CryptoPunks: A CNN Application

1st Michel de Buren
dept. of Finance
HEC, University of Lausanne
Lausanne, Switzerland
michel.deburen@unil.ch

2nd Eloïse Robyr
dept. of Finance
HEC, University of Lausanne
Lausanne, Switzerland
eloïse.robyr@unil.ch

3rd Paolo Trifoni
dept. of Finance
HEC, University of Lausanne
Lausanne, Switzerland
paolo.trifoni@unil.ch

***Abstract*—This project applies machine learning and image recognition techniques to a unique dataset of pixel art images: CryptoPunks. The objective of this project is to use 92 different Convolutional Neural Networks (CNN) to the data set in order to identify each of the 92 possible features that can be present in any given CryptoPunk. The data set was downloaded through the OpenSea API, and an image containing 10'000 CryptoPunks was retrieved directly from the official LarvaLabs GitHub page. Results show that the selected model, made of 3 Convolutional layers and 2 Fully-Connected layers, successfully classifies most of the targeted features, some remaining difficult to properly categorize even for the average person.**

***Index Terms*—Neural Networks, Image Processing, Feature Extraction, Convolutional Neural Networks, CryptoPunks.**

I. Introduction

The popularization of cryptocurrencies in the 21st century has allowed many people to realize the potential of crypto-encrypted objects and blockchain technology. Cryptography is now even present in Art for example. A Non-Fungible Token (NFT), is a unit of data stored on a block-chain that certifies a digital asset to be unique. An example of a NFT could be a digital painting for example and the main benefit of such an object is that once it is owned it cannot be removed or destroyed by anyone else. Many platforms now use this technology to sell unique digital concepts that cannot be replicated or deleted without the owner's permission, but can usually be traded for other NFT's or cryptocurrencies. The possibility of trading such objects allows for speculation on their prices, which is likely the reason NFT's are gaining popularity lately.

One of the actors in this newly formed business is the NBA, which developed its own platform for trading video clips of basketball games. The logic behind the utility of such a concept remains obscure to many, since these video clips can be watched infinitely many times on YouTube for free. These clips are traded for millions of dollars with future speculation in mind.

In this project, we will consider a precise kind of NFT, CryptoPunks. They are 8-bit digital characters (see Fig.1). These characters have different traits, such as clothes, accessories, race, eye color, etc. We decided to take these characteristics and use a Supervised Machine Learning algorithm to classify the 92 possible traits a CryptoPunk can have.

Fig. 1. Examples of CryptoPunks with many different attributes.

II. Description of the research question and relevant literature

In this project, we create 92 different Convolutional neural Networks (CNN) to properly classify the 92 attributes. Details on the precise structure of our CNN will be given in the methodology section.

Our project is an application of a concept that has been used multiple times for image classification which is its most widespread use. Image recognition is useful in many different fields, such as social media or self driving cars among many others.

The origin of CNNs can be traced back to the Neocognitron, introduced by Kunihiko Fukushima in 1980. His work already contained a basic type of layer still present in CNNs today: convolutional layers.

The first image recognition system that used CNNs was introduced by Yann LeCun et al. (1989). Their CNN was already able to recognise handwritten ZIP code numbers, although the kernel coefficients were hand designed (kernels will be introduced in the following section).

LeNet-5, a pioneering 7-level convolutional network by LeCun et al. in 1998, is what really popularized CNNs. It was applied by several banks to recognize hand-written numbers on checks and proven to be extremely efficient.

Since then, a lot of progress has been made in the field, such as using GPU support to reduce computation time, or changes in CNN structure. We can now train models with millions of parameters on very large datasets. CNNs are currently used in a variety of research and industries. Every year, records for computation time and accuracy are broken thanks to constant improvement of specific CNN structures for each considered dataset. Examples of these structures are the AlexNet CNN, which stacked multiple convolutional layers directly on top of

each other instead of having a pooling layer between them. This new concept earned its creators, Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton, a victory in the 2012 ImageNet ILSVRC challenge. More recently in 2015, the ResNet, which won the ILSVC 2015 challenge had a 152 layers deep structure. It is important to note that progress in this specific field is also correlated with progress in computer performance.

III. A CNN Introduction

A CNN is a Deep Learning algorithm typically used in imagery classification. It takes an input image, discovers characteristics and assigns weights related to their importance. It is then able to differentiate one image from another by learning weights and biases of the given characteristics. Convolutional networks were inspired by biological processes. The connection pattern between nodes is based on the connections present in the part of the brain that processes visual information.

So far, this is the same as a regular feed-forward network. The need for CNN arises when we consider much bigger images. For 256 x 256 pixel images for example, the first layer of a regular Neural Network will have more than half a million parameters, which not only slows the computational speed, the obtained network will have issues generalizing the train data. It will be much harder to classify new data. CNNs are used to reduce the number of these parameters to more usable numbers, thereby increasing computational speed and model performance.

Another issue with regular feed-forward networks is that they don't take into account the relationship between space and pixels : pixels close to one another are generally more correlated than those that are far apart. CNNs are able to implement this concept.

A. The kernel

A kernel is a combination of weights w_1 and w_2 and inputs x_1 and x_2 that will lead to an output, such as for example :

$$x_1 w_1 + x_2 w_2 = y \tag{1}$$

In the case of a CNN, since layers aren't fully connected, kernels' equations are much smaller. In the above example, imagine the weights are:

$$(w_1, w_2) = (-1, 1) \tag{2}$$

Therefore, (1) becomes:

$$y = x_1 - x_2 \tag{3}$$

and y is maximised when:

$$(x_1, x_2) = (1, 0) \tag{4}$$

Since x_1 and x_2 are our inputs, and pixel RGB values are generally normalized to [0,1] when working with Neural Networks, this particular unit is maximized when there is a large shift in their values. 1 being the maximum value possible and 0 being the lowest value possible. This kernel would for example be used to detect the "edge" features of the image or places where pixels have high value changes. In a CNN, nodes in the same layer can share the same kernel, so that all units perform the same operation in different places on the image. This kernel sharing is one of the main strengths of CNNs.

B. 2D Convolution

Since pixel images are being considered, 2D Convolution is performed. It is important to note that if we are considering 3 different channels, namely RGB (Red, Green, Blue), the inputs in the structure will be in three dimensions. Technically, this is still considered as 2-D Convolution because the channel order has no importance. The kernel will basically not vary according to the channel dimension, since it will always cover the 3 channels at the same time.

In the 2D Convolution Layer, the kernel will go through each input, acting as a scanner on the image. The size of a kernel is arbitrary but will usually be chosen according to the input, and can be seen as a matrix as shown in Fig.3. In this case the depth dimension would be the different channels. Width and height are the dimensions of the inputted image (see Fig 2).

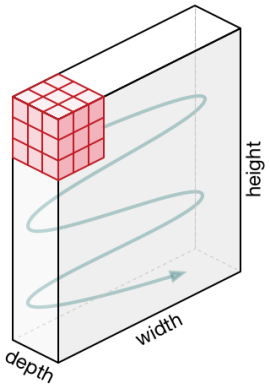

Fig. 2. Example of a kernel structure.

A convolution layer will perform the following :

Since the input is made of pixels with different RGB values, a dot product operation will be performed between the kernel and the portion P of the image over which the kernel is hovering for every possible kernel location on the image. In other words, a dot product (element wise sum) between the red square in Fig. 3 and the first P matrix (made of columns 0 to 2 and rows 0 to 2 in the case of a 3x3 kernel) will be made. The kernel will then shift to the right (Figure 3 shows a step size of 1) and do the same with the new matrix P (made of columns 1 to 3 and rows 0 to 2 for a kernel step size of 1 column). This process is repeated until the entire input image is covered and is repeated for every color channel and every kernel. Figure 3 will make this description easier to follow.

The output of a convolutional layer in the simplest case, where the input is of size (N, C_{in}, H, W) and output (N,

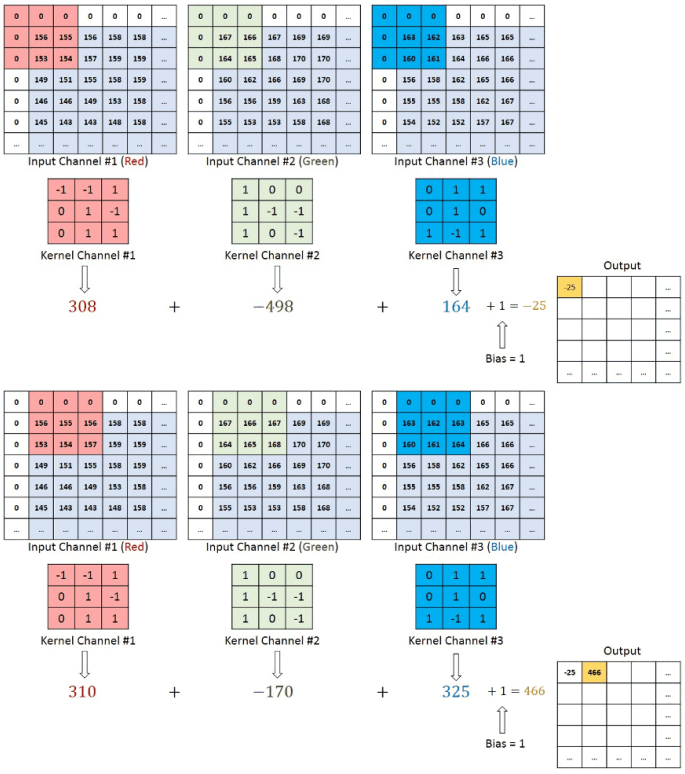

Fig. 3. Example kernel and input dot product with bias for 3 channels and step size of 1.

C_{out}, H_{out}, W_{out}) can be described by the following:

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in-1}} weight(C_{out}, k) \star input(N_i, k) \quad (5)$$

Where $\star$ is the valid 2D cross-correlation operator, N is a batch size, H is a height of input planes in pixels, W is width in pixels. and C denotes a number of channels. The Convolution Layer's output will therefore be of size H×W×C, where H and W are the same as the input's.

Therefore, back in our example, the result of each multiplication is added to the corresponding results of the other channels and to a bias to obtain the output. This entire process is repeated in every convolutional layer in the CNN structure, once per kernel.

The objective of the Convolution layer is to extract the features of the image. Usually the first layers only detect edges, colors and other general features. With added layers, the CNN adapts to all the features of an image and recognizes it surprisingly well.

C. Pooling Layer

Between the convolutional layers, one will usually find the Pooling layers. Pooling layers were created to increase computational speed and are based on dimensionality reduction. If in a given CNN one has 100 kernels for example, given a 256x256 image as input, the resulting output will be 224x224x100 after just one layer, assuming a kernel of 32x32. Since most CNNs are made of multiple layers, the computational power required to process them would be too big. That is why pooling was invented. The most used type is Max-Pooling which works by returning the maximum value in the portion of the image that is covered by the filter. The filter is the kernel corresponding to the Max-Pooling method. This would allow for example to reduce the 224x224x100 output to a 112x112x100 output in the case of a 2x2 Max-Pooling filter. In the figure below, an example of a 2x2 filter that performs Max-Pooling is given. Each color represents a different filter position on the image. As planned, the size of the input went from a 4x4 to a 2x2 matrix.

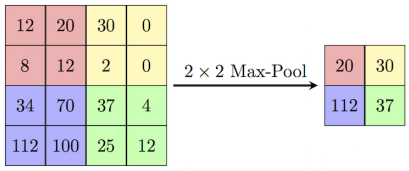

Fig. 4. Example of Max-Pooling with a 2x2 filter.

D. Flattening

Flattening is a way to convert the 3D output of the Pooling layer to a one dimensional vector. This step is essential for the final steps of the CNN. The goal of the entire process is to classify the input data properly, which is done by a Fully-Connected Layer (FC-Layer) situated at the end of the CNN structure. FC-Layers usually require inputs to be flattened before they are used.

E. Fully-Connected Layer

The Fully-Connected Layer operates on a flattened input (Nx1) where each individual input (1x1) is connected to all the nodes that follow. This is a classic multi-layer Perceptron structure, with an activation function that adds non-linearity to the model. Back propagation is applied to every iteration of training : the error in classification is calculated and is taken into account by the model for the next iteration to improve classification. This step requires a loss function, which is what the CNN wants to minimize at every iteration (more information on activation functions is provided in subsection I and Loss functions in section H). Over a series of EPOCHS, which represent a number of iterations over a whole dataset, the model is then able to distinguish features and classify them using a conversion to probability. This can be done by the SoftMax function for example. The goal of this layer is to classify the data, where as previous layers where used to extract features.

F. SoftMax Classification

Before the output is computed, it is important to bring down the data to numbers between 0 and 1, which represent the probability that a given input belongs to a specific class. This is the role of the SoftMax function:

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \quad for\ i, j = 1, 2, \ldots, K \tag{6}$$

where x_i is the input.

G. Output Layer

The final layer of the structure is the output, which depends on what type of classification we are looking for. In the Fig.3 example, which is related to the MNIST database for classifying hand-written numbers, the final layer is made of 10 nodes that indicate which number the model is predicting from 0 to 9.

H. Loss functions & Optimisation

Loss functions are essential to any Neural Network model. The idea is to reduce the classification error as much as possible by minimizing the loss function. Loss functions are made so that training the model penalizes the deviation between the predicted output of the network and the real data label the model is trying to predict. Many such functions exist, but in the scope of this project the Average Mean-Squared Error was used:

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i; W))^2 \tag{7}$$

Where y_i is the actual output the model is attempting to predict, W is the vector of weights and $f(x_i; W)$ is the prediction. The network is "trained" meaning that it will look for the the weights W which achieve the lowest possible loss:

$$W^* = \arg\min_{W} J(W) \tag{8}$$

Concretely, network training here means to perform stochastic gradient descent which is an approximation of gradient descent. The difference is that the gradient, which is normally calculated with the entire dataset, is replaced by its estimation, calculated with a randomly selected subset of the data. Every iteration of the stochastic gradient descent will therefore compute the following :

$$W := W - \eta \nabla J(W) = W - \frac{\eta}{n} \sum_{i=1}^{n} \nabla J_i(W) \tag{9}$$

where η is a step size, or learning rate and ∇ is the calculated gradient of the error function with respect to the model's weights. This is an application of backpropagation: the calculation of the gradient proceeds backwards through the network, starting with the computation of the final layer's gradient using its corresponding weights. The gradient of the first layer of weights is calculated last. Computations of the gradient from one layer are used in the computation of the gradient of the previous layer. This backwards error usage allows for increased efficiency when computing the gradient of each layer compared to the normal approach of calculating them separately.

I. Activation functions

Activation functions are now a staple in Neural Networks. Their role is to define the output of a node given an input or set of inputs. These key functions are required in order to introduce non-linearity in the model, allowing it to classify more complex inputs. Activation functions are generally present in every layer of a CNN except for the first and last ones. Recently, the most popular has been the ReLU activation function:

$$Relu(z) = max(0, x) \tag{10}$$

Where x is the input to the node. Its popularity doesn't imply that it is without flaws. The main one is the dying ReLU problem: some nodes can remain inactive for all inputs which decreases the models capacity. A "dead" node will always output the same value, regardless of the input provided. It is unlikely that this node will ever recover since the function's gradient at 0 is also 0. The function will the always return the value 0. The dying ReLU issue usually happens when the learning rate is set too high. With a proper learning rate, this problem arises less frequently.

Advantages are mainly related to efficiency and the fact that it is rescalable:

$$\max(0, ax) = a \max(0, x) \text{ for } a \geq 0 \tag{11}$$

This property will be useful when normalizing or standardizing input data for example. ReLU is reliable, making it the best default option in most models and will also be the activation function of choice in this project. Other activation functions exist, but it is not necessary to introduce them as they were not used.

Concretely, an example of a CNN structure in its entirety would be the structure shown in Fig.5. Note that the Padding concept will not be introduced as it wasn't used in this project. This particular structure is related to the MNIST dataset.

J. Dropout

Dropout is method employed to minimize the chances of overfitting model predictions to the data and provides a way of approximately combining many different neural network structures efficiently. It is called dropout because nodes and their incoming and outgoing connections can be temporarily removed from the network. These nodes are dropped with probability $(1-p)$ or kept with probability p. The consequence is that only the reduced model is trained on the data. The removed nodes are then brought back to the model structure.

For a neural network with n units, applying dropout will result in 2^n possible "thinned" neural networks. When testing the model, the 2^n networks are approximated by the initial network weighted by a factor p. The expected value of the output of any node is the same as in the training stages. This is the main advantage of this method: although it produces 2^n neural nets, at test time only one needs to be tested. By avoiding training all nodes on all training data, dropout decreases overfitting. The method also significantly improves training speed.

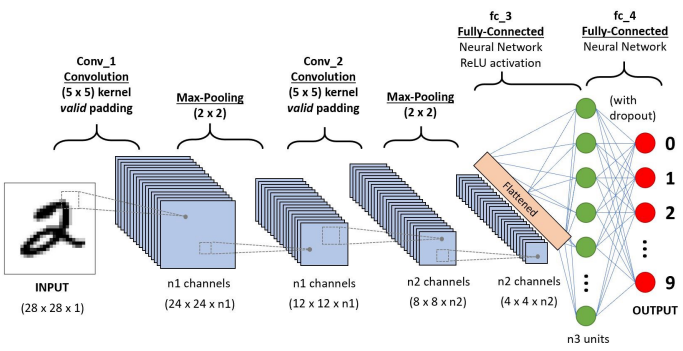

Fig. 5. Example CNN structure related to the MNIST dataset.

IV. Data Set Description

A CryptoPunk is a 24x24 8-bit pixel art image generated algorithmically. Most are punk looking guys and girls, but there are a few rarer types mixed in: Apes, Zombies and even Aliens. Every Punk has their own profile page that shows their attributes as well as their ownership/for-sale status and price.

Their creators, Larvalabs Studio, are using the Ethereum blockchain which allows for arbitrary computer code to be executed on the blockchain and the results of the execution to be stored forever. The code can be used by anyone to trade CryptoPunks, allows users to know for certain that there are only 10'000 Punks in existence and guarantees that no stealing can occur since the code cannot be modified, only used. All information regarding CryptoPunks can be found on the Larvalabs website. The Opensea API was also used to gather the different traits of each CryptoPunk. The details on the gathered data are provided in the Code Implementation section.

Each CNN has a dataset made of 8'000 CryptoPunks that have the attribute that is being recognized and an additional 8'000 that do not have it. The latter 8'000 are randomly selected out of the total 10'000. Obviously, an attribute is not necessarily present on 8'000 CryptoPunks. An oversampling is therefore implemented. This will be further developed in the Reasoning section. The total oversampled dataset for each trait is therefore made of 16'000 CryptoPunks.

V. Methodology

A. Network Structure

The networks are all convolutional. Each has:

- A Convolutional layer (number 1) where the input's number of kernels is 1, transitioning to an output with 32 kernels. The kernel used is of size 2x2 and will be the same for all Convolutional layers. The ReLU activation function was used.
- A Convolutional layer (number 2) where the input's number of kernels is 32, transitioning to an output with 64 kernels. The RELU activation function was used in this layer, as well as the Max-Pooling method to speed-up computation, with a filter of size 2x2.
- A Convolutional layer (number 3) where the input's number of kernels is 64, transitioning to an output with 128 kernels. The RELU activation function was used in this layer, as well as the Max-Pooling method to speed-up computation, with a filter of size 2x2.
- The Dropout method is implemented between the Convolutional Layers and the Fully Connected layers with a rate of 20%.
- A Fully Connected layer of kernel size 128 to 512, with the ReLU activation function.
- A Linear layer 512 to 2, reducing our input to only two possible choices.

A given CNN must be able to tell if its given attribute is present on the image or not. The output is then converted to a probability using the SoftMax function, which is an essential step for us to interpret the result and evaluate the model's performance. The output of the SoftMax function will be the probability of a CryptoPunk having the considered attribute or not.

B. Reasoning: Model and structure Choices

The reason we included 92 different CNNs is that for some attributes, for example the Alien attribute, only appear on a few Punks out of the 10'000. Making a single CNN with many kernels and layers was possible. It would have assessed the 92 traits at once, but it wouldn't have recognised this particular characteristic (being an Alien or not) correctly. The dataset would not have been balanced. The model wouldn't have been able to accurately tell when a Punk is an Alien or not. We therefore believed that a CNN per attribute was the best approach.

In most cases, such as for the Alien race CNN for example, oversampling was used. Since this method will duplicate the 7 Aliens until they number 8'000, the idea is for the model to recognise what is not an Alien, rather than recognising Aliens directly, since it doesn't have enough input diversity to do so. For more numerous attributes, the model is able to properly recognize them.

The CNN structure was chosen arbitrarily by trying different combinations of FC-layers and Convolutional layers. The structure which was used was the one which gave the best performance out of the ones that were tried.

Note that binary classification is also possible with only one node in the output layer, but this wouldn't have allowed for the use of the Softmax function. Since the conversion to classification probability is essential, two output nodes had to be included.

Since we consider 24x24 images initially, we believe a 2x2 kernel was the best approach. Bigger kernels would have provided faster computation times at the cost of increasing prediction error. Since CryptoPunks are intricate and small sized, in order to have good model performance a small kernel seemed best.

VI. Code implementation

The project was coded in a single main.py file in order to facilitate the use of Google Colab, which was used to run the code since personal computers took too long when dealing with the 92 CNN. The entirety of the code was designed to that it can be run on any PC. If the required files are missing, they will be automatically downloaded. Since the last OpenSea API update, this will not work unless this project's code is updated as well (see the Self-Criticism paragraph for more explanation). The deep learning module used in this project was PyTorch.

A. Data Gathering

The OpenSea API, which is an NFT Marketplace, was used to gather the traits of each Punk. Threading was used to send the requests to the API in order to speed-up the process.

The Punks themselves were obtained by algorithmically cropping them out of the image that contains all 10'000 of them. This image was taken from the GitHub page of the CryptoPunks project. All the obtained data was inserted in .csv files.

B. Data Cleaning

A lot of data was downloaded with the Opensea API. It was then cleaned so that only the CryptoPunks and their corresponding attributes remain. The attributes were all converted to dummy variables. The starting dataset for the traits is therefore made of 10'000 rows, 1 for each CryptoPunk, and 93 columns. The first one is the CryptoPunk ID and the 92 next columns are the dummies for each attribute.

The Punks in the image which contains the 10'000 CryptoPunks are displayed in their ID order. The top left CryptoPunk has an ID number of 0. The one to its right is number one, etc., until number ID 9999 in the bottom right of the image. Therefore, when cropping them out of the image, the CryptoPunk images were inserted in a folder with their corresponding ID number.

The results of these 2 steps are one dataframe with all the attributes, and a folder of all CryptoPunk images and their corresponding ID.

C. Data Building

An 80/20 data split is performed. This step is essential in order to see if there is data overfitting. It will also allow to test the model's performance on the 20% subset of the data which it has never seen, since it will only be trained on the 80% subset. The result of the split is two sets of data, one containing 8'000 Punks (the training dataset) and the other 2'000 (the testing dataset).

Data Building is performed on both. The first step was to link each Punk image to its corresponding traits through the Punk's ID present in both databases. All images were first converted to grayscale and then to an array representing each pixel's grayscale value. Since elements of the array can take values between 0 to 255, they were all divided by 255 to obtain values between 0 and 1, which is a useful step in order to improve the model's efficiency. When the data is not normalized, the shared weights of the network have different calibrations for different features, which can make the cost function converge very slowly. Normalizing the data makes the cost function much easier to train.

Then, two datasets per attribute are created. For each attribute, we obtain one dataset containing all the CryptoPunks which have the considered attribute and one with all the Punks which don't have it. At this point, oversampling was used in order to make up for the differences in number. Some traits are present on more CryptoPunks than others. The two datasets per attribute are therefore oversampled up to 8'000 observations. The goal is to obtain balanced dataset across all 92 attributes.

Both datasets for each attribute still have all the columns made of dummies for the other attributes. These are not needed anymore. They will be replaced by a one hot vector. The one hot vector for the Cryptopunks that have the considered attribute is [1,0] and for those who don't have it the one hot vector is [0,1]. The two datasets are then combined into one for each attribute.

The results of the previous steps are 92 different dataframes, one per attribute. Each dataframe is made of 16'000 rows and 2 columns. In the first column there is the image corresponding to each CryptoPunk in grayscale normalized values and in the second the one hot vector. Each total oversampled dataframe was shuffled.

D. Data training

The model was trained using Google Colab. The reason is that our computers couldn't handle the workload needed. Even with the use of Goocle Colab, which allows users to use computational resources from Google, the project still took 2.5 hours to run. The tqdm progress bars were implemented in order to see the model's progress.

E. Performance Metrics

The usual performance metrics are described in this section. They will be used to showcase the results.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{12}$$

Accuracy shows how much we predicted correctly, out of all the CryptoPunks in a CNN's dataset. This metric was also computed for the entire 92 CNNs to obtain the model's total accuracy.

$$Precision = \frac{TP}{TP + FP} \tag{13}$$

The precision metric works in the following way: out of all the CryptoPunks the model predicted as positive, how many are actually positive.

$$Recall = \frac{TP}{TP + FN} \tag{14}$$

The recall metric shows the following: the percentage of Punks that we correctly predicted as having the trait compared to the total number of Punks which have this trait.

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2 * TP + FP + FN} \quad (15)$$

Finally F1 is a weighted average of the precision and recall values, where an F1 score reaches its best value at 1 and worst value at 0. It is useful to compare models.

VII. Results

There are many interesting points that must be brought up in this section. Let us start with a concrete example showcasing some particular CryptoPunk features and their corresponding estimated F1. The 3 worse F1 and 3 best F1 are selected and compared.

F1 is the harmonic mean between recall and precision, and is a good way or displaying the models overall performance in binary classification, keeping in mind a perfect model has a F1 score of 1. The three worse are shown in Table 1.

TABLE I

Worst F1 traits	
Trait	F1
Rosy Cheeks	0.041025641025641
Spots	0.1649484536082474
Blue Eye Shadow	0,178897025171624

Rosy Cheeks, Spots and Blue Eye Shadow were very hardly recognisable by the model. Their representation on CryptoPunks is shown in Fig.7, 8 and 9.

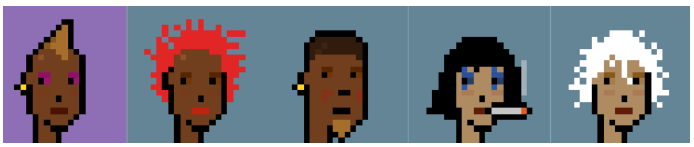

Fig. 6. Examples of CryptoPunks with the Rosy Cheeks attribute.

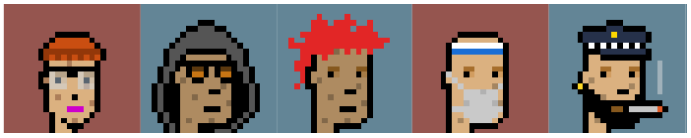

Fig. 7. Examples of CryptoPunks with the Spots attribute.

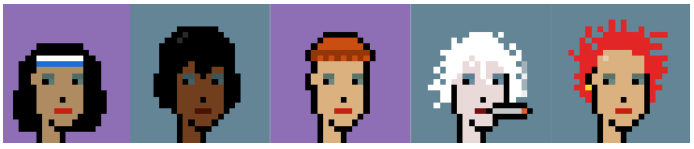

Fig. 8. Examples of CryptoPunks with the Blue Eye Shadow attribute.

It turns out that even for us it is hard to recognise some of these particular attributes, and in some cases, the colored pixels representing rosy cheeks or spots are simply non-existent or hidden by other attributes. Since images were converted to grayscale, it is likely that the difference in color was not sufficient. The CNN simply couldn't tell the difference between a skin color pixel and an attribute pixel. This can be corrected by keeping the RGB values at the cost of computational time. The Blue Eye Shadow case is slightly different. The errors are probably due to the existence of other eye shadow colors, positioned in the exact same way around the eyes of the Punks. The CNN could therefore not differentiate them properly. Another explanation is the lack of trait exclusivity in the model. CryptoPunks can't have Eye Shadows of different colors or two pairs of glasses, or two skin colors. This additional information would probably have resulted in better results overall.

The attributes with the best F1 are reported in Table 2.

TABLE II

Best F1 traits	
Trait	F1
3D Glasses	1
Classic Shades	1
Wild White Hair	1

Some examples of two perfect F1 attributes are shown in Fig.9 and 10.

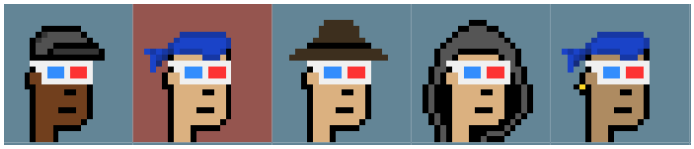

Fig. 9. Examples of CryptoPunks with the 3D Glasses attribute

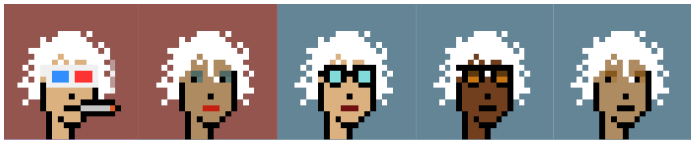

Fig. 10. Examples of CryptoPunks with the Wild White Hair attribute

As we can see, there is a major difference in visibility on the image compared to the previous attributes. In fact, this time we are able to recognize these attributes at a simple glance without any effort, much like the CNNs in these two cases. The best performing CNNs all had attributes with consistent shapes and sizes. They are also relatively big compared to the input's size and are not hidden by other attributes.

All in all, this shows that our model performed well, only falling short in cases where there is not much that can be done.

As a further proof of this argument, Fig.11 shows two graphs of the average accuracy & loss over 92 models, for both in-sample (training) data and out-of-sample (testing) data for one EPOCH, or equivalently 160 model steps.

Both measures' improvement rate becomes almost negligible from the 80th step on wards, which suggests that the

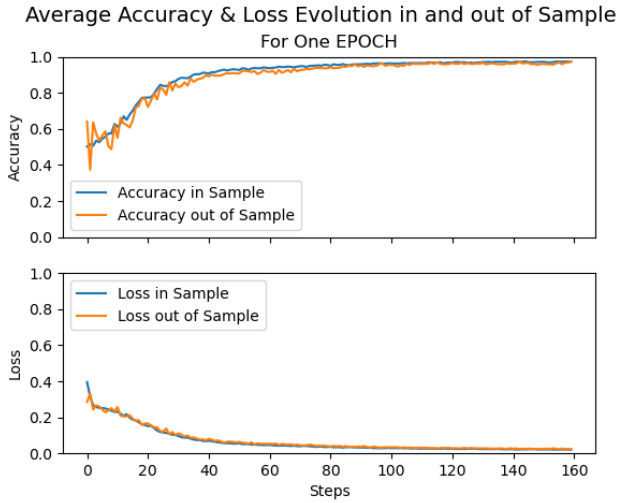

Fig. 11.

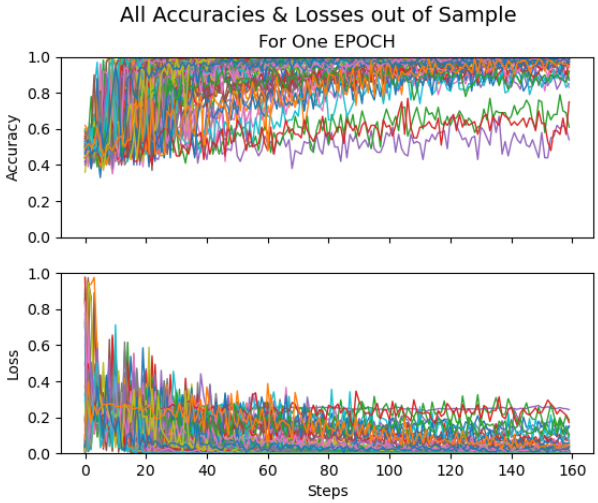

Fig. 13.

number of iterations could be halved to achieve similar results on average. The losses graphic is a pleasant surprise as it shows strong model performance overall. When faced with the testing dataset, the model shows similar results to the testing dataset.

Fig.12 shows the same measures with the entire 12 EPOCHS.

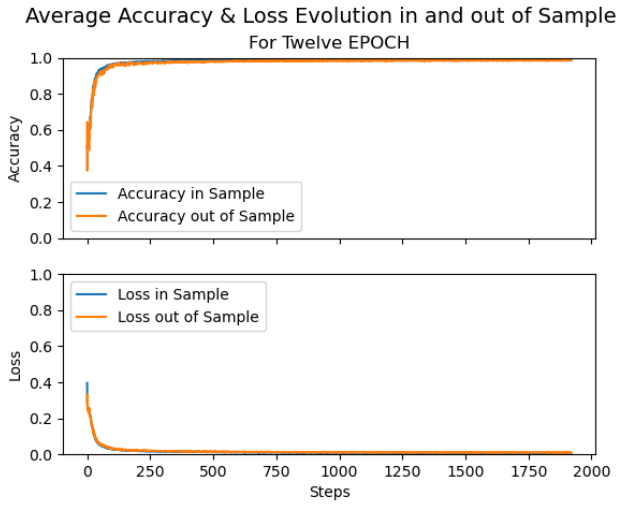

Fig. 12.

One might think that having this many EPOCHS doesn't provide any benefit from the second or third EPOCH on, but previous graphs showed average metrics only. The dropout implementation allows for the use of many EPOCHS without overfitting the data. The reason this many were used can be seen in Fig.13.

The individual metrics per CNN show diverse results for the reasons previously discussed at the begging of this section. Additional EPOCHS were therefore added in order to have satisfying metrics for most CNNs or until convergence was obtained, meaning the model's limit was reached, and no further significant improvement could be measured. Fig.14 shows the big picture of the considered metrics over all 12 EPOCHS for all 92 CNN.

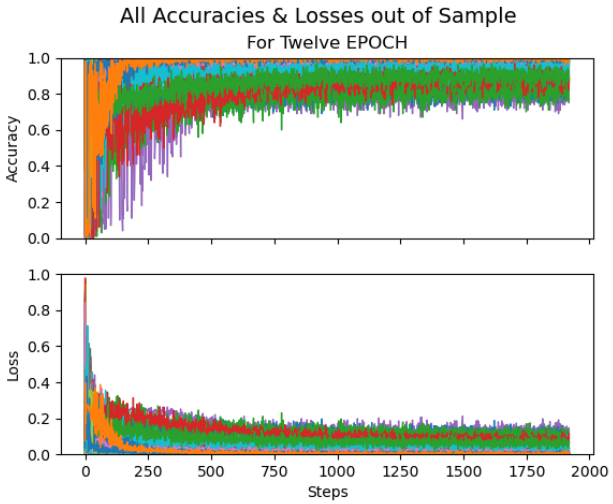

Fig. 14.

12 EPOCHS was the amount that produced the best results, which is why it was used in the end. Finally, Fig.15 shows the accuracy, precision, recall and F1 our model achieved in total.

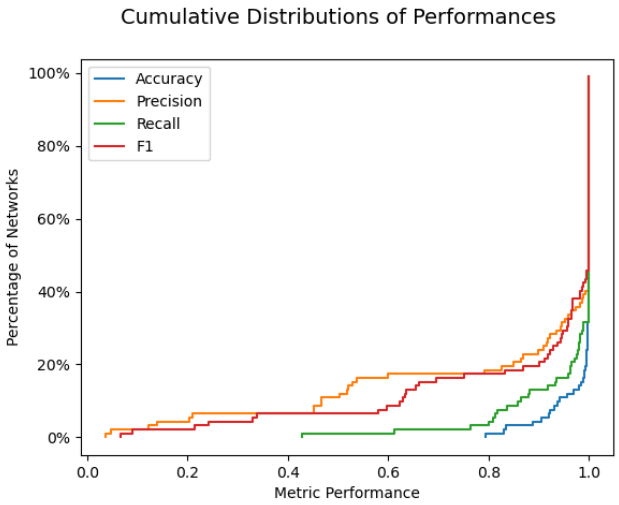

Fig. 15.

Fig.15 shows that the overall performance is satisfactory. First and foremost, we were happy to see that 50% of our models achieved perfect metrics showing strong results. No models achieved accuracy lower than 0,8. About 80% of the models showed a precision and F1 of at least 0,8. Finally about 90% of models achieved a recall of 0,8 at least.

This figure also shows that improvements can be made, but for a first CNN implementation, results seem satisfactory.

VIII. Self-Criticism

Looking back on our project, and having covered topics related to our problematic, we are able to tell that some improvements and corrections can be made.

Firstly our CNN's structure is definitely sub-optimal, and we are aware that better results can be attained with the proper layer combinations. Finding the best model structure is the object of many studies. Given our number of CNNs and computation time, there is room for improvement through trial and error.

Code efficiency can be improved. As previously mentioned, even with the implementation of different methods known to improve computation time, the model still takes 2,5 hours to run on high performance hardware. A potential solution to this issue is an autostop, which would avoid useless training. The portability can also be improved. The total project file weighs more than 2 GB.

Using PyTorch wasn't wise. TensorFlow would have been a simpler module to use as it is more straight-forward and was also introduced in class. We chose the harder path by using PyTorch.

Our data was gathered through an API called OpenSea as we specified above. The issue is that since the start of the project, a possible API update happened which made us unable to run our code, although it was working fine at the start. This means that our code cannot be run at the moment. Our data gathering section in the code would have to be updated. This issue only happened once the project's code was finished, so it doesn't change our results in any way.

Converting inputted images to grayscale means that features relating to color only will be difficult to recognise. We were aware of this from the start and chose to keep this choice in order to further reduce computation time.

IX. Conclusion

This project was focused on applying image recognition techniques to CryptoPunks to extract their attributes related to shapes, sizes and positions. The attributes were recognized through the implementation of 92 different CNNs to each trait. Results show that most attributes were successfully recognized by our models and those that weren't are in fact hard to recognize even for human eyes. Better performance can certainly be achieved by restructuring the network as there are many ways to approach an image recognition problem. The maximization of performance and reduction of computation time have been the focus of many datascientists worldwide. This project contains many arbitrary choices that can certainly be improved upon, such as adding convolutional layers, changing the structure to have a convolutional layer followed by a pooling layer and repeating this sequence many times, changing the kernel size to a column vector that "scans" the input much like a printer would, etc. All these options and many more must be tested by implementing them one at a time and then combining them in order to obtain the best performing model possible. Future versions of this application should consider implementing these options.

Overall, we are extremely pleased with the results given this was our first CNN implementation. Hopefully our approach to this problem can inspire future projects.

X. References and Sources

References

[1] S. Saha. (2018, December 15). *A Comprehensive Guide to Convolutional Neural Networks— the ELI5 way*. Available:https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[2] Intel AI (2016, July 14). *Convolutional Neural Networks*. Available:https://www.youtube.com/watch?v=SQ67NBCLV98

[3] D. MWiti (2018, May 8). *Convolutional Neural Networks: An Intro Tutorial*. Available:https://heartbeat.fritz.ai/a-beginners-guide-to-convolutional-neural-networks-cnn-cf26c5ee17ed

[4] Larvalabs (2017, December 28). *GitHub page of the CryptoPunks project*. Available:https://github.com/larvalabs/cryptopunks

[5] Larvalabs. *Official page of the CryptoPunks project*. Available:https://www.larvalabs.com/cryptopunks

[6] Opensea (2017, December 28). *Opensea marketplace for trading CryptoPunks*. Available:https://opensea.io/assets/cryptopunks Fukushima, K. (2007).

[7] R.Venkatesan, B.Li (2017, October 23). *Convolutional Neural Networks in Visual Computing: A Concise Guide.*.

[8] D.Ciresan, U.Meier, J.Schmidhuber,(2012, June). *Multi-column deep neural networks for image classification*.New York, NY, United States

[9] K.Fukushima (1980). *Neocognitron*. Kansai University, Japan

[10] Srivastava, Nitish; C. Geoffrey Hinton; A.Krizhevsky, I.Sutskever, R.Salakhutdinov (2014). *Dropout: A Simple Way to Prevent Neural Networks from overfitting*.Journal of Machine Learning Research.