Neural Networks

Binary Classification and Image Generation of Cats and Dogs

Statistical Methods For Machine Learning
Sistemi Intelligenti Avanzati

Academic Year
2021-2022

Sara Regali

sara.regali@studenti.unimi.it

Contents

1 Introduction **3**
- 1.1 Neural Networks . . . 3
- 1.2 Convolutional Neural Network . . . 3
- 1.3 Binary Classification . . . 3
- 1.4 Loss Functions . . . 3
- 1.5 Binary Crossentropy . . . 3
- 1.6 Mean Squared Error . . . 4
- 1.7 Optimizers . . . 4
- 1.8 K-Fold Cross Validation . . . 4
- 1.9 Accuracy . . . 4
- 1.10 Generative Adversarial Network . . . 4

2 Libraries **6**

3 Preprocessing **7**

4 Binary Classification Module **9**
- 4.1 Model Compilation . . . 9
- 4.2 Training and Validation of a model . . . 9
- 4.3 Fully Connected Perceptron . . . 11
- 4.4 Convolutional Neural Network . . . 11
- 4.5 Transfer Learning . . . 13
- 4.6 Results . . . 15

5 Image Generation Module **16**
- 5.1 Generator . . . 16
- 5.2 Discriminator . . . 16
- 5.3 Losses . . . 17
- 5.4 Training step . . . 17
- 5.5 Binary Crossentropy . . . 19
- 5.6 Mean Squared Error . . . 19

6 Results **21**

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. The code for the Image Generation module was adapted from this tutorial [1]. In the tutorial, a different dataset and preprocessing phase were used, and no experimental results to compare were provided.

1 Introduction

The following is a report of the project available at *this link*, developed for the Statistical Methods For Machine Learning course along with Sistemi Intelligenti Avanzati course at University of Studies of Milan, during the academic year 2021-2022. The project compares the performances of different neural networks trained for the task of binary classification of images of cats and dogs taken from *this dataset*, with the use of cross-validation. Furthermore, it proposes a way to generate new images of cats and dogs, starting from the same dataset using a Generative Adversarial Network (**GAN**).

1.1 Neural Networks

A neural network is a collection of interconnected nodes that works with processes that mimic the way the human brains operates (hence the name Neural), used to model complex relationships between inputs and outputs or to find patterns in data. In the work explained in this report, Neural Networks will be exploited to classify data and generate new ones.

1.2 Convolutional Neural Network

A Convolutional Neural Network is a specialized type of neural network model designed for working with two-dimensional image data, that use convolution layers that apply **convolution**, a linear operation that involves the multiplication of a set of weights with the input. The technique was designed for two-dimensional input, so the multiplication is performed between an array of input data and a two-dimensional array of weights, called a filter or a kernel. The filter is applied systematically to each overlapping part or filter-sized patch of the input data to detect features and later discover them anywhere in the image.

1.3 Binary Classification

In Machine Learning, binary classification is denoted as the task of classifying the elements of a set into one of two classes. This is typically done by providing a set of labelled data (already classified) to the predicting model, which will be used as a **training set** by the model itself to learn how to correctly classify examples that were never seen before. After the training phase is over, the model will be tested and its performances evaluated using a new set of unseen datas, called the **validation set**. In this report, binary classification will be used to discern pictures of cats from pictures of dogs.

1.4 Loss Functions

The loss of an algorithm is a measure of the goodness of its predictions. In a more precise way, it's the disparity between the predicted lable and the correct one. The simplest loss function for classification is the **zero-one loss**, which is defined in the following way:

$$l(y, \hat{y}) = \begin{cases} 0 & \text{if} \quad y = \hat{y} \\ 1 & \quad\quad otherwise \end{cases}$$

1.5 Binary Crossentropy

Binary Crossentropy or Logaritmic loss is a loss function which is defined in the following way.

$$l(y, \hat{y}) = \begin{cases} \ln \frac{1}{\hat{y}} & \text{if} \quad y = 1 \\ \ln \frac{1}{1-\hat{y}} & \text{if} \quad y = 0 \end{cases}$$

It will be used as a training function for the Binary Classification module and during the training of the GAN in the Image generation module.

1.6 Mean Squared Error

Mean Squared Error is a loss function defined as following.

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y, \hat{y})^2$$

Where N is the number of samples we are testing against. It will be used in comparison with the Binary Crossentropy loss function during the training of the GAN in the Image generation module.

1.7 Optimizers

An optimizer is an algorithm used to minimize the loss function of a predictor by changing the weights and learning rate of a Neural Network. In this project, for all models the **Adam** optimizer will be used, which is a gradient descent algorithm that tweaks its parameters iteratively to minimize a given function to its local minimum. The **learning rate** will be used to determine how small the steps to get to the local minimum should be.

1.8 K-Fold Cross Validation

K-Fold Cross Validation is a technique used to estimate the loss of a given learning algorithm. With Cross Validation, the entire dataset is split into K subsets (or **folds**). The model is then trained with (K-1) of the subsets and tested with the remaining *i-th* one. This is done exactly K times and the final Cross Validation estimate is computed by averaging the errors on the testing part of each fold. In this project, 5-fold Cross Validation will be performed to estimate the zero-one loss of the classifier for the binary classification task.

1.9 Accuracy

Accuracy is defined as the number of correct predictions made by the model, divided by the total number of predictions. In other terms, it's the percentage of samples correctly predicted by the model. This metric will be used to evaluate the performances of the different models used in this project.

1.10 Generative Adversarial Network

In Machine Learning, a GAN is a technique that learns to generate new data starting from a training set. GANs are based on two neural networks that work competing against each other. The first neural network is defined as **Generator**, and learns the patterns in the training data producing new images out of input noise vectors; the second one is called **Discriminator** and it classifies the images as "real", meaning they come from the training set, or "fake", meaning that they were produced by the generator. The Generator's job hence, is to fool the Discriminator, whose only job is

reduced to the one of binary classification. In this work, a **DCGAN** (Deep Convolutional Generative Adversarial Network), a GAN that explicitly uses convolutional and convolutional-transpose layers in the discriminator and generator respectively, will be used to generate new images of cats and dogs.

2 Libraries

Various libraries will be used during the development of this project.

- `Tensorflow 2.9.1` a machine learning ad artificial intelligence library with a focus on training and inference of deep neural networks.[2]
- `Keras 2.9.0` a deep learning API running on top of Tensorflow with a focus on enabling fast experimentation.[3]
- `Scikit-learn 1.1.2` a machine learning library that features various classification, regression and clustering algorithms along with many preprocessing and parameter tuning tools.[4]
- `Matplotlib 3.5.3` a comprehensive library for creating static, animated, and interactive visualizations in Python.[5]
- `Numpy 1.23.1` a library for the support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.[6]

3 Preprocessing

After downloading the images in local storage, a preprocessing phase is necessary to filter out corrupted images from the dataset. This was done with the following function.

```
def remove_corrupted_images(path: str = None):
    img_paths = glob.glob(os.path.join(path, '*/*.*'))
    bad_images = []

    print("Corrupted images:")
    for image_path in img_paths:
        try:
            img_bytes = tf.io.read_file(image_path)
            tf.image.decode_image(img_bytes)
            Image.open(image_path)
        except (
                tf.errors.InvalidArgumentError,
                UnidentifiedImageError
        ) as e:
            print(f"Found corrupted image at: {image_path} - {e}")
            bad_images.append(image_path)

    if len(bad_images) == 0:
        print("No corrupted images found.")
    else:
        print("Removing them...")
        create_dir(TRASH_PATH)
        create_dir(TRASH_PATH + CATS)
        create_dir(TRASH_PATH + DOGS)
        for path in bad_images:
            print(f"Moving image: {path} to Trash")
            shutil.move(f"{path}", f"{TRASH_PATH}{path[len(IMGS_PATH):]}")
```

Images that throw an exception while trying to be opened are moved to the `Trash/` directory inside their respective label's directory.
The next step is to scale down and transform to grayscale the remaining images, in order to have feature vectors with a limited range of values (0 to 255 in this case). This is done by exploiting the `ImageDataGenerator` class from the `keras.preprocessing.image` module that offers a clever feature to generate batches of data from images inside a directory by specifying its path.

```
    gen = ImageDataGenerator(
        rescale = 1/rescale,
        validation_split = validation
    )

    train_dataset = gen.flow_from_directory(
        IMGS_PATH,
        target_size=size,
        class_mode='binary',
        batch_size=batch_size,
```

```
    subset="training",
    seed=123,
    color_mode="grayscale"
)
```

- `rescale` is the rescaling factor, that ensures the image's pixels will be scaled down to values from 0 to 255.
- `validation_split` is the fraction of images reserved for validation.
- `IMGS_PATH` is the local directory where the dataset is saved. It is important that the images are sorted in other directories by class (eg. Dog or Cat) beforehand, so that the generator can assign the labels to the image based on the directory they were found.
- `target_size` is the dimension to which all images will be resized to.
- `class_mode` determines the type of label arrays returned. In this case is binary, which means there will be 1D binary lables.
- `batch_size` sets the size of the batches of data. This is an hyperparameter.
- `seed` is an optional random seed for shuffling (which is by default set to TRUE.)
- `color_mode` if set to 'grayscale', converts images to have 1 channel.

With `rescale = 255.`, `validation = 0.0` and `size = (50, 50)` the training dataset is now ready to be used in both the Binary Classification and the Image Generation module.

4 Binary Classification Module

This module focuses on training three different Neural Networks for the task of binary classification of cats and dogs images, and later uses 5-fold Cross Validation as a tool to compute the accuracy of the constructed models. Each model is trained for 30 and 60 epochs, with a batch size of 32 and 64, with a total of four experiments per model.

4.1 Model Compilation

For all models the same optimizer and compilation parameters are used:

```
adam_optimizer = tf.optimizers.Adam(learning_rate=0.001)

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

4.2 Training and Validation of a model

5-fold Cross validation is implemented exploiting the `KFold.split()` method from the `sklearn.model_selection` library. An instance of the `KFold` class is created with the following parameters:

```
k_fold = KFold(n_splits=N_OF_FOLDS, shuffle=True)
```

with `N_OF_FOLDS` set to 5 as a project specifics, and shuffle set to `True` for better performances. On the `k_fold` object is then called the `split(train_dataset)` method, which returns two arrays containing the training and test set (batch) indices for that split. The training and validation subsets are then created iterating over the batches of the training set as explained in the following code:

```
for train, test in k_fold.split(train_dataset):
    for image_batch in train_dataset:
        if train_dataset.batch_index in train:
            X_train = np.insert(X_train, 1,
                                np.array(image_batch[0]), axis=0)
            y_train = np.insert(y_train, 1,
                                np.array(image_batch[1]), axis=0)
        else:
            X_test = np.insert(X_test, 1,
                               np.array(image_batch[0]), axis=0)
            y_test = np.insert(y_test, 1,
                               np.array(image_batch[1]), axis=0)
```

If the index of the current batch is contained in the `train` array, the images and the corrisponding labels are inserted in the `X_train` and `y_train` numpy array for training. Otherwise, they are inserted in the `X_test` and `y_test` arrays for validation.
The model is then trained and consequently validated with the `.fit()` method from the `keras.Model` API, which returns an object with the recorded training and validation losses, along with the training

and validation metrics specified, which will be accuracy in this project. The `.fit()` method is called on the model with the following parameters:

```
history = model.fit(
    X_train,
    y_train,
    validation_data=(X_test, y_test),
    batch_size=BATCH_SIZE,
    verbose=1,
    epochs=N_OF_EPOCHS,
    callbacks=[reduce_lr_acc, early_stopping, mcp_save],
)
```

Where:

- `X_train` supplies the input data
- `y_train` supplies the target data or labels for the input data.
- `validation_data` are the data used to evaluate the loss and accuracy of the model at the end of each epoch. This is set as the test subset previously assembled.
- `batch_size` sets the size of the batches of data. This matches the batch size used for the preprocessing phase.
- `verbose` sets the verbosity mode as 'progress bar'.
- `epochs` sets the number of epochs to train the model over.
- `callbacks` sets a list of callbacks to apply during training.

In particular, the following callbacks are used during the model training:

- `ReduceLROnPlateau` used to reduce learning rate of a 0.1 factor if accuracy stops improving after 7 epochs.
- `EarlyStopping` stops the training if accuracy hasn't improved in the next 15 epochs.
- `ModelCheckpoint` used to save the model every time accuracy improves.

4.3 Fully Connected Perceptron

The previously described structure is exploited for the training and validation of a Fully connected Perceptron, whose architecture is described below:

```
model = Sequential()

model.add(Input(shape=(IMG_HEIGHT, IMG_WIDTH, CHANNELS), name="input_layer"))
model.add(Flatten(name='flatten_hidden_Layer'))
model.add(Dense(1024, activation='relu', name='hidden_Layer'))
model.add(Dense(1, activation='sigmoid', name='output_Layer'))
```

This model was chosen for academic purposes, to show how a single hidden layer is still able to achieve good results. Results for the various experiments are shown below.

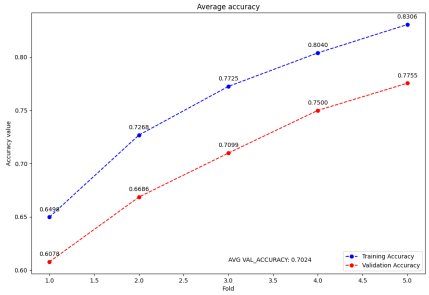

Figure 1: Plot for 30 Epochs and Batch Size 32

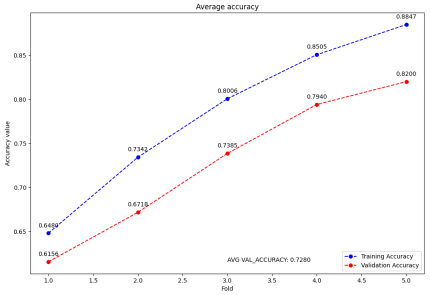

Figure 2: Plot for 30 Epochs and Batch Size 64

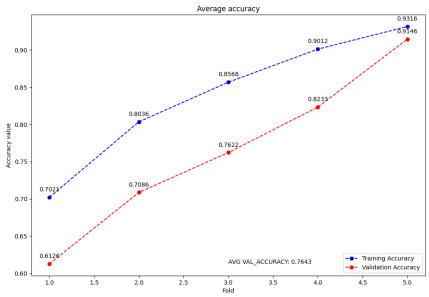

Figure 3: Plot for 60 Epochs and Batch Size 32

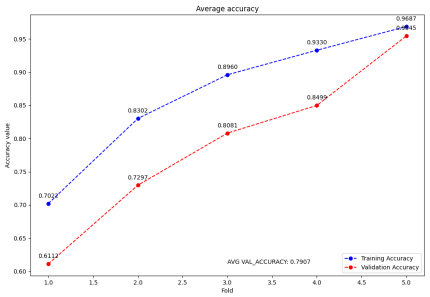

Figure 4: Plot for 60 Epochs and Batch Size 64

The plots show the typical behaviour for a model's accuracy: it grows overtime and the training curve is higher than the validation curve.

4.4 Convolutional Neural Network

The Convolutional Neural Network created for this project was the following:

```
model = Sequential()

model.add(Conv2D(32, (3, 3), activation='relu', input_shape=
                (IMG_WIDTH, IMG_HEIGHT, CHANNELS), name='conv0'))
```

```
model.add(MaxPooling2D(pool_size=(2, 2), name='max_pool0'))

model.add(Conv2D(64, (3, 3), name='conv1'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2), name='max_pool1'))

model.add(Conv2D(128, (3, 3), name='conv2'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2), name='max_pool2'))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))

model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

The results obtained using this model are shown below.

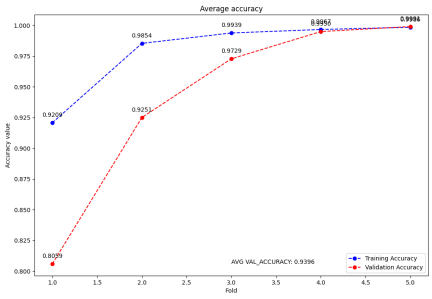

Figure 5: Plot for 30 Epochs and Batch Size 32

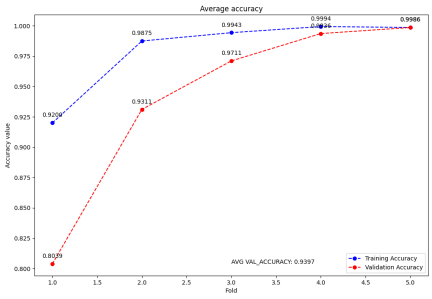

Figure 6: Plot for 30 Epochs and Batch Size 64

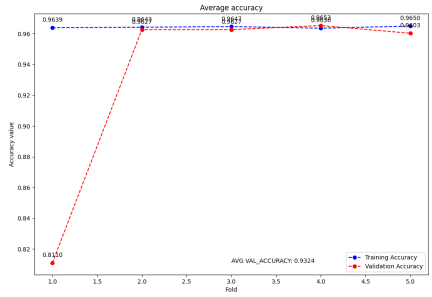

Figure 7: Plot for 60 Epochs and Batch Size 32

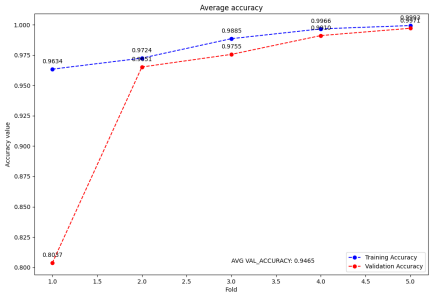

Figure 8: Plot for 60 Epochs and Batch Size 64

These plots also exhibit the typical behaviour for accuracy, with a lesser distance between the training and validation curve. The third plot (60 epochs and batch size 32) shows how the model starts to overfit by the last fold, since the validation accuracy curve start to lean downwards. In the other plots it is not noted any significant behaviour.

4.5 Transfer Learning

The intuition behind Transfer Learning is that if a model is trained on a large and general enough dataset, this model will effectively serve as a generic model for a different but related problem. The last model in this project consists of a pretrained Convolutional Neural Network adapted for the binary classification task, the ResNet50.
The model architecture is shown below.

```
input_layer = Input(shape=(IMG_WIDTH, IMG_HEIGHT, 3))
resnet_model = ResNet50(weights='imagenet', input_tensor=input_layer,
                        include_top=False)
last_layer = resnet_model.output    #Takes out the last layer
flatten = Flatten()(last_layer)
# Last layer for binary classification
output_layer = Dense(1, activation='sigmoid')(flatten)
model = Model(inputs=input_layer, outputs=output_layer)
# We are making all the layers intrainable except the last layer
for layer in model.layers[:-1]:
    layer.trainable = False
```

Since the ResNet model was originally trained to be used with an input shape of (224, 224, 3), which is incompatible with the chosen `IMG_WIDTH` and `IMG_HEIGHT`, it is necessary to set the argument `include_top=False` to be able to specify a custom input shape. Weights are pretrained using the ImageNet database, and the last layer is first flattened and then adapted to work for binary classification. Ultimately, all layers but the last one are set as **not trainable** so that their learning rate is zero and their weights can't be updated. This may help with speeding up network training and to prevent those layers from overfitting to the new data set. Furthermore, since ResNet was originally ment to work with images as RGB values but in this project they are in grayscale, the image values in the `X_train` and `X_test` must be repeated three times to achieve the three channels necessaries before calling the `.fit()` method on the model.
Results using the custom ResNet model are shown below.

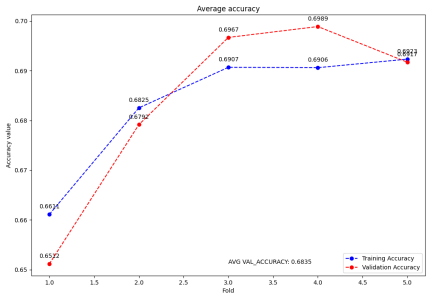

Figure 9: Plot for 30 Epochs and Batch Size 32

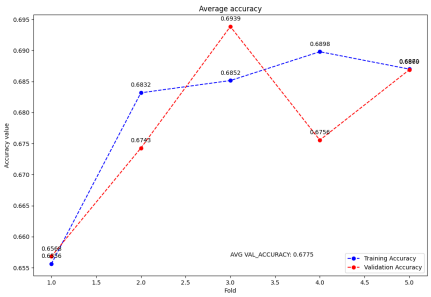

Figure 10: Plot for 30 Epochs and Batch Size 64

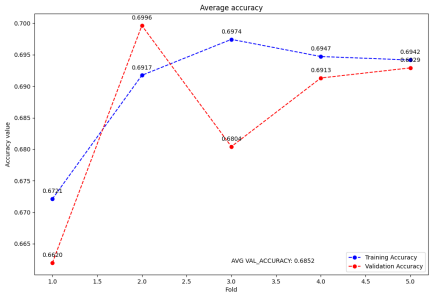

Figure 11: Plot for 60 Epochs and Batch Size 32

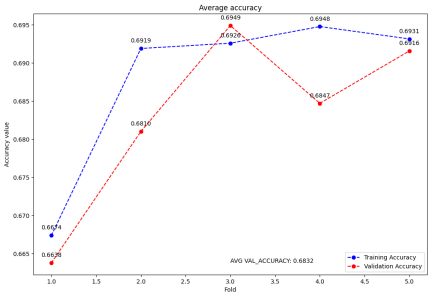

Figure 12: Plot for 60 Epochs and Batch Size 64

The plots show how the validation dataset seems to be unrepresentative of the model used. This is higlighted by the fact that the training curve looks considerably ok, but the validation function behaves noisily around the training curve, especially in the last three plots. It was proved examining the plots of the other models used, how they exhibited the typical behaviour for training and validation accuracy, so data seems not to be the problem in this case but rather the actual model used.

4.6 Results

The following table shows the mean validation accuracy values recorded across all folds for each model used, for every combination of number of epochs and batch size chosen.

Epochs and Batches	Perceptron	CNN	ResNet50
30E, 32B	0.7024	0.9396	0.6835
30E, 64B	0.7280	0.9397	0.6775
60E, 32B	0.7643	0.9324	0.6852
60E, 64B	0.7907	0.9465	0.6832

The best performances for each model are higlighted in red. As expected, the CNN model is the one that generally performed better across all experiments with a maximum accuracy of 95%. The Perceptron model achieves 79% accuracy at its best, while the ResNet model only stops at 69%. The poor performance of the latter can be explained by the fact that it was originally designed for images with an input shape of 224x224, while the chosen input shape was set at 50x50 for performances reasons. The lack of additional features to extract from the small images might have been what led the model to be unfitted for the characteristics chosen for the problem.
In all three models, the best performances were achieved with the highest number of epochs, which was set to 60. This is straightforward since more epochs means that the model has more time to improve, even with the `EarlyStopping` callback set.
With the number of epochs set, in both the Perceptron and CNN model the best performances are achieved with a batch size of 64. This follows what was proven in litterature, which is that a smaller batch size has faster convergence to a good solution, but in generally doesn't guarantee that the model itself will converge to the global optima. Raising the batch size can overcome this problem but it can drop the model's ability to generalize, which is to adapt and perform whell when given unseen data. In this case, raising the batch size led to better performances. With the ResNet model though, when using a bigger batch size performances slightly dropped but not in a significant way.

5 Image Generation Module

This module focuses on developing a DCGAN to generate images of cats and dogs and later discusses the impact that the use of different loss functions had on the training itself. The loss functions chosen for this project were Binary Crossentropy and Mean Squared Error.

5.1 Generator

As previously stated in the subsection 1.10, the Generator is a Neural Network which produces an image starting from a random noise vector. The Generator architecture used in this project is shown below.

```
model = Sequential()

model.add(Dense(IMG_WIDTH // SCALE_FACTOR * IMG_WIDTH // SCALE_FACTOR * 256,
                use_bias=False, input_shape=(NOISE_SHAPE,)))
model.add(BatchNormalization())
model.add(ReLU())
model.add(Reshape((IMG_HEIGHT // SCALE_FACTOR, IMG_WIDTH // SCALE_FACTOR, 256)))
model.add(Dropout(0.4))

model.add(Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same',
                          use_bias=False))
model.add(BatchNormalization())
model.add(ReLU())
model.add(Dropout(0.4))

model.add(Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
                          use_bias=False))
model.add(BatchNormalization())
model.add(ReLU())

model.add(Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',
                          use_bias=False, activation='tanh'))
```

The Generator uses `Conv2DTranspose` layers that, starting from a `Dense` layer that takes a random noise vector as input, upsample it several times to reach the final image size of 32x32x1. Furthermore, it uses `BatchNormalization` layers to normalize the input features to have mean close to 0 and variance close to 1.

5.2 Discriminator

The Discriminator is the Neural Network responsible to classify the generated images as real or fake. The architecture chosen for the discriminator will be trained to output positive values for real images and negative values for fake images.
The discriminator's architecture is shown below.

```
model = tf.keras.Sequential()
model.add(Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                 input_shape=[IMG_WIDTH, IMG_HEIGHT, 1]))
```

```
model.add(LeakyReLU())
model.add(Dropout(0.3))

model.add(Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
model.add(LeakyReLU())
model.add(Dropout(0.3))

model.add(Flatten())
model.add(Dense(1))
```

The discriminator uses `LeakyReLU` instead of `ReLU` as activation function for it leads to better performances because it allows the pass of a small gradient signal for negative values instead of passing a gradient of zero in the back propagation passage.

5.3 Losses

In the project two functions are used to calculate the generator's and the discriminator's losses respectively.

```
def discriminator_loss(real_output, fake_output):
    real_loss = loss(tf.ones_like(real_output), real_output)
    fake_loss = loss(tf.zeros_like(fake_output), fake_output)
    return real_loss, fake_loss


def generator_loss(fake_output):
    return loss(tf.ones_like(fake_output), fake_output)
```

The discriminator's loss is splitted in the loss calculated for the real images and the loss calculated for the generated images. This gives a measure of how well the discriminator is able to distinguish real images from fake ones. The generator's loss is calculated only on the fake images, and it gives a measure of how well the generator was able to trick the discriminator.

5.4 Training step

The following function performs the training step for the generator and discriminator, and returns the loss of the generator and the losses of the discriminator over real and fake images.

```
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        r_disc_loss, f_disc_loss = discriminator_loss(real_output, fake_output)
```

```
        disc_loss = r_disc_loss + f_disc_loss

    gradients_of_generator = gen_tape.gradient(gen_loss,
                                       generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
                                           discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator,
                                generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
                                    discriminator.trainable_variables))
    return gen_loss.numpy(), r_disc_loss.numpy(), f_disc_loss.numpy()
```

First of all, a noise vector is created for the generator to produce the images, then the discriminator is used to classify real and generated images. The respective losses are then computed, then gradients are used to update the generator and discriminator. The `@tf.function` annotation causes the function to be compiled and leads to better performances when in graph execution.
The `train_step` function is executed for each batch of the dataset, for every epoch specified. The full training cycle is explained below.

```
epoch_losses = []
for epoch in range(i, epochs):
    losses = [0.0, 0.0, 0.0]
    for image_batch in dataset:
        # Train the model for each batch in the train set
        gen_l, r_disc_l, f_disc_l = train_step(image_batch[0])
        losses = [add_value_to_avg(losses[0], gen_l, dataset.batch_index),
                  add_value_to_avg(losses[1], r_disc_l, dataset.batch_index),
                  add_value_to_avg(losses[2], f_disc_l, dataset.batch_index)]

    epoch_losses.append(losses)

    if epoch == 0 or (epoch + 1) % SAVE_IMAGES_INTERVAL == 0:
        print("Saving generated images")
        display.clear_output(wait=True)
        generate_and_save_images(generator,
                                 loss_name,
                                 epoch + 1,
                                 seed)
    # Produce plot and save the model
    if (epoch + 1) % SAVE_PLOT_INTERVAL == 0:
        print("Saving model and plotting discriminator losses.")
        checkpoint.save(file_prefix=checkpoint_prefix)
        plot_losses(epoch_losses, img_gen_dir, loss_name, epoch + 1, BATCH_SIZE)
```

The `epoch_losses` array keeps track of the average losses per epoch, while the `losses` array keeps track of the average loss for each batch. Images are saved after the first epoch and every 5 epochs,

while the model and the plots are saved every 10 epochs.
For both losses, the GAN will be trained for 100 epochs with a batch size of 64. The latter was chosen after it resulted to be the best-performing size in the experiments run for the Binary Classification module.

5.5 Binary Crossentropy

Binary Crossentropy is the first loss function to be used for training the GAN. Some of the generated images are shown below.

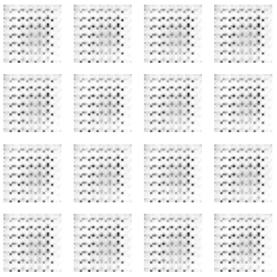

Figure 13: Epoch 1

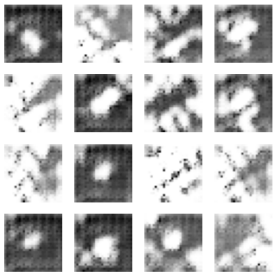

Figure 14: Epoch 15

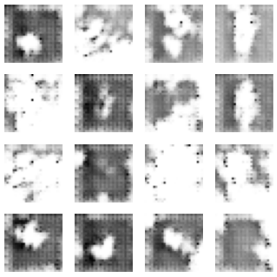

Figure 15: Epoch 25

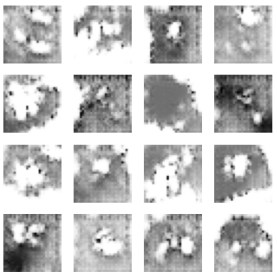

Figure 16: Epoch 50

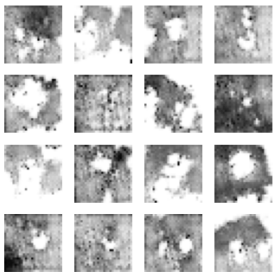

Figure 17: Epoch 75

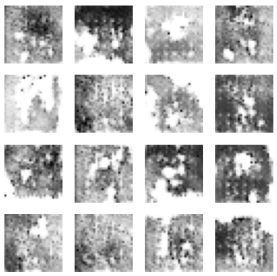

Figure 18: Epoch 100

By epoch 25 some of the patterns start to emerge and silhouettes of cats and dogs are visible in the center of the pictures. This develops in the following images, but by epoch 100 most images appear to be similar to each other. This could be a sign of **mode collapse**, in which the generator keeps producing a similar set of output.

5.6 Mean Squared Error

A limitation of the Binary Crossentropy loss function is that it is primarily concerned with whether the predictions are correct or not, and less so with how correct or incorrect they might be. To overcome this issue, it may be useful to change the Binary Crossentropy loss with the Mean Squared error one, for the latter penalizes generated images based on their distance from the decision boundary. This provides a strong gradient signal for generated images that are very different or far from the existing data and addresses the problem of saturated loss, resulting in a more stable training process and the generation of higher quality and images.

Results obtained using the mean squared error loss function are shown below.

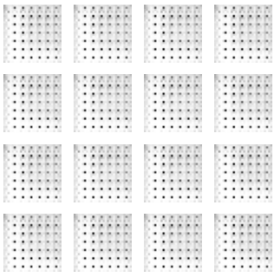

Figure 19: Epoch 1

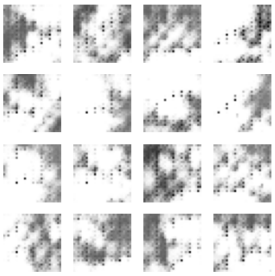

Figure 20: Epoch 15

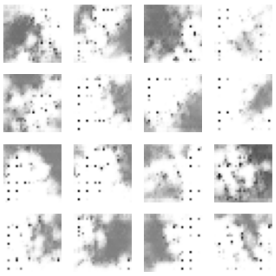

Figure 21: Epoch 25

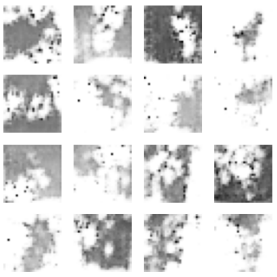

Figure 22: Epoch 50

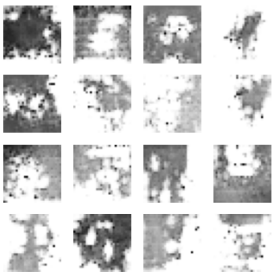

Figure 23: Epoch 75

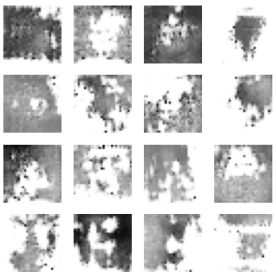

Figure 24: Epoch 100

With this loss function we see that the silhouettes of cats and dogs start to appear by epoch 50, a bit later than the previous model, and that the general images differ from one another, though it's still difficult to clearly recognize either cats or dogs.

6 Results

Resulting plots for the generator and discriminator's losses are shown below.

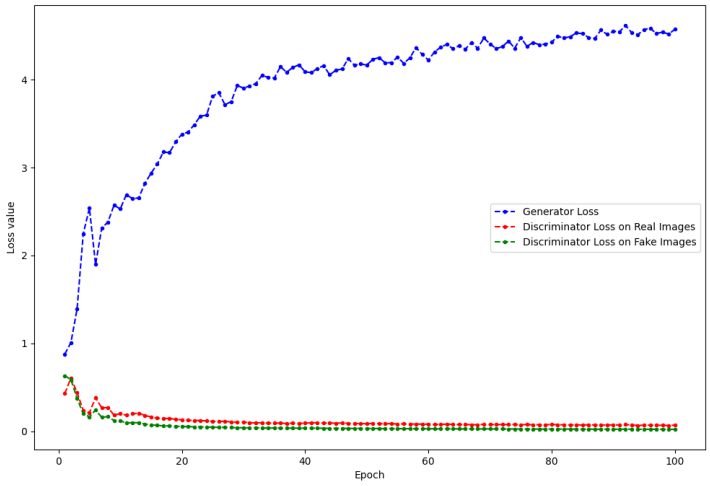

Figure 25: Plot for Binary Crossentropy loss for 100 Epochs and Batch Size 64

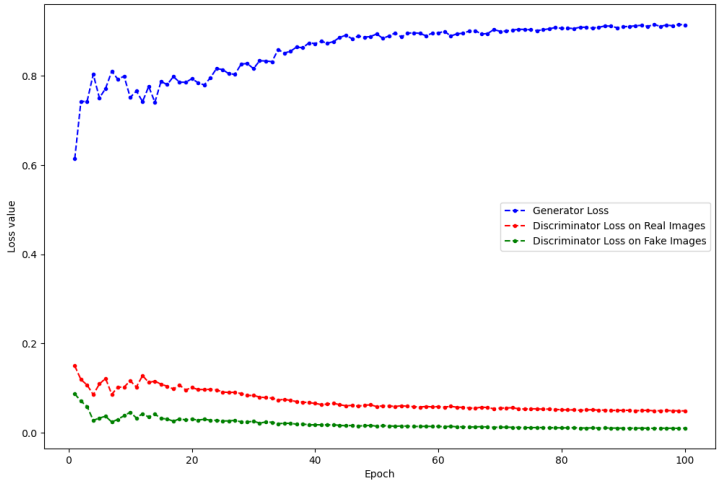

Figure 26: Plot for Mean Squared Error loss for 100 Epochs and Batch Size 64

It should be noted that unlike Mean Squared Error, Binary Crossentropy may take arbitrarly high values, but this doesn't effect the metrics used for validating the plots. In the plot for Mean Squared Error, it is shown how all three of the losses stabilize after epoch 70 circa, which is the expected behaviour for a GAN, while in the plot for Binary Crossentropy, the generator fails to stabilize as it loss keeps increasing, although with decreasing steepness. This, coupled with the fact that images

generated by epoch 100 appear to be very similar among them, could be a sign of a failure but not necessarily. In fact, increasing the number of epochs could give the generator time to stabilize its loss and produce better images. Of course, it is important to keep checking the intermediate images in order to detect possible failures of the model.
In both plots we see that both the discriminator's losses are very low, which could mean that the generator has found a set of fake images really easy for the discriminator to identify. As previously stated, increasing the number of epochs and checking for mode collapse or convergence failure (when a visually bad set of data keeps getting generated) could reveal wether the model is still in its learning phase, or if it exhibits some kind of failure.
Regardless, the Mean Squared Error loss function seems to lead to slightly better results both in terms of loss values and quality of the generated images.

References

[1] T. T. Authors, Deep convolutional generative adversarial network (2019).
URL https://www.tensorflow.org/tutorials/generative/dcgan?hl=en

[2] G. B. Team, Tensorflow (2015).
URL https://www.tensorflow.org/

[3] F. Chollet, Keras (2015).
URL https://keras.io/

[4] D. Cournapeau, Scikit-learn (2007).
URL https://scikit-learn.org/stable/

[5] J. D. Hunter, Matplotlib (2012).
URL https://matplotlib.org

[6] T. Oliphant, Matplotlib (1995).
URL https://numpy.org/