Fooling face-recognition software using a neural network

Applied Machine Learning Presentation

Sofus Stray, Kristoffer Kvist, David Dedenbach, and Elias Najarro

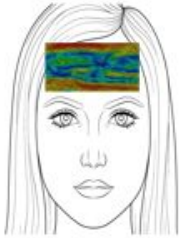
Introduction

- Trick face recognition ML using a generated image
- Further focus - Only apply to localized region of interest (ROI)
- Many different possible approaches

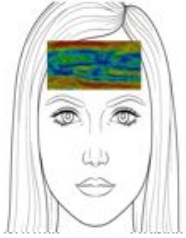
End goal - Be able to create a “sticker” somewhere on the face that can trick face-recognition

Generator

ROI selection and fill algo



Discriminator



Human/Dog image classifier

DOG!!

Fixed ROI - Basic Information

- Pre-training for discriminator: typically 50 epochs → roughly 90% accuracy
- Generator takes (100, 1) random noise array as input

Loss functions: Binary Cross entropy

- 2 different loss functions for discriminator: adding loss for fake images after pre-training
- Generator loss is discriminator confidence on human fake images

David Dedenbach

Results: Discriminator mostly wins...

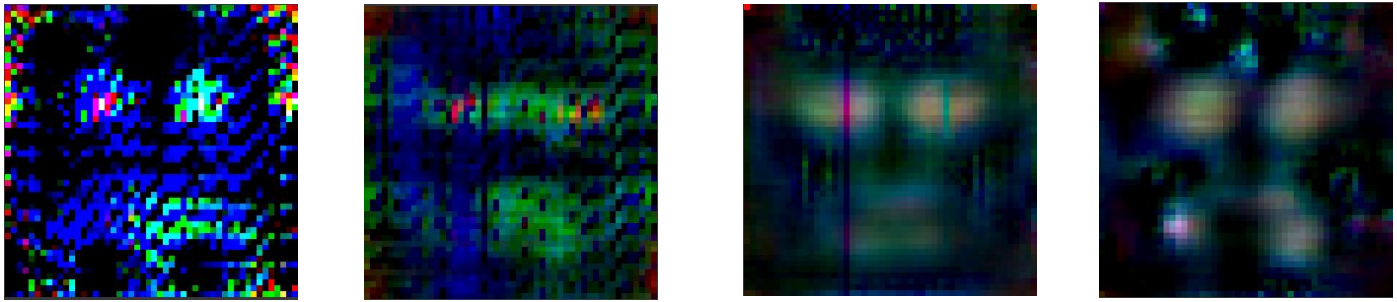
Confidence before / after pattern was applied

- 100 epochs training
- Adjust learning rates to try and find a point of even competition

David Dedenbach

Fixed ROI - Generator Progression

Epochs 2 10 50 150



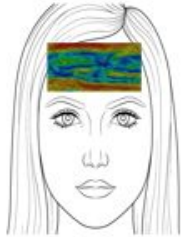
Discussion

- Good first result
- Not quite a sticker yet
- More HPO necessary

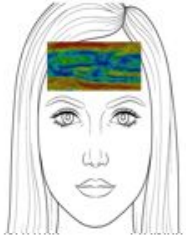
Kristoffer Kvist

Generator

ROI selection and fill algo



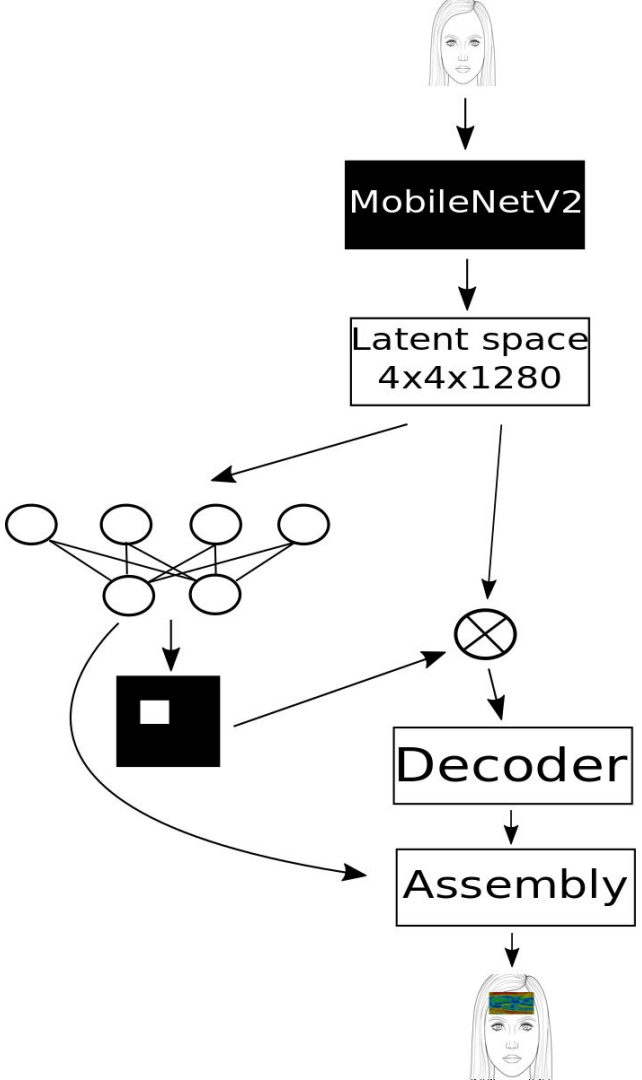
Discriminator



Human/Dog image classifier

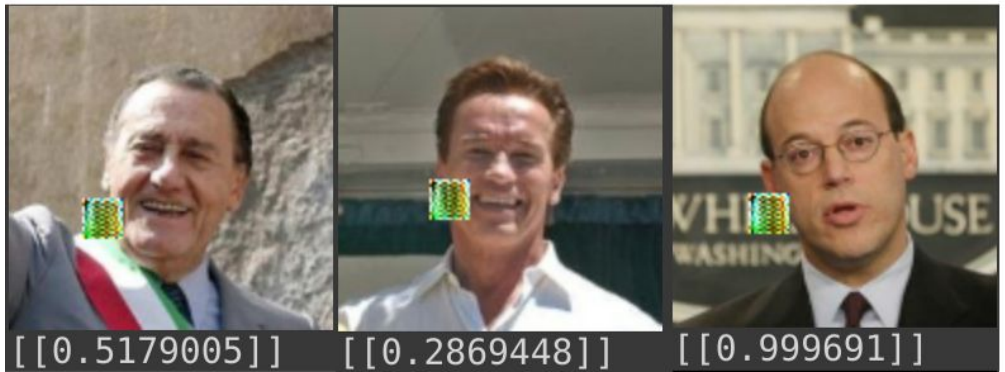
DOG!!

Deterministic approach

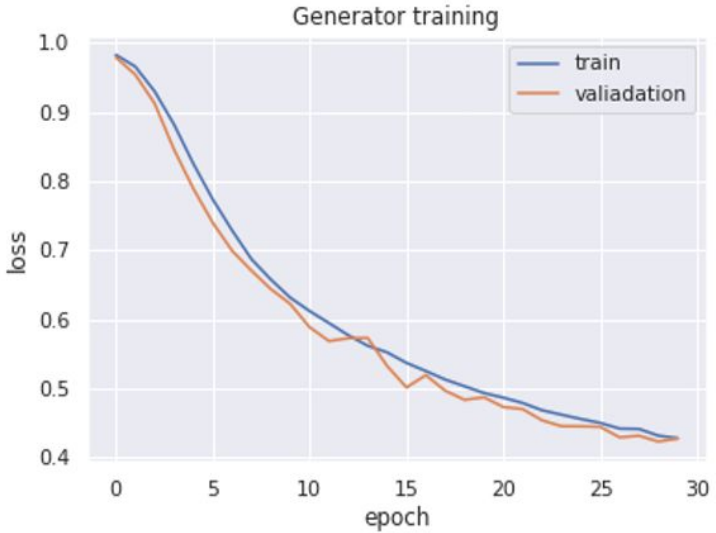


Pretraining Results

Kristoffer Kvist



A testing set of 1082 faces is augmented by the generator. The discrminator scores 37 % accuracy on the returned images. Doing the same with 1000 dog images gives an accuracy of 99.9 %.



Results after 200 epochs of adversarial training

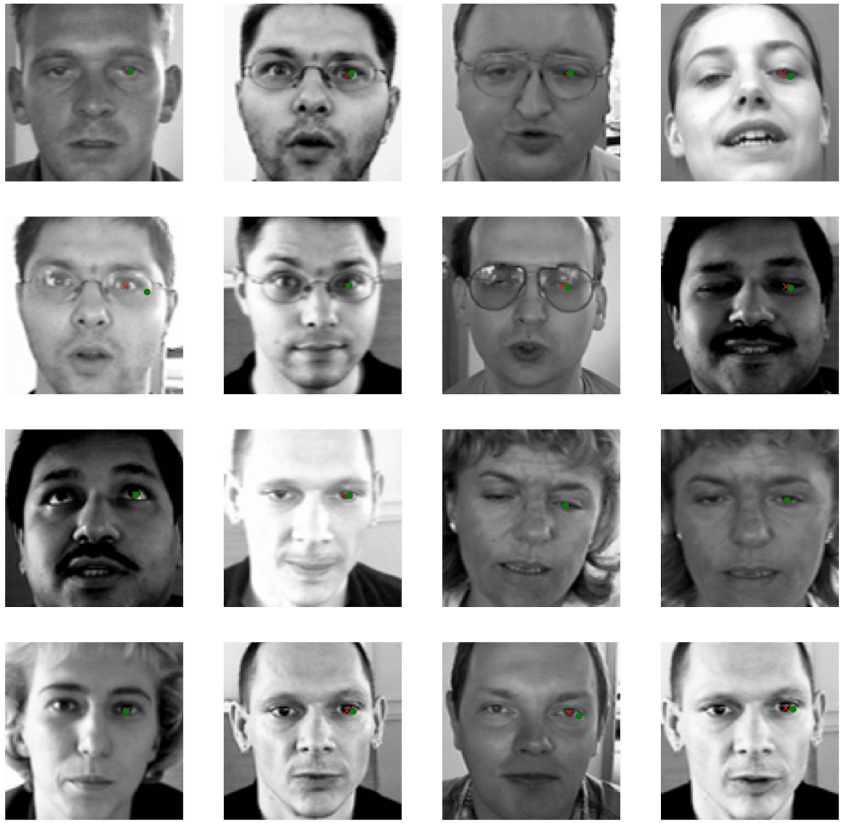
Kristoffer Kvist

Generative Adversarial Network on Regression

- Find position of left eye instead of classification of human/dog
- ~7000 Grayscale images
- Fixed ROI

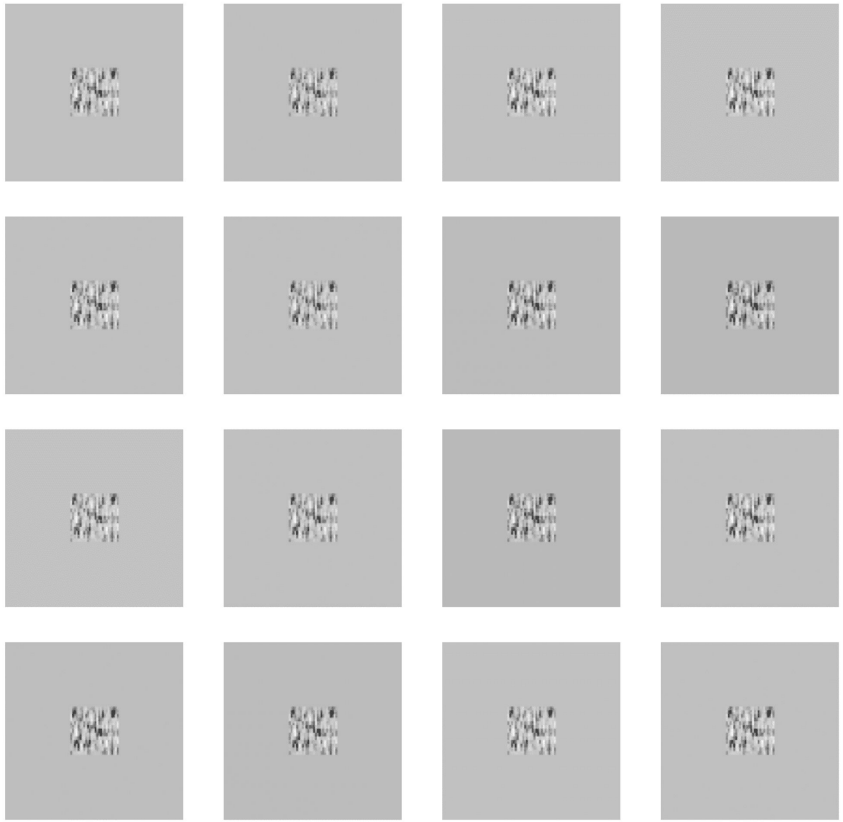
Results (discriminator wins)

- Discriminator does well
- Generator cannot advance noise-map
- Prediction on real and modified image are the same

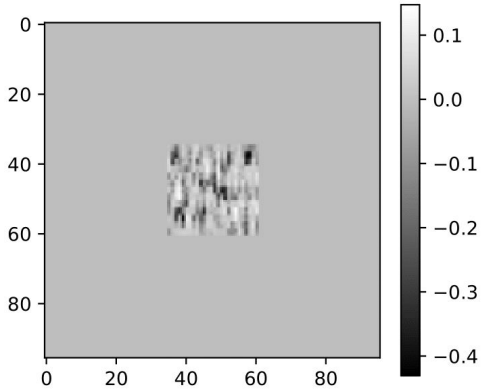


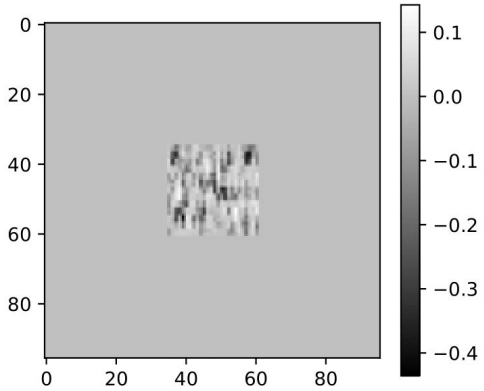
Results (discriminator wins)

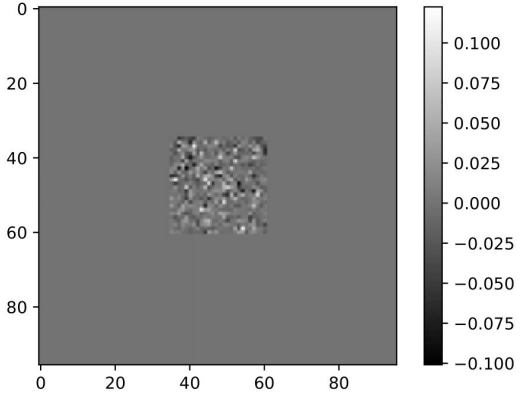
- Noise maps all the same



Results (discriminator wins)

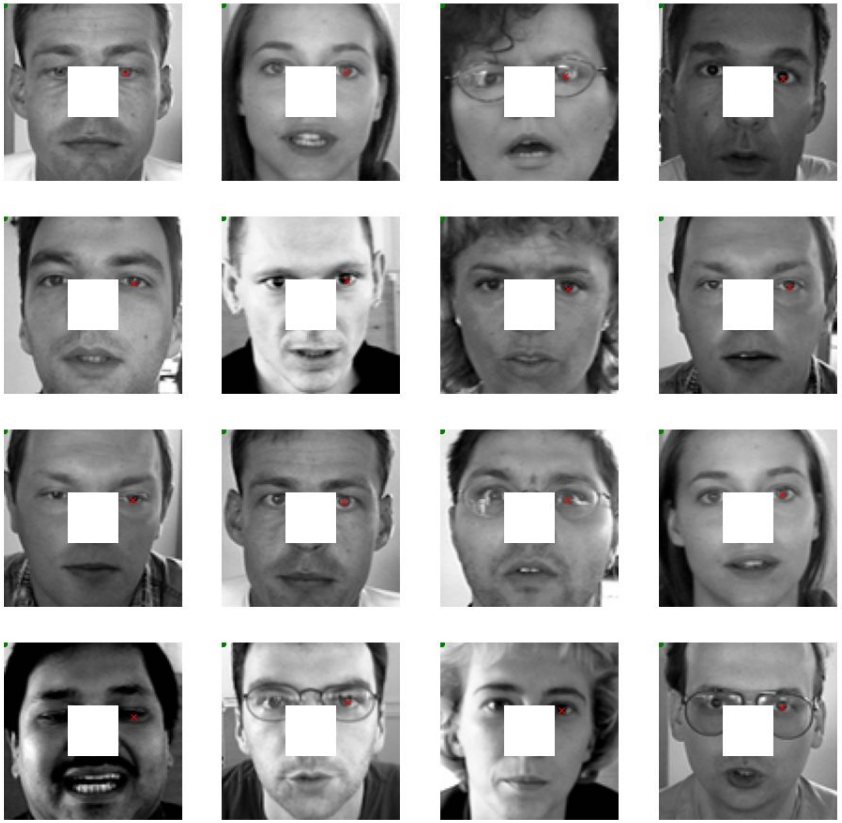






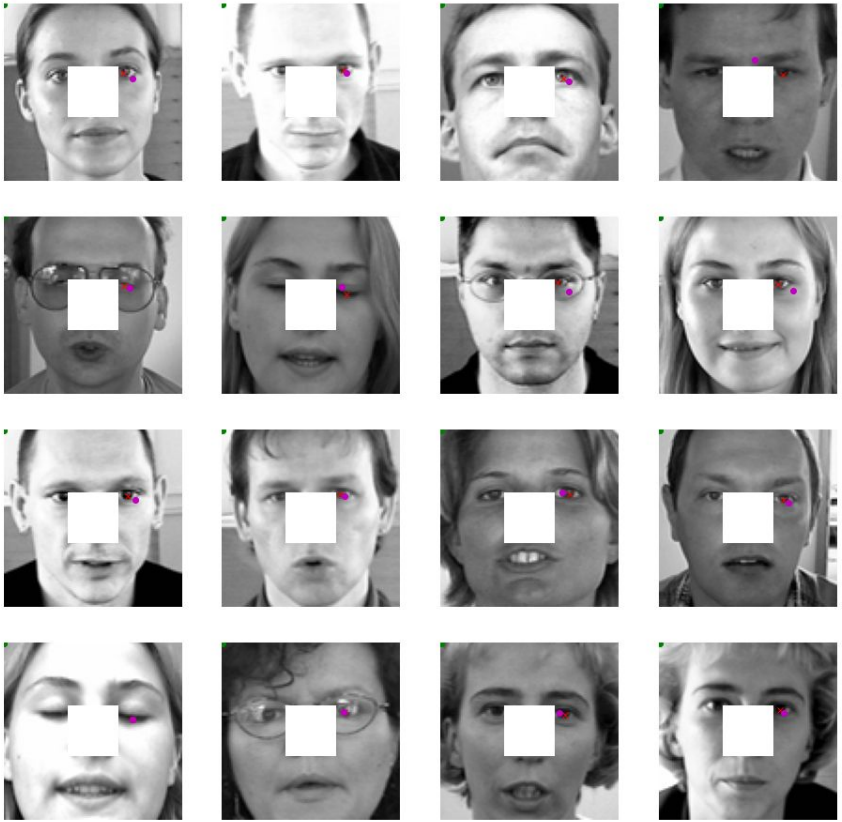
Results (generator wins)

- Noise-maps all the same
- Discriminator gets stuck (can’t predict real image either)
- Generator finds that blank slate is most efficient



Results (discriminator ignores generator)

- Noise-maps again all the same
- Discriminator correctly predicts real image, naturally fails on generated image
- Generator again finds that blank slate is most efficient



Discussion and improvements

- Both work to a degree individually
- One performs terribly if other performs too well
- Limited dataset
- Experiment with hyperparameters
- Train on more facial features and a variable ROI

Face-detector/tracking system: MTCNN

Can we generate a pattern that could be printed on a face mask to trick it?

Two metrics:

1. Reduce the confidence
2. Trick the detector position

Face-tracking are the base of **face-recognition** systems

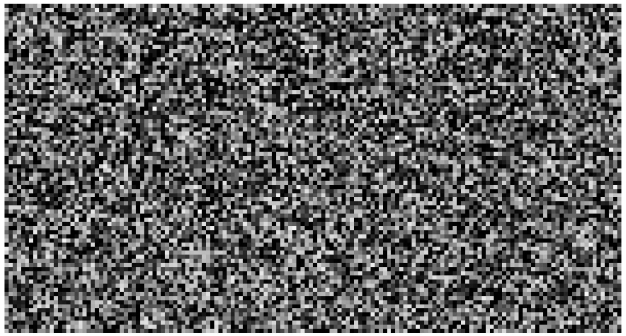
ES approach

- Generate a pattern on the face-mask area

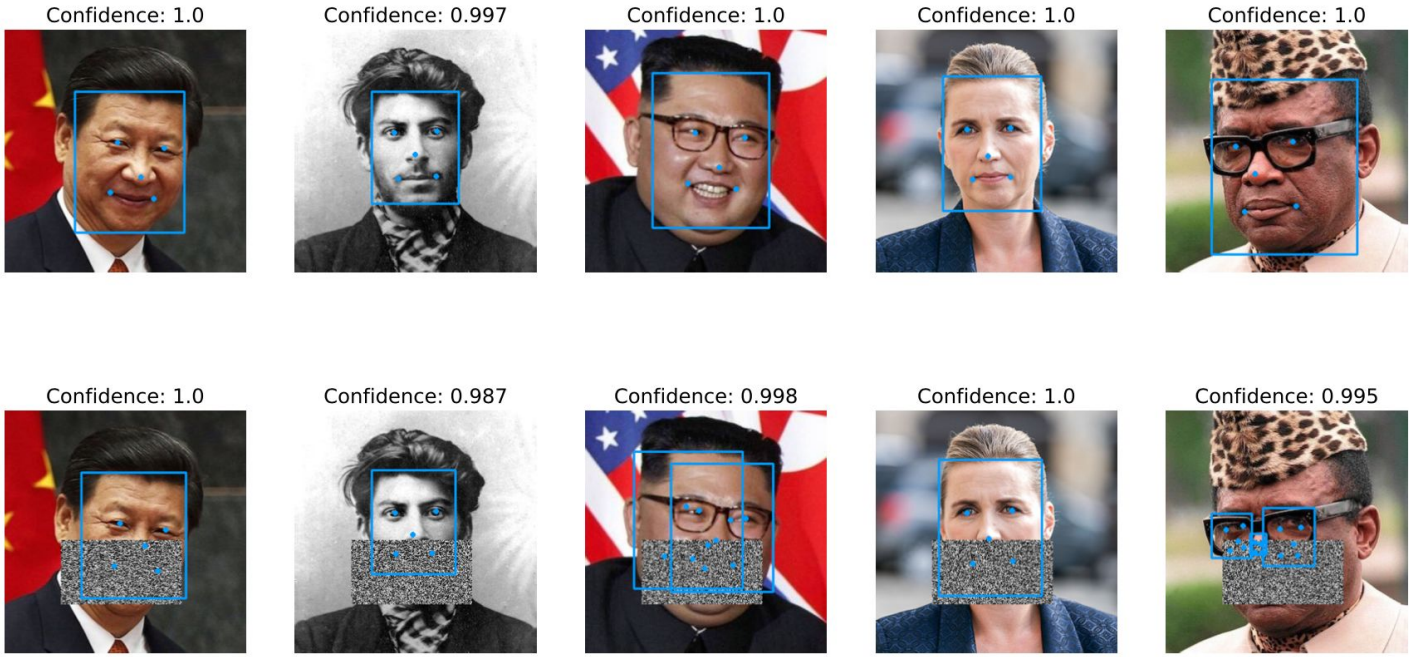
- Approach 1: optimise the pattern directly
- Approach 2: using activations of network pre-trained on faces

- We use a Evolutionary Strategy (ES) optimisation: gradient free

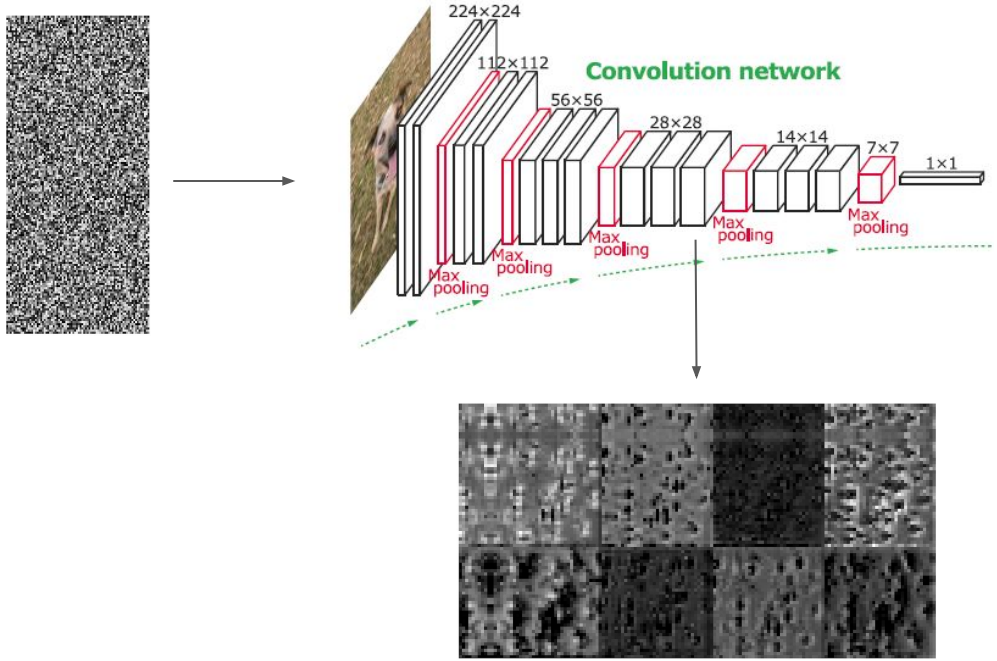
Sub-approach 1: Optimise the pattern directly



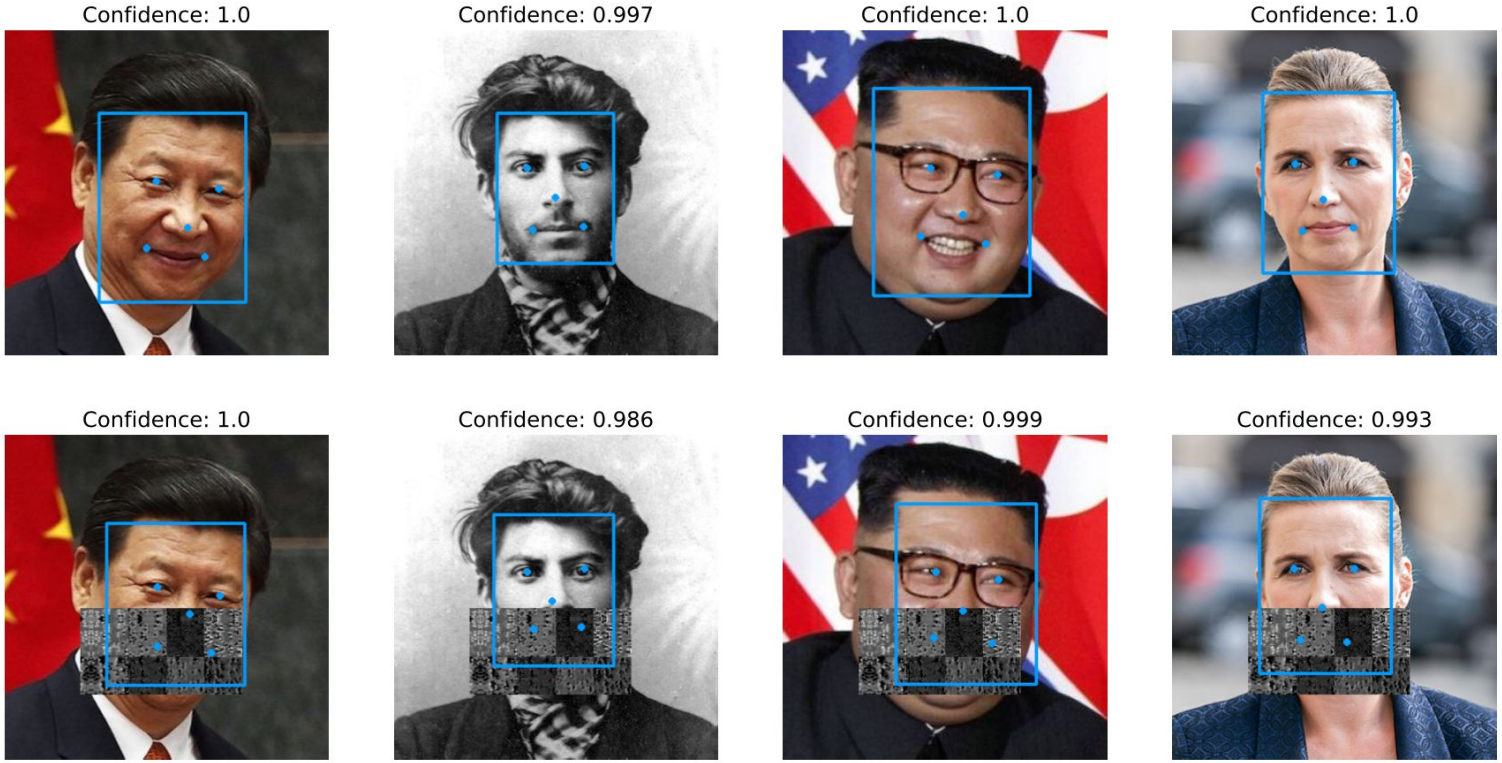
Results: Optimise the pattern directly



Sub-approach 2: Optimise the the noise fed to a pre-trained network



Results: pre-trained network activations



ES approach: Conclusions

- MTCNN face-tracking in very robust
- Pattern seem to work best when covering just below eyes
- Need more training time, convergence too slow / parameter space too big

Conclusion

Many different approaches, all with varying results

Applied ML

Sofus Stray, Kristoffer Kvist, David Dedenbach, Elias Najarro

June 2020

1 Introduction

The goal of this project was to create an algorithm capable of generating patterns of pixels that when hovering over a small part section of the face would fool a face detection algorithm into not detecting the face. We took two approaches, one based on the framework of generative adversarial networks where we training the discriminator doing the face detection as well as the generator algorithm simultaneously. And a second approach based on evolving the candidate patterns and evaluating against a state-of-the-art pre-trained face detection system.

This project is divided into four different algorithms, as all team members took a slightly different approach and we will compare the results in our 15 minute presentation.

Kristoffer started by defining a generator network with a versatile region of interest as well as a discriminator network, both of them in Tensorflow. He later translated that to a GAN. Sofus and David started on a GAN in Tensorflow, ending up with slightly different algorithms with a more fixed ROI. Elias used MTCNN[1] as the face dectection algorithm to beat, however instead of using gradient descent, he used a evolutionary strategy to train the pattern generator.

Contributions on this project have been equal but fractured. Each member mainly worked on their specific approach, but everyone participated in the group meetings and contributed equally to the overall project.

2 Adversarial Networks Based on MobileNetv2 with a Deterministic Generator for ROI Placement as well as Image Generation. - Kristoffer Kvist

All text and underlying code presented in this section is produced by Kristoffer Kvist (nmd499). As you read this, you are encouraged to look at the slides with Kristoffer Kvist written in the top right corner, as these will provide you with figures explaining the architecture of the generator and show the results from training the networks.

2.1 Discriminator

The discriminator utilizes transfer learning of the convolutional part of MobileNetv2, with the "imagenet" weights loaded through Keras. This convolutional network creates a $4 \times 4 \times 1280$ latent space,

[1] https://github.com/ipazc/mtcnn

which is then flattened and a dense layer with one neuron and sigmoid activation is put on top. The weight of the convolutional part is now frozen, and only the 2×1280 weights and biases of the dense layer is trained. The loss is a simple binary cross entropy. The training is performed on 14000 labelled images, consisting of equal amounts of dogs and human faces. Dogs are labelled by 0.0 and faces are labelled by 1.0. After 10 epochs the discriminator is able to classify the 1082 validation images with 99.7% accuracy. One could have proceeded by unfreezing some of the convolutional layers and fine tuned these with a slower learning rate. However, due to the large accuracy this was omitted.

2.2 Generator

2.2.1 Architecture

The generator designed as a modified auto encoder takes an image as input an decides where to place the sticker and subsequently what it should look like. It does not take any random noise as input as the other generators presented and is thus deterministic in the sense that supplying the same image gives the same output every time at a certain training state. It starts by creating a latent space using MobileNetv2, with the same weights as in the discriminator. Once again these weights are frozen. The values of the lantent space are flattened and given to a two sequential dense layers that output the coordinates for the sticker. These coordinates specify the region that shall be referred to by the ROI. The ROI consists of a square of hard coded size (see later section), and the dense layer thus only provides the coordinates for the upper left corner of this square. When using the output from a dense layer to create the coordinates, one must have a way of converting any set of real numbers into valid coordinates, such that these are not out of bounds, for instance. This problem was solved by applying a sigmoid activation function after each dense layer, to squeeze the output in between zero and one, these outputs are then multiplied by the respective image dimensions to get the coordinates.

Initially I tried converting the output of the dense layers using modding, which worked poorly since the loss function then became discontinuous, and then so does the gradients! In the slide showing the results of model 1, the consequences of this is shown.

From the ROI coordinates and the square size, a binary map is created. This map has the same outer dimensoins ad the original image and contains one value within the ROI, and another outside the ROI. These values are added as weights to the network. This binary map is now resized, and multiplied entry wise by each layer of the latent space, thus outputting the dimensions $4 \times 4 \times 1280$ - same as the latent space. The idea of this multiplication is to supply the latent space with the position of the ROI, such that the generated image can be adjusted accordingly. The latent space is now decoded by transverse convolutions to finally output the same shape as the input image, $128 \times 128 \times 3$. This image is then truncated by a sigmoid function, to ensure that the sticker pixels will be within the range of real images. The generated sticker is then resized, to the size of the square, and inserted in the input image, at the specified position.

2.2.2 Loss and ROI Positioning

The loss function is simply the output of the discriminator when supplied with the augmented images. The initial idea was make the generator decide the size of the ROI and then add the area, or a function of this area as a loss to the generator. But when this was implemented it only optimized the weights due to this second area loss, and did not manage to fool the discriminator at all. Even with loss functions with a very steep gradient at one for the discriminator loss, and a very flat gradient at zero for the area loss, using exponential and power functions respectively, the area of

the ROI kept converging at zero. For this reason I chose to hard code the size of the ROI. This also meant that I could now divide the ROI localazation and the generative part of the network, since it did not have the opportunity to reduce the ROI as the quality of generation got better, as I initially intended. However with the already trained network I chose not to make this division. However, to accomplish this division and focus the training on the placement of the ROI i froze all layers except the dense layers deciding the position of the ROI, during part of the adversarial training. This however did not yield a better positioning of the ROI, and by better i mean all ways being placed in the face. There can be two reasons for this. One is that the placement of the sticker is not that important for the discriminator, the other one is that the latent space simply do not hold the necessary location information to place the sticker in an intelligent way. If the second reason is correct it would also mean that the first reason is correct since the discriminator is based on MobileNetv2 as well, so if the positional information is washed out in the latent space is would not care where the sticker is placed.

2.2.3 Pretraining

The network was pretrained with 7000 facial images through 30 epochs, with a sticker of side length equal to one tenth of the respective vertical and horizontal image dimensions. After this training the discriminator recognized only 37.2% of the augmented images as faces, and the rest as dogs, which should be compared to the 99.7% accuracy it had on non-augmented images. Even in instances where the sticker did not cover the face, many augmented face images were recognized as dogs anyway. So with the weights of the discriminator frozen, it is fairly easy to fool the discriminator.

2.3 Adversarial training

The fact that the generator could so easily be trained to fool the discriminator, calls for an adversarial setup where both networks are trained simultaneously. One adversarial training step consists of a set of subsequent steps. First to batches are passed to the adversarial method, one batch containing face images and one containing dog images. The face images are now passed to the generator which creates and augmented version of the face batch. The discriminator is now called three times, one using each of the data sets i.e. one call with the original faces, one call with dogs, and one call with the augmented faces. The losses are now calculated using binary cross entropy, such that the output of the discriminator from the augmented images are compared to zeros in the generator loss and ones in discriminator loss. The discriminator loss also contains the discriminator output from the original face images compared to ones, and the the discriminator output from the dog images compared to zeros. By compared i mean calculating the cross entropy. Ones the losses are obtained the gradients are calculated and the updates are made, all handled by Tensorflow. In my implementation these updates are done simultaneously, such that both generator and discriminator are trained in each step.

2.4 Results

The final results of the adversarial training, which are shown in the slides, show that the generator does indeed manage to fool the discriminator from time to time, but also that the discriminator is able to detect some faces with large certainty. This is actually a great result showing that the adversarial approach is working as it should i.e. gradually increasing both networks simultaneously so that they keep challenging each other. The generator was not able to trick commercial face detection however, this should not be expected since I have trained my network to specifically fool a certain discriminator. On the other hand, the ease of fooling the classifier while it was not trying

to defend itself by training simultaneously, shows that most networks are probably pretty easy to attack if you target them specifically.

2.5 Further work and improvements

The most important improvement would be a more sophisticated discriminator, that would give the position of the face and its most important landmarks. The landmark locations part of this network could be used in the part of the generative network that sets the position of the ROI, to accomplish better positioning of the sticker. Most optimally some state of the art face detection software such as MTCNN should be modified to run within Tensorflow, such that it could be used in the loss function of the generator, which would enable the generator to attack this algorithm directly.

Another nice improvement would be to solve the problem of the multiple loss functions of the generator, that I described in the "Loss and ROI Positioning" section, such that the size of the ROI could be balanced with the generators ability to fool the discriminator. Setting some terminal failure rate of the discriminator one could then generate the smallest possible stickers that would still enable the generator to fool the discriminator as often as this terminal value requires.

Furthermore the architecture of the generator has not been optimized. During so automatically would be very computationally intensive unless you have significant domain knowledge when it comes to generative networks, that will allow you to set some pretty strict bounds for the optimization. Without large experience with generative networks any advanced optimization in the vast parameter space of my model seems very hard. The architecture as it stands is thus merely a product of trying to mimic success full auto encoders, and keeping the number of trainable parameters down. It has 1.1 million as it stands now. Most auto encoders use more transpose convolutional layers but this gives a lot of parameters to train i.e easily 10 million.

2.6 Learning Outcome

The majority of my work, and thereby also learning outcome, has been spend on exploring advanced functionalities of Tensorflow necessary to create the custom model. This work has made me confident in putting together almost any model using Tensorflow. However, in this process I also realized that the great performance of Tensorflow comes with the restricting rigidity that you cannot step outside the Tensorflow domain at any point in your model or loss function, as this will disable Tensorflow to acquire the relevant gradients. Furthermore discovering transfer learning has enabled me to do, especially image classification, but hopefully also many other machine learning tasks WAY more efficiently.

3 GAN with fixed ROI - David Dedenbach

I tried to set up a simple face detection algorithm to distinguish between pictures of faces and dogs. Using this algorithm as discriminator and a second neural network to manipulate a noise input of fixed size and position, I trained a simple General adversarial network.

For implementation I used custom Keras models with the gradient tape session, using the framework of a tensorflow guide [2].

[2]https://www.tensorflow.org/tutorials/generative/dcgan

Dataset choice

To have the noise layered over the face consistently, we needed a dataset of photos with faces in the middle of every photo. We chose to use a subset of the "Labeled Faces in the Wild" dataset [3] consisting of roughly 1000 images. We added equal as many images of dogs from the "Stanford Dogs" dataset[4].

3.1 Architecture

3.1.1 Discriminator

The discriminator is implemented through a custom keras model. The input flows into a 2D convolutional layer, followed by a 2D pooling layer and a LeakyReLU activation function. This is repeated a second time, before flattening the result and narrowing it down to single dense layer perceptron in several steps. I used sigmoid activation after the last layer, which gives a value in the interval [0,1] to work as a classification algorithm. This sets up the network to predict "No Face" (equal to 0) and "Face" (equal to 1).

As an optimizer for the discriminator, I started out with 'Adam', which seems to be the standard for this type of application.

I added pre-training for the discriminator where it learned to distinguish between humans and dogs. This worked pretty well. For all the results you can see in my slides, I used 50 epochs of pre-training to get the discriminator to a good level, before I let it compete with the generator. After pre-training the discriminator regularly reaches an accuracy of over 90 percent when predicting a test sample of human and dog images.

As for the learning rate, I started out on $1e-4$, in the later stages of the algorithm I adjusted the learning rates of discriminator and generator to try and manually create a more even competition between the algorithms (more about this in a later section).

3.2 Generator

As the discriminator, the generator is a custom keras model. The architecture is a little different though, beginning with a dense layer followed by two 2D-deconvolutional layers and a 2D pooling layer between those. The result is then brought into the correct format and returned. For this network, I continued using the 'Adam' optimizer after trying around with some others. I also started out with a $1e-4$ learning rate.

The generator network features region of interest fixed in position and size, which is layered over the middle of the original picture. The original pictures had a size of 120×120 pixels, the noise was layered over a 50×50 area. The input is simply an array of random floats in normal distribution.

This is of course not a very versatile generator, especially keeping in mind that the end goal was a pattern, printable as sticker, that could fool the discriminator. However it is a good basic model to get a simple GAN to work.

3.2.1 General adversarial network

To combine discriminator and generator into a GAN, I first let the generator put a noise map on each human image that was in the pre-selected batch. Then all pictures of the batch were fed to the

[3] http://vis-www.cs.umass.edu/lfw/

[4] http://vision.stanford.edu/aditya86/ImageNetDogs/

discriminator, now including the images with noise patterns. This means that the discriminator is still training on standard human and dog images, only getting a new challenge in the noise patterns.

3.3 Loss functions and hyper parameters

Loss functions

As my algorithm works with a binary classification problem, I used binary crossentropy as loss function for all losses. For the discriminator, different loss functions are needed in pre-training and GAN. The pretraining loss function adds up the losses of wrongly classifying dog images and human images. In the GAN, we additionally need the loss of the discriminator wrongly classifying images that were altered by the detector before. I kept the other two contributions and also continued feeding standard images to the discriminator, as it should not compromise its ability to classify those images.

For the generator, I simply took the confidence of the discriminator as a loss function. There are more complicated possibilities, for which I didn't have the time anymore. For example I would like to try a weak element in the loss function, which punishes the generator for changing big areas of the original pictures to see, if it can work effectively in smaller areas.

Hyper parameters

As already mentioned, I tried to use the optimizers 'Adam' and 'RMSprop' for both algorithms, starting with 'Adam'. Ir changed to 'RMSprop' on the discriminator algorithm, because it seemed to be working better for my network. In particular, it solved a problem that I had for a long time, where the networks predictions would generalize to zero, apparently it got stuck in a local optimum of the gradient.

However, at the end of the project, I can conclude in general that the choice of optimizer is not as important as other hyper parameters for this project. After figuring out the range of other hyper parameters, both optimizers ran reasonably well.

The two parameters that were more important, are batch size and the learning rates of the algorithms. I started out using a batch size of 32. In the course of the debugging for the discriminator algorithm generalizing to zero, I lowered this to 4, which made it way more stable. I gues that the high batch size caused the algorithm to generalize over too many different pictures, leading to really flat gradient descents that don't really change the weights of the algorithms.

Secondly, the learning rates are really important for two reasons. First, I ran into many local optima with the generator in the beginning. A higher learning rate resolved that. The more important feature is though that the generator and discriminator can be balanced through the learning rates. I didn't quite get to a satisfying point there, so this would need some more optimization.

My preferred learning rates evolved from $1e-4$ on both algorithms to a little lower for the discriminator ($1e-5$) and a little higher (between $1e-5$ and $5e-4$) for the generator. The learning rates are really hard to balance, if the discriminator learning rate gets too low, the generator simply stays on it's pattern, because it doesn't need to evolve. For this I would've liked to try an approach with adaptive learning rates.

3.4 Results

There were two standout pattern types that appeared as generatro outputs: One pattern that simply covers a big part of the faces, making it pretty much impossible for the discriminator to detect the face. This mostly appeared in the early stages of the algorithm, when I had the discriminator on

a very low learning rate ($1e-6$ or lower). The generator then has simply no reason to change the pattern, as it's winning. The other typical pattern shows a very dense cluster over the eyes as well as the mouth angles, making it the much more interesting one. I explored this one in greater detail and the results can be found in the slides.

Usually the discriminator won against the generator with ease - although there were cases, where the generator created some successful patterns, especially after adjusting the learning rates.

3.5 Problems and Next Steps

I had a lot of problems getting the algorithm to run. Using Tensorflow datasets as input for the networks, I had some cryptic error messages for quite a while, which were solved by importing 'tensorflow.keras' instead of just 'keras'. While costing a lot of time, this at least provided a great learning experience for me, as I was forced to read about the tensorflow and keras library in detail.

Now, presenting this algorithm with a fixed ROI wasn't my final goal. My next step would've been to try and implement Shapley values with the discriminator algorithm to see, which pixels are most important for the detection of the face. Those insights could've been used by the group to find new, more precise regions of interest, potentially even as weigths for the generator to know, where to put the noise clusters.

In general, I would've liked to make the generator algorithm more sophisticated and potentially make it work with MTCNN to compete against a state-of-the-art face recognition algorithm. Also, the existing algorithm definitely needs some more work. With more time it would've been convenient to do some Hyper Parameter Optimization especially on batch size and learning rates, but also on optimizers etc., especially to avoid having an algorithm generalizing to zero again. Implementing a bigger sample size would have been beneficial as well to avoid overfitting, although we are working in a GAN here and that shouldn't really be a problem. But combined with the enormous amount of epochs I ran (up to fifty thousand), I would like to explore a little more with bigger and more versatile datasets. The fixed ROI is not optimal, as it does not take into account at all, where the face actually is on the picture. I minimized this problem by using the deepfunneled version of the LFW dataset, which has images of faces in roughly similar positions. This would also have been a starting point for refining the algorithm, so it adjusts to the properties of each picture.

4 Generative Adversarial Networks with focus on regression - Sofus Stray

So far, we have mostly considered a binary classification scheme - either the image is of a human or a dog. This section deals with a regression problem, where the GAN algorithm tries to predict the position of a person's left eye.

The implementation uses a similar architecture to section 3, using Keras models and a gradient tape with Tensorflow[5].

4.1 Dataset

Unlike the previous sections, the dataset for regression naturally requires labels of identifying features. As such, the chosen dataset is "Facial Keypoints Detection"[6] provided by Dr. Yoshua Bengio

[5] https://www.tensorflow.org/tutorials/generative/dcgan
[6] https://www.kaggle.com/c/facial-keypoints-detection

of the University of Montreal. James Petterson. The specific dataset used is a format-modified version[7] by Omri Goldstein.

The dataset features 96x96 pixel grey-scale images of webcam-focused faces. The angle and position of these faces are near homogenous, and as such, the algorithm needed is not as complex as it would be had the position and angle of the faces been more varied. The primary reason for choosing this dataset is the available labels. Note that it also provides labels for other parts of the face, but the algortihm only trains on the left eye position for simplicity.

4.2 Approach

We follow a GAN approach, with a generator creating a noise-map that tries to fool a discriminator into misplacing the position of the eye. The discriminator attempts to find the eye position for both unmodified and generated images.

4.2.1 Generator

The generator takes a Nx100 Gaussian noise map and applies a dense layer before reshaping it into a 16x16x1 array. This reshaping is necessary, as the next step is to deconvolute it. We run two deconvolutions with a 2D max pooling layer in-between, before a final reshape that then represents the actual noise-map. The dimensions of this noise-map is 26x26 pixels. We then "average" this noise-map onto the center of the original image. This procedure requires padding he 26x26 pixel space with a zero-array until the dimensions are the same and then, following the averaging, multiplying the resulting image by 2 in order to normalize it. The result is a 96x96 image with a 26x26 noise-map in the middle.

The position of the noise-map is fixed in this algorithm. A more sophisticated approach would initially attempt to find the region of interest (ROI) and then fine-tune the map itself. This approach is more simple in order to verify whether our GAN network works on a more primitive task. There is also good reason to believe that the ROI in this algorithm would simply be straight over the eye, which is not very interesting scientifically speaking. A generator with a varying ROI would need a discriminator that looks for multiple facial features.

The averaging of the noise-map over the image is not ideal, as it does not correctly reproduce the "sticker" concept that we originally aimed for. Nevertheless, it is an approach that closely imitates the ideal and is more practical to implement. We do not expect averaging versus replacement of the image with the noise-map to yield significant changes. To see the noise-map, however, we do need to subtract the generated image from the original image.

4.2.2 Discriminator

The discriminator takes a 96x96 pixel image (either unmodified or generated) and attempts to find the (x,y) position of the eye. We do this via two convolution nodes, each followed by a leaky rectified linear unit activation function and a dropout function to prevent overfitting. We then flatten the result and feed it into a 2-dimensional dense layer that represents the (x,y) coordinates. This position is a float and not constrained to a particular pixel. The labels of the dataset match this idea. The final dense layer uses a rectified linear unit which is very useful for regression problems.

[7] https://www.kaggle.com/drgilermo/face-images-with-marked-landmark-points

4.2.3 Loss Function

The loss function for the discriminator is a simple mean-absolute-error (MAE) between the (x,y) coordinates of the prediction and the labels. It will try to reduce the error of a sum of both the MAE from the unmodified images and the generated images. The weights of this sum can be changed so that the discriminator can "focus" more on getting the real or generated images correct. In our case, we give them equal weights.

The loss function of the generator only concerns itself with the discriminator's ability to predict the generated images. As it wants the discriminator to have a high loss on these, the function is simply 1 over the MSE from the discriminator's prediction. As such, the worse/better the discriminator is, the more the generator is rewarded/punished.

4.3 Implementation

The algorithm has an issue where the discriminator will generalize to zero for both the real and generated image if the generator performed too well, and sometimes even randomly even if the generator was not training at all. The latter was fixed in part by introducing better activation functions and changing the learning rate, but the generator still trips up the gradient of the discriminator unless certain conditions apply, see below.

While the approach is simple in theory, it is more sophisticated in practice. In most GANs, the networks work together. The generator often tries to generate an image, and the discriminator helps it along by telling it where to improve. In this case, however, the discriminator wants the generator to fail and vice versa. When the GAN works like that, is it to be expected that one of the networks "win".

The biggest hurdle is then to balance out the two networks. In this specific case, one of two things can happen. The first is that the generator wins early and figures out a shape that can fool the discriminator and the latter is unable to correctly predict the left eye of the generated image. In this case, the loss function punishes the discriminator for every action, and it is even unable to predict the unmodified image positions. The second is that the discriminator win early and the generator is unable to alter the initial noise map because the gradient is zero no matter what changes are made. The ideal would naturally be that the generator finds a noise-map, the discriminator figures out how to find the left eye despite the noise-map, the generator creates a better noise-map and so on. In most cases, this does not happen.

In order to facilitate this adversarial growth, we set up pretraining. Initially, we train the discriminator only, then optionally the generator only, and then both of them in an adversarial network. The specifics here are myriad. Pretraining the discriminator on only real images results in the generator winning the second it gets started. Pretraining the discriminator on real and random noise image makes it too robust for the generator to have gradients initially. No pretraining results in the generator winning.

In practice, the pretraining is done by freezing the weights on either model at specific epochs. Changing how many epochs pass before a model is permitted/denied training can vary the result, although even a single epoch of discriminator training makes the generator lose out.

The loss-function can also be modified, as stated earlier. Depending on the factor in front of the two loss functions in the discriminator sums, it either wins or loses early. Hyperparameter optimisation could provide help, but the optimisation in this case would be a balance between either network winning, which is difficult to do in practice.

4.4 Results

Based on the discussion above, the results of various different implementations is shown in the presentation. We run with a batch size of 32 for 300 epochs. In the first case, we allow adversarial training at the start. In the second, we freeze each network for 100 epochs each in order to pretrain (generator frozen from 0 to 100, discriminator frozen from 100 - 200). We also run a third time where the discriminator ignores the generated images and only tries to predict the real ones.

In case the generator wins, the noise-map is a white square and the discriminator guesses (0,0) for the eye position. This white square representation is most likely the result of the pixel-values for the noise-map going towards infinity and there may still be noise if a more robust normalization is done. However, at the moment, the network cannot perform this normalization, and any pixel-variation is immediately smeared out when we normalize it as is.

In case the discriminator wins, the noise-map is nearly unchanged from the default noise and the discriminator guesses the exact same position for both versions.

If the discriminator ignores the generated image, then the discriminator generalizes to zero for the generated images and correctly predicts the real ones, as expected.

4.5 Discussion

It is quite clear that both the generator and discriminator work individually, and both converge towards either the correct eye position or a noise-map that heavily disrupts the guess.

The obvious goal would be a discriminator that does not generalize to zero even if it loses, and a generator that does not have zero gradients if the discriminator performs too well early.

As mentioned in the dataset section, there is a good chance that the discriminator suffers from the homogeneous nature of the images, and that a better variety would result in a more robust network that nevertheless could fail with some noise-maps. This possibility would be good to explore in future prospects. A larger and more general dataset for a more sophisticated algortihm would in general be preferred in order to generalize it better. A challenge is of course finding these datasets that also contain facial feature labels. Figuring out how to properly normalize the current white noise-maps is definitely the first step towards improvement, however.

The two networks were not developed as much as hoped. With some hyperparameter optimization, better knowledge of neural networks, and experimentation with optimisers, learning rates, and activation functions, a discriminator and generator that could be equal adversaries may have been possible to create.

A further goal for the project, given more time, would be introducing more facial features, allowing the position of the ROI to change, and steer the noise-map towards certain patterns or shapes that would be possible to create on a sticker. At the moment the two networks are very simple and fairly haphazardly built, and something more complex would be very interesting to build.

5 MTCNN - Elias Najarro

The goal of this approach was to come up with pattern generator capable of fooling MTCNN, a state-of-the-art face detection system, which takes an image and outputs the frame regions where it detects the presence of faces and the corresponding confidence of each detection. In this scenario, the trained model would generate a pattern that when covering a partial region of the face would confuse MTCNN.

In order to trick MTCNN two metrics are possible, either the presence of the adversarial pattern leads to a low confidence on the detected face position, or better, it detects the face on the wrong location but with high confidence. Both loss function are implemented in the attached code.

During evaluation the generated patterns where place in the area covering the mouth where a face mask would be place, the idea was to be able once the optimal pattern found to be able to print them on a face mask so the concept could be demonstrated on a real-time face tracking systems.

5.1 Pixel pattern optimisation

We came up with two approaches, in the first one we start off by generating a random pixel pattern, which then we optimise.

5.2 Pre-trained neural network activations

On the second approach, we use an convolutional network (InceptionResnetV1) that has been pre-trained to perform face recognition named VGGFace2 : we feed the network with a input noise vector and used the generated activations of a hidden layer as the pattern generator; the reasoning is that the patterns generated by the activations of a network that has previously learned an internal model capable of doing face recognition might be more likely to generate pattern capable of confusing MTCNN. In this last approach we optimise not the weights but the noise vector fed to the network.

Evolutionary strategies (ES)

In order to optimise both the pixel pattern and the VGGFace2[8], rather than using gradient descent we use an evolutionary strategy (ES). Following Brownlee's definition [9], "Evolution Strategies are inspired by the theory of evolution by means of natural selection. Specifically, the technique is inspired by macro-level or the species-level process of evolution (phenotype, hereditary, variation) and is not concerned with the genetic mechanisms of evolution (genome, chromosomes, genes, alleles)". That is, ES can be seen as implementing the principles of evolution disregarding the gene-based encodings and cross-over breeding mechanisms of individuals. Nevertheless, we still have a fitness function that we seek to optimise and which is responsible of creating selective pressure on our population of candidate solutions in order to lead us iteratively to a global optimal solution.

In order to be able to run faster our algorithm, we have implemented it using the *Python* multi-processing library[10], so that each CPU core is capable running a simulation in parallel. See attached code.

5.3 Loss functions and hyper parameters

Loss functions

In order to optimise our generators two different loss function where used: the first one -referred as *distance* in the code- rewards the model for generating pattern that makes the position of the detected face by MTCNN to move away from it's real position; specifically we used the left eye as reference point. The second metric -*confidence* in the code- tries to decrease the confidence with which the face is detected. The generated pattern where evaluate on a small set of hand-picked images.

[8] https://www.github.com/ox-vgg/vggface2
[9] https://github.com/clever-algorithms/CleverAlgorithms
[10] https://docs.python.org/3/library/multiprocessing.html

Training parameters

We used population of 10 generators, trained over 100 generations. The different ES parameters are the ones indicated a default parameters in the code.

Implementation

The solutions where implemented in Python, the pre-trained MTCNN runs with Tensorflow as backend, and we modified a pre-trained InceptionResnet that had been trained on the VGGFACE2 dataset so it would output not just the last layer activations but all the hidden activations of the hidden layers. As ES algorithm, we used OpenAI's implementation, introduced this paper [11], and nicely explained here[12].

5.4 Results

As presented in the slides, this approach worked in cases where the generated pattern made MTCNN believe that more faces were presented in the pattern. It did manage to decrease the confidence of the detector by some points, but not a significant amount.

5.5 Problems and Next Steps

Although the fitness of the solutions seemed to increase over generations, the limitation on computational resources was a deadend. Running MTCNN on a single 300x300 pixels images took 0.3 seconds, which made that evolving only 10 candidates solutions 100 generations took several hours on a 4-cores machine. Futhermore, the more interesting approach of using the activations of the pre-trained InceptionResnet network added extra load to the computation.

6 Conclusion

The problem of attacking face detection systems was approached in four different manners. Three approaches were using a discriminator build partially or entirely by them selves and one was using MTCNN as discriminator. Generally we had good success in fooling a pretrained discriminator of our own making, while MTCNN seemed pretty robust. We did however only apply one of the approaches to the MTCNN, due to the fact that using MTCNN within Tensorflow requires heavy modification. So we do not know if MTCNN is just insusceptible to this approach or generally really robust. Two of the approaches succeeded in producing printable stickers that could easily be tested in the real world as a further study. Even though we were able to trick some of our discriminators while these had frozen weights, the discriminators quickly got the upper hand when the adversarial training begun. This is however, quite unsurprising when you consider the large amount of trainable parameters in a generative auto encoder, compared to the number of trainable parameters in a convolutional classification algorithm, with the parameters of the convolutional part frozen, and only the dense "top" training, which is still very effective. This concludes that fooling a specific face detection algorithm might not be so hard as long as this algorithm remains stationary. If however the face detection system starts training on your generated images to defend itself, it quickly learns to detect the augmented faces, despite the attackers efforts.

[11] https://arxiv.org/abs/1703.03864
[12] https://openai.com/blog/evolution-strategies/