SeeFood

Ívan, Bjarki, Matteo

24th November 2022

1. Introduction

Out of three possible projects to pick from, our group settled for Project 3. This project has its focus on image enhancement. In the day and age of the Internet, images are key for the success and development of many industries and fields. They represent information that stimulates vision more than text and are therefore more powerful at fulfilling some important requirements and do so quickly. Being able to improve the quality of images places individuals at the top of the ever growing stack of competitors. Advertising, for example, relies on images heavily and companies use them to represent their product(s) and themselves; quality is essential to grab others' attention and belittle other competing brands.
Here, we explore a few approaches to implement AI-driven image enhancement through the use of neural networks. We will present approaches that yielded acceptable results, attempts that lead to failure or poor results and mention ideas that were not implemented.

2. Data

Our dataset of choice is the Pokémon image dataset. It consists of 819, 256x256 coloured images of different pokemons. No particular thought process led to us preferring this set over others as we simply went for the easy route and decided to stick to the assignment instruction's suggestion.

3. Methods

Fully connected network

We started this autoencoder process by creating a basic fully connected neural network with an input flatten layer, 4 dense layers and a reshape output layer just to get a baseline for how the autoencoders work.

```
model.add(Input((img_size,img_size)))
model.add(Flatten())
model.add(Dense(encoding_dim, activation="relu"))
# Middle layer
model.add(Dense(encoding_dim/2, activation="relu"))
model.add(Dense(encoding_dim, activation="relu"))
model.add(Dense(img_size**2, activation="relu")) # Get ima
model.add(Reshape((img_size, img_size)))
```

The model was trained for 400 epochs with no regularization as this model performed so poorly on the training data that it had no room for any more underfitting.

64 to 64 grayscale convolutional autoencoder

After getting a baseline with the fully connected we now decided to use an autoencoder with convolutional layers and max pooling layers.

```
# Encoding part

model_2.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model_2.add(MaxPooling2D((2, 2), padding='same'))

# Decoding part

model_2.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model_2.add(UpSampling2D((2, 2)))
model_2.add(Conv2D(8, (3, 3), activation='relu', padding='same'))
model_2.add(Conv2D(1, (3, 3), activation='relu', padding='same'))




model_2.compile(optimizer=Adam(lr=5e-3), loss='mse')
```

We tested out min pooling, max pooling and average pooling with various results but the max pooling ended up giving the best results. The choice for max pooling makes the most sense for this application as we want the edges to be clearer and noise to be reduced.
The best results were obtained with only one pooling layer as adding more removed too much data from the image.

64 to 64 color convolutional autoencoder

Very similar methods were used for the greyscale autoencoder and the same neural was used with some slight modifications.

```
# Encoding part

model_5.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model_5.add(MaxPooling2D((2, 2), padding='same'))

# Decoding part

model_5.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model_5.add(UpSampling2D((2, 2)))
model_5.add(Conv2D(8, (3, 3), activation='relu', padding='same'))
model_5.add(Conv2D(3, (3, 3), activation='relu', padding='same'))




model_5.compile(optimizer=Adam(lr=5e-3), loss='mse')
```

With some testing, this architecture ended up giving the best results. The exact same model as used above but with 3 convolutional output layers to account for 3 colors.

64 to 256 grayscale image convolutional upscaler

After only doing N to N autoencoding to increase the resolution we decided to create image upscalers which increase the resolution of images. I.e. we create a neural network that takes in a smaller image size in our case 64px x 64px and scales it up to 256px x 256px.

This was achieved with the following architecture

```
model_3 = Sequential()
model_3.add(input_img)


# Upscale

model_3.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model_3.add(UpSampling2D((2, 2)))
model_3.add(Conv2D(16, (3, 3), activation='relu', padding="same"))
model_3.add(UpSampling2D((2, 2)))
model_3.add(Conv2D(1, (3, 3), activation='relu', padding='same'))




model_3.compile(optimizer=Adam(lr=5e-3), loss='mse')
```

There are no pooling layers in the upscalers as they ended up decreasing the final quality.

64 to 128 color image upscaler

After creating the 64px to 256px greyscale upscalers we created a new neural network that scales colored images from 64px to 128px. This was achieved with a similar network as before but with some modifications to upsampling layers and output layers.

```
model_6 = Sequential()
model_6.add(input_img)

# Upscale

model_6.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model_6.add(UpSampling2D((2, 2)))
model_6.add(Conv2D(8, (3,3), activation='relu', padding='same'))
model_6.add(Conv2D(3, (3, 3), activation='relu', padding='same'))




model_6.compile(optimizer=Adam(lr=5e-3), loss='mse')
```

.

64 to 64 image colorizer

Another experiment we decided to perform was creating a network which predicts color in the images. This was done by splitting the images into greyscale images (X) and colored images (y). Then a convolutional neural network was created with the following architecture.

```
# Convolutional layers

model_7.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model_7.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model_7.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model_7.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model_7.add(Conv2D(8, (3, 3), activation='relu', padding='same'))
model_7.add(Conv2D(3, (3, 3), activation='relu', padding='same'))


model_7.compile(optimizer=Adam(lr=5e-3), loss='mse')

model_7.summary()
```

After some research and testing we decided to not use any as they returned worse results from fewer features.
The model was also trained for only 5 epochs as training longer only meant the neural network would pick the safest color scheme which was greyscale which also meant we sacrificed quality.

Regularization

Regularization was not needed in any of the models as the convolutional layers picked up on all of the patterns found in the pokemon dataset. The validation loss and training loss always followed a similar curve.

One use case for regularization for this model would be if it was tested for other datasets, the models definitely generalized well for the pokemon dataset but might not work as well for other images.

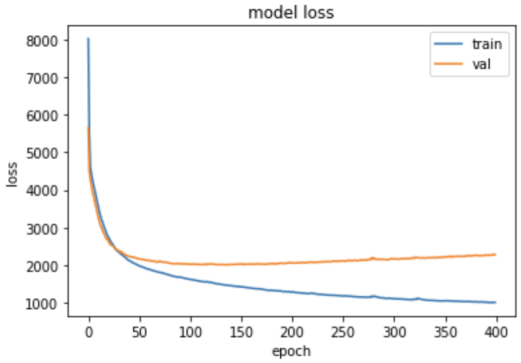
The only outlier of the models that ended up overfitting was the fully connected neural network but the performance on the training set was so bad that there was no reason to use it any further.

4. Results and discussion

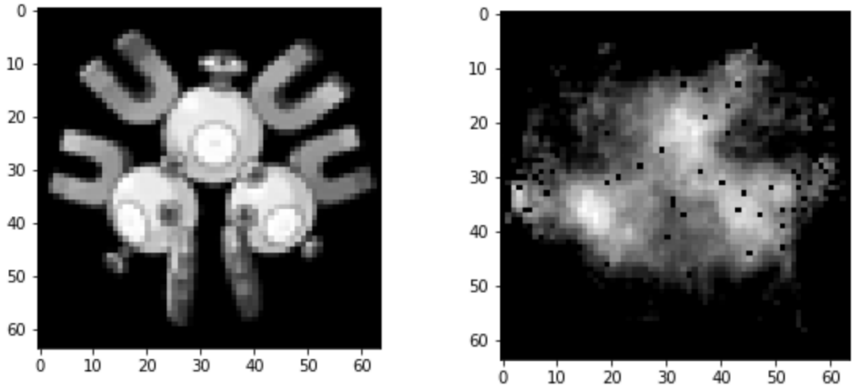
Fully connected network

The fully connected network performs very poorly for increasing the image quality of images. It merely catches the general shape of the image but nothing more and overfit to the training data. It does however give us a nice insight of how powerful CNNs really are compared to normal fully connected neural networks.

Loss during training:



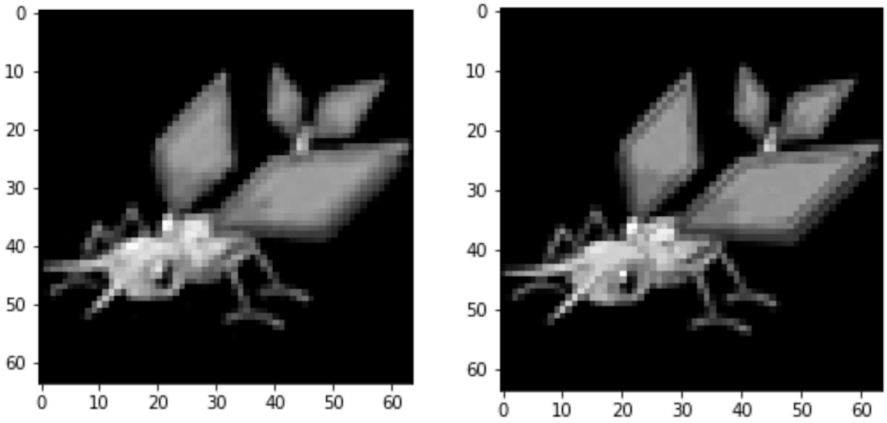
Before: After:



64 to 64 grayscale autoencoder

The 64x64 grayscale autoencoder yields incredible results. The image size is maintained – the resolution does not change. Hence, adding details cannot be obtained through sheer quantity of pixels: we must improve the actual colors stored in the pixels at hand. Parts of the image that lose details and become blurred are reconstructed faithfully. This is powerful: we retain details even though there is not as much room for these details to fill. In addition, other areas become crisper and we are able to distinguish separate parts of a pokemon that we otherwise would miss out on due to the blurred nature of the compressed image.

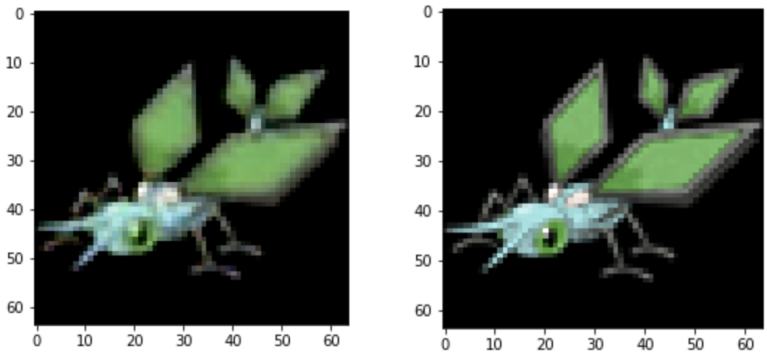
Before: After:



64 to 64 color autoencoder

Much like the 64x64 grayscale autoencoder, the results speak for themselves. Everything from the greyscale autoencoder applies with the addition that we can appreciate the contribution of color and how it adds needed detail and helps reduce the monotonicity in color caused by averaging when reducing scaling the image down.

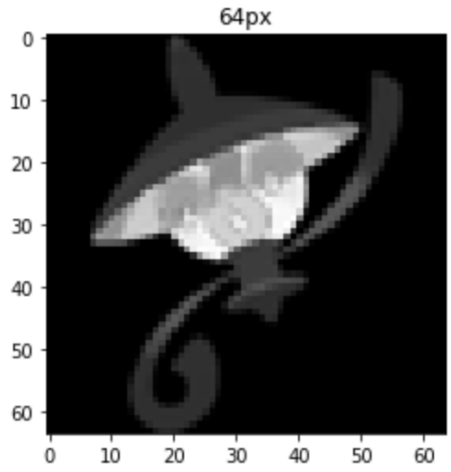
Before: After:

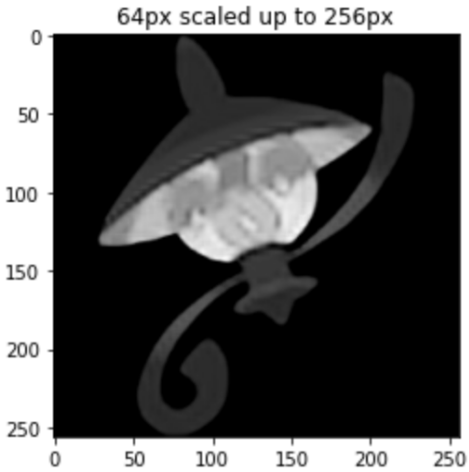


64 to 256 grayscale image upscaler

Results here are good. The outer border of pokemons is crisp. Pixelation is not easily noticeable when looking at inner areas of the pokemon – the fact that we have color probably helps with this. Patterns are reconstructed faithfully and the quality of the color retention cannot be ignored.

Before: After:

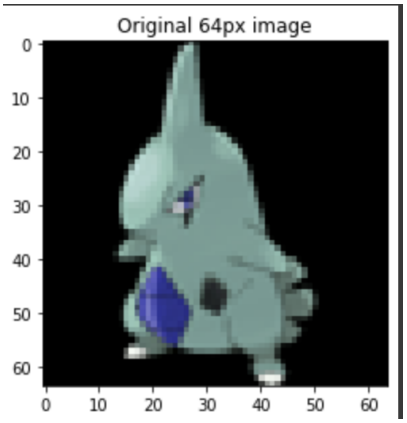


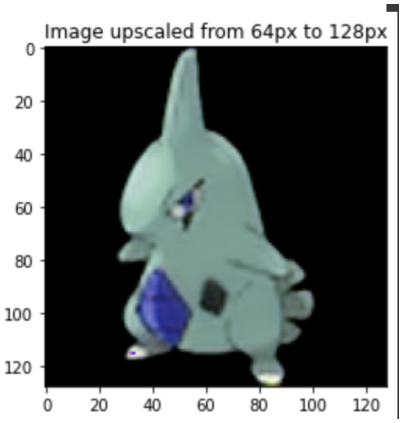


64 to 128 color image upscaler

Some traces of pixelation are perceivable within the upscaled image which resulted from the application of the upscaler. Pixelation is most noticeable within inner areas of the pokemon whereas the border with the black background does not suffer as much. In fact, the outer borders are quite decent and really sell the fact that the image is larger and as such there is more room for detail.

Before: After:

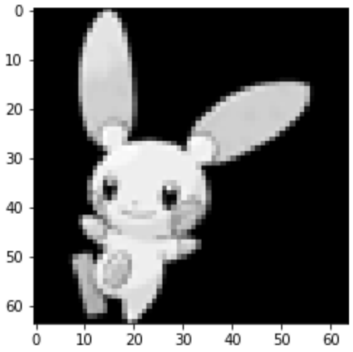




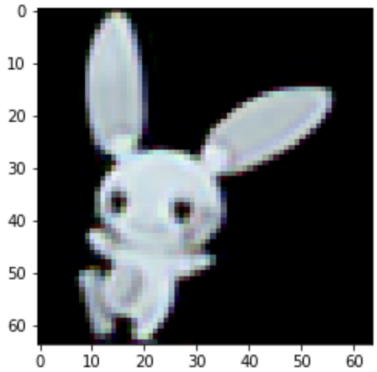
64 to 64 image colorizer

The coloriser yielded underwhelming results. We can see a hint of color and are usually able to identify a slight resemblance to the test set image but results are not satisfactory at large. The images remain mostly gray and not highly saturated.

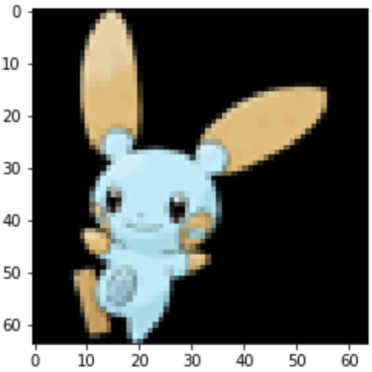
Before:



After:



Actual:



5. Conclusion

Here we have a ranking of all the neural networks in order of performance/usefulness:

1. The 64 x 128 autoencoder worked best for improving the image quality. The neural network can definitely be scaled up for further benefits.

2. 64 x 64 (both) autoencoder. This network works best for creating clearer edges in images and reducing noise. Can be scaled up for further benefits.
3. 64 x 256. Some improvement but the upscaled image didn't look like it had the quality it was supposed to have. Not as useful as the top 2 networks.
4. 64 x 64 colorizer. Didn't really do anything, slightly more color in the output image. Might have some use cases for example creating a more realistic for example in a dark room filter
5. Fully connected. Outputs a pixelated version of the original image. No real use cases.