 'rb') as f:
        x_test = np.load(f)
    # Generate the corresponding labels
    y_train = get_labels(x_train.shape[0])
    y_test = get_labels(x_test.shape[0])
    # Instantiate and train the backend model
    model = make_backend()
    fit_info = model.fit(x=x_train, y=y_train,
                         epochs=epochs,
```

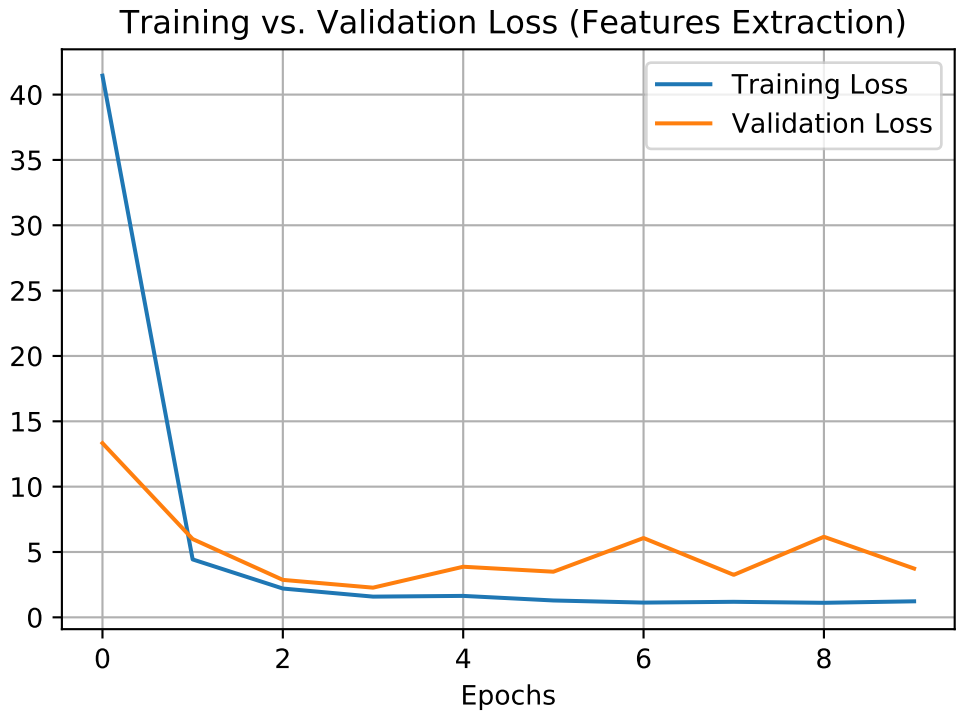
```
                    verbose=1,
                    validation_data=(x_test, y_test))
    return x_test, y_test, fit_info, model
```

Now that everything is setup, let's start training the backend model.

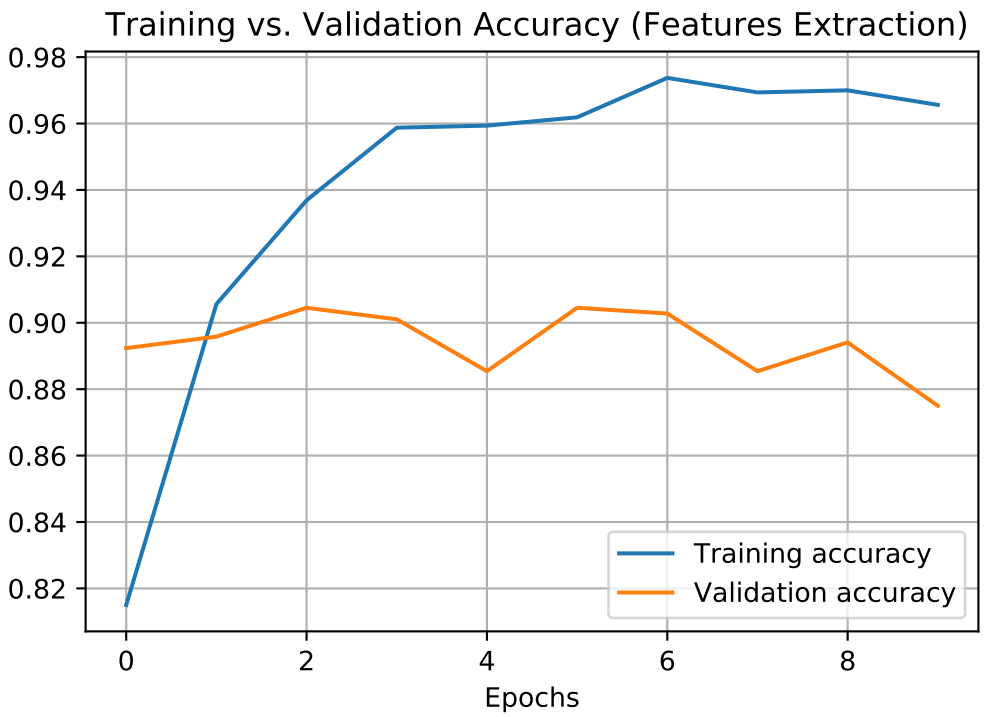
[38]:
```
x_test, y_test, fit_info, model = train_on_cnnfeatures()
models['Backend'] = {'model': model, 'history': fit_info.history}
```

```
keras_dir: /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data/backend
keras_h5: /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data/backend.h5
Model "backend" not found in: /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data
Epoch 1/10
50/50 [==============================] - 1s 9ms/step - loss: 41.4473 - accuracy:
0.8150 - val_loss: 13.3288 - val_accuracy: 0.8924
Epoch 2/10
50/50 [==============================] - 0s 7ms/step - loss: 4.4316 - accuracy:
0.9056 - val_loss: 5.9905 - val_accuracy: 0.8958
Epoch 3/10
50/50 [==============================] - 0s 7ms/step - loss: 2.2039 - accuracy:
0.9369 - val_loss: 2.8659 - val_accuracy: 0.9045
Epoch 4/10
50/50 [==============================] - 0s 7ms/step - loss: 1.5847 - accuracy:
0.9588 - val_loss: 2.2707 - val_accuracy: 0.9010
Epoch 5/10
50/50 [==============================] - 0s 7ms/step - loss: 1.6404 - accuracy:
0.9594 - val_loss: 3.8713 - val_accuracy: 0.8854
Epoch 6/10
50/50 [==============================] - 0s 7ms/step - loss: 1.2903 - accuracy:
0.9619 - val_loss: 3.4906 - val_accuracy: 0.9045
Epoch 7/10
50/50 [==============================] - 0s 6ms/step - loss: 1.1283 - accuracy:
0.9737 - val_loss: 6.0680 - val_accuracy: 0.9028
Epoch 8/10
50/50 [==============================] - 0s 6ms/step - loss: 1.1882 - accuracy:
0.9694 - val_loss: 3.2502 - val_accuracy: 0.8854
Epoch 9/10
50/50 [==============================] - 0s 6ms/step - loss: 1.1120 - accuracy:
0.9700 - val_loss: 6.1642 - val_accuracy: 0.8941
Epoch 10/10
50/50 [==============================] - 0s 6ms/step - loss: 1.2260 - accuracy:
0.9656 - val_loss: 3.7325 - val_accuracy: 0.8750
```

[39]:
```
plt.plot(models['Backend']['history']['loss'], label='Training Loss')
plt.plot(models['Backend']['history']['val_loss'], label='Validation Loss')
plt.grid('both')
plt.xlabel('Epochs')
plt.title('Training vs. Validation Loss (Features Extraction)')
plt.legend()
plt.show()
```



[40]:
```
plt.plot(fit_info.history['accuracy'], label='Training accuracy')
plt.plot(fit_info.history['val_accuracy'], label='Validation accuracy')
plt.grid('both')
plt.xlabel('Epochs')
plt.title('Training vs. Validation Accuracy (Features Extraction)')
plt.legend()
plt.show()
```



[41]:
```
save_model(models['Backend']['model'], 'backend')
```

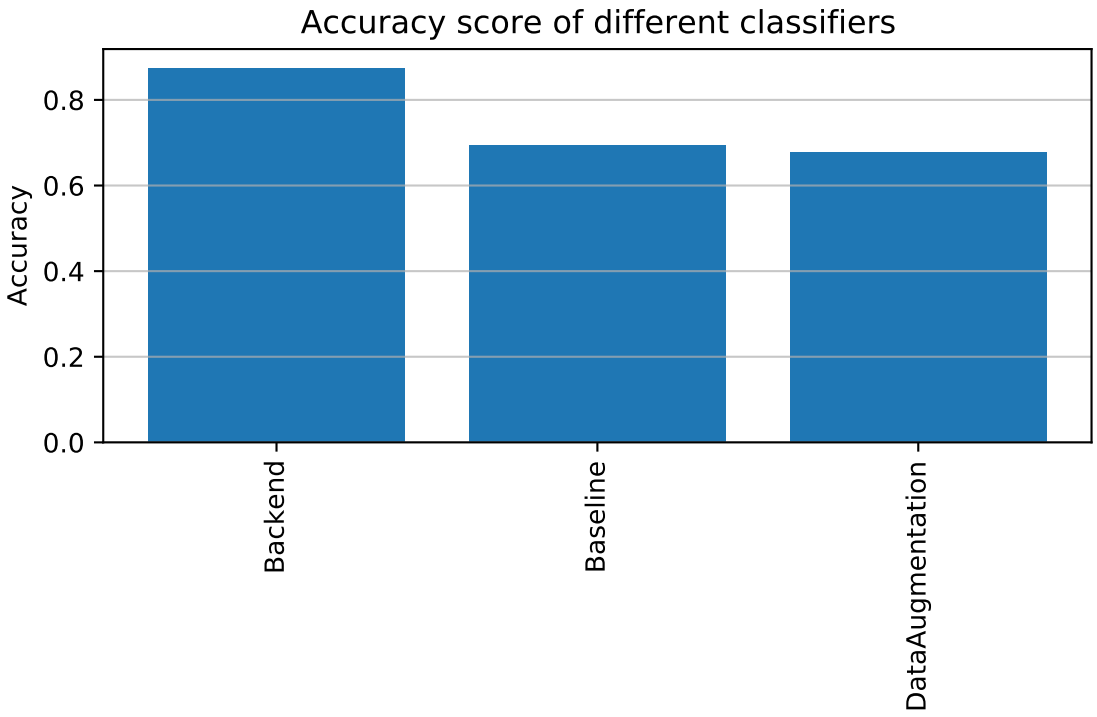
```
INFO:tensorflow:Assets written to: /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data/backend/assets
Model saved at: /content/drive/MyDrive/Colab
Notebooks/dat340/assignment_5/data/backend.h5
```

[42]:
```
loss, accuracy = models['Backend']['model'].evaluate(x=x_test, y=y_test,␣
↪verbose=0)
models['Backend']['accuracy'] = accuracy
models['Backend']['loss'] = loss
print(f'Test loss: {loss:.4f}\nTest accuracy: {accuracy:.4f}')
```

```
Test loss: 3.7325
Test accuracy: 0.8750
```

[43]:
```
plot_accuracy()
```

```
Baseline accuracy: 0.694s
DataAugmentation accuracy: 0.679s
Backend accuracy: 0.875s
```



We can see that the accuracy is now significatly higher compared to the previous models. This is because the VGG16 network is **more capable of detecting and extracting significant features** from the given images.

The VGG16 network was trained on powerful hardware and on a large dataset like ImageNet, and became a state-of-the-art CNN network for image classification. Compared to our baseline, VGG16 consists of many more layers, *i.e.* is deeper, and this generally trades-off accuracy with training time. Moreover, more layers typically means extracting higher level features, *i.e.* not just simple shapes like edges and color contrasts, but also features like wheels, windows, sky patches, *et cetera*. On top of that, its CONV layers include many more parameters, *i.e.* filters/kernels, which are able to extract many more features compared to our smaller CONV layers.

Since there are more significant and more features coming from the input image, our classifier is somewhat facilitated in classifying car images. Intuitively, the VGG16 network can pass to the backend model more precise information on whether there are features of cars in the supplied image. Based on that, the backend model can then better discriminate between images of cars that contain those features.

1.7 Part 4: Visualizing the learned features

Let's first take a look at the dimensions of the kernel weight in the first VGG16 CONV layer.

[44]:
```
first_layer_weights = vggmodel.get_weights()[0]
first_layer_weights.shape
```

[44]:
```
(3, 3, 3, 64)
```

We can see that there are 64 kernels and each and every one of them is a $3 \times 3 \times 3$ cube, or also a three-channel image of size 3×3.

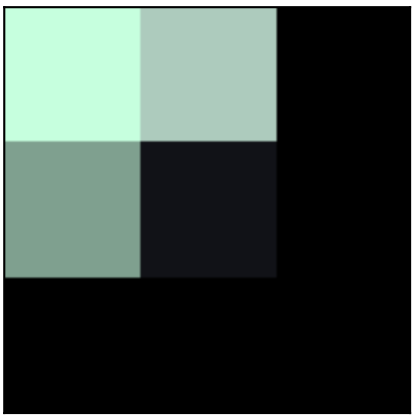
For other Convolutional layers in the network, each kernal shape might be of $3 \times 3 \times n$, or $5 \times 5 \times n$, meaning that there are n channels for each kernel "image". Because of that, it might be cumbersome to visualize those layers' weights and so I'll only focus on the first layer.

The following is a function to extract and make more visually appealing a single kernel image.

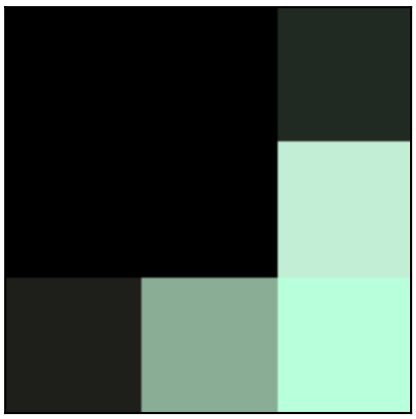
[45]:
```
def kernel_image(weights, i, positive):
    # Extract the convolutional kernel at position i.
    k = weights[:,:,:,i].copy()
    if not positive:
        k = -k
    # Clip the values: if we're looking for positive values, just keep the
    # positive part; vice versa for the negative values.
    k *= k > 0
    # Rescale the colors, to make the images less dark.
    m = k.max()
    if m > 1e-3:
        k /= m
    return k
```

Let's finally plot all the 64 kernel weights of $3 \times 3 \times 3$ "pixels"/numerical values.

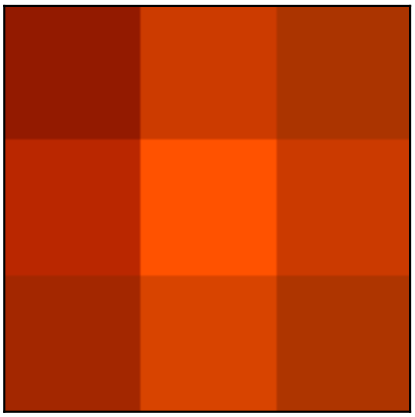
[46]:
```
weights = first_layer_weights = vggmodel.get_weights()[0]
num_filters_to_plot = 8
for i in range(num_filters_to_plot * 2):
    plt.subplot(1, 2, (i % 2) + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(kernel_image(weights, i // 2, positive=i % 2 == 0))
    plt.xlabel(f'{"positive" if i % 2 == 0 else "negative"} kernel n.{i // 2}')
    if i % 2 == 1:
        plt.show()
```



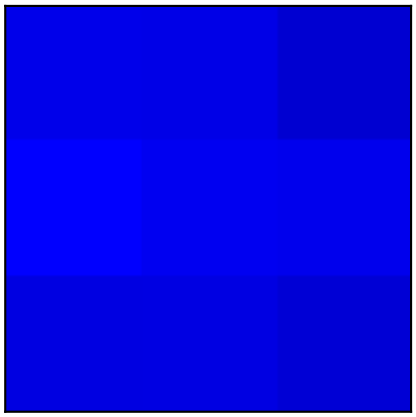
positive kernel n.0



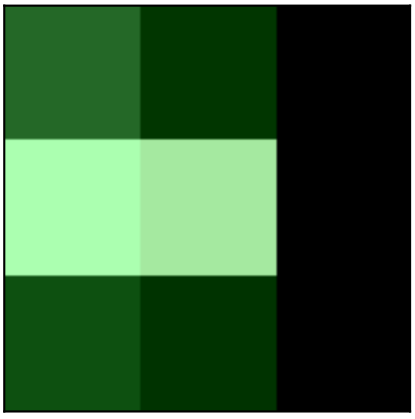
negative kernel n.0



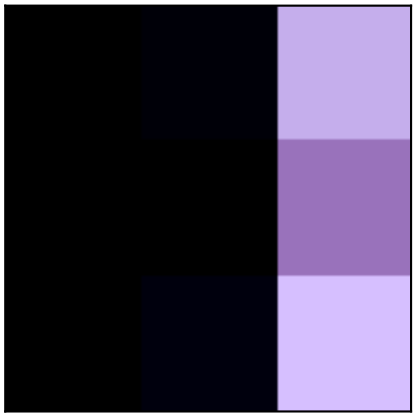
positive kernel n.1



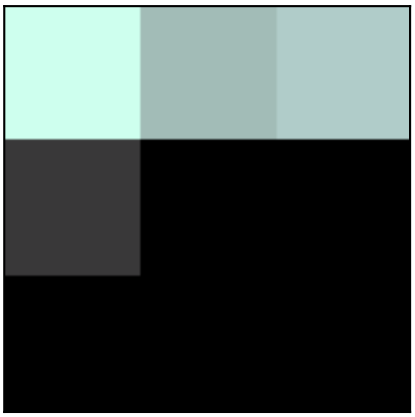
negative kernel n.1



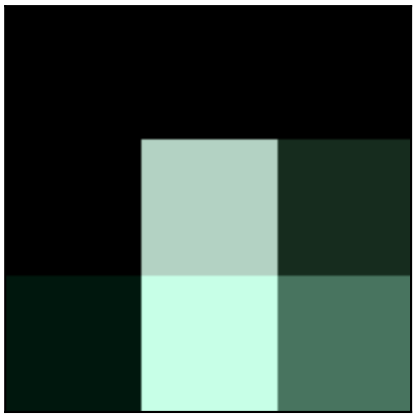
positive kernel n.2



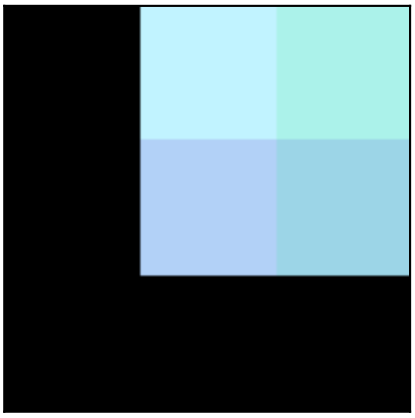
negative kernel n.2



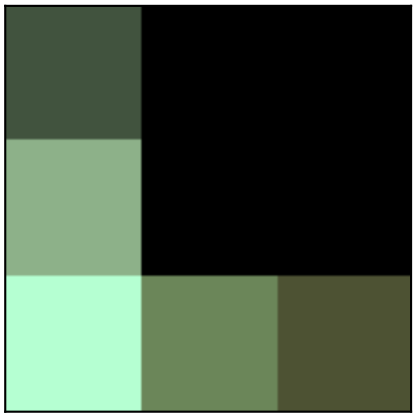
positive kernel n.3



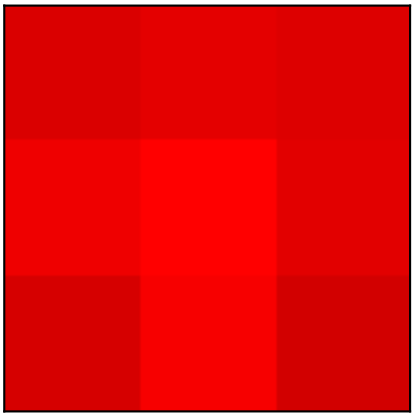
negative kernel n.3



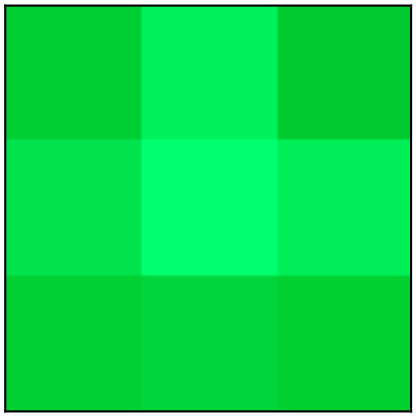
positive kernel n.4



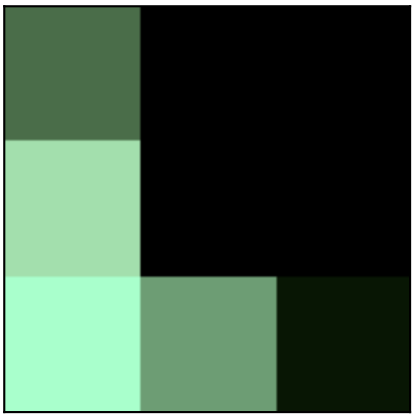
negative kernel n.4



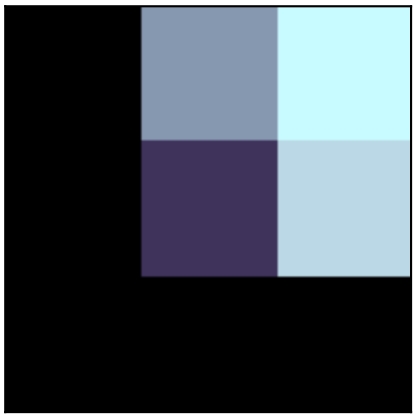
positive kernel n.5



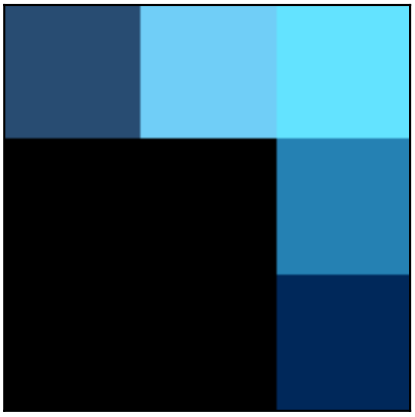
negative kernel n.5



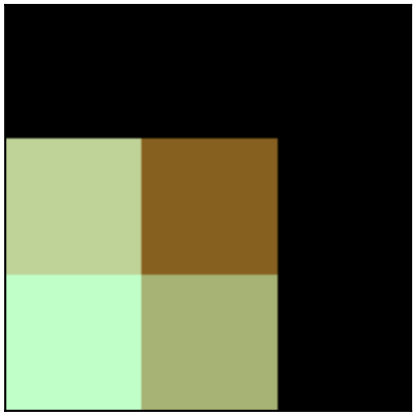
positive kernel n.6



negative kernel n.6



positive kernel n.7



negative kernel n.7

By looking the the images, it is hard to identify the features of all the kernels. However, by looking for example at kernel n.0 and kernel n.7 that they identify/detect a gradient of light-dark pixels. In particular, kernel n.0 a gradient in the top-left corner, whereas the kernel n.7 a gradient in the bottom-left corner.

2 Converting Notebook to PDF

The following two cells can be ignored for grading, as they just convert this notebook into a PDF file.

[ ]:
```
%%capture
!apt-get update
!apt-get install -y texlive-xetex texlive-fonts-recommended␣
 ↪texlive-plain-generic
!apt-get install -y inkscape
!add-apt-repository -y universe
!add-apt-repository -y ppa:inkscape.dev/stable
!apt-get update -y
!apt install -y inkscape
```

[ ]:
```
%%capture
import re

ASSIGNMENT_NAME = 'DAT340 - Assignment ' + ASSIGNMENT_ID.split('_')[1]
pdf_dir = os.path.join(os.path.abspath(''), 'drive', 'MyDrive')
pdf_dir = os.path.join(pdf_dir, 'Colab Notebooks', 'dat340', ASSIGNMENT_ID)
pdf_filename = re.escape(os.path.join(pdf_dir, ASSIGNMENT_NAME)) + '.ipynb'

!jupyter nbconvert --to pdf --TemplateExporter.exclude_input=False $pdf_filename
```

[ ]:
```

```