The following article is a part of the project and describes how computer vision and CNN work. I decided to upload it on GitHub also as a PDF to make it more easily accessible.

Intro - Computer Vision

People are, in general, really good at picture recognition. Is it a person? How old? In what mood? It only takes us a fraction of a second and we extract all this information. Computers, on the other hand, struggle with this task. So, how does a computer see

the picture you uploaded?

Hopefully you did not spend a lot of your time looking for the sharpest best looking picture that was taken with your new 48-megapixel camera when playing with FaceGuesser. While it would make sense that higher resolution should help our algorithm perform better, in reality, the very first thing that happens to the uploaded image is this:

Our program resizes the image to 128x128 pixels and desaturates it. This saves loads of computation time. While we lose details, it allows us to train our model in a reasonable time even on our home computers (or cheap servers 😅). It all boils down to what we are looking for. If we are making a program that detects if there is a face in the picture or not, we can probably get away with even lower resolution (like 32x32) and still get the job done. If we are looking for finer details, we need to find the sweet spot between performance and accuracy. But this is still not what the computer sees. It is actually

much closer to this:

```
157 157 133 90 54 49 42 45 40 32 31 32 27 26 27 25 27 26 25 23
24 26 27 26 25 25 26 29 31 31 30 30 33 33 35 33 30 32 35 35 25
27 28 27 25 24 28 31 36 31 29 33 36 36 36 38 40 46 53 56 58 63
70 77 86 85 81 80 86 90 92 94 93 83 79 85 104 98 87 80 84 83 61
65 56 55 48 48 50 49 64 51 78 106 83 54 37 40 54 43 44 42 45 52
52 68 88 127 159 163 163 162 161 160 160 160 160 161 160 159
160 160 160 160 160 160 160 160
```

All our program sees is an array of numbers. Each number represents one pixel in our picture. The closer the number is to 255, the more intense (=brighter) it is. The closer it is to 0, the darker it gets. The numbers in our example above represent the first row of pixels. There are 127 more lines just like this and 16,384 numbers in total. If we had not resized the picture and used the original (441x593px), we would get 261,513 values. In addition to that, if we had also kept our image colorful, we would get 784,539 values 🤯.

Chapter 1 - Patterns

To recognize what the picture shows, our program looks for patterns. To give you a grossly oversimplified example, let's say that our program knows that people who are smiling tend to have pixels with higher values at a certain position. Because as we now know, in our program's eyes, higher values represent bright pixels, we can guess that this could be the result of people showing white teeth while smiling. If the program notices these high values at the correct place, it will decide

that the person is smiling

Of course this one pattern would not be enough to recognize if a person is smiling and it would not help at all with the age or gender either. That's why there are hundreds or thousands of such patterns and each gives our program an indication what it is looking at. After all these patterns have been applied they all take a vote. Some of them have much stronger votes than others but can be outvoted if a lot of weaker patterns agree. Our pattern looking for white teeth would probably be one of the more vocal voters in our pattern community, but we should be careful not to listen to it too much. What if the person is, in fact, not smiling but shouting and while shouting

they are showing teeth?

Image is in the public domain

While it is in theory possible to teach our program all these patterns manually, it would be extremely time-consuming and not a lot of fun. Why not let the computer do all this work for us instead? This is when the *Convolutional neural network (CNN)* comes into play. CNNs are used to find patterns in an image by convoluting an image over and over again (don't google it, it will be explained 😉). Does it sound complicated? It is. But we already established that we are not afraid of gross oversimplification and, thanks to Python frameworks already prepared for us by people much smarter than me, we do not really need to understand that much to get our program going.

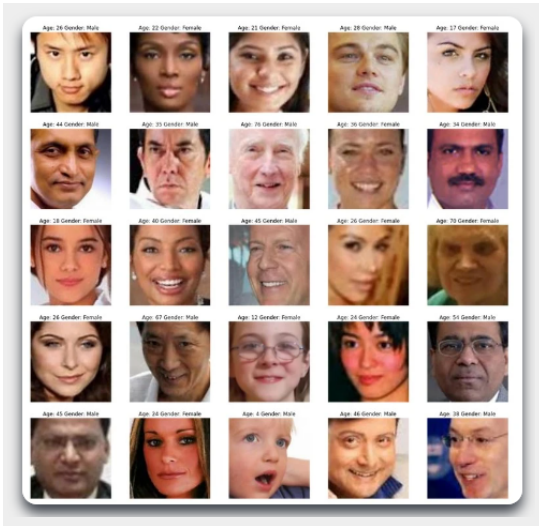
Chapter 2 - Data, data, data

So, should we use this magical CNN to teach our program these patterns now? As Kryten from Red Dwarf would put it:

"A superlative suggestion, sir. With just two minor flaws. One, we don't have any data to use while teaching. And two, we don't have any data to use while teaching. Now I realise that technically speaking that's only one flaw but I thought that it was such a big one that it was worth mentioning twice."

If our program is to teach itself, it needs data to learn from. A lot of them. It is going to go through these data hundreds of thousands of times and learn what patterns to look for. This is called *Backpropagation*. Therefore, not only do we need thousands of pictures, we also need them labeled, so that we can tell our program which is the correct answer. Luckily for us, there are a few of such databases on the internet accessible for free like the UTKFace with over

25,000 labeled pictures.

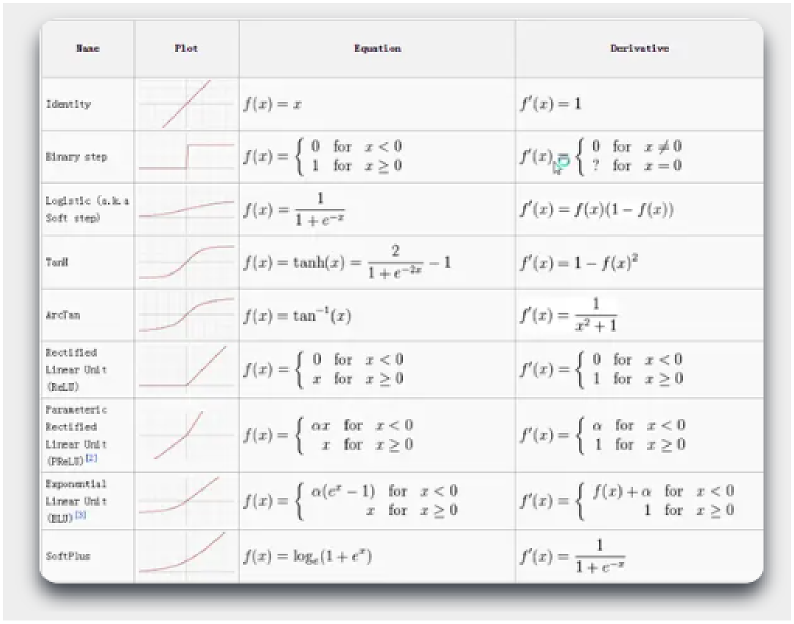


These pictures have been labeled with the age and gender, which is exactly what we need. It goes without saying that the quality of this input dataset plays a huge role in how our program will perform. Now all we need to do is to use CNN to learn from these pictures.

Chapter 3 - CNN intro

In the following few chapters we will briefly talk about Convolutional Layers, Overfitting and Optimizers. All these terms sound pretty complicated, but I promise you, we will spend no time at all talking

about this:



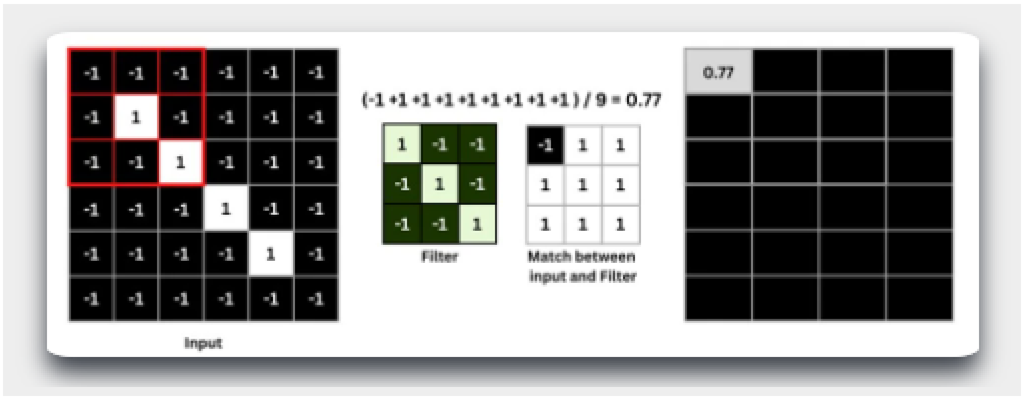
Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU)[2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU)[3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x)+\alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1+e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

And instead we will spend much more time talking about taking a shower. What has personal hygiene in common with optimizers 🤨? Stay tuned...

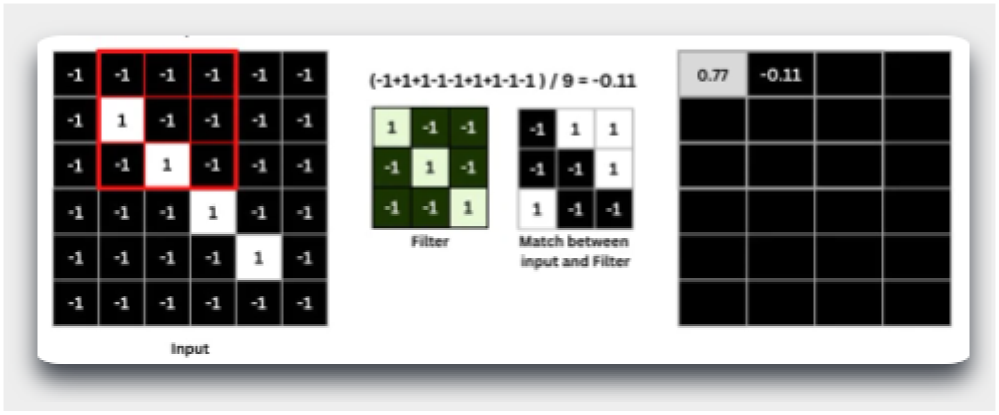
Chapter 4 - Convolution

When we speak of convolution in terms of CNN, we talk about looking for patterns in a picture. This is done through applying filters (= kernels) to a picture. These filters are usually of 3x3 pixels that represent the pattern we are looking for. The program applies this filter on each 3x3-pixel section in our input image and calculates how much it fits with the pattern we are looking for. Let's say that we want to find out if our input image has any bright diagonal lines and where they are. Then the operation would look

a little bit like this



Our program compares the filter with the selected part of our input and starts comparing pixel by pixel if the pattern matches. The higher the total match is in this 3x3 section, the higher the "score" will be. Once it is done, it moves one pixel to the right and does the exact same thing.



As you can see, unlike the first section, this second part does not match very well with our diagonal pattern we are looking for. Therefore, the result will be a negative number. We now know that the pattern we are looking for is not in this section of the picture. And it keeps going like this until it goes through all the 3x3 pixel sections in our image. Let's take a look at a real example now. Here we can see how these simple 3x3 filters can help us recognize

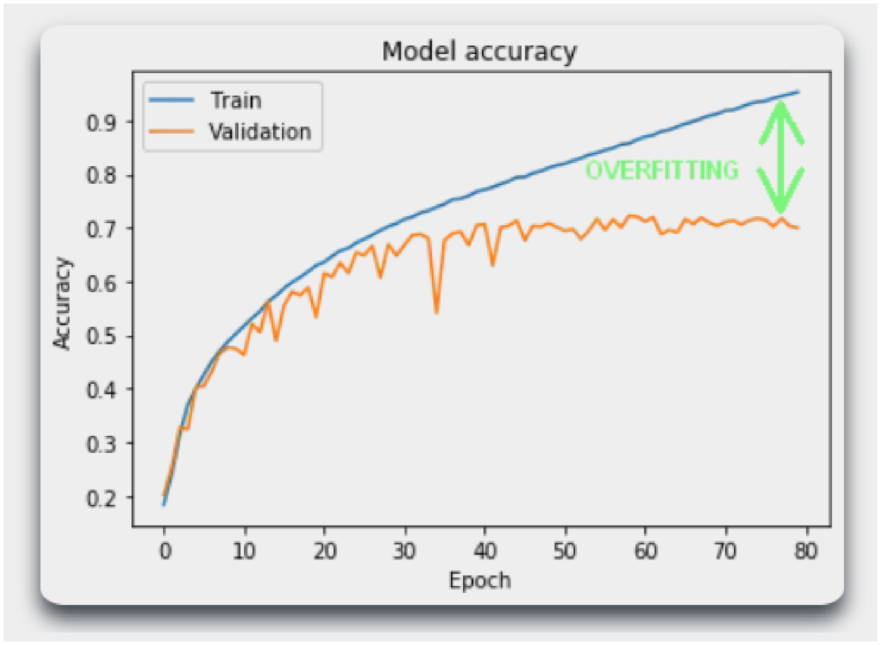
basic structures of our picture:

When FaceGuesser starts learning how to recognize what's in the picture, it will through the cca 25,000 pictures, apply literally hundreds of these filters on each picture and look for matching patterns. These filters will start very simple just looking for general outlines and edges, like in the picture above. However, as the program goes through the pictures again and again, this filtering will start to generate more specific outcomes. Congratulations, now you understand the basics of convolution in CNN.

Chapter 5 - Overfitting

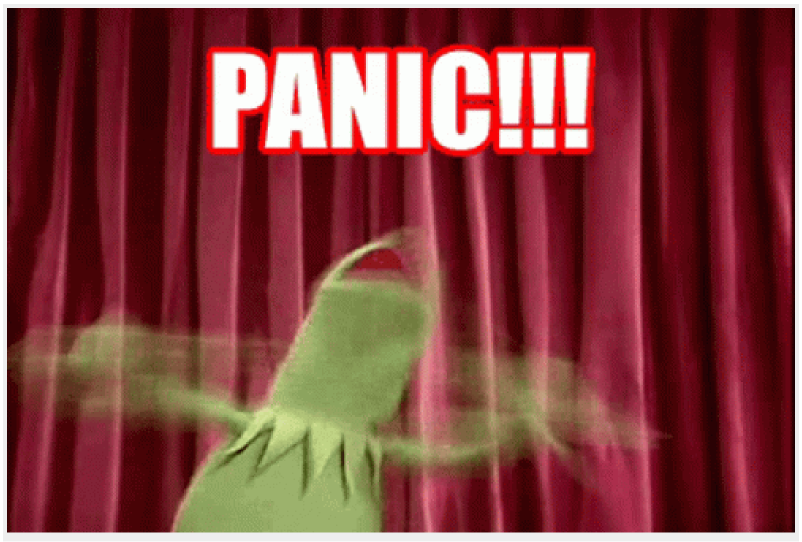
Alright, now we have the data, we understand convoluting and it is time to let our program learn. Out of the 25,000 pictures we have at our disposal, let's dedicate 80% to learning and we will use the rest randomly picked 20% to periodically test our progress. To begin with, let our program go through all of our pictures 80 times (these are called *epochs* and 80 is a bit of an overkill for our purposes, but bear with me). After each epoch, we will measure how well we are able to guess and see

our accuracy:

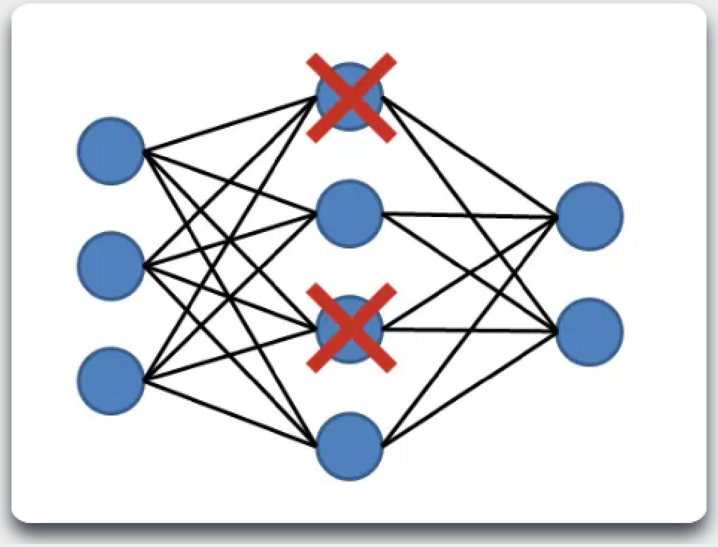


It seems that FaceGuesser is doing an amazing job with 80% of pictures dedicated to learning (blue color), but when it comes to actual accuracy testing (orange), it is not so hot. So, what is happening here? Why is the performance so much better for the testing pictures? This is a really common problem in machine learning and it is called *overfitting*. It means that our program concentrates on the wrong patterns when generating results. What does it mean exactly and is there anything we can do about it?

Or is it time to panic?

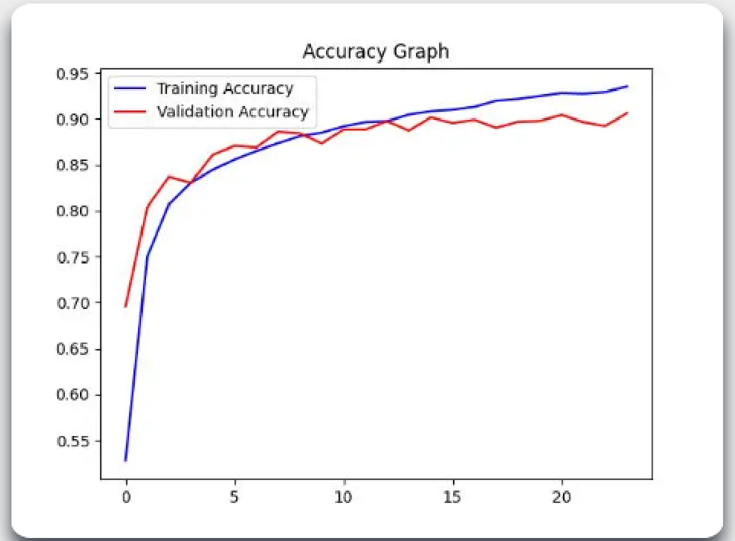


Let's start with an example. What will happen if our data are not perfect? What if, for example, in our data 80% of all people wearing glasses are women? Our program could very easily decide that based on this pattern, wearing glasses is a strong indication that the correct answer for gender should be female. Or what if only 30% of men are smiling in a picture whereas 70% of women are? Does it mean that smiling indicates gender? While this assumption is completely wrong, from a computer's point of view, it would be a very strong pattern that would affect the end result. We need to avoid this if possible. Luckily for us, we have a thing called dropout layers:



Dropout layers tell FaceGuesser to ignore certain randomly picked patterns. This forces the program not to heavily rely on a few chosen metrics and to spread the voting rights more evenly when deciding which pattern will get a say in deciding the result. This will help us get

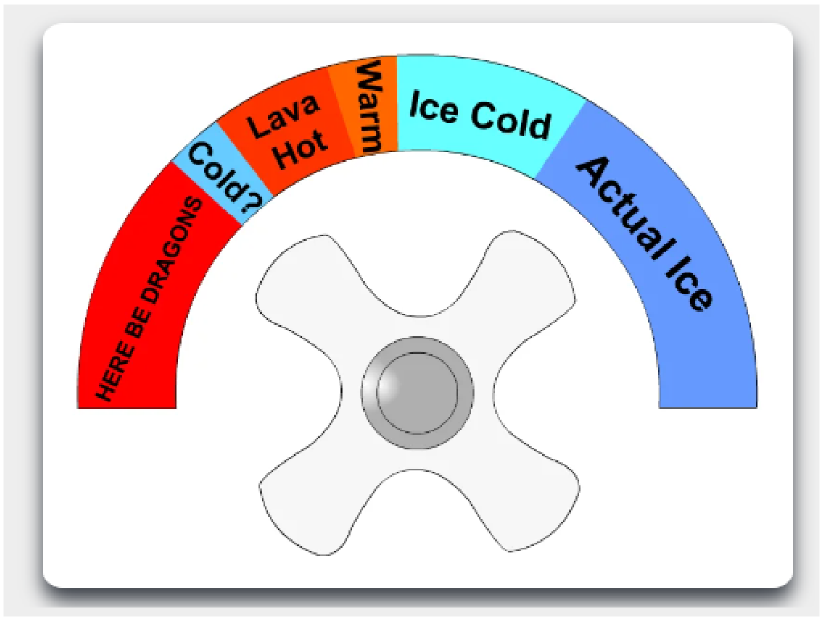
better overall results.



Chapter 6 - Optimizers

Voting has been mentioned a couple of times. It is the time when all the "patterns" get together and vote what they see in the picture. Is it a male or female? Young or old? Everybody casts their vote, but these votes have different *weights.* Now you may ask, what is the mechanism behind deciding which pattern gets to vote and how strong the vote (=weight) will be? How much should this be adjusted? And when should we stop adjusting and accept our current setting as the best possible result? This is a big topic in machine learning and there is no one correct answer. If this does not make a lot of sense at this point, worry not because we have

our shower analogy.



Let's put machine learning aside just for a moment and imagine that we are trying to find the best possible temperature for taking a shower. We define the best result as the temperature that causes the least discomfort. Or in other words, temperature that causes the biggest pleasure, but because of how machine learning works (= tries to minimize loss), let's stick with our discomfort definition. Let's also imagine that we know nothing about showers. Showers are a completely new concept to us (just like to many people at an anime convention). So, how do we find the best result? The *naive method* would be to send a few hundred or thousand people to take a shower each at a different temperature, ask them how much they hated it, write it down and get our answer by comparing the values. This would most definitely give us the right result no matter what, but

it would take forever.

A much faster method would be to send one person in the shower, record their rate of discomfort, make the shower just a little bit colder and then send another person and do the same. Did their discomfort decrease? Good, we are on the right track and can keep lowering the temperature. Did it increase? This tells us to go in the opposite direction and make the water hotter. Also, we can compare the rate at which the discomfort changed and based on that assume how much we can adjust the temperature. Now, this would be significantly faster than our previous method, but it lacks the brute force of the *naive approach* that will always get the correct answer no matter what.

Disclaimer - no animals were hurt during our shower experiment

So what could go wrong with our new method? What if our shower volunteers are very picky and it does not matter to them what the temperature is unless it is just absolutely perfect? In this case, it could seem that no matter if we adjust a little bit to the left or to the right, the temperature does not play any role in shower discomfort. Or what if the shower is broken and adjusting it to the left, which usually means warmer water, actually causes the water to get slightly colder and it would be the exact temperature we are trying to find? These are similar problems that *optimizers* try to solve in our code. The way this applies to our machine learning scenario is that an *optimizer* is working in the background and adjusts how much different patterns can vote for what they think is the right answer. It just gives a little bit more vote to pattern A and a little bit less of a vote to pattern B and observes if it improved the accuracy, or, again, to be more precise - if it decreased the loss. In theory, just like with our shower analogy, it could try out all possible combinations, but it is just significantly faster to try experimenting with the weights and see where it will get us.

Chapter 7 - Advanced CNN

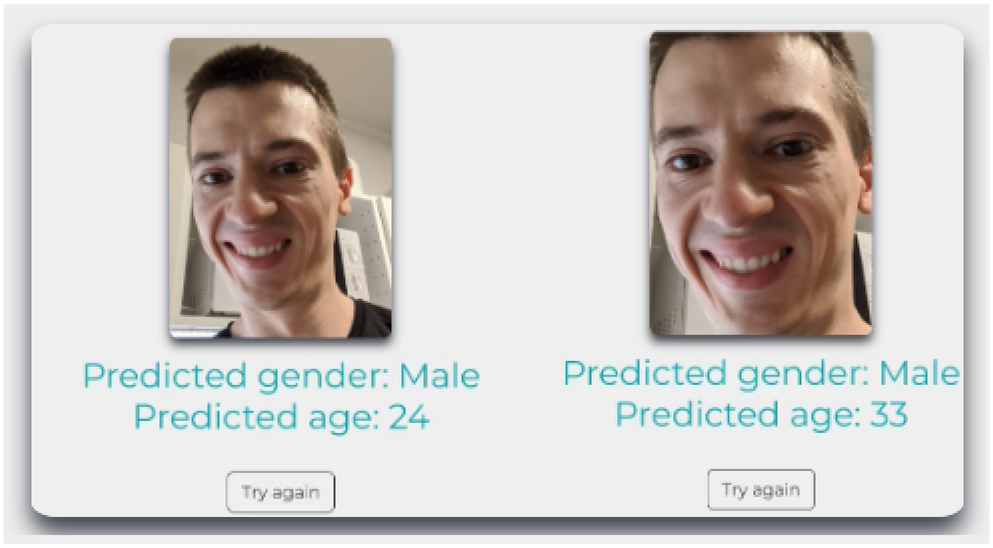
Of course there is much much more that goes into CNN. We did not even mention fully connected layers, hyperparameters, pooling layers, loss and activation functions and so on. However, I had a choice to either cover all these things or spend three paragraphs on a shower analogy and I made my decision.🤠

For more info on CNN, definitely check out this website. I promise you, it is nothing short of awesome.

Chapter 8 - Why it Sucks

Alright, so all this makes sense, but you uploaded your picture and FaceGuesser gave you a completely wrong answer. Why? First of all, I would suggest you try to crop the picture differently and try again. Make sure that it shows only your face and there is no background visible if possible. Even very small changes can bring a

very big difference in the accuracy.



Predicted gender: Male
Predicted age: 24

Predicted gender: Male
Predicted age: 33

If it still sucks, it is because my model was compiled on a home computer in a matter of a few minutes and I used only a limited number of pictures. To get the best possible results we would need a much faster computer, let it run for a few days and use millions of pictures. That being said, I hope you had fun playing with FaceGuesser and you learned something new in this article. Thanks for reading 😊