Binary Classification of GAN-generated High-Resolution Images via Convolutional Neural Networks

Alice Schiavone, Università Statale di Milano

Statistical Methods for Machine Learning, Algorithms for Massive Datasets
(September 2021)

1 Abstract

This paper reports the reasons and problems met during the development of a Convolutional Neural Network that gives binary answers to the query "does the person in the image wear glasses?". Before analysing the results, the reader can get an understading to how and why CNNs were born to solve the task of image classification in machine learning. The main issues regard the scalability of the network pipeline and the search for hyperparameters. The power of a simple CNN, when fed properly, gave a satisfying solution to the task with little resources and a limited number of tests.

2 Introduction

Since the concept of Artificial Neural Network has been introduced in 1943 [1], this computing system has helped to develop powerful and versatile machine learning algorithms, which led scientists to solve large and complex tasks. These include classifying billions of images, by assigning them a label from a finite set. Specifically, we focus on the problem of determining if the people represented in the given pictures have glasses or not (Figure 2). Although the task can be presented briefly, its implementation requires many techniques and refinements which are meant to solve some of the biggest challenges that a machine learning algorithm can encounter, especially when dealing with a intricate network. The reader will experience the process of defining such issues, exploring different solutions and how they work, evaluating the best alternative.

Figure 1: Faces generated by Prof. Heaton's GAN

3 Dataset

The images come from a public dataset published on Kaggle [2] by Professor Jeff Heaton of Washington University in St. Louis, under the Attribute-ShareAlike 4.0 Creative Commons license (CC BY-SA 4.0). Prof. Heaton generated these images through a GAN (Generative Adversarial Network), a powerful generative model designed in 2014 [3], that since its introduction has found many applications. In this case, Prof. Heaton used a GAN to generate 5000 images of human faces, using a 512-dimensions latent vector, and later asked his students to develop a machine learning algorithm that would be able to recognise whether each computer-generated-person wore glasses or not. We will do the same, but to frame the problem (and develop better solutions), we will assume our algorithm purpose, which wasn't previously given.

In fact, based on the context, the different implementations would be quite distant from each other, and subject to the reader imagination; this image classifier could be used by the police to detect if driving people with a myopia diagnosis are actually wearing glasses: this would require fast computation times with an incoming stream of data from a low resolution street camera. Maybe it could be used to detect how many models in Milano Fashion Week runaways wore glasses this year, to decide if they are fashionable right now: we would then have little data (only hundreds of models) but many different images and styles. Of course we could not choose one application and try to have the model to be useful for many of them, but this is usually not the case and the images from the dataset are very similar to each other, and unlikely to fit to realistic portraits because of the well centered frame and perfect lighting.

These are the reasons why we chose to implement a model that would serve the image generative pipeline, by evaluating the effectiveness on the model of generating people that wear glasses or not. This is in fact one issue presented by Prof. Heaton, because the dataset contains uncertain images which cannot be classified even by a human eye (Figure 3). An image classifier could be useful to improve the GAN and discard images that could not be used for other purposes.

Figure 2: Example of uncertain images

To sum up, the important characteristics of our data and related problems are:

1. This is a binary classification task, with one output: the probability of the image of being 'no glasses' (0) or 'glasses' (1). Discard images with too much uncertainty (close to 0.5);
2. High resolution of 1024x1024 pixels: too high for a neural network;
3. High number of images: we only have 5000 available, but we assume that many more would be generated and evaluated throughout the generative pipeline;
4. Same frame, position and lightning for all images: our classifier will likely overfit the data, but the model will be tested on similar images and not on real life captured photos;
5. The data set comes with labels only for the first 4500 images. The last 500 will not be used for testing, instead the labeled ones will be divided into training, validation and testing sets to evaluate accuracy;
6. Along the labels, the .csv file contains also 52 attributes that represent the parameters used by the GAN to generate the images. We will ignore this to focus on a traditional image classification task;

4 Network Architecture

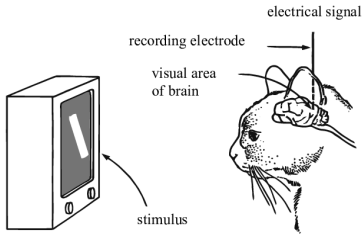

Figure 3: The experiment done by Hubel and Wiesel on a domestic cat visualising bars of light. [4]

In 1981, David H. Hubel and Torsten Wiesel receive the Nobel Prize in Physiology or Medicine for their work in sensory processing and the development of the visual system. In particular, in 1959, they projected lights in patterns in front of a cat, which was previously anesthetized to insert a microelectrode in its visual cortex. [5] They noticed that not all neurons react to lines of different angles equally, in fact some activate when seeing horizontal lines and some when seeing vertical ones. Moreover, not every neuron sees the same, they instead react to only a small portion of the visual field. By overlapping the receptive fields of different neurons, the brain puts together the overall image seen by our eyes. In order to detect all the complex patterns, some neurons react to combinations of the straight lines detected by the 'lower-layers' connected to

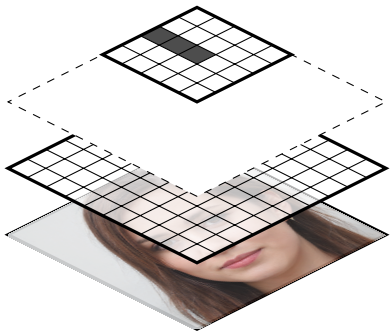

Figure 4: A convolutional layer scans the input image pixels with a filter to output a new feature map. In this toy example, a filter of size 5×5 detects a vertical line of 3 pixels.

them. This means that the brain is an intricate network of layers, that rely on different levels to notice patterns, starting from simple ones.

This research prompted scientists working on artificial neurons to develop an architecture that resembles what an organic brain normally does, to detect patterns in digital images. The first to introduce this concept was Kunihiko Fukushima, who used a multilayered neural network with hierarchies for character recognition of Japanese handwritten characters in 1979. [6] He called it a neocognitron, and after 18 years of research, Yann LeCun introduced the LeNet-5 architecture, a 'complex decision surface that can classify high-dimensional patterns, such as handwritten characters, with minimal preprocessing'. [7] This architecture also classified digits of hand-written characters, and was successfully applied by banks to read cheques because it outperformed other models. This event enstablished the neocognitron evolution as the standard for image classification tasks, because instead of having fully connected layers as previous neural networks, it introduces the concept of *convolutional layer*.

Normally, a neural network layer has all its input nodes connected to the next layer, which means every pixel of the input image is computed at once. Instead, a convolutional layer gives meaning to the pixel positioning in the image grid, and computes the weights for a "window" of values, that moves over the image and whose output is directly connected to the next layer (See Figure 4). This is not a fully connected layer anymore, and strongly reduce the number of computations: instead of having 1 input and 1 output value, we have multiple inputs (given the size of the window) and 1 output. Combining these layers we get a *convolutional neural network* (CNN). It "remembers" the position of the pixel, with respect to the others in its window; this allows the lower levels to detect small patterns, and higher layers to combine them. A *filter* or *kernel* is the neuron's weights represented as a small image, usually of the size of 3x3, 5x5 or 7x7. The filter is applied to the receptive field of the neuron in order to detect patterns: it performs an element-wise multiplication and sums up the

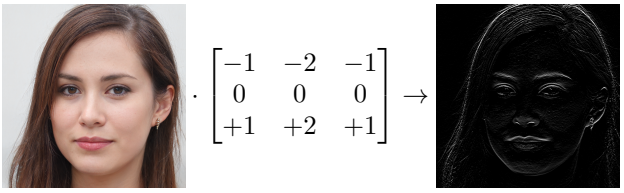

Figure 5: Feature detection example: Sobel filter (3×3) applied to detect horizontal lines in the image. A convulation layer learns its own filter.

result to form the *feature map*. (See Figure 5) A convolutional layer is a stack of multiple filters and their respective feature maps, which individually share the same parameters and bias for all neurons.

Given that images are usually very repetitive, a CNN need less training images for an image classification model, because it can detect different features after learning a kernel that can scan the image in group of pixels, instead of detecting features in single pixels like in the standard deep neural network.

5 Data Preprocessing

The data set comes with 4500 labeled and 500 unlabeled images. To evaluate the model, we will need labels, and so we discard the unlabeled ones. The images are $1,024 \times 1,024$ in width and height, with 3 *rgb* channels and PNG format. They are centered to show faces of people with or without glasses, with almost no background and with a similar frame throughout all the data set. The image classification can output only two classes, each of which takes the following portion of the entire set:

Glasses	No Glasses
.635	.365

The 'No Glasses' class is the minority class, but we still consider the data set as balanced because it still takes a large portion of it. Also keeping in mind the small number of data points, we will not discard images in order to reach a 50-50 distribution of the classes.

The resolution of the images needs to be taken care of: if we want a fast computation, we need less parameters, the nature of the convolutional layer is not enough to reduce them. While loading the data as needed (see Section 7 about Scalability), we resize the images to a new resolution. Testing different resolutions, with a non-tuned model, we get the following accuracy on the validation data set, after 20 epochs:

Resolution	32×32	64×64	128×128
TFrecord file size	**1.1M**	3.6M	13M
Validation accuracy	.8611	**.8733**	.8444

It is clear the best results come from the 64×64 resolution, with little more space required with respect to the 32×32 resolution, but because we don't have enough resources to train the model as one would like, we chose the 32×32 resolution to have faster computation times and try more tests. It is important to note to the reader that the tests done by the author are not extensive: this means that the reported results are the outcome of only one specific run, but represent the intuitive optimization done to reach the best possible solution at the end of the overall process.

6 Hyperparameters optimization

The example sets sizes and the image resolution are parameters that can be taken care of during the data pre-processing phase. Instead, in this section we focus on *hyperparameters*, whose value is used to control the learning process. It is easy to notice that in the context of Convolutional Neural Networks the number of hyperparameters can get big pretty quickly.

First, we consider the batch size, which is the number of images on which the model trains at a time. The *mini-batch* size in this implementation corresponds to the batch size of TFRecords files. This is done because it would be too computational expensive to compute *stochastic gradient descent* on each training example. It would be more accurate, but not realistic. Remember that *SGD* is used to minimize the loss between the predicted label by the output neuron and the true label for that example, or when using mini-batches, for a set of examples. The weights of the network are the inputs to the loss function that we want to minimize. Instead of *SGD* we used *Adam*, which complicates the hyperparameter tuning problem by adding two more parameters, but usually performs better that *SGD*. *Adam* is based on "adaptive estimates of lower-order moments" [8].

The final model was trained for 3 different batch sizes to see, in this particular problem, which was the best result. As done previously, each test is only run once. Repeating the same test, with the best resolution, but with different batch sizes we get:

Batch size	32	64	128
Computation time	**33.4s**	40s	34.4s
Validation accuracy	.895	**.899**	.887

Given the results, we will use batch size of 32 examples. The accuracy is slightly less than the better batch size of 64, but given the uncertainty of the experiment we will emphasize the slower computational time.

The lower computational time is also possible because of the EarlyStopping callback, that gives the training a reason to stop, in this case when the validation loss doesn't improve after some number of epochs (this itself is another hyperparameter).

In order to lower the learning rate of Adam, we use a learning scheduler that computes the next value of the learning rate as follows [9]:

```
def decayed_learning_rate(step):
  return initial_learning_rate * decay_rate ^ (step / decay_steps)
```

This exponential decay function is used during the model training to compute more efficiently the *Adam* optimization.

The architecture of this Convolutional Neural Network is fairly simple: for every Convolutional Layer with an activation function, we add a pooling layer and a Batch Normalization layer. These are indeed hyperparameters.

The pooling layers purpose is to down sample the convolutional layer features, so that they are not dependent on the pixel positions. The batch normalization layer normalizes its inputs so that the output μ is close to 0 and σ is close to 1, making the network faster and more stable, even though the reasons of why it works so well are still debated. [10]

Other hyperparameters include the dropout rate, the window size and the stride value. The last two didn't result as significant as the dropout rate, which instead was very much important to solve the problem of classification. In fact, if the dropout rate is not tweaked as needed or if the dropout layer is not in the network at all, the results get pretty bad, with the validation accuracy getting stuck pretty early with no further improvement.

The final layer of the network has a very important job: output the final prediction for that example. To do so, it needs a function, which in itself is another hyperparameter. For binary classification, you could use two output neurons with a *softmax* activation function, which lights up the neuron which stands for 0 if the prediction is 'no glasses', and 1 if it is 'glasses'. This is obviously redundant as you could use only one neuron which doesn't light up if the prediction is 0, and lights up if it is 1. This could be done easily by the unit step function (Figure 6), but because we use backpropagation to update the weights of a network deeper than one layer, we need a differentiable activation function. The unit step function derivative is always 0, and non differentiable at $x = 0$. Backpropagation will fail because the gradient descent won't be able to update the weights in a useful fashion.

$$\text{f(x)}= \begin{cases} 1 & \text{if } x \geq 0.5 \\ 0 & \text{otherwise} \end{cases} .$$

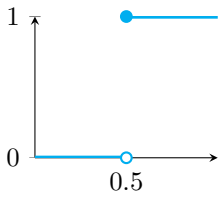

Figure 6: Unit step function.

That is why we use the sigmoid activation function (Figure 7) for the final

layer, and neuron, of this CNN: it mimics the unit step function, because it always outputs values between 0 and 1, but has the required properties to apply backpropagation. It is also called logistic function.

It is an estimate of the probability of an instance belonging to a certain class, where 1 stands for "this example is of class P" and 0 for "this example is not of class P", which is a perfect binary classifier. If the probability falls below 0.5, sigmoid will assign it to the negative class ('no glasses'), otherwise to the positive one ('glasses').

$$sigmoid(x) = 1/(1 + e^{-x})$$

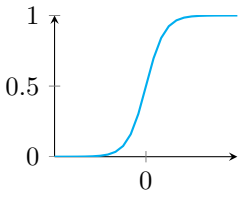

Figure 7: Sigmoid function.

The rectified linear activation unit (Figure 8), or ReLU for short, is the simple function that outputs the given input value if its positive, otherwise it returns zero. Or simply, the *max()* function defined as $f(x) = max(0, x)$. Its derivative is easy to compute (needed to update the weights of the network) when assuming the derivative in 0 is equal to 0, it doesn't require an exponential computation (like tanh and sigmoid) and its linearly increasing slope ensures that the output is a true zero value (not a very close value to zero, like tanh and sigmoid). For these reasons, ReLU has become the standard activation function for the hidden layers of most neural networks.

$$\text{ReLU} = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}.$$

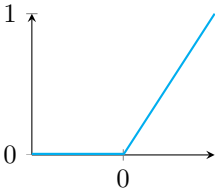

Figure 8: ReLU function.

Last but not least, the model needs to evaluate its outputs while training. To compute the loss between the predicted value and the true label, we use the logistic loss, also known as the cross entropy loss, with only two classes, C_1 and C_2:

$$CrossEntropyLoss = -\sum_{i=1}^{C=2} t_i log(s_i) = -t_1 log(s_1) - (1 - t_1) log(1 - s_1).$$

This function needs an input in the range of (0,1), so that is why it is paired with the sigmoid activation function in the previous layer. We like this function because when the model estimates a probability close to 1, but the instance is negative, it will output a big cost, and vice versa. We cannot compute the value that minimizes *log loss*, but we rely on *Adam* to find the local minimum.

Given the number of huge number of possible combination of parameters (number of layers and filters, max or average pooling, number of neurons), we implement a hyper parameter search model with Keras Tuner. In particular, we use Keras built-in method *RandomSearch* to randomly try different combinations. This method is not very efficient, because it doesn't implement any strategy in order to keep track on which combinations work better, but it can still be useful to detect important parameters that couldn't be noticed before (see more about it in Section 8, Results).

7 Scalability

A big challenge for CNNs is rooted in the premises of the problem: it needs a lot of images. The reader could think that a the order of magnitude of 'a lot' stands in the range of 10^3 or 10^4, but in reality a deep learning algorithm needs to be fed with much more images to be able to give acceptable results. In this particular application, we had only 4.500 labeled images. Even if these number could not be enough for a CNN, it is certainly too big for most of the computational units that users casually own for their daily tasks. The set of all images, with resolution of 1024×1024 size is around 6GB. These 6GB need to fit in RAM to be processed by the neural network all together, for more than once. Considering all the hyperparameters and computations, the space and time required by the network to see all the images once can easily skyrocket. For example, the first layer of the network is made of $1024 \times 1024 \times 3$ nodes, for a total of 3,145,728 nodes only for the first layer.

The first step to solve this problem is to use a GPU instead of a CPU to make the required computations. The simple operations required by a neural network are easily solved by a GPU, because for the most part they are all matrix multiplications, which don't require the full power of a CPU. Instead, the hardware of a GPU was designed to solve this particular operation, even if for other purposes. In fact, for each N-node, there are N^2 scalar multiplications and N sums of N numbers, with a time complexity of $O(N^3)$. This result can be improved by implementing different algorithms, but to this day it's still impossible to compute a matrix multiplication with time complexity lower or even close to $O(N^2)$, which is still huge compared to the required number of computations. The GPU higher number of cores gives the possibility of making more parallel computations, which are also efficiently fed by a dedicated RAM. A neural network computation can easily be parallel, because of the architecture itself which implements nodes which are generally independent from the others.

Given the parallel nature of a CNN, it is easy to image that we can get better results connecting more GPU in a cluster, so that we have many, many cores.

This idea alone could virtually solve all deep neural networks, but in reality a GPU is an expensive piece of hardware, both to build and to maintain.

If we zoom in the problem, we could notice that the reason why the networks takes so much to converge, is not in the layers themselves, but rather at the very beginning of the process: the data cannot reach the network as fast as the network outputs the results. This is called a *bottleneck*, which is a common problem in the data pipeline of big data implementations. The reasons is deeply rooted in the hardware itself, rather than how the software engineer wrote its code. The highest cost of any network is its *communication cost*, which is the cost of making the information travel through the network, usually from disk to main memory. This cost can not be bypassed easily, so we need to find ways around it.

The huge problem with CNN is given by images: they are big and annoying. Each image format has its rules, but even then each pixel carries a big amount of information that cannot easily be compressed. The first logical thing to do is to reduce the resolution of our images. After all, we can still detect patterns if the image is of a very low quality, why wouldn't a machine? So after our initial investigation, we decide to reduce the images from $1024 \times 1024 \times 3$ to $32 \times 32 \times 3$. We now have 3,072 initial nodes vs the previous 3 millions. However, this is not enough. The network still takes a lot of time to compute just one epoch.

The final improvement was made by using *TensorFlow Records*, which is TensorFlow way of storing binary data. This is the format that TensorFlow suggests when dealing with big data sets. Using binary data decreases read-/write time from disk, consequently improving the overall performance. We need to specify the structure of the data that we want to store as binary strings, then serialize it and finally write it to disk. At the same time, we divide data in chunks, so that they can be dealt with more easily when distributed in a parallel system. After resizing the images to $32 \times 32 \times 3$ resolution, we get that a chunk of 450 images needs only 1.1MB to be stored. From the initial set of all images of 6GB, after storing the data as binary strings we need only 0.011GB for all the data set. We can easily notice that this method decreases computational time immensely and, as showed in Section 5, the use of TFRecords did not decrease validation accuracy. One could argue that the process of loading the data, applying such heavy preprocessing and then train the network doesn't bring a net improvement to a standard implementation. This is not true, because the preprocessing takes about 4-5 minutes in total, compared to the hours required to train network that is not efficiently fed, but even if the time needed its more, it still doesn't take in consideration that the initial dataset can be entirely discarded after writing it as binary strings, and could be used many more times to run more tests and to look for better hyperparameters. This approach is scalable, time and space efficient, and comfortable for the user.

8 Results

While exploring the data, it was also possible to notice that some labels are incorrect. With such low resolution it can be difficult even for a human to say if one person is without glasses (see image (7) in Figure 9). While this could be solved by having more resources (GPUs) and increasing the resolution, we cannot solve the "human-in-the-loop" problem so easily. This is fine, as long as we consider that this will introduce Bayes error, because the training set labels are inherently stochastic.

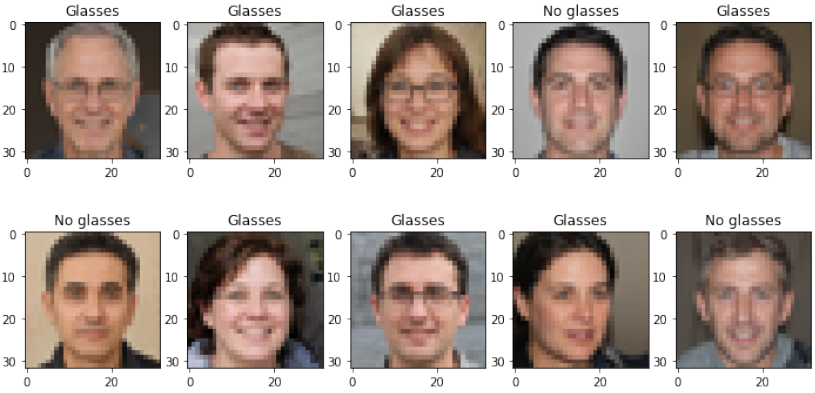

Figure 9: 10 images from the training set: image (2) and (9) are not labeled as expected, or are uncertain.

After training an initial model tuned by various tests, we notice that the accuracy stops to improve after a number of epochs as low as 20 or 30, because it reaches 1.00 accuracy (Figure 10). This result could seem very good, however by comparing it to the accuracy computed on the validation set, it is clear that this model overfits a lot, with a loss of around .409 until the end of the 100 epochs. [1]

In order to test more combination of hyperparemeters, two models were trained with two different strategies: the first model hyperparameters were randomly searched and tested by a Keras built-in method, which trained 5 different combinations of network architecture and finally trained the best one; the same was done for another model that had the same architecture as the initial model, but instead tuned the learning rate, in particular its ability to increase/decrease to find the local minimum. At the end, the models were evaluated.

Model	Accuracy Evaluation	Loss
Untuned Model	**0.887**	.388
Tuned Architecture	0.867	.320
Tuned Learning Rate	0.869	**.318**

[1] The number of epochs is totally arbitrary, it could be possible that a neural network that seems stuck for many epoch actually 'wakes up' later in training, but this would require more time and resources.

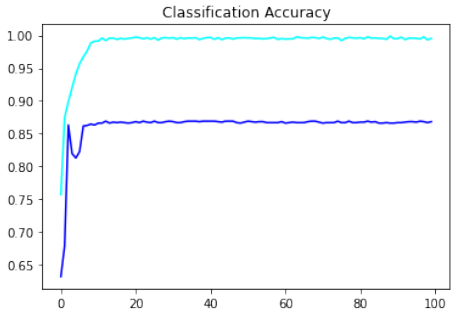

Figure 10: Classification Accuracy after 100 epochs: untuned model.

As you can see in table about the Accuracy Evalutation, the best accuracy result was by the untuned model. This should not come as a surprise, because the search for the best hyperparameters was totally random, and its improvents to the initial architecture are expected to show after many trials. However, the best loss was on the model with a tuned learning rate: this could suggest that spending more time tweaking the learning rate could bring more success than on changing the architecture. The same can be concluded when looking at the plotted functions of the accuracy and validation accuracy on the three models in Figure 11.

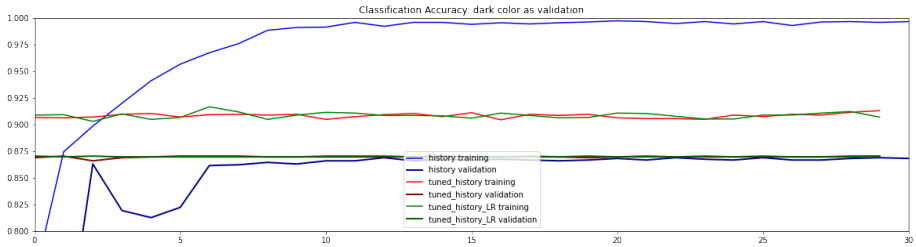

Figure 11: Classification Accuracy after 30 epochs: untuned model, tuned model on network architecture, tuned model on learning rate.

In the end, we can evaluate how our model predicts the label on data it (and we) has never seen before. In Figure 12 we can see that the model preforms well, and due to the probability of being the picture portraying a person with glasses is very close to 0 or 1. The method that prints the pictures also prints if the model is not sure about the label, when the probability is between 0.4 and 0.6, which was the initial objective of this project.

The results are encouraging and can be further improved by applying some ideas that came while analysing the overall process and outcomes. Another technique that was useful to detect wether the architecture was performing well or was too deep or shallow, was to print the feature maps produced by each convolutional layer and their pooling layer. This gave a direct insight on what

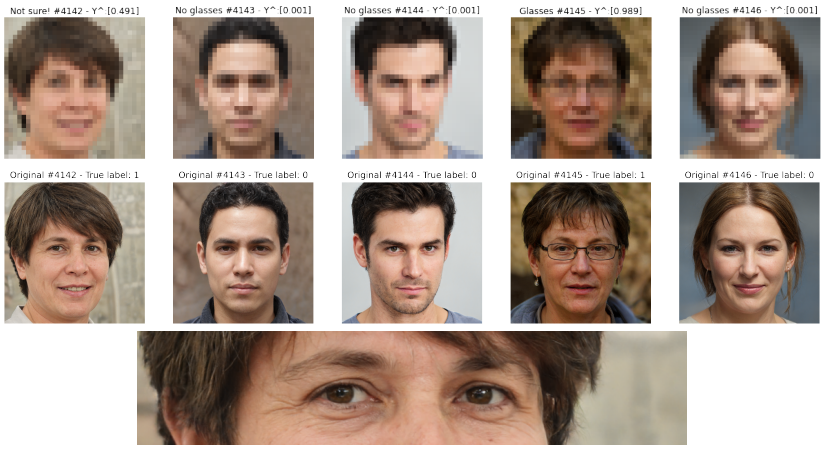

Figure 12: Predictions on some images from the test set. Detail of the eyes of the first image, which has $\hat{Y} = 0.491$, which implies a state of uncertainty by the model.

the network was seeing, and which tweaked actually proved useful. Starting from the higher layers (see Figure 13) where the face of the person is still discernible, we finally get to the last layer where only a handful of pixels lights up.

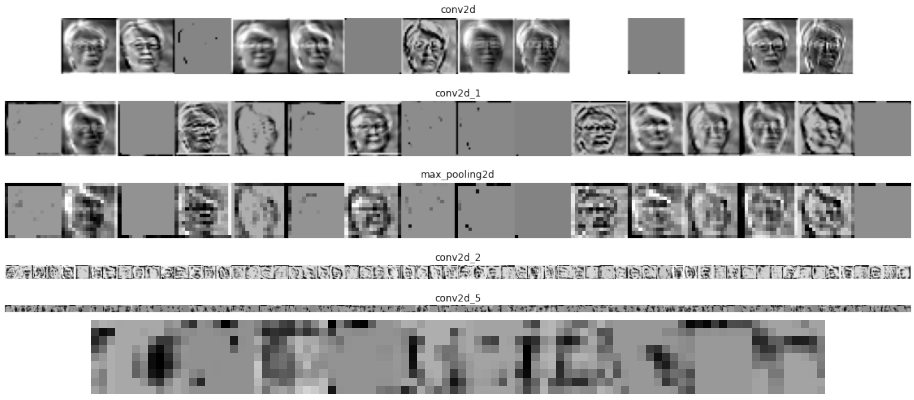

Figure 13: Some feature maps. (Top: higher layers, bottom: lower layers). Detail of last 10 tiles of the last layer.

9 Discussion

Something that was considered during the development of the project was the application of image transformations, but ultimately it was left out of the final tests for the following reasons:

- If we imagine that this model would enter the evaluation pipeline of the GAN that generated these images, it would make more sense to generate more data, not to get new ones from transformations.
- It was preferred to focus resources on tuning and understanding how the hyperparameters work, rather than simply get better results by a *"brute force"* approach.

While many improvements are required to this project in order to fully function as a real-life application, building it has been a fun journey. I hope the reader realized, as the author did at the end of it, that the intuition of the brilliant people that applied biology to the task of image classification by computers is not enough to replicate the astounding capabilities of an organic being.

Regardless of the limitations, some ideas to improve the final model are:

- Increase the number of GPUs: the code is already set to work this way, but could not be tested
- With more computing power, the resolution of images could be improved to get better results
- Tweak the learning rate, which could turn out to be the solution to large overfitting
- Test a deeper network for more epochs
- Get more data

References

[1] McCulloch, W.S., Pitts, W. "A logical calculus of the ideas immanent in nervous activity". In: *Bulletin of Mathematical Biophysics 5* (1943), pp. 115–133. DOI: https://doi.org/10.1007/BF02478259.

[2] Prof. Jeff Heaton. *Glasses or No Glasses*. 2020. URL: https://www.kaggle.com/jeffheaton/glasses-or-no-glasses.

[3] Goodfellow, Ian; Pouget-Abadie, Jean; Mirza, Mehdi; Xu, Bing; Warde-Farley, David; Ozair, Sherjil; Courville, Aaron; Bengio, Yoshua. "Generative Adversarial Networks". In: *Proceedings of the International Conference on Neural Information Processing Systems* (2014), pp. 2672–2680. DOI: https://arxiv.org/abs/1406.2661.

[4] Anh Nguyen. *Understanding Neural Networks via Feature Visualization: A Survey - Scientific Figure on ResearchGate*. 2019. URL: https://www.researchgate.net/figure/Fig-In-the-classic-neuroscience-experiment-Hubel-and-Wiesel-discovered-a-cats-visual_fig1_335707980.

[5] David H. Hubel. "Single Unit Activity in Striate Cortex of Unrestrained Cats". In: *the Journal of Physiology* (1959), p. 147. DOI: 226-238.

[6] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biol. Cybernetics 36* (1980), pp. 193–202. DOI: https://doi.org/10.1007/BF00344251.

[7] Y. Lecun; L. Bottou; Y. Bengio; P. Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE (Volume: 86, Issue: 11, Nov. 1998)* (1998), pp. 2278–2324. DOI: https://doi.org/10.1109/5.726791.

[8] Diederik P. Kingma ; Jimmy Lei B. *Adam: a Method for Stochastic Optimization*. 2014.

[9] TensorFlow documentation. *tf.keras.optimizers.Adam*.

[10] Santurkar, Shibani ; Tsipras, Dimitris ; Ilyas, Andrew ; Madry, Aleksander. *How Does Batch Normalization Help Optimization?* 2018.