IDL Assignment 1

Group 76: Jasper Mens (s2015242) and Laurens Müller

November 2023

Type	Parameters
Flatten	
Dense	`n=200, activation = RELU`
Dense	`n=100, activation = RELU`
Dense (output)	`n=10, activation = Softmax`

Table 1: Base MLP model based on the example given in the book, from which we vary the optimizer (Adam or SGD) and the input size ($32 \times 32 \times 3$ for CIFAR or $28 \times 28 \times 1$ for MNIST).

Task 1: Keras Basics

In this task we experiment with variations to the MLP and CNN architectures shown in tables 1 and 2, and search for the best-performing architectures on the Fashion MNIST-dataset. Then, we take the 3 best architectures and apply them to the CIFAR-dataset. To facilitate this, we set up a simple script that iterates through all the possible combinations of (boolean) changes to the default models. For the CNN, we try changing the optimizer (Adam or SGD) the number of convolutional layers (3 or 5), and the number of dropout layers (none or 2). For the MLP, we only vary the optimizer type. We train the models for 30 epochs, using the categorical cross-entropy as a loss function and a 90/10 train/test split. An overview of the CCN and MLP model differences and the corresponding performance can be seen in table 3 and table 4 respectively. We notice a few things:

- Even though the loss and the accuracy improve considerably while training the MLP using the Adam optimizer, its performance on the validation set doesn't. In fact, after improving a bit, the loss actually begins to increase after 10-15 epochs (see figure 1). This does not happen when using the Stochastic Gradient Descent optimizer, where both the loss and the accuracy keep improving during training (see figure 2). This is overfitting induced by the omission of drop-out layers in the first case.
- We observe the same effect while training the CNN using the Adam optimizer, whereas using SGD results in *better* loss and accuracy on the validation set during most of the training process, slowly approaching the performance on the test set. This suggests that using the Adam optimizer results in considerable overfitting on the training set, so we decide to use SGD on the CIFAR set.
- Even though the CNN-architecture is varied considerably, the accuracy on the validation set changes only very little, especially when using the Adam-optimizer but also with the SGD-optimizer when omitting the dropout layers. In the latter case, the same initial improvement followed by the worsening of the loss is observed as with the Adam optimizer.
- Even though the MLP gets close when using SGD, the three best-performing architectures are all CNNs using SGD. Interestingly, the MLP with SGD actually has a better loss than the CNN with SGD, 5 convolutional layers and no drop-out layers. Conversely, the CNN with SGD, 3 convolutional layers and no drop-out layers is one of the three winners.

Type	Parameters
Conv2D	`n=64, kernel_size=7, strides=1`
MaxPool2D	`pool_size=2, strides=1`
Conv2D	`n=128`
Conv2D	`n=128`
MaxPool2D	`pool_size=2, strides=1`
Conv2D	`n=128`
Conv2D	`n=128`
MaxPool2D	`pool_size=2, strides=1`
Flatten	
Dense	`n=128, activation = relu`
Dropout	`rate=0.5`
Dense	`n=64, activation = Relu`
Dropout	`rate=0.5`
Dense (output)	`n=10, activation = Softmax`

Table 2: Base CNN model taken from the book, to which we vary the optimizer (Adam or SGD), the number of convolutional layers (3 or 5), the number of drop-out layers (none or 2) and the input size ($32 \times 32 \times 3$ for CIFAR or $28 \times 28 \times 1$ for MNIST) to find optimally performing architectures.

Finally, we apply the three winning architectures to the CIFAR-dataset (see table 5. We notice that the performance of the latter two (CNNs with SGD, 2 drop-out layers and 3 resp. 5 convolutional layers) is slightly better than on the Fashion MNIST dataset. Interestingly, the one candidate without drop-out layers performs worse, and the training- and validation-performances don't converge, just like the other CNN with SGD without drop-out layers discussed above. We therefore conclude that the best candidates are CNNs using

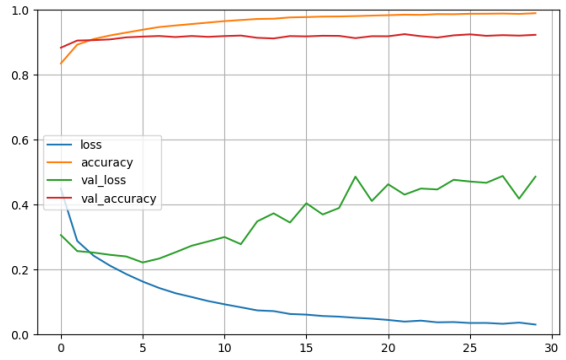

Figure 1: Example of overfitting: improvement on training set and worsening on validation set when using the Adam optimizer (3 convolutional and no drop-out layers)

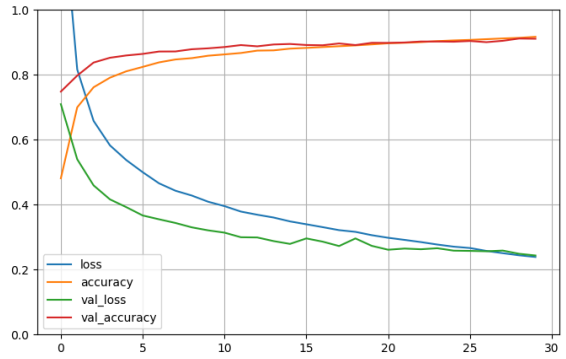

Figure 2: Example of improvement both on training and validation set when using the SGD optimizer (5 convolutional and 2 drop-out layers)

SGD with 2 drop-out layers, and that the variation of the convolutional layers has the least effect for that architecture, suggesting at least a local optimum.

T2: Tell The Time

This task revolves around developing a CNN capable of reading the time from images of analog clocks. We try out three approaches (classification, regression, and a combination of the two), and compare the results.

One important note is that conventional ('linear') accuracy is not quite the right metric for this application, as the time displayed on a clock is cyclical, such that, for example, the difference between `11:55` and `00:05` is only 10 minutes. Thus, we introduce a "common sense" accuracy, which is simply the mean error with this caveat in mind. To compute this metric, some unit conversion is needed. The labels from the dataset are converted to integers between 0 and $12 \times 60 = 720$, representing minutes since midnight/noon, such that the linear error `t_true - t_pred` can be expressed in minutes. The cyclic deviation (i.e., the "common sense" error) is then simply the minimum of `720-linear_error` and `linear_error`.

Optimizer	Conv. Layers	Dropout layers	Loss	Accuracy
`Adam`	3	0	`0.4864`	`0.9232`
`Adam`	3	2	`0.3633`	`0.9258`
`Adam`	5	0	`0.3905`	`0.9198`
`Adam`	5	2	`0.3669`	`0.9144`
`SGD`	3	0	`0.2594`	`0.9104`
`SGD`	3	2	`0.2551`	`0.9088`
`SGD`	5	0	`0.3331`	`0.9020`
`SGD`	5	2	`0.2558`	`0.9132`

Table 3: CNN Performance (loss and accuracy on validation set) on Fashion MNIST-dataset after 30 training epochs

Optimizer	Loss	Accuracy
Adam	`0.4668`	`0.8990`
SGD	`0.3051`	`0.8894`

Table 4: MLP Performance (loss and accuracy on validation set) on Fashion MNIST-dataset after 30 training epochs

Model 1: Classification

By grouping the labels into n_c intervals of $720/n_c$ minutes, we can approach the problem using n-class classification. While models with higher n_c should be capable of more accurate results, we expect this accuracy to come at a cost. Thus, we begin with large intervals, and slowly ramp up the number of classes after we have found a model that performs well enough on the simplest case. In total, we try three different n_c, starting with a modest $n_c = 12$ and doubling twice from there to 24 and 48.

Of course, this classification approach requires distinct labels. Thus, we write a simple routine that rounds the time down to integer multiples of $720/n_c$ minutes past noon, such that `class_label = int( (hours * 60 + minutes) * n_c/720)`. Note that because of this floor division, we interpret the model prediction as being perfectly centered within the class interval. For instance, when dividing the clock into 30-minute intervals ($n_c = 24$), the predicted time corresponding to class `0` is `00:15`.

It is furthermore worth mentioning that this gives us a lower bound to the common sense deviation. When $n_c = 12$, the error can never be less than 30 minutes, even when the classification is perfect. Since each clock time is equally represented in the dataset, the resulting mean error can therefore never be under 15 minutes.[1] It will be good to keep this in mind when comparing the performance of the models discussed below.

[1] Technically, less than 15 minutes is possible. For instance, if the test set only contains images of clocks showing `XX:30`, this metric would obviously return a mean error of 0 minutes. So, this lower bound only holds as long as the test set is large enough and the labels are isotropically distributed.

Optimizer	Conv. Layers	Dropout layers	Loss	Accuracy
SGD	3	0	0.2714	0.9116
SGD	3	2	0.2435	0.9106
SGD	5	2	0.2435	0.9112

Table 5: Winning architectures performance (loss and accuracy on validation set) on the CIFAR-dataset after 30 training epochs

Type	Parameters
Conv2D	n=32, kernel_size=4, strides=2
MaxPool2D	pool_size=4, strides=4
Conv2D	n=32, kernel_size=2, strides=1
MaxPool2D	pool_size=2, strides=2
Flatten	
Dropout	rate=0.15
Dense	n=144
Dropout	rate=0.15
Dense	n=144
Dense (output)	n=12, activation=softmax

Table 6: Fiducial model architecture for the clock-reading CNN classifiers. The convolutional and dense layers all use the ReLu activation function. With a batch size of 32, and a Nadam optimizer with exponential learning rate decay but otherwise default parameters, this model is capable of reaching 80% test set accuracy within 100 epochs, with a common sense error of only 28 minutes.

Type	Parameters
Conv2D	n=64, kernel_size=4, strides=2
Conv2D	n=64, kernel_size=4, strides=2
MaxPool2D	pool_size=4, strides=2
BatchNorm	
Conv2D	n=64, kernel_size=2, strides=1
Conv2D	n=64, kernel_size=2, strides=1
MaxPool2D	pool_size=2, strides=2
BatchNorm	
Flatten	
Dropout	rate=0.5
Dense	n=360
BatchNorm	
Dropout	rate=0.5
Dense	n=360
BatchNorm	
Dense (output)	n=12, activation=softmax

Table 7: Final model architecture for the clock-reading CNN classifiers. The batch size, activation functions, initializations, and optimizer parameters are all identical to the ones used on the fiducial model, save for the speed of learning rate decay, which is doubled. This model is capable of reaching up to 96% test set accuracy within 100 epochs, with a mean error of only 16 minutes.

Tuning for full hours

After some initial exploration, we settle on the fiducial model architecture shown in table 6. We use the Nadam optimizer with default parameters, aside from using a scheduled learning rate. We use a gentle exponential decay, such that the learning rate halves every 40 epochs, starting from 0.0002. We let the training run for 200 epochs, but employ early stopping such that training stops if the training loss has not improved for 5 epochs. Finally, as a loss function we simply use the built-in categorical cross-entropy function.

With this configuration, the classifier is capable of reaching a test set accuracy of roughly 80% within 100 epochs of training, with a CSE (common sense error, as described above) of 28 minutes. This is a good start, but it would be nice if it converged faster and the accuracy were higher. So, we experiment and tune the parameters some more. We will now walk you through our experiments and corresponding thought process, leading up to our final classifier (table 7). An overview of the models described here can be seen in table 8, and the accuracy progression of some highlight models can be seen in figure 3.

1. First, we try increasing the batch size in an attempt to decrease the training duration. We set the batch size to 256 and leave everything else untouched. We find that, while this does decrease the compute time per epoch, the number of necessary epochs is much higher now. Furthermore, the model starts to overfit significantly, and the final performance is much worse overall.

2. While the increased batch size only made things worse, perhaps batch normalization would help mitigate these problems. We try a run with an even larger batch size of 512, and implement batch normalization between the convolutional stacks. This does improve things a little, but the performance is still far worse than the fiducial model, with a lot of overfitting. So, we return to a batch size of 32.

3. Judging from the past two experiments, batch normalization did seem to improve the situation quite a bit. So, we try a run with batch normalization after every pooling and dense layer. While this yielded strictly worse test performance than model 0, this is mostly due to overfitting, as the training accuracy soars to around 95%, converging extremely quickly. If we can combat this overfitting, this model shows great promise.

4. To combat this overfitting, we ramp up the droupout rates in for the dense layers to 0.5. While this may seem drastic at first, such high rates are actually relatively common in convolutional nets. This iteration yielded a test accuracy of 81% in 160 epochs, with a CSE of 25

minutes. While the overfitting problem has been completely eliminated, the particularly long convergence tail suggests that there is still room for improvement.

5. We now try doubling the number of convolutional nodes per layer, with the goal of raising the accuracy ceiling. In previous experimentation, we found that adding more nodes this way generally only causes overfitting. Hopefully, the high dropout changes this picture. And indeed, overfitting is not an issue, and this model achieves an accuracy of 83% after about 150 epochs, with a CSE of 23 minutes. While this is an improvement over the fiducial model, it is only incremental, and the convergence tail is still very long. As this doubling of convolutional nodes is computationally expensive and only results in incremental improvements at best (and may still introduce overfitting issues if taken too far), we keep this model as our current best, and seek further improvement elsewhere.

6. Perhaps a less gentle learning rate curve could improve the situation. For the next run, we double the decay rate, such that it now halves every 20 epochs instead. This model reaches 80% accuracy after 117 epochs, with a CSE of 26 minutes. While this is not exactly an improvement in terms of accuracy, the convergence is faster.

7. We decide to push our luck, and double the learning rate decay rate once again. The model now reaches 53% in 72 epochs, and a CSE of 45 minutes. Clearly, we have overshot the mark here, so we return to the decay rate of the previous iteration.

8. Although adding more nodes to the convolutional stacks did not do much, we have not yet explored adding more convolutional *layers*. So, we simply duplicate the `Conv2D` layers. Interestingly, the performance is just about on-par (or slightly worse) than the previous ones. In part, we attribute this to the strides of the extra convolutional layer decreasing the size of the images as they enter the dense stack (down from 8×8 to just 3×3), meaning fewer weights overall, and likely less information to work with for the dense stack. Another potential explanation is that the dense stack is simply incapable of utilizing the full potential of the added filters.

9. We attempt to combat both of these potential causes by setting `strides=2` for the first max-pooling layer (down from 4), and raise the number of nodes per dense layer from 144 to 360 (normally, we would probably use 128 and 256, but we feel that these clock-related numbers are more fun here). The smaller strides mean that the dense layers now have 7×7 (flattened) images to work with, which should be a lot better, and as an added bonus the second convolutional stack has more information as well. It does not take long for this model to start to stick out above the competition, as the test accuracy easily exceeds the previous best result within about 30 epochs. Our only concern is that the test accuracy is quite erratic, so performance may not be as reliable as one would hope. This is a trend that we have seen before in other models, where the test accuracy effectively does a random walk around the training accuracy, yielding final performance metrics that depend sensitively on the specific epoch at which training stopped, i.e., unreliable models. Luckily, this model managed to outgrow its rebellious phase, and the variations died out after 50 or so epochs. Finally, after 96 epochs, the early stopping callback halts the training at an unprecedented 96% test set accuracy and a CSE of just over 16 minutes (remember: the lower bound is 15 minutes). We see very little room for improvement here, so we cash our proverbial chips now, and move on to training on smaller time intervals.

Smaller intervals

Satisfied with the performance on $n_c = 12$, we now move on to progressively smaller intervals. We begin by doubling the number of classes. The model now gives us a test set accuracy of 93% after 109 epochs, and a CSE of 9.8 minutes. For comparison, the lower bound of the mean error is now 7.5 minutes, so there is a slight degradation of performance across all three metrics (class accuracy, training time, and CSE). Perhaps this discrepancy could be made up by simply adding more nodes in one way or another, but such measures are hardly necessary with this performance.

Sadly, the common sense error plateaus at this value, as doubling n_c again yields a CSE of 9.6 minutes, and 88% accuracy after 106 epochs. It seems like we have reached the limit of what this model is capable of.

Out of interest, we retrain one more time, this time using $n_c = 720$, i.e., single minute intervals. Now, performance is absolutely abominable, with 15% accuracy after 128 epochs, a CSE of 67 minutes, and a huge amount of overfitting. In part, this is due to the limits of our hidden layers, but another significant problem is that there is only so much training data. The full dataset contains only 25 samples of each of the 720 minutes in the day, all of which viewed from different angles and lit differently, so there may simply not be enough information to let the model pick up a pattern. This is only made worse by the fact that there is no reward for almost getting it right in classification; a prediction is either correct or wrong, so being one minute of is treated the same as being three hours off.

While we are able to get a mean deviaton of about 10 minutes using this approach, we suspect that any further accuracy gains would come at a steep price in terms of model complexity and training time, and getting a decent prediction down to the minute seems to be far beyond reach.

Model 2: Regression

Some of the aforementioned problems are solved by changing to a regression strategy instead. Here, getting close to the true answer is explicitly rewarded, which makes sense in the context of reading a clock.

The main difference between this approach and classification is of course the loss function. Where we used the categorical cross-entropy before, we can now use the mean error

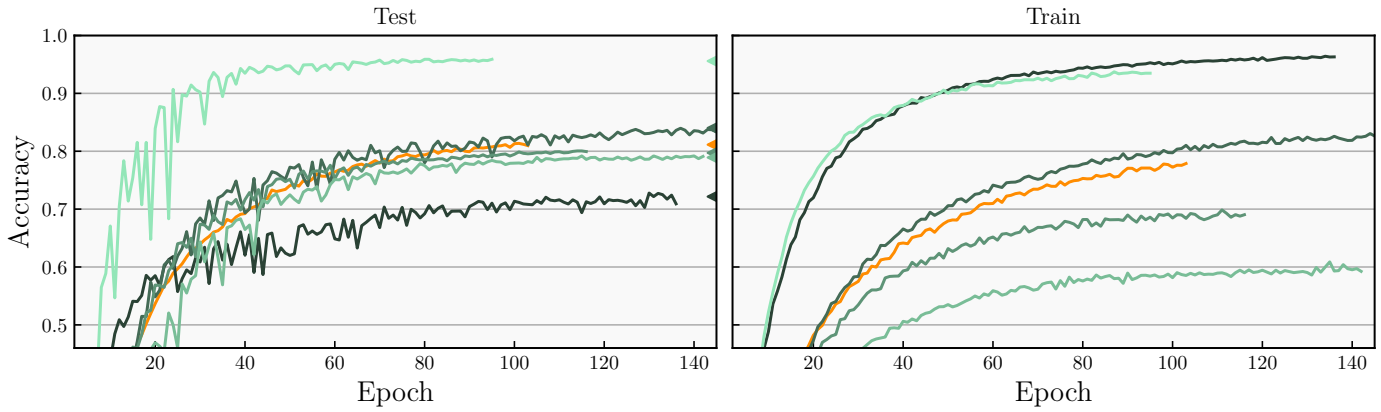

Figure 3: Accuracies during training for some of the standout models throughout our experimentation (see table 8). The orange curve corresponds to the fiducial model, after which the brightness of the lines denotes the order of execution, with the color getting brighter for later models. The final model, although it does show strong variability initially, dwarfs the previous iterations in terms of test accuracy.

		% Acc	Epochs	CSE	Notes
0	*	81	100	28	Fiducial model
1		62	175	44	Increased batch size: no speedup, lots of overfitting
2		67	135	40	Big batches plus batch norm: still bad, but BN helps
3	*	72	137	35	BN with fiducial batch size: faster, but stil overfits
4		81	160	25	Combat overfitting with harder dropout: no more overfitting!
5	*	83	150	23	Previous model + doubled convolutional nodes: pretty good!
6	*	80	117	26	Faster learning rate decay: not quite better, but converges faster!
7		53	72	45	Even faster LR decay: terrible! go back!!
8	*	79	143	23	Extra convolutional layers: maybe the extra info isn't being put to use
9	*	96	96	16	More dense nodes and smaller pooling strides: perfection!

Table 8: Overview of the hyperparameter optimization for the $n_c = 12$ clock classifier. The accuracy progression during training is plotted in figure 3 for the models marked with $*$.

(generally either absolute or squared error- we opt for absolute, but we did not notice much of a performance difference between the two).

We use the architecture of the classification model (table 7) as a starting point, only changing the output layer from n_c dense nodes with sigmoid activation, to one single node with a linear activation function. Sadly (though not unexpectedly), this model did not perform quite as well as before, so we do some further tuning. Because the convolutional stack seemed to be quite capable of getting a decent grasp of the orientation of the clock and its hands before, we restrict our experimentation to the dense stack.

The first thing that we notice about the regression training is that there there seems to be a lot of headroom for overfitting, i.e., the dropout rate is probably too high. Furthermore, the mean absolute error seems like it could be smaller. Our goal here is to at least match the classification model's performance, so there is some work to be done. After dialing back the dropout rate a bit (to $0.2 - 0.3$), we first try a smaller learning rate, which may allow us to reach a more 'narrow' minimum. We experiment with smaller initial learning rates, as well as higher decay rates, but no real improvements are found, so we move on.

Next, we try bolstering the dense layers. Since we already have quite a large number of nodes in each of the layers, we instead begin by doubling up each of the dense layers. Finally, this increased the performance somewhat, so we try pushing a little harder by adding an additional dense stack (set of dropout+dense layers), which finally delivers performance we are happy with. This final model (table 9) reaches a CSE of 12.5 minutes after 86 epochs of training, which is quite decent, but not as good as we might have hoped. Perhaps better hyperparameter combinations exist, but we suspect that adding more weights will come with diminishing returns, so a different approach might be more fruitful.

There are a few problems with the regression approach. First, the output has to be discontinuous between 719 and 0 minutes, as that is how time works. This is not great, as it means closely related inputs do not correlate to closely related outputs. Another problem is that both hands have to be read correctly in order to get a good loss value. This may cause problems in situations where one of the hands is correctly

Type	Parameters
Conv2D	`n=64, kernel_size=4, strides=2`
Conv2D	`n=64, kernel_size=4, strides=2`
MaxPool2D	`pool_size=4, strides=2`
BatchNorm	
Conv2D	`n=64, kernel_size=2, strides=1`
Conv2D	`n=64, kernel_size=2, strides=1`
MaxPool2D	`pool_size=2, strides=2`
BatchNorm	
Flatten	
Dropout	`rate=0.3`
Dense	`n=360`
Dense	`n=360`
BatchNorm	
Dropout	`rate=0.3`
Dense	`n=360`
Dense	`n=360`
BatchNorm	
Dropout	`rate=0.3`
Dense	`n=360`
Dense	`n=360`
BatchNorm	
Dense (output)	`n=1, activation=linear`

Table 9: Final model architecture for the regression implementation. The model contains 1.9 million trainable weights, making it a bit heavier than the classification model. It is capable of reaching a common-sense error of just under 13 minutes after 86 epochs.

identified, but the other one is not, because the model can only assume that both must be incorrect. This may slow down training and affect performance.

Model 3: Multihead

We have seen that the classification approach works extremely well when $n_c = 12$, and we have seen that the regression approach may have some trouble with confusing the hours and minutes hands. So, a natural next step is to combine the two approaches such that the hours are taken care of by a classifier and the minutes by a regression model.

Because the underlying features are likely going to be the same, we train a multi-head model that sends the output of the convolutional stack to two distinct dense stacks, which will independently predict the hours and minutes labels respectively. To keep things simple, we just copy the architectures from tables 7 and 9, and merge them at the convolutions such that they share all of their filters. The resulting model has two output heads, which each have their own loss function and ground truth labels. The relative weights of these loss functions can be tuned, such that one head has more influence over the loss than the other. In this context, we would want to emphasize getting the hours right, as the hours hand obviously the larger impact on the time. After some playing around, we find that we get great performance with a 99%/1% weight distribution[2].

The minutes head seems to rush toward a mean absolute error of less than 5 minutes, while the hours head takes a little longer to converge to a decent value. It appears that the increased dense stacks for the regression head are being put to much better use than before, so one may be able to get away with trimming some of those extra layers off. Any excess layers do not seem to be getting in the way, however, as we are able to reach a common-sense error of just 2.5 minutes within 80 or so epochs!

We notice that the convergence of the hours loss appears to be slower than before, and the test loss is much more erratic. Every few epochs, the test loss suddenly jumps up or down, suggesting less reliable performance than before, but the amplitude of these fluctuations appears to decrease slightly as the training proceeds, so we are not too worried. To be safe, we repeated the training a few times to get a sense of the spread of the results, which ended up being quite narrow.

Task 3: Generative Models

For this task, we use the Convolutional and Variational Autoencoder (CAE resp. VAE) and Generative Adversarial Network (GAN) from the notebook kindly provided together with the assignment. All three make use of the downsampling/upsampling or encoding/decoding components represented in tables 10 and 11 respectively. Because this task focuses on images, these components contain convolutional layers. The encoders compress (encode) images into a smaller feaure space representation, from which the decoders can in turn recover the images (or, in the case of the VAE and GAN, even generate entirely novel ones).

We apply these models to several image datasets, and see what we can learn from the results. We try three datasets: pictures of cats, pictures of pistachios, and some pixel-art sprites. In order to keep the computational cost and training time down somewhat, we resort to downsampling our images to $32 \times 32 (\times 3)$. This degrades the image quality quite a bit,

[2]Note that this isn't necessarily as trivial as letting the hours be worth roughly 60 minutes, as the loss of the hours head (categorical cross-entropy) does not directly translate to a common-sense deviation.

Type	Parameters
Conv2D (input)	`default`
n_down × Conv2D	`default`
Flatten	
Dense (output)	`(output_shape), act=sigmoid`

Table 10: Encoding convolutional net: (default) parameters are kernel_size=(3,3), strides=(2,2), padding=same, filters=128 and activation=relu. We have used n_down=4 downsampling layers and a sigmoid output activation function.

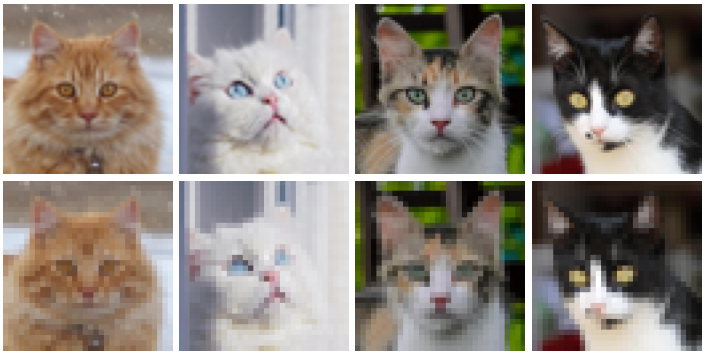

Figure 4: Examples of cat images from the original dataset. Top layer: original images. Bottom layer: their downsampled $32 \times 32 \times 3$ counterparts. There is quite a bit of degradation, but you may use this as a reference for an appropriate viewing distance/zoom level for evaluating the generated images later on.

Figure 5: Cat images from the dataset and their reconstructions by our CAE after 100 epochs of training

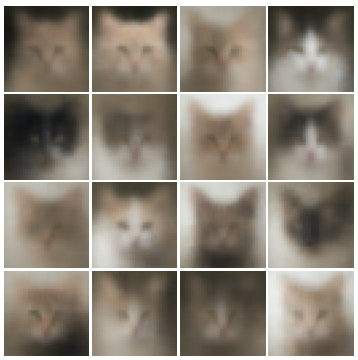

Figure 6: Images generated from random vectors in the latent space of our VAE after 20 epochs of training on the cat dataset

but if viewed from large enough distances, this is not an issue. Figures 4 and 11 show the difference in image quality, with the top panels at a resolution of 64×64, and the bottom panels at 32×32. The pixel-art dataset is not downsampled, and kept at its native resolution of 64×64. For the sake of report structure and lay-out, we have moved the non-cat dataset images to the appendices at the end.

A CAE consists of an encoder connected to a decoder; during training, the encoder learns to represent images in 256-dimensional (at least, we use a latent space dimensionality of 256) vectors, and the decoder learns to reconstruct the orginal images from these vectors. The model is trained based on the difference between input and output. Examples can be seen in figures 5 and 12.

The VAE also uses the trained decoder described above, which is fed with random samples from a diagonal Gaussian distribution over (in our case) 32-dimensional feature-space. Since the backpropagation algorithm is ill-defined on random nodes, samples are reparametrized using the average (μ) and standard deviation (σ) of the distribution of the latent representation: a sample $z = \mu + \sigma\epsilon$, where ϵ represents the random value used for sampling. The output of the VAE therefore consists of new images. Examples of cat faces and pistachios constructed from random latent vectors are shown in figures 6 and 13. Note how blurry these output images are. Even after longer periods of training (up to several hundred epochs), this does not change.

Type	Parameters
Dense (input)	`4×4×64, (latent_dim)`
Reshape	`shape=(4,4,64)`
n_up × Conv2DTranspose	`(default), n_filters=128`
Conv2D (out)	`(output_default)`

Table 11: Decoding deconvolutional net: (default) parameters are kernel_size=(3,3), strides=(2,2), padding=same and activation=relu. The (output_default) parameters are kernel_size=(3,3), padding=same, filters = 3 and activation = sigmoid. We have used (latent_dim)=32 latent dimensions, n_up=4 upsampling layers and a sigmoid output activation function.

Finally, the GAN again consists of a decoder (Generator), tasked with turning random noise in latent space into novel images, and an encoder (Discriminator) that attempts to discriminate between these 'fake' images and 'real' ones from the training set. During training, the Discriminator is fed with both real and fake images and learns to classify them as such, meanwhile the Generator uses the Discriminator's performance as feedback to learn to produce more convincing images. After training, like with the VAE, random noise vectors from the latent space can then be used to generate new images, which hopefully look convincing enough to the human eye. We train our GANs for 500 epochs, using the default parameters in the notebook. The results can be seen in figures 7 and 14.

The CAE, VAE and GAN therefore share many components, but differ in their overall architecture. First of all, the CAE doesn't contain random nodes - it simply reconstructs input images from their latent representation. Both the VAE and the GAN are capable of outputting new images, whereas the CAE essentially only compresses and reconstructs images from the training set. The clearest difference between the two generative model architectures can be expressed in terms of their loss functions: where the VAE essentially uses the difference between the original and the reconstructed images (like the CAE), the GAN lets the Discriminator function as the arbiter of similarity between training images and the output.

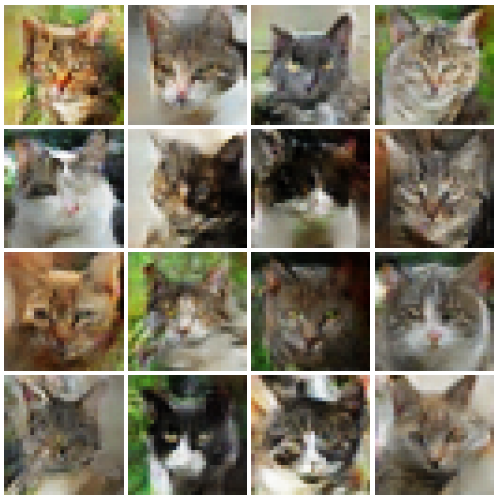

Figure 7: Sample of cats generated from random latent space vectors by our GAN after 500 epochs of training.

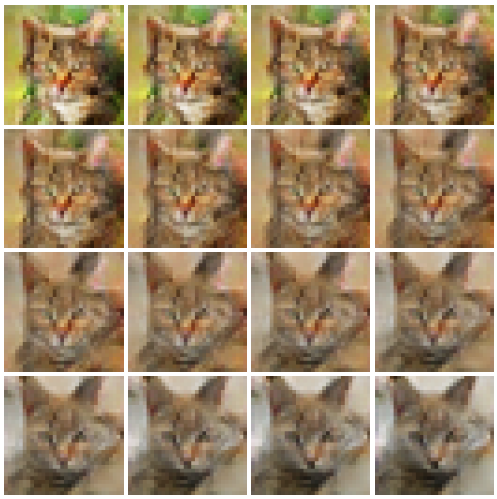

Figure 8: Interpolation between the top-left and bottom-right cats in figure 7, as produced by the Generator in our GAN.

The GAN and VAE approaches also differ strongly in terms of output. Where the output of the VAE tends to be blurry, the output of a GAN is often very erratic, with sharper features. This is most evident in the early phases of training. Furthermore, the output of VAEs tends to contain less variation when compared to GANs, which can also be seen to some extent in our results.

It is worth discussing our dataset choices briefly. We started by looking for $64 \times 64 \times 3$-sized images, and found the pixel-art set first. When applying the VAE model to these 'Tiny heroes', however, we noticed a distinct lack of variation in the generated images (see figure 9). We also noticed the poor quality of the images generated by the GAN, even after 100 epochs of training[3], as illustrated in figure 10. Even after double-checking the sampling process and varying the model parameters, we were unable to find a convincing explanation. It did turn out, however, that the dataset was itself generated using a generative model. Perhaps this could be the cause of our problems, but either way it makes the results less interesting to us. Next, we tried the pistachio set. The relatively simple basic shape seemed appealing, and the images seemed to maintain a lot of information after downsizing to 32×32. Application of the VAE to this dataset lead to another (more prozaic) problem, however: the VAE produced images managed to produce images of convincingly pistachio-shaped blobs, but consistently without the characteristic slit separating the shell hemispheres (see figure 13). We think that this has to do with the inconsistency in the location and prominence of the slit. Furthermore, there is no 'uncanny valley' in pistachios, which is something that is particularly interesting with generative models. So, we decided to look for other (preferably larger) datasets with features in more consistent places, containing 'uncanny valley'-susceptible subjects. It did not take long for us to land on the cats dataset. For a more technical run-down of the datasets (including links), see appendix A.

[3] in hindsight, this may have been exacerbated by the higher resolution of these images

A Datasets

The dataset we have been referring to as the 'cat face' dataset is actually part of a larger set of images meant for classification of animals. The set is called *Animal Faces*, and it contains 16 thousand images of animal faces in total, split among the categories *cat*, *dog*, and *other*, each containing roughly 5000 images (5153 *cat* images, to be precise). The images have a native resolution of 512×512, which is quite a lot more than our networks can handle, so significant downsampling was definitely needed. The dataset can be found at https://www.kaggle.com/datasets/andrewmvd/animal-faces.

The pistachio dataset is part of the *Pistachio Image Dataset* available at https://www.kaggle.com/datasets/muratkokludataset/pistachio-image-dataset/data. The full set contains just over 2000 600×600 images of pistachios in various orientations, split into the varieties *Kirmizi* (1232 images), and *Siirt* (916 images). Naturally, these two varieties of pistachio are wildly different, so we decided to only use the larger *Kirmizi* set.

The pixel-art dataset is called *TinyHeroes* dataset, and can be found at https://github.com/AgaMiko/pixel_character_generator. The full set contains 4×914 64×64 images of game sprites facing one of four different directions. We exclusively used the 914 front-facing images.

B Division of Labor

Task 1		Task 2		Task 3	
Code	Report	Code	Report	Code	Report
Laurens	Laurens	Jasper	Jasper	Jasper	Both

Table 12: Approximate division of labor for this assignment.

C Tiny Heroes

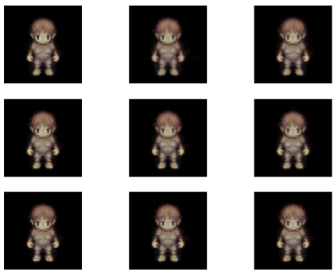

Figure 9: Pixel-art images generated by the VAE after 20 epochs. To our eye, all of these are identical.

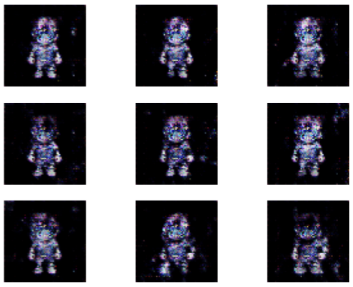

Figure 10: Pixel-art images generated by the GAN after 100 epochs. Note that neon zombies are not represented in the training set, so this is quite poor performance. Furthermore, the lack of variety in the samples is potentially problematic.

D The Pistachio Dataset

Figure 11: Pistachio images before and after downscaling.

Figure 12: Pistachio images (top) with their CAE reconstructions (bottom).

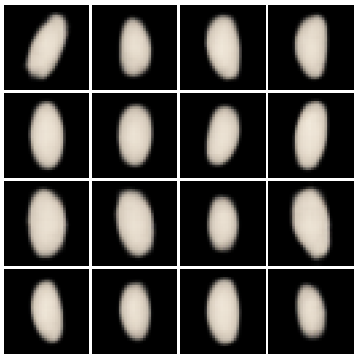

Figure 13: Pistachio images generated from random vectors in the latent space of the VAE (100 training epochs). The shapes look decent, but there is a distinct lack of those characteristic pistachio nut slits, so they mostly just look like fuzzy images of rocks.

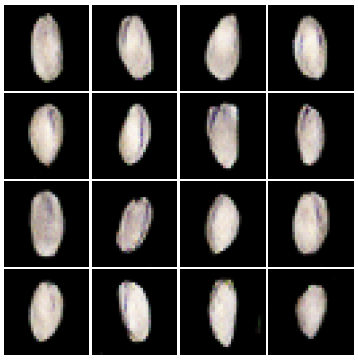

Figure 14: Positively mouthwatering pistachio images generated from random vectors in the latent space of the GAN, after 500 epochs of training.

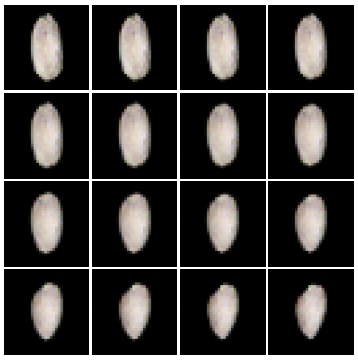

Figure 15: Interpolation between GAN generated pistachio images. Indeed, these are all quite convincing pistachios. Oh, how proud our parents must be.

E Bonus GAN Output

Because we are so proud of our GAN output, and we have a page to spare, please enjoy some more pistachios and cats!

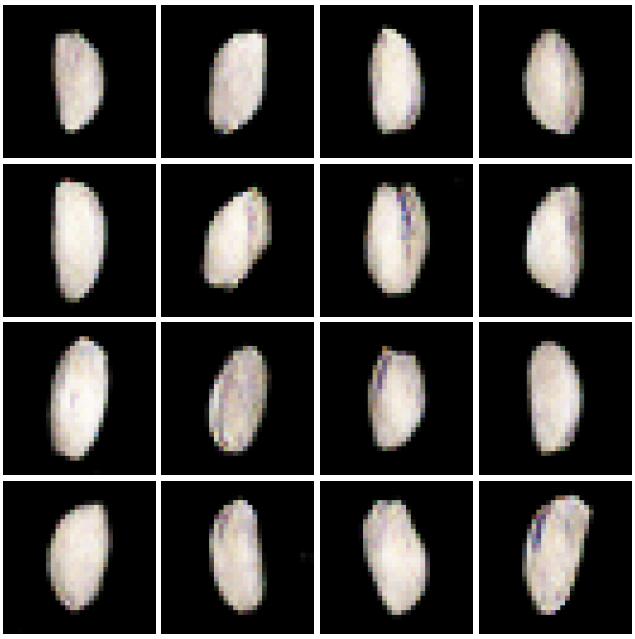

Figure 16: Pistachio samples

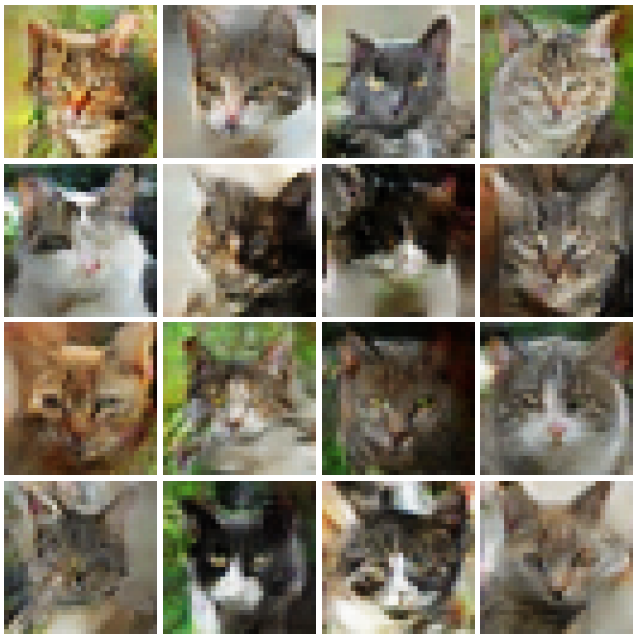

Figure 18: Cat samples

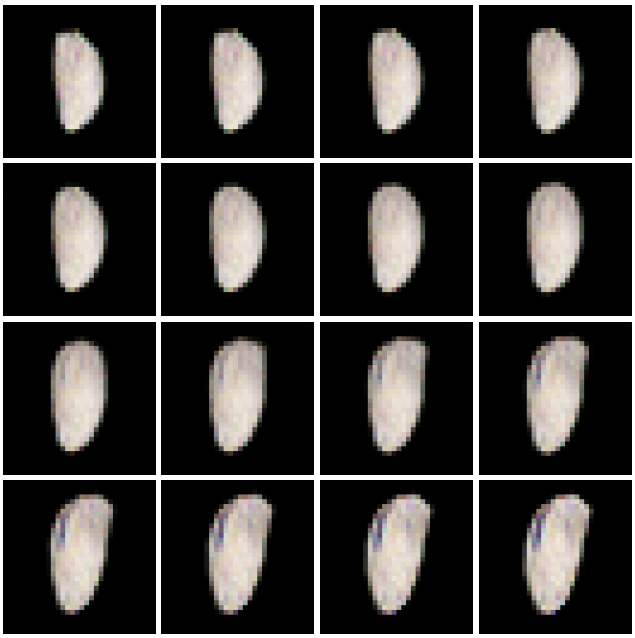

Figure 17: Pistachio interpolation

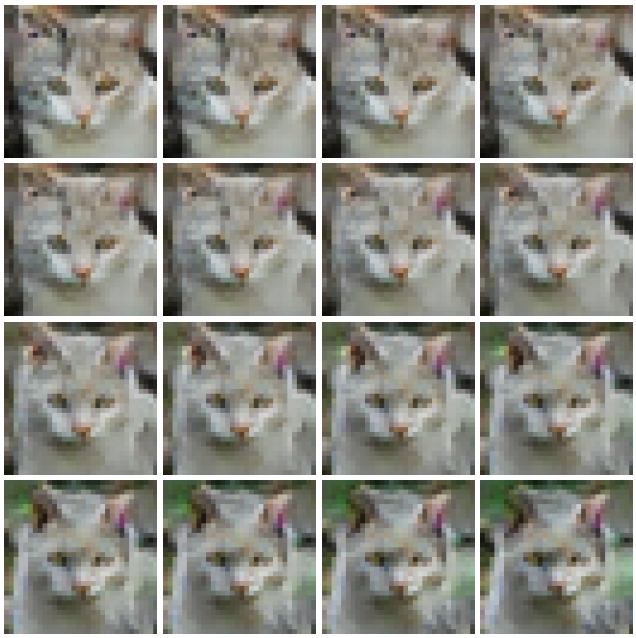

Figure 19: Cat interpolation