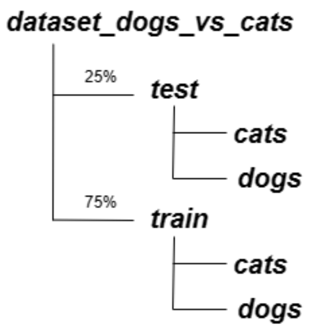
Dogs vs Cats

The goal of this project was to understand how we could build a convolutional neural network (CNN) model to perform a binary classification between cats and dogs using Keras, and how different approaches could affect the results.

Different subjects were explored such as different architectures (VGG with 1, 2 and 3 blocks), regularization techniques (dropout and data augmentation) and transfer learning (VGG16 and ResNet50).

Dataset

The original dataset was retrieved from Kaggle and contains 12500 images of cats and 12500 images of dogs. The dataset is coloured (RGB format) and it was reshaped to 200x200 format and divided into 75% for training and 25% for validation.



Defining and running the models

In this section, it’s given a small explanation of the models that were used, how we have defined them and ran them, with the code also presented.

In this section, it’s given a small explanation of the models that were used, it’s shown the implementation of the code and the results obtained. The models that were studied were:

- One Block VGG
- Two Block VGG
- Three Block VGG
- Three Block VGG + Dropout
- Three Block VGG + Data Augmentation
- VGG16
- ResNet50

1. VGG

First, we started by exploring the VGG architecture. This consists of using a convolutional layer with 3x3 filters followed by a max pooling layer, which forms a block.

We have observed the difference in the results when using one, two and three VGG blocks.

As it can be seen from the results, with the increase in the number of VGG blocks there is an improvement in the accuracy of the model. However, there is also an overfitting of the data at around 5 epochs. To study how overfitting could possibly be reduced, regularization techniques such as dropout and data augmentation were applied.

1.1. One Block VGG Model

1.1.1. *Defining the model*

```
def one_block_vgg():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
   padding='same', input_shape=(200, 200, 3)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(1, activation='sigmoid'))
    # compile model
    opt = SGD(learning_rate=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

1.1.2. *Fitting and evaluating the model*

A function to plot the accuracy and loss of the model both in train and validation datasets was created. This function was used for every model and the code used is the following.

```
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='blue', label='train')
    pyplot.plot(history.history['val_loss'], color='orange', label='test')
    # plot accuracy
```

```
        pyplot.subplot(212)

        pyplot.title('Classification Accuracy')

        pyplot.plot(history.history['accuracy'], color='blue', label='train')

        pyplot.plot(history.history['val_accuracy'], color='orange', label='test')

        pyplot.show()
```

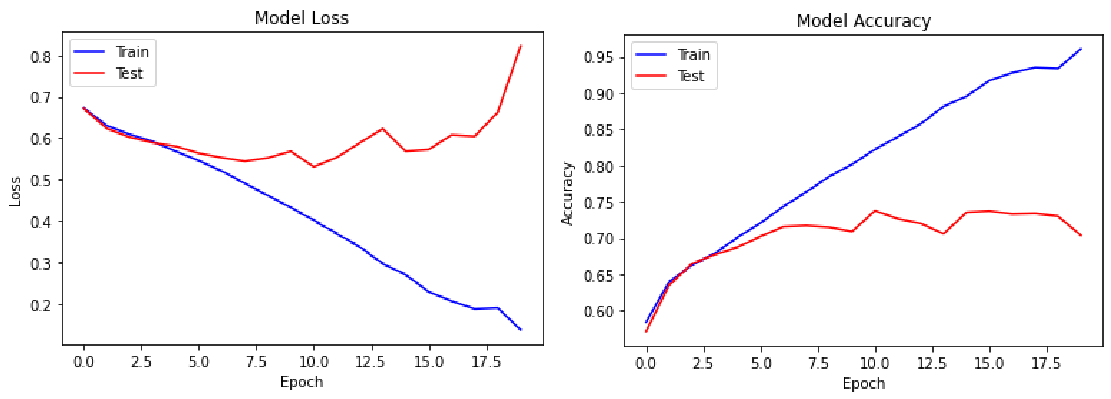
To fit and evaluate the model, a function was created where the defined model is called, fitted to the data and evaluated. The *summarize_diagnostics()* function is also called to plot the accuracy and the loss. This function is named as *run_test_harness_1()* and it's used for the VGG 1, 2 and 3 models and for the added dropout model.

```
def run_test_harness_1(model_string):

        #print(model_string)

        # define model

        if model_string == "1blockvgg":

                model = one_block_vgg()

                ep = 20

        if model_string == "2blockvgg":

                model = two_block_vgg()

                ep = 20

        if model_string == "3blockvgg":

                model = three_block_vgg()

                ep = 20

        if model_string == "3blockvgg_dropout":

                model = three_block_vgg_dropout()

                ep = 50

        # create data generator

        datagen = ImageDataGenerator(rescale=1.0/255.0)

        # prepare iterators

        train_it = datagen.flow_from_directory('dataset_dogs_vs_cats/train/',

        class_mode='binary', batch_size=64, target_size=(200, 200))

        test_it = datagen.flow_from_directory('dataset_dogs_vs_cats/test/',

        class_mode='binary', batch_size=64, target_size=(200, 200))

        # fit model
```

```
history = model.fit(train_it, steps_per_epoch=len(train_it),
validation_data=test_it, validation_steps=len(test_it), epochs=ep, verbose=1)
# evaluate model
_, acc = model.evaluate(test_it, steps=len(test_it), verbose=0)
print('> %.3f' % (acc * 100.0))
# learning curves
summarize_diagnostics(history)
```

1.1.3. Results

Accuracy: 70.363%



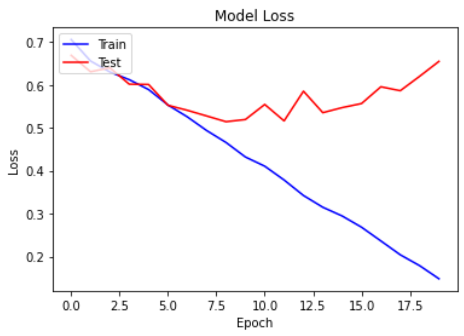
1.2. Two Block VGG Model

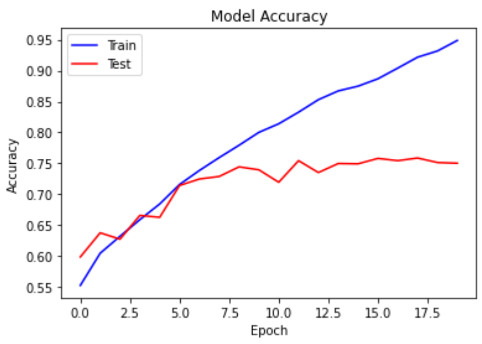
1.2.1. Defining the model

```
def two_block_vgg():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
   padding='same', input_shape=(200, 200, 3)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
   padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(1, activation='sigmoid'))
    # compile model
    opt = SGD(learning_rate=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])

    return model
```

1.2.2. Results

Accuracy: 75.028%





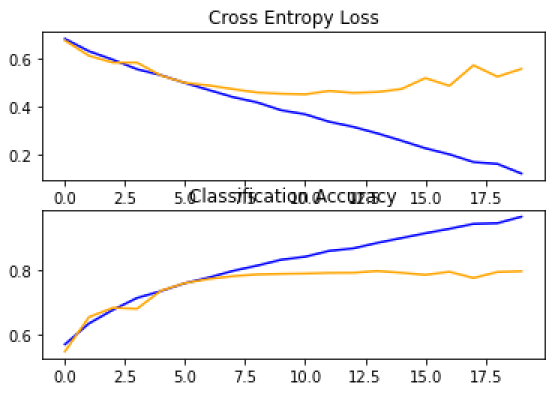
1.3. Three Block VGG Model

1.3.1. Defining the model

```
def three_block_vgg():

    model = Sequential()

    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',

   padding='same', input_shape=(200, 200, 3)))

    model.add(MaxPooling2D((2, 2)))

    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',

   padding='same'))

    model.add(MaxPooling2D((2, 2)))

    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',

   padding='same'))

    model.add(MaxPooling2D((2, 2)))

    model.add(Flatten())

    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))

    model.add(Dense(1, activation='sigmoid'))

    # compile model

    opt = SGD(learning_rate=0.001, momentum=0.9)

    model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])

    return model
```

1.3.2. Results

Accuracy: 79.486%



2. Regularization techniques

To understand how the overfitting problem can be addressed we have explored dropout and data augmentation. The VGG3 model was used as the baseline model to implement these techniques.

2.1. Three Block VGG + Dropout Model

As it has been said before, dropout was applied has a regularization technique to address overfitting. It consists of randomly disabling a fraction of inputs to a layer during training. A dropout rate of 20% was applied after each VGG block and a dropout rate of 50% was applied after the fully connected layer.

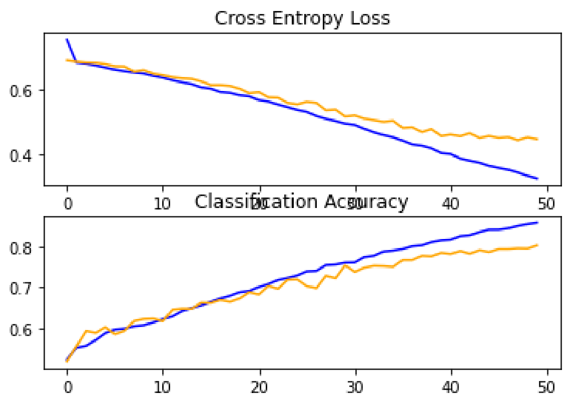
From the results it can be observed that both accuracy and overfitting have been improved.

2.1.1. Defining the model

```
def three_block_vgg_dropout():

        model = Sequential()

        model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',

        padding='same', input_shape=(200, 200, 3)))

        model.add(MaxPooling2D((2, 2)))

        model.add(Dropout(0.2))

        model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',

        padding='same'))

        model.add(MaxPooling2D((2, 2)))

        model.add(Dropout(0.2))

        model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
        model.add(MaxPooling2D((2, 2)))
        model.add(Dropout(0.2))
        model.add(Flatten())
        model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
        model.add(Dropout(0.5))
        model.add(Dense(1, activation='sigmoid'))
        # compile model
        opt = SGD(lr=0.001, momentum=0.9)
        model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
        return model
```

2.1.2. Results

Accuracy: 80.295%



2.2. Three Block VGG + Data Augmentation Model

In order to implement data augmentation a new test harness function was developed to fit and evaluate the VGG3 model.

By making transformations in the original dataset, data augmentation can improve the model performance by both increasing the size of the dataset and making it more robust to noise. In this case, it was applied a 10% random horizontal and vertical shift, and a random horizontal flip.

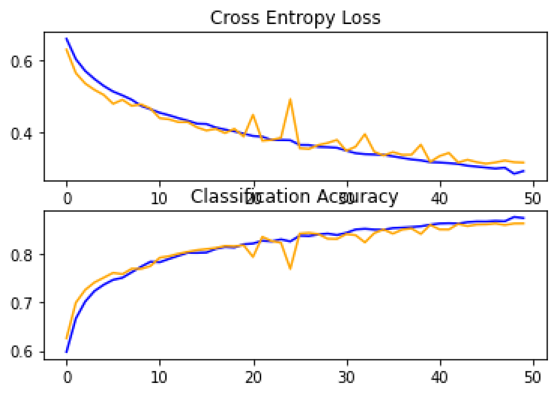
When compared with dropout, data augmentation shows a better model improvement with an accuracy of 86,34%. More interesting, the overfitting problem seems to have been solved.

2.2.1. Fitting and evaluating the model

```
def run_test_harness_2():
        # define model
        model = three_block_vgg()
        # create data generators
        train_datagen = ImageDataGenerator(rescale=1.0/255.0,
        width_shift_range=0.1, height_shift_range=0.1, horizontal_flip=True)
        test_datagen = ImageDataGenerator(rescale=1.0/255.0)
        # prepare iterators
        train_it = train_datagen.flow_from_directory('dataset_dogs_vs_cats/train/',
        class_mode='binary', batch_size=64, target_size=(200, 200))
        test_it = test_datagen.flow_from_directory('dataset_dogs_vs_cats/test/',
        class_mode='binary', batch_size=64, target_size=(200, 200))
        # fit model
        history = model.fit_generator(train_it, steps_per_epoch=len(train_it),
        validation_data=test_it, validation_steps=len(test_it), epochs=50, verbose=0)
        # evaluate model
        _, acc = model.evaluate_generator(test_it, steps=len(test_it), verbose=0)
        print('> %.3f' % (acc * 100.0))
        # learning curves
        summarize_diagnostics(history)
```

2.2.2. Results

Accuracy: 86.340%



3. Transfer learning

In order to implement transfer learning, we had to resize the images in the dataset to 224x224, since both VGG16 and ResNet50 have been trained in images with this shape.

In our case, the goal of using transfer learning is to use the pre-trained models to work has feature extractors by removing the fully connected layers (by using “include_top=False”) and add new ones that are made for our classification problem.

From the results we can see that the accuracy results were very good. However, there is an overfitting problem in both models, which could again possibly be solved with regularization techniques.

3.1. VGG16

3.1.1. Define the model

```
def vgg_16():

    # load model

    model = VGG16(include_top=False, input_shape=(224, 224, 3))

    # mark loaded layers as not trainable

    for layer in model.layers:

        layer.trainable = False

    # add new classifier layers

    flat1 = Flatten()(model.layers[-1].output)

    class1 = Dense(128, activation='relu', kernel_initializer='he_uniform')(flat1)

    output = Dense(1, activation='sigmoid')(class1)

    # define new model

    model = Model(inputs=model.inputs, outputs=output)

    # compile model

    opt = SGD(lr=0.001, momentum=0.9)

    model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])

    return model
```

3.1.2. Fitting and evaluating the model

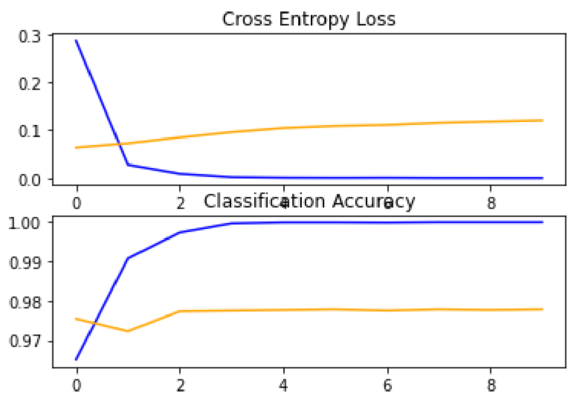
For the VGG16 and the Resnet50 a new test harness function had to be defined.

```
def run_test_harness_3(model_string):

    #print(model_string)

    # define model
```

```
 if model_string == 'vgg16':
         model = vgg_16()
 if model_string == "resnet50":
         model = resnet_50()
 # create data generator
datagen = ImageDataGenerator(featurewise_center=True)
# specify imagenet mean values for centering
datagen.mean = [123.68, 116.779, 103.939]
 # prepare iterator
train_it = datagen.flow_from_directory('dataset_dogs_vs_cats/train/',
class_mode='binary', batch_size=64, target_size=(224, 224))
test_it = datagen.flow_from_directory('dataset_dogs_vs_cats/test/',
 class_mode='binary', batch_size=64, target_size=(224, 224))
# fit model
history = model.fit(train_it, steps_per_epoch=len(train_it),
 validation_data=test_it, validation_steps=len(test_it), epochs=10, verbose=1)
# evaluate model
_, acc = model.evaluate(test_it, steps=len(test_it), verbose=0)
print('> %.3f' % (acc * 100.0))
# learning curves
 summarize_diagnostics(history)
```

3.1.3. Results

Accuracy: 97.779%



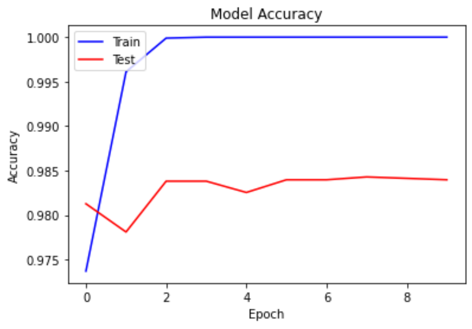
3.2. ResNet50

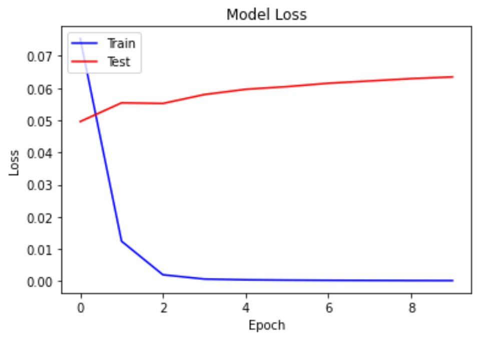
3.2.1. Define the model

```
def resnet_50():
	# load model
	model = ResNet50(include_top=False, input_shape=(224, 224, 3))
	# mark loaded layers as not trainable
	for layer in model.layers:
		layer.trainable = False
	# add new classifier layers
	flat1 = Flatten()(model.layers[-1].output)
	class1 = Dense(128, activation='relu', kernel_initializer='he_uniform')(flat1)
	output = Dense(1, activation='sigmoid')(class1)
	# define new model
	model = Model(inputs=model.inputs, outputs=output)
	# compile model
	opt = SGD(learning_rate=0.001, momentum=0.9)
	model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
	return model
```

3.2.2. Results

Accuracy: 98.398%





Conclusion

The main goal of this project was to explore Convolutional Neural Networks by solving a binary classification task between cats and dogs. Different subjects were explored with the main objective of understanding how they can be implemented in practice and how they can affect the results, such as different model architectures (VGG1, VGG2 and VGG3), regularization techniques to improve model performance (dropout and data augmentation) and transfer learning (VGG16 and ResNet50).

Regarding VGG1, VGG2 and VGG3, we can see that with the increase in the number of VGG blocks there is an increase in the accuracy of the model, but there is also an overfitting problem. In order to study how this overfitting could be addressed, we have considered the VGG3 as the baseline model and applied dropout and data augmentation.

Both techniques have improved accuracy and overfitting with data augmentation performing better in both of these aspects and actually solving the overfitting problem.

When it comes to transfer learning, two models were explored VGG16 and ResNet50. The fully connected layers were removed, and new ones were added that were suited to our classification problem. The two models have shown really good values of accuracy (almost 100%), but also overfitting. In order to solve this, regularization techniques could be again applied.

Although multiple other aspects could be further explored such as different techniques or even parameter values (for instance different dropout rates), this project enabled us to understand how CNNs can be used for classification problems, and the impacts that different techniques can have in the model performance.