Project SSY098
VAE- Variational Autoencoder

Robin Nilsson
Chalmers University of Technology
Gothernburg
robn@student.chalmers.se

Abstract/Introduction

A common problem within computer vision and training machine learning algorithms for recognition is the size of the training set, which tends to be too small. One way to battle this problem would be to simply expand the training set by e.g. taking more picture. But this, if even possible which it doesn't necessarily have to be, is a time and cost consuming task many which to avoid. Another method which in the last years has grown in interest is to create an algorithm which automatically creates new (fake) images which are as real looking as possible. For this to work the algorithm should at least 1) create new training data which are realistic and 2) not be a replica of already existing images since we then could simply reuse that data.

In this project I attempt at creating a Variational AutoEncoder (VAE) to reconstruct digits from 0-9 using the MNSIT dataset. A VAE is a dual network with one network which is tough to compress images to only their most important features, and one which is trained to sample from the compressed data and create new realistic images. By also compressing the data such that it is as close to a standard normal distribution as possible, the sampling is then simply from $N(0, I)$.

Four different networks were implemented and both the ADAM and SGDM update schedules where implemented, although SGDM returned poor result so the results were left out. Theory of the VAE were based on the original VAE paper, [2], and a tutorial paper [1] both recommended in the assignment paper. Additionally three theoretical questions were answered in section 3. At the end I will also test the VAE using a classifier net with training images both from the MNIST data set and images created from the VAE.

Contents

1. Method

$$\underset{Input\ probability\ dist.}{p_I(x)} \xrightarrow[Encoder\ p_\theta(z|x)]{} \underset{latent\ space}{p_\theta(z)} \xrightarrow[Decoder\ p_\theta(x|z)]{} \underset{Output\ probability\ dist.}{p_O(x)}$$

Schema 1: VAE schema. Here p's are probability distributions with parameters θ.

In Schema 1 the VAE schema is outlined. We would in the optimal case have that $p_I(x) \approx p_O(x)$, since we wish to have the property that sampling from $p_I(x)$ would be the same as sampling from $p_O(x)$. This property would ensure the newly created images $\hat{x} \sim p_O(\hat{x})$ would resemble the

original images $x \sim p_I(x)$ given that the $\hat{x}$ are constructed as explained above.
But we have a problem here; we know $p_I(x)$ and we want $p_O(x)$ to be as above, so basically it is also known. But everything in between is unknown, things which we wish to estimate. Let's start with doing some simplifications/assumptions; first let $q(z|x) = N(\mu_z, \sigma_z|x) \approx p_\theta(z|x), p_\theta(z) = N(0, I)$ and $p_\theta(x|z) = N(f(z), \sigma^2 I|z)$ be multivariate normal distributions (MND). f here is the decoder, taking sampled values from the latent space and returns an image. We can make these assumptions since take $p_\theta(z)$ for example, letting it be a standard multivariate normal distribution makes no difference since we then can simply choose an appropriate decoder to match the distribution. Here we also approximated the encoder with $q(z|x)$ since the encoder in general is intractable, which we can see from the denominator in Bayes theorem.

By letting the encoder and decoder be neural networks with weights and biases $\{\mathbf{W}_i^e, b_i^e\}_{i=1,..,n}$ and $\{\mathbf{W}_i^d, b_i^d\}_{i=1,..,m}$ and assuming $q(z)$ to be normal distributed $N(\mu_z, \sigma_z^2)$, we write the above procedure as the following schema,

$$
\begin{array}{ccc}
Input\ sample & & \text{latent space} \\
x & \xrightarrow[\{\mathbf{W}_i^e, b_i^e\}_{i=1,..,n}]{} & \begin{array}{c} \mu_z = \mathbf{W}_n^e h + b_n^e \\ log\sigma_z^2 = \mathbf{W}_{n-1}^e h + b_{n-1}^e \\ z = \mu_z + \sigma_z * \epsilon, \ \epsilon \sim N(0, I) \end{array} \\
 & & Output\ reconstr.data \\
 & \xrightarrow[\{\mathbf{W}_i^d, b_i^d\}_{i=1,..,m}]{} & f(z)
\end{array}
$$

Schema 2: VAE schema in practice. Here $\{\mathbf{W}_i^e, b_i^e\}_{i=1,..,n}$ and $\{\mathbf{W}_i^d, b_i^d\}_{i=1,..,m}$ represents the weights and biases for the encoder and decoder networks respectively. Here the two last layers of the encoder are not dependent on each other.

h here is the encoder's last neurons before being fully connected to u_z and $log\sigma_z^2$ as described above. Both u_z and σ_z are vectors both of the same length $latent\_dim$. As can be seen from the schema is that we directly use the values from the output of the encoder when reconstructing $(N(\mu_z, \sigma_z^2))$, but for creating new images we would want to sample from a standard MND. The weights and biases are thus (partly) updated by using the following loss function (per input data/image),

$$L_i = \underbrace{-D_{KL}(q(z|x)||p_\theta(z))}_{=\frac{1}{2}\sum(1+log(\sigma_z^2)-\mu_z^2-\sigma_z^2)} + \underbrace{\frac{1}{L}\sum log(p_\theta(x|z))}_{=-C||x-f(z)||^2}. \quad (1)$$

and for a minibatch with M input datas we simply use $L_M = \frac{1}{M}\sum L_i$ as the loss described by [2] and [1] combined. The first term of equation 2 is the KL divergence of the approximate and the latent space distribution measuring how different they, as distributions, are. We see thus by minimizing the first term we have approximately a standard MND[1]. The second term hasn't so much to do about the distributions, but simply is a measure how well we reconstructed the image as seen by that the term results in the norm of the difference of x and f(z) times a constant $C = log(\frac{1}{\sqrt{(2\pi)^k|\sigma|}}\sigma^{-1})$ (see the code for derivation).

The MNSIT data set contains images of digits of size [28 28 1] which after the encoder should be compressed to the size of 2*$latent\_dim$, one vector for μ_z and one for $log\ \sigma_z^2$. The decoder takes a [1 1 $latent\_dim$] input size and should produce a [28 28 1] image ($f(z)$) using transposed convolution. What to consider when designing the networks in between follows the same principles as *e.g.* normal convolutional networks; More weights will make the networks more robust (until overfitting) but is slower in general. 4 different networks were implemented, and the specifications and layouts of the networks are available in the appendix A.

2. Experimental Evaluation

To train the networks mainly 20 000 of the 60 000 images were used, since using more results in long execution times and with 20 000 train images the results were decent. Similarly the number of epochs was chosen to be 40, the minibatch size to 400, and per default the 'adamupdate' function was used to update the weights and biases although the 'sgdmupdate' function (with momentum 0.9) was also implemented. The initial learning rate was set to 0.005 and for the SGDM algorithm it was updated using a time based schedule, while for the adam algorithm it was set as a constant value. As specified in the assignment description the latent dimension was set to 2, which results in the last layer of the encoder being of size 4 (two for the mean, and two for the log variance).

2.1. VAE results

In figure (1) the z distribution of the test images using network 1 is displayed. Digits 4, 3, and 2 are almost entirely 'covered' by another digit. This means that when reconstructing digits or creating new ones, these digits will probably be difficult to fully attain, and might include features of other digits as we will see later. The final ELBO loss was 21.9 for the test images.

In figure (2) the z distribution of the test images is displayed after training the VAE with network 2. As can be seen

[1] We assume $p_\theta(z) = N(0, I)$, thus we compare the output distribution of the encoder q(z | x) with a standard MND.

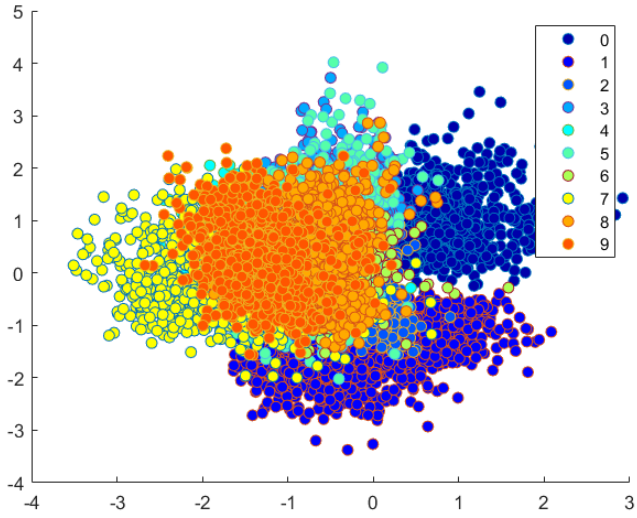

Figure 1: *Distribution of $z = \mu + \epsilon * \sigma$ for the test image batch, using network 1 with 40 epochs. The network was trained with 20 000 training images and the learn rate was set to 0.005.*

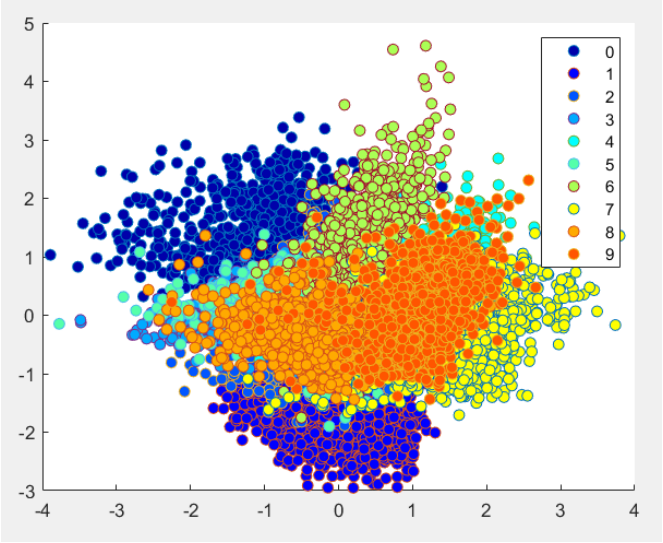

Figure 2: *Distribution of $z = \mu + \epsilon * \sigma$ for the test image batch, using network 2 with 40 epochs. The network was trained with 20 000 training images and the learn rate was set to 0.005.*

from figure (2) most digits seem to be separable for the VAE, although some seems to be almost inseparable, like 4 and 9. Inspecting the training data, many 4's were similar looking to 9's and vice versa, which could explain the result. The final ELBO loss was 21.5 for the test images.

Figure (3) displays the result using network 3 and the same setup as before. Here we see a clear diversion and that the network fails completely in separating the digits. The reason for this was a really slow convergence, or that the

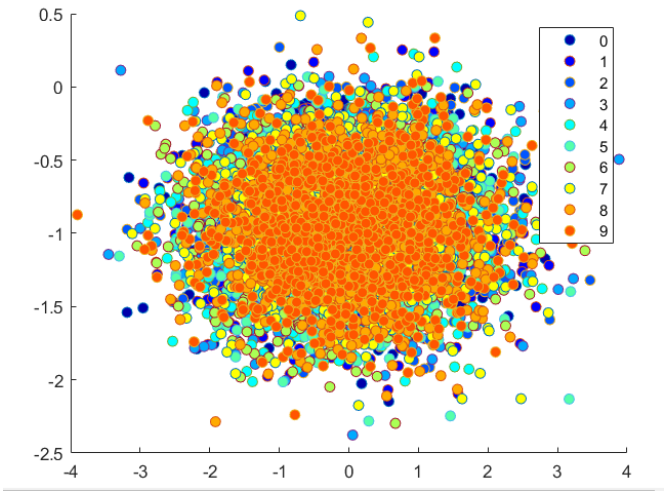

Figure 3: *Distribution of $z = \mu + \epsilon * \sigma$ for the test image batch, using network 3 with 40 epochs. The network was trained with 20 000 training images and the learn rate was set to 0.005.*

algorithm got stuck in a local minimum. While networks usually converged with an ELBO loss of 21-22, network 3 almost completely stopped at an ELBO loss of 32.

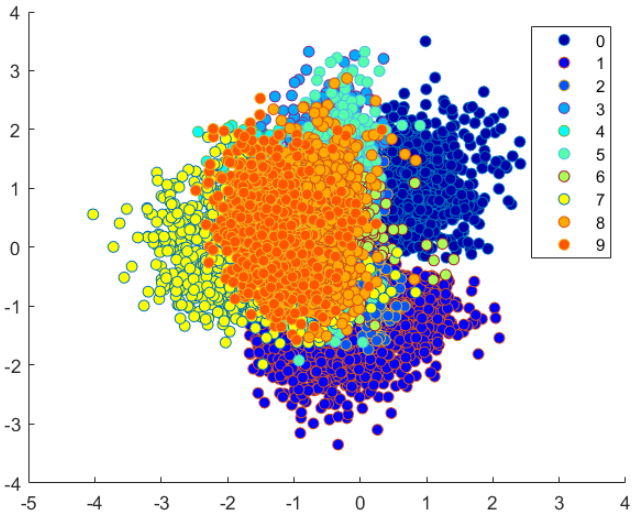

Figure 4: *Distribution of $z = \mu + \epsilon * \sigma$ for the test image batch, using network 4 with 40 epochs. The network was trained with 20 000 training images and the learn rate was set to 0.005.*

In figure (4) the result from training the VAE with network 4 and the above mentioned setup is displayed. The result is almost the same as for network 2, with an ELBO loss of 22.

In figure (5) the result of sampling directly from a standard MND three times which is then passed through the decoder

(i.e. fully synthetic images) is displayed. I would assume the middle to be a 6, the right a 4 and the left somewhere between an 8 and 9. In the right image we notice how the 4 is almost turning to a 9, displaying the issue mentioned above. This can also be seen from figure (9) and (10) in appendix B, where both reconstructed 9's and 4's have blurry tops. We also see that the network tries to reconstruct compressed 2's as 6's.

Network 1, 2 and 4 gave similar results, and one of them barely performed better then the others. Judging purely by the ELBO loss network 2 performed best, but yet again, barely. Network 4 seems to have the least executing time despite being larger, and ran for about 20 minutes compared to network 1, 30 minutes and network 2, 35 minutes.

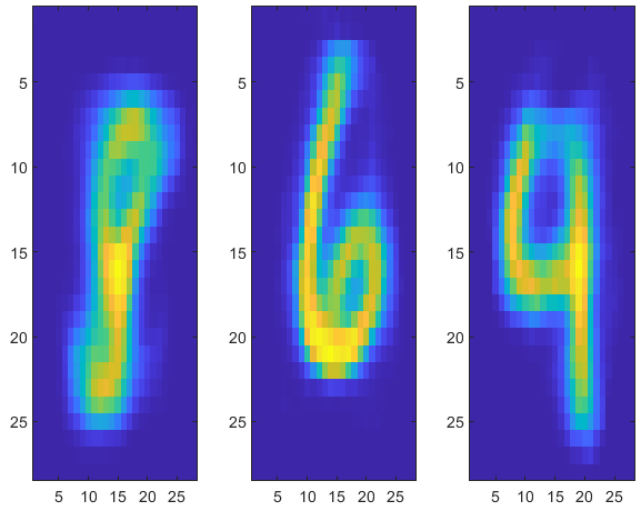

Figure 5: *Synthetic images with z directly sampled from $N(0, I)$ using the 2nd trained network, number of training images 20 000, number of epochs 40 and learn rate set to 0.005.*

Lastly, in figure (6) network 4 was executed for 80 epochs with a 60 000 images large training set. As expected the simulation resulted in the digits more precisely were separated in the z distribution. This also resulted in the reconstructed images being more similar to the real images, as seen in figure (11). Though this might just be an example of overfitting and divergence from the $N(0, I)$ distribution we wish to attain, and taking samples from $z \sim N(0, I)$ results in no digits, see figure (7).

2.2. Synthetic images for classification (advanced part)

To this end we will train a classifier for the digits, and see if using synthetic images when training the classifier can improve the accuracy. We do this by first finding the

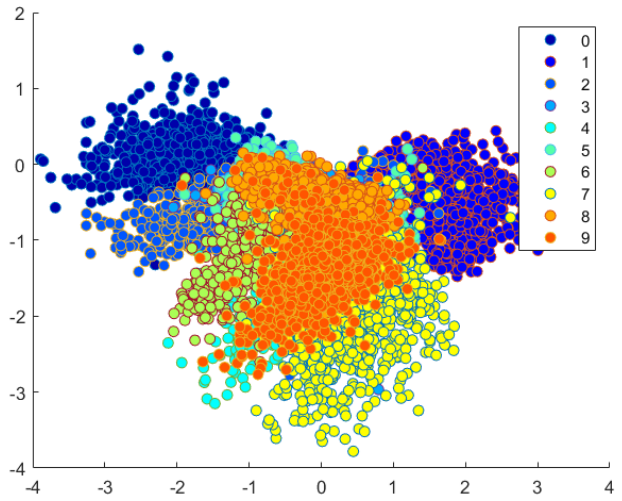

Figure 6: *Distribution of $z = \mu + \epsilon * \sigma$ for the test image batch, using network 4 with 80 epochs. The network was trained with 60 000 training images and the learn rate was set to 0.0005.*

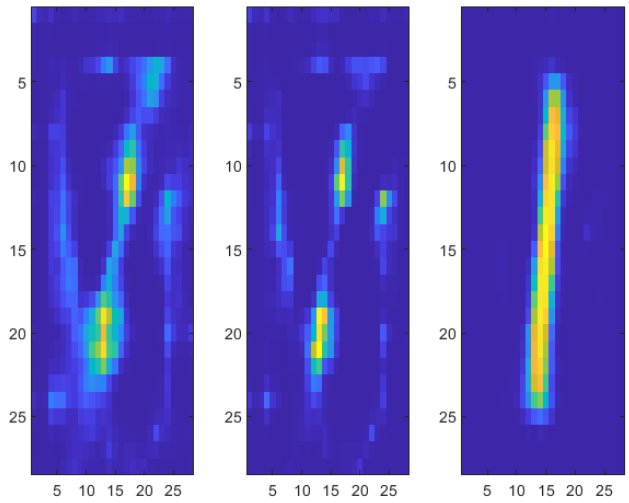

Figure 7: *Synthetic images with z directly sampled from $N(0, I)$ using the 4th trained network, number of training images 60 000, number of epochs 80 and learn rate set to 0.0005.*

mean x,y positions of the z distribution for each digit, and then sample $z \sim mean(digit) + N(0, 0.1 * I)^2$ which we feed into the decoder. As such we will have new pairs (image, label)'s. This might impose some limitations on the synthetic images set though and if the digits z distribution

[2]This is only one way to sort of automate the labeling and reconstruction of the digits. As specified in the text it might be a limited version. Another possible way would be to manually pick out the areas for each digit, and yet another could be to classify it using cross entropy and thresholds. A third way could be to compress real images, add Little noise to them and then feed them to the decoder.

representation are not separated enough it might be difficult to label the synthetic images correctly, or get nonsensical images. Using only the mean might also give lack of diversity in the synthetic images for each digit.

As described in the assignment description the VAE will be trained on half the training images, and for the other half we train the classifier using 1) 20 % of the images + equally many synthetic images 2) 50 % and 3) 100 %.

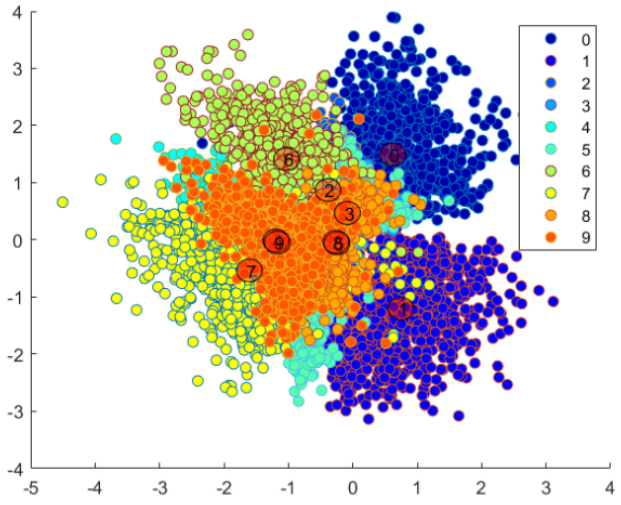

Figure 8: *Distribution of $z = \mu + \epsilon * \sigma$ for the test image batch, using network 4 with 80 epochs. The network was trained with 30 000 training images and the learn rate was set to 0.005. The circles with numbers indicates where the mean x and y position for respectively digit is located.*

In figure (8, see appendix) the z distribution for the trained network is displayed. The result is not perfect; some digits are almost fully covered by other digits. For instance the mean position values of 4 and 9 are almost identical. In figure (12) the training of the classifier is displayed, without using any synthetic images, and will be used for comparison. Without using synthetic images we reached an accuracy of 97.24 %. Considering the result of the VAE it will probably be hard to get better results or even similar.

For 1), 2) and 3) we got the results 96.96 %, 96.82 %, 97.00 % respectively. Thus we didn't get better performance, but instead worse, likely due to the reasons mentioned above. Figure (16) endorses this statement. In figure (16) I instead used the trained network where the full training set was used, and also trained over more epochs. The results were 98.41 % accuracy on the test set, thus an improvement. The main difference between the z distributions were that the digits were more separated and the means further apart.

3. Theoretical Part

3.1. ELBO loss

Since the problem is quite specific, I think explaining with my own words is rather difficult but I will give it a try.
Lets start from the expression derived in [2],

$$log(p_\theta(x^{(i)})) = D_{KL}(q(z|x) \,||\, p_\theta(z|x)) + ELBO\_loss. \tag{2}$$

The only difference from the practically used loss function (ELBO_loss) is hence the first term of the RHS, measuring how close $q(z|x)$ is to $p_\theta(z|x)$ as probabilistic distributions. The problem here is $p_\theta(z|x)$ as mentioned before, due to it being (mostly) intractable. This can be seen by using Bayes theorem,

$$p_\theta(z|x) = \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(x)} = \frac{p_\theta(x|z)p_\theta(z)}{\int p_\theta(x|y)p_\theta(y)dy}. \tag{3}$$

Here the problem lies in the denominator; the integral. Integrating a conditional distribution can be very difficult, since they often are very complex. A way to go around this issue would be to use variational inference methods to approximate it, and that is what we do when implementing $q(z|x)$. This means on the other hand that we can't really calculate the KL divergence term. But we do want $q(z|x) \approx p_\theta(z|x)$, and if we assume this to hold and write out the KL divergence term,

$$D_{KL}(q(z|x) \,||\, p_\theta(z|x)) = \sum q(z|x) log\left(\frac{q(z|x)}{p_\theta(z|x)}\right)$$

we have that it should approach 0 due to log(1) = 0. We also have the property that a KL divergence always is positive, so all in all this means that we wish to minimize the KL divergence (first term). Or, if we instead fix the $log(p_\theta(x))$ term we should maximize the ELBO loss, since then that would equally mean that we are minimizing the first term. And since we can't do the former we are left to the maximizing. And yet again we wish $p_\theta(x)$ to be close to the input distribution, which is why we in theory can fix it.

3.2. VAE-GAN

Schema 3 displays the main idea of the VAE-GAN algorithm for the forward propagation in the training loop. Here we have three networks to train; the encoder, decoder and the discriminator. We can see how the VAE and the GAN can be constructed from combinations of the networks; (encoder, decoder) = VAE, (decoder, discriminator) = GAN. We pass an image through the VAE and use the reconstructed image, the input image and a fully synthetic image as input to the discriminator. Thus we can train the encoder, decoder and discriminator both jointly and as

GAN:

$\hat{x} = f(\hat{z}), \hat{z} \sim noise$

$\underset{x}{Real\ samples} \xrightarrow[Discr.]{\text{Discr.} \searrow}$ synthetic / real image

VAE-GAN:

VAE

$\underset{x}{Input\ sample} \xrightarrow[Discr.]{\text{Discr.} \searrow}$ synthetic / real image

Schema 3: Main idea of the VAE-GAN algorithm; exchange the random noise sampling with the VAE.

they normally would be trained (with some slight modifications). Here for instance we can use the discriminator to determine whether the VAE can reconstruct images on a realistic level or not, since if it can, it should also be able to create synthetic realistic images fooling the discriminator.

By using the reconstructed image, passing it through the discriminator with the real image we can train the discriminator and decoder as a normal GAN. Thus we enable the realistic reconstruction element in the training. By encoding and decoding using the VAE we enable the algorithm to decompose the important elements of the image and reconstruct it, thus we can with a trained network reconstruct images with variations. Combining these elements we have an algorithm that can create realistic synthetic images just from sampling a standard normal distribution, in theory.

To accomplish this we need the updates of the encoder and decoder to be dependent of the discriminator. By recommendations from the original paper authors [3] the networks are updated as follows: the discriminator is updated as in a normal GAN and the decoder as the generator in the a GAN + a reconstruction loss term. The encoder is updated with the reconstructed term + the KL loss as above. For the VAE the reconstructed loss term was set as $log(p_\theta(x|z))$, now we will let it be $log(p_\theta(D_l(x)|z))$, i.e. instead of measuring using pixel-to-pixel values we measure the l'th discriminator layer output using the reconstructed image against the output using real image. We do this for each layer. By experience the authors also mention why the encoder shouldn't be updated with a loss function based on the GAN loss.

3.3. Task driven losses

OBS! I assumed below that you meant that the classifier should be trained jointly with the encoder and decoder. If not then the middle paragraph can be removed (starting with 'This however..').

Well, similarly to the GAN attempting to say if an image is real or not, we could use a classifier to see if we can classify the image or not. In our case we already have labelled data, so exchanging the GAN to a classifier and using the labels to determine how good the classifier is could be one idea (see Schema 4). In this case we won't get a binary response as for the GAN (real image / synthetic image) but multiple classifications and as such the output and hence back propagation should instead use the sigmoid function.

VAE

$\underset{x}{Input\ sample} \xrightarrow[Class.]{\text{Class.} \searrow}$ classification

Schema 4: VAE with a classification network.

As such, like the VAE-GAN we could update with following losses,

$$\begin{aligned} Encoder\ loss &= KL\_loss + log(C_l(x)|z) \\ Decoder\ loss &= log(C_l(x)|z) + C\_loss \\ Classifier\ loss &= C\_loss \end{aligned}$$

where C_loss could *e.g.* be the standard cross entropy loss function for both real and reconstructed. $log(C_l(x)|z)$ as before is the difference between each layer in the classifier net between the real image and the reconstructed image.

This however might lead to some issues, for instance early on in the network when the decoder reconstructed images doesn't look like anything. Then we would also train the classifier on images which should have no correspondence, and as such shouldn't be classified. Following this thought there are a couple of things we could do to improve the situation; 1) we could calculate $logp(x|z)$, and if it is above a certain threshold (*i.e.* difference between real and reconstructed is 'too' great) we only train the classifier on the real image. 2) we could add one more classification for nonsensical reconstructions and yet again, determine this by *e.g.* using $logp(x|z)$ (if $logp(x|z) > threshold$ then 100 % false image). This would be more made like the GAN but with a classifier instead of 'real image'.

Instead of all this we could also simply just add the cross entropy of the classifier output as an extra term to the VAE loss, times a regularisation term. Without testing any propositions, the scheme is closely related to the VAE-GAN scheme, and as such one can argue that when using the classifier as such should produce somewhat good results. But instead of the better quality images as from the GAN, possibly we could get a better separation between the different features of the images.

References

[1] C. DOERSCH. Tutorial on variational autoencoders. *arXiv.org*, 1606 05908, 2021.

[2] D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv.org*, 1312 6114, 2014.

[3] A. B. L. Larsen and et.al. Autoencoding beyond pixels using a learned similarity metric. *arXiv.org*, 1512 09300, 2016.

[4] F. Mohr. Teaching a variational autoencoder (vae) to draw mnist characters. https://towardsdatascience.com/teaching-a-variational-autoencoder-vae-to-draw-mnist-characters-978675c95776.

Appendices

A. Networks

Listing 1: Network 1. 21380 / 25089 encoder / decoder learnable parameters.

```
net.Encoder = dlnetwork(layerGraph([
    imageInputLayer(pvars.imageSize,'Name','
        input_encoder','Normalization','none')
    convolution2dLayer(3, 32, 'Padding','same'
        , 'Stride', 2, 'Name', 'conv1')
    reluLayer('Name','relu1')
    convolution2dLayer(3, 64, 'Padding','same'
        , 'Stride', 2, 'Name', 'conv2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(2 * pvars.latent_dim,
        'Name', 'fc_encoder')
    ]));
net.Decoder = dlnetwork(layerGraph([
    imageInputLayer([1 1 pvars.latent_dim], '
        Normalization','none','name', 'Input1'
        )
    transposedConv2dLayer(7, 64, 'Cropping', '
        same', 'Stride', 7, 'Name', '
        transpose1')
    reluLayer('Name','relu1')
    transposedConv2dLayer(3, 32, 'Cropping', '
        same', 'Stride', 2, 'Name', '
        transpose2')
    reluLayer('Name','relu2')
    transposedConv2dLayer(3, 1, 'Cropping', '
        same', 'Stride', 2, 'Name', '
        transpose3')
    ]));
```

Listing 2: Network 2. 21124 / 62017 encoder / decoder learnable parameters.

```
net.Encoder = dlnetwork(layerGraph([
    imageInputLayer(pvars.imageSize,'Name','
        input_encoder','Normalization','none')
    convolution2dLayer(3, 32, 'Padding','same'
        , 'Stride', 2, 'Name', 'conv1')
    reluLayer('Name','relu1')
    convolution2dLayer(3, 64, 'Padding','same'
        , 'Stride', 2, 'Name', 'conv2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(2 * pvars.latent_dim,
        'Name', 'fc_encoder')
    ]));
net.Decoder = dlnetwork(layerGraph([
    imageInputLayer([1 1 pvars.latent_dim],'
        Name','i','Normalization','none')
    transposedConv2dLayer(7, 64, 'Cropping', '
        same', 'Stride', 7, 'Name', '
        transpose1')
    reluLayer('Name','relu1')
    transposedConv2dLayer(3, 64, 'Cropping', '
        same', 'Stride', 2, 'Name', '
        transpose2')
    reluLayer('Name','relu2')
    transposedConv2dLayer(3, 32, 'Cropping', '
        same', 'Stride', 2, 'Name', '
        transpose3')
    reluLayer('Name','relu3')
    transposedConv2dLayer(3, 1, 'Cropping', '
        same', 'Name', 'transpose4')
    ]));
```

Listing 3: Network 3. 136644 / 72961 encoder / decoder learnable parameters but using dropout. Inspired by the network found in [4]

```
net.Encoder = dlnetwork(layerGraph([
    imageInputLayer(pvars.imageSize, '
        Normalization','none','name', 'Input1'
        )
    convolution2dLayer(4, 64, 'stride', 2,'
        Padding', 'same', 'name', 'convlayer1'
        )
    reluLayer('name', 'relu1')
    dropoutLayer(pvars.keep_prob, 'name', '
        drop1')
    convolution2dLayer(4, 64, 'stride', 2, '
        Padding', 'same', 'name', 'convlayer2'
        )
    reluLayer('name', 'relu2')
    dropoutLayer(pvars.keep_prob, 'name', '
        drop2')
    convolution2dLayer(4, 64, 'stride', 1, '
        Padding', 'same', 'name', 'convlayer3'
        )
    reluLayer('name', 'relu3')
    dropoutLayer(pvars.keep_prob, 'name', '
        drop3')
    fullyConnectedLayer(2*pvars.latent_dim,'
        name', 'fullyconn1')
```

```
    ]));

net.Decoder = dlnetwork(layerGraph([
    imageInputLayer([1 1 pvars.latent_dim], '
        Normalization','none','name', 'input2'
        )
    transposedConv2dLayer(7, 64,'stride', 7,'
        Cropping', 'same', 'name', 'tconv1')
    dropoutLayer(pvars.keep_prob, 'name', '
        dropt1')
    reluLayer('name', 'trelu1')
    transposedConv2dLayer(4, 32, 'stride', 2,
        'Cropping', 'same', 'name', 'tconv2')
    dropoutLayer(pvars.keep_prob, 'name', '
        dropt2')
    reluLayer('name', 'trelu2')
    transposedConv2dLayer(4, 1, 'stride', 2, '
        Cropping', 'same', 'name', 'tconv3')
    ]));
```

Listing 4: Network 4. 403972 / 25089 encoder / decoder learnable parameters.

```
net.Encoder = dlnetwork(layerGraph([
    imageInputLayer(pvars.imageSize, '
        Normalization','none','name', 'Input1'
        )
    fullyConnectedLayer(512,'name', '
        fullyconn1')
    reluLayer('name', 'Relu1')
    fullyConnectedLayer(2*pvars.latent_dim,'
        name', 'latent space')
    ]));
net.Decoder = dlnetwork(layerGraph([
    imageInputLayer([1 1 pvars.latent_dim], '
        Normalization','none','name', 'Input1'
        )
    transposedConv2dLayer(7, 64, 'Cropping', '
        same', 'Stride', 7, 'Name', '
        transpose1')
    reluLayer('Name','relu1')
    transposedConv2dLayer(3, 32, 'Cropping', '
        same', 'Stride', 2, 'Name', '
        transpose2')
    reluLayer('Name','relu2')
    transposedConv2dLayer(3, 1, 'Cropping', '
        same', 'Stride', 2, 'Name', '
        transpose3')
    ]));
```

B. Reconstruction results and other images

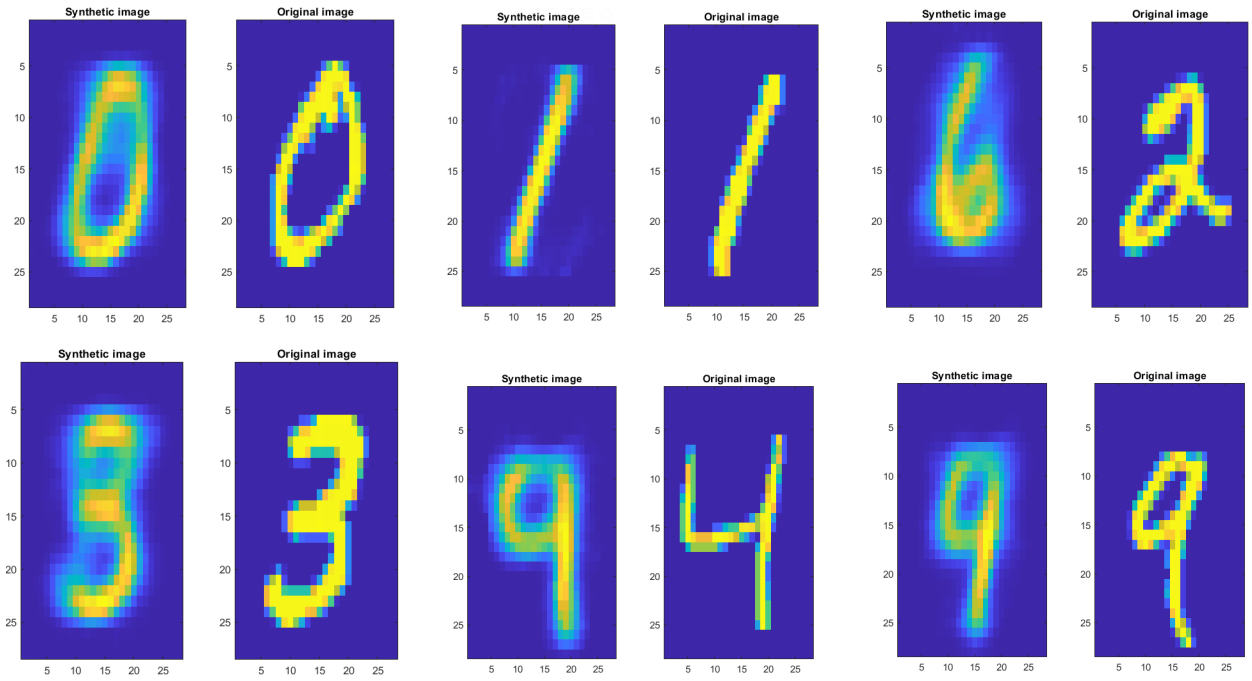

Figure 9: *VAE reconstruction and the original input image using network 2 and 'adamupdate'. The network was trained on 20 000 train images during 40 epochs with learn rate set to 0.0005.*

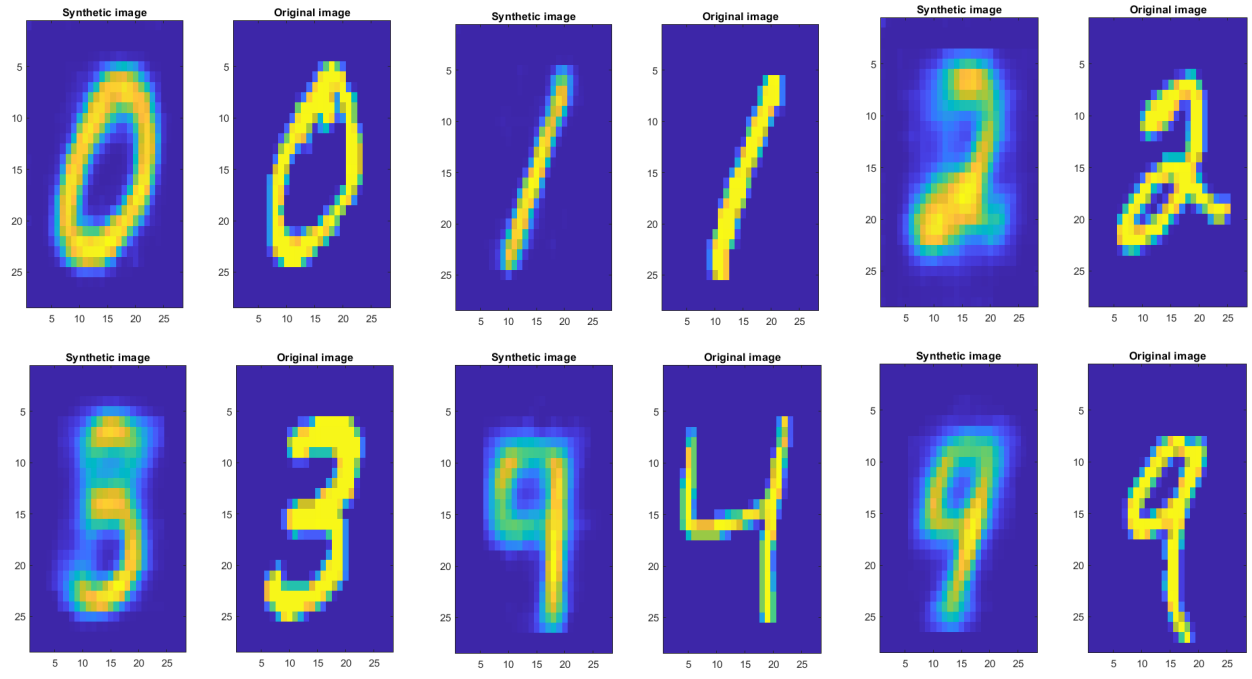

Figure 10: *VAE reconstruction and the original input image using network 4 and 'adamupdate'. The network was trained on 20 000 train images during 40 epochs with learn rate set to 0.0005.*

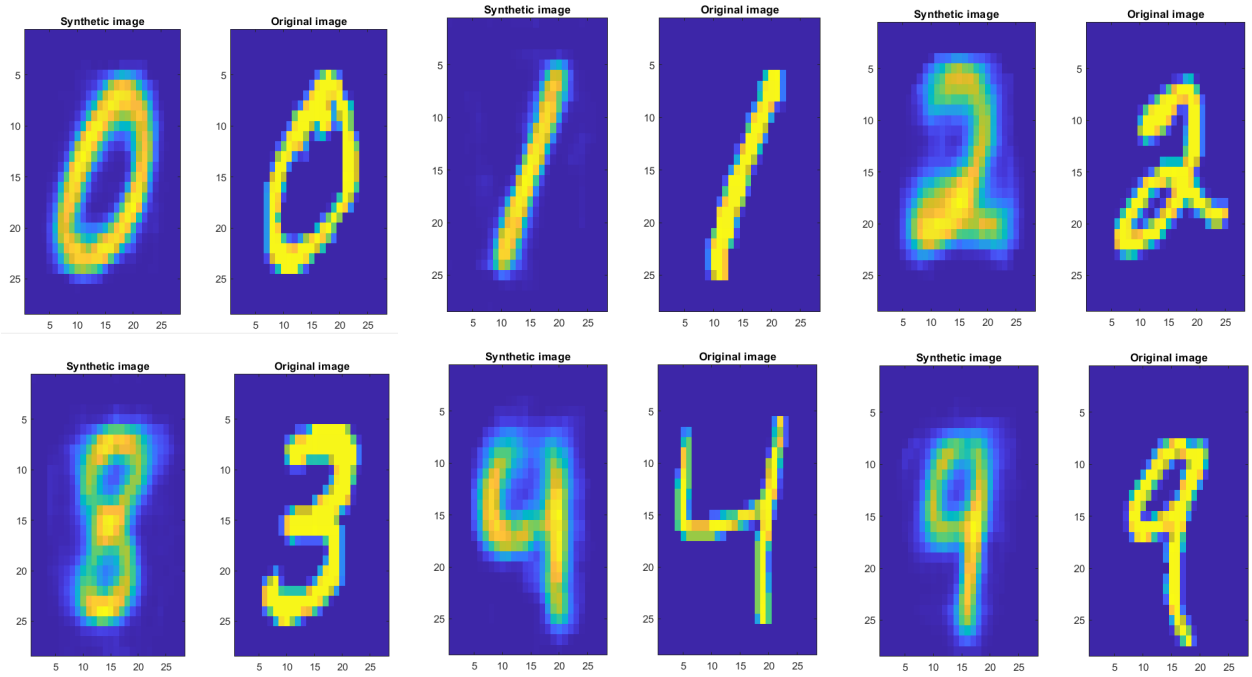

Figure 11: *VAE reconstruction and the original input image using network 4 and 'adamupdate'. The network was trained on 60 000 train images during 80 epochs with learn rate set to 0.0005.*

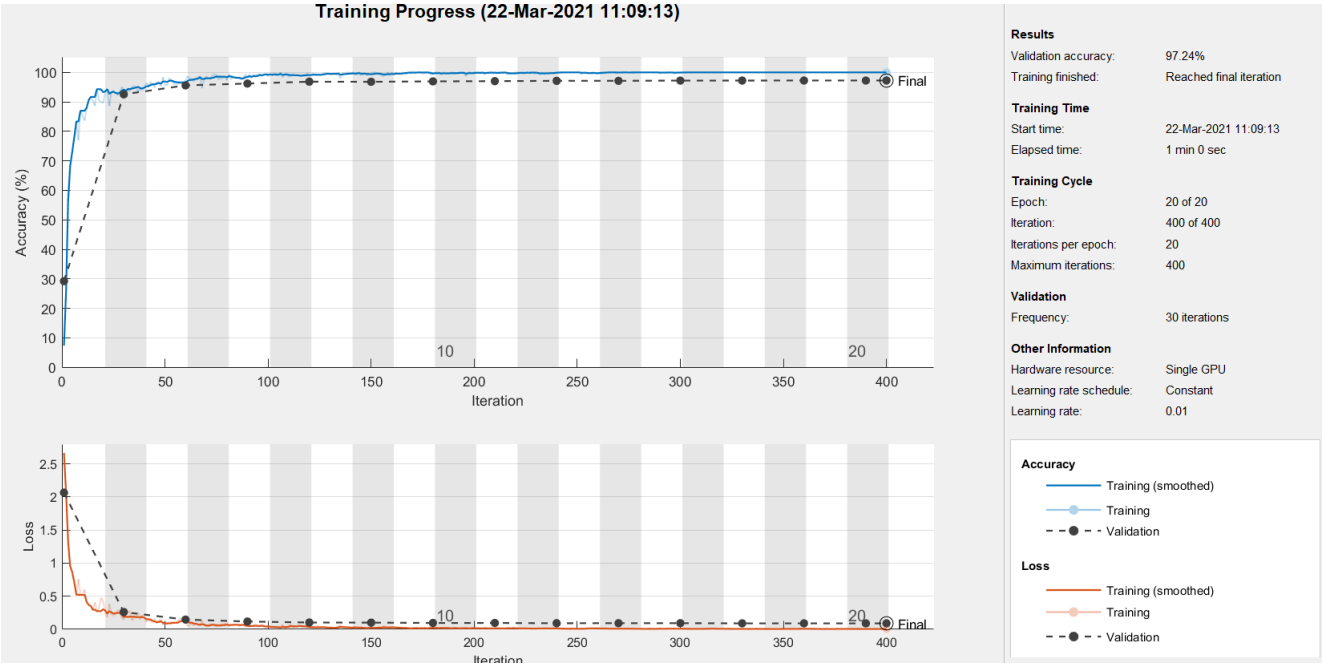

Figure 12: *Trained classifier on 0.5×0.2 of the total number of images without including synthetic images.*

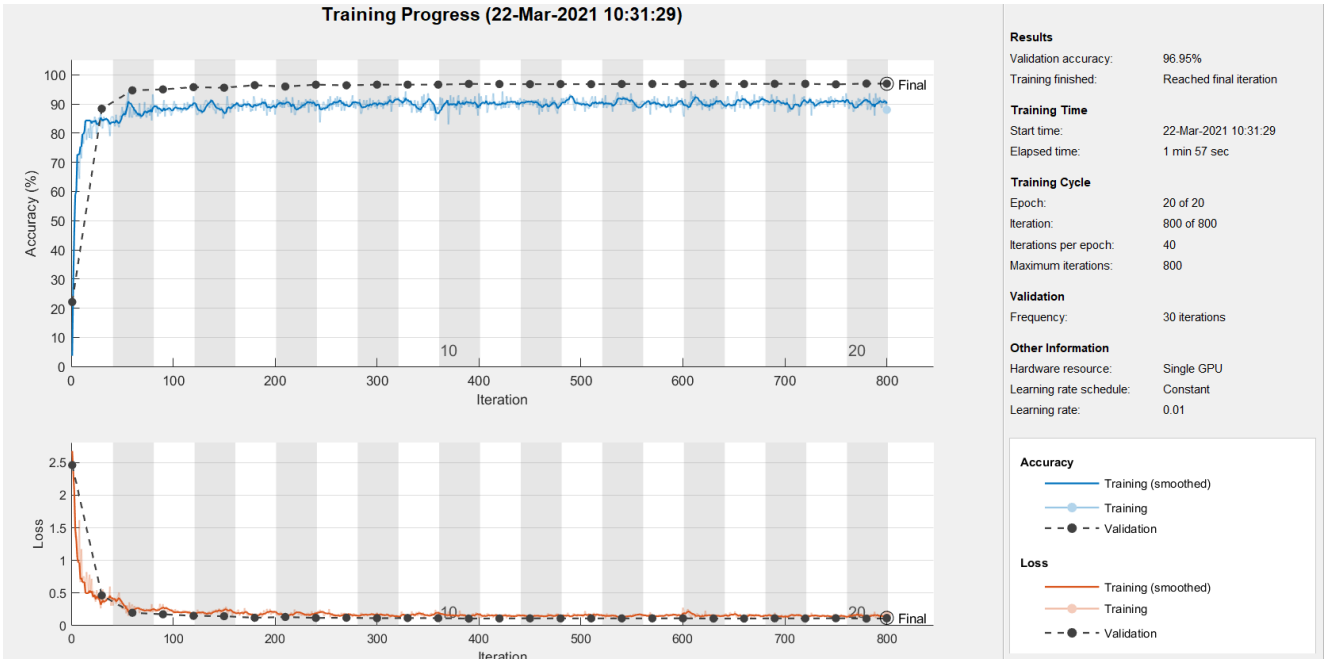

Figure 13: *Trained classifier on 0.5×0.2 of the total number of images + synthetic images.*

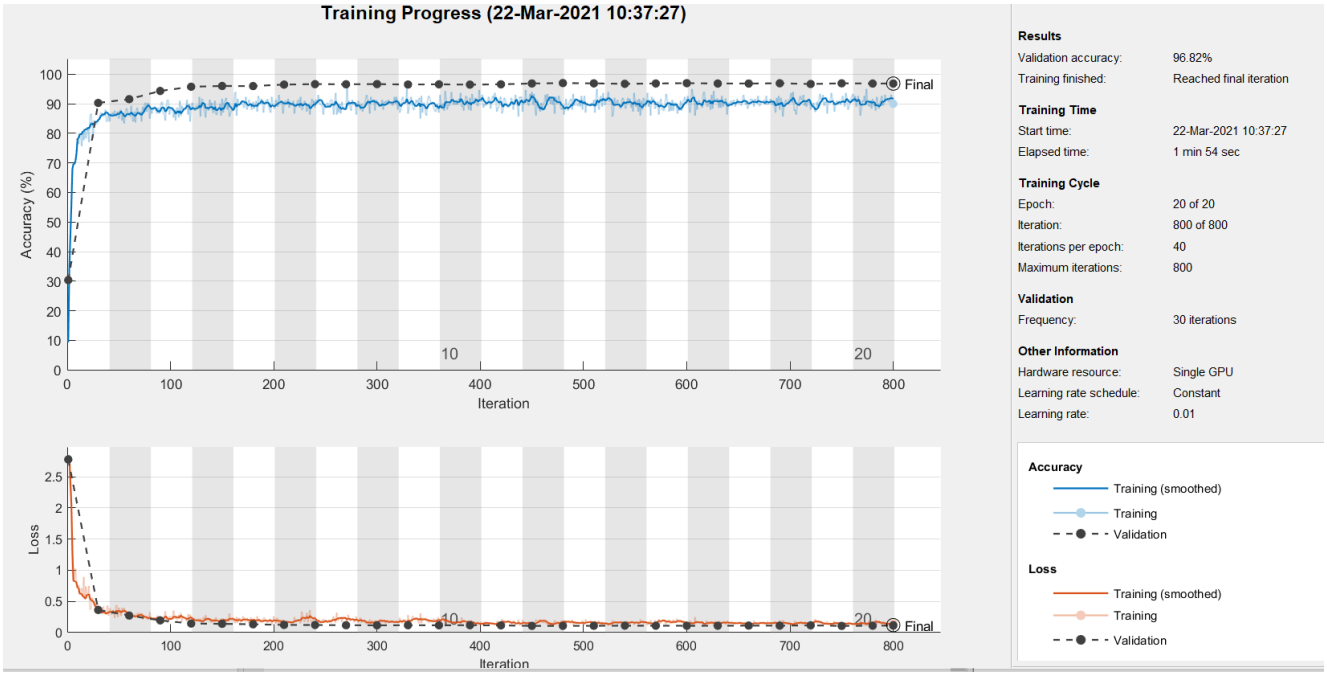

Figure 14: *Trained classifier on 0.5×0.5 of the total number of images + synthetic images.*

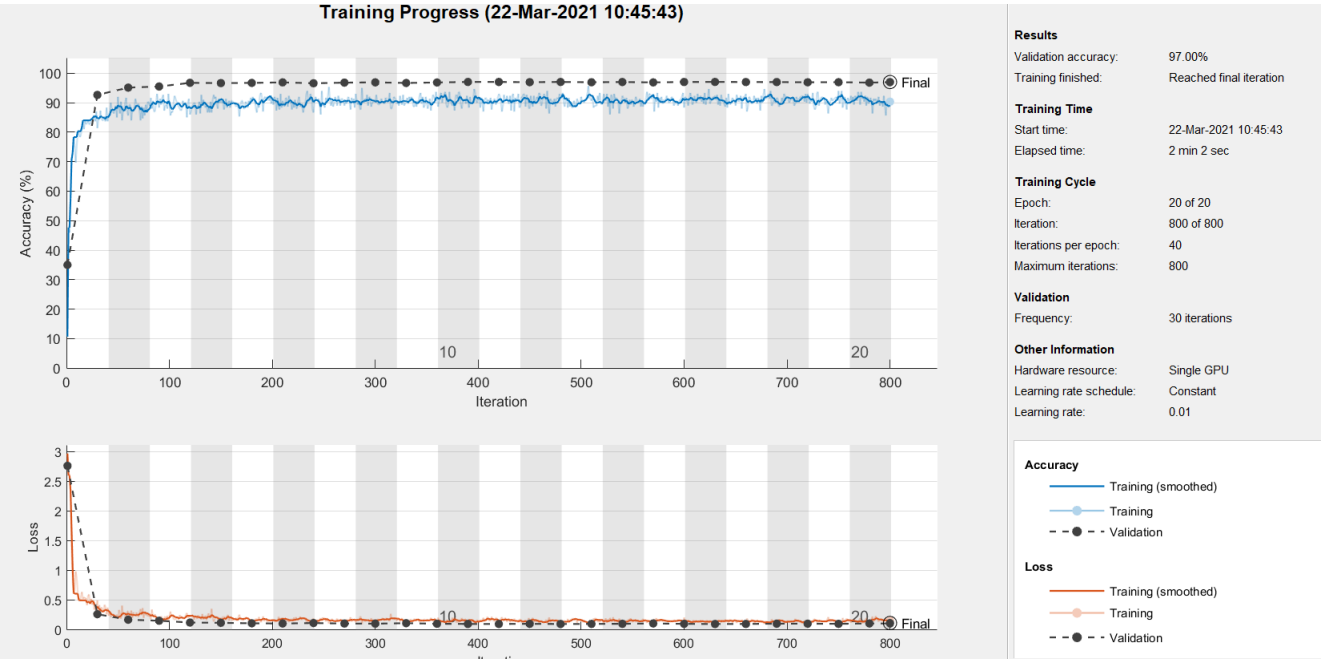

Figure 15: *Trained classifier on 0.5 of the total number of images + synthetic images.*

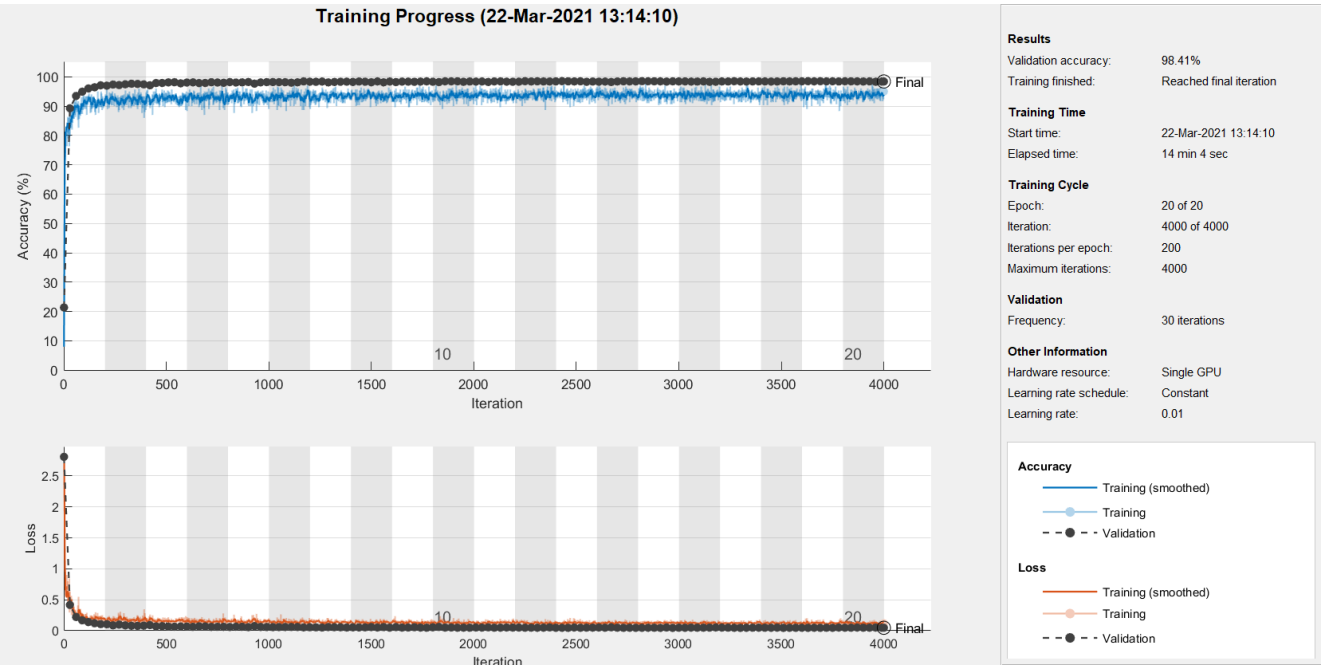

Figure 16: *Trained classifier on 0.5 of the total number of images + synthetic images. The VAE was trained using network 4, with the full training set and learn rate set to 0.0005.*