Introduction to Deep Learning - Assignment 1

Colin M. Poppelaars, Menno van der Eerden and Tim van der Vuurst - Leiden University

November 10, 2023

Introduction

The overarching theme of the assignment shown in this report is building neural networks (NNs) on image data namely for the purpose of classifying, predicting and generating. The tasks in this assignment are laid out to teach us how to build neural networks using APIs. In task 1, we will build multi-layer perceptrons (MLPs) and convolutional neural networks (CNNs) using the Keras and Tensorflow APIs to classify the fashion MNIST and CIFAR-10 datasets, two well-known datasets. The goal of this task is to build up some intuition while working with MLPs and CNNs. In task 2, we applied our newly gained intuition to predict the time shown on clocks in a dataset of clocks, which is done by building a custom CNN. In task 3, we train generative models such as Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs) on a custom dataset to generate new images that were not part of the original data.

1 Task 1 - Learning Keras API for TensorFlow

1.1 MNIST examples from Keras

The official Keras GitHub has lots of examples[1] of Keras implementations on various datasets, including two on the MNIST number data. They implemented both a simple multi-layer perceptron (MLP) and a convolutional neural network (CNN) and, even without optimal parameters, claim to achieve test accuracies of 98.40% and 99.25% respectively. One of the used tricks is implementing batches. This works in the following way. The full training set consists of x samples on which the network may be trained. We can however split this data into batches of size y ($1 \leq y \leq x$) and train our network on individual batches each epoch. If, for example, $x = 950$ and $y = 100$, we will have 10 batches where the final one only has the last 50 entries in the training set since x and y are not divisible without remainder. This has two distinct advantages. First, the memory usage is strongly limited in this way since the network is trained on smaller batches of data each time meaning it requires less memory to train. Second networks tend to train faster with batches since every epoch the weights are updated n times, where n is the number of batches instead of just once per epoch without batches ($x = y$). The biggest drawback of using this method however is that models tend to perform worse with smaller batch sizes, namely they perform worse in estimating the gradient during the optimization. This trick of batches with a batch size of 128 is implemented in the Keras examples as well, resulting in the training of the MLP taking only about 1 minute and the training of the CNN of approximately 3 min on Google Colab with TPU acceleration.

1.2 Fashion MNIST and CIFAR-10

Here, we implement an MLP and CNN on the MNIST Fashion data and experiment to find the top 3 working hyperparameters for each of these models and subsequently implement these models on the CIFAR-10 data. First, we must introduce the data[2].
The MNIST Fashion data consists of 70,000 28x28 pixel gray-scale images of articles of clothing in 10 different categories, with pixel values between 0 and 255. Keras has implemented a function to load this data, automatically splitting it into 60,000 training instances and 10,000 test instances. Similarly, the CIFAR-10 data consists of 60,000 32x32 pixel colour images of various modes of transportation (such as aeroplanes) and animals, totalling 10 mutually exclusive classes. Keras splits this into 50,000 training instances and 10,000 test instances. For more detailed information such as the distinct classes and origin of the data, we refer to the respective websites of the data.
We built the MLP inspired by the example set by A. Geron[3] on the same data. The same holds for the generic set-up of a CNN using Keras. There is, however, a wide range of hyperparameters that may create the optimal models. In order to search for these we applied a brute-force method where we iterate over various combinations of batch size, number of epochs, optimizers, and learning rates (η) as presented in table 1. Other than this we experimented with various layers and nodes in these layers to see which combination worked best.

η	**Batch size**	**Epochs**	**Optimizer**
0.0005	32	5	Adam
0.001	64	10	Adamax
	128	20	RMSProp
	256	30	SGD
	512		Adagrad
			Adadelta
			Nadam

Table 1: Various values of hyperparameters we experimented with. Note that order does not hold meaning in this table. Here η is the learning rate.

[1] Which may be found here.
[2] The MNIST Fashion data may be found here, the CIFAR-10 data may be found here.
[3] See Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, chapter 10 (MLP) and chapter 14 (CNN).

1.3 Results

Here we present the results of the training of models on the Fashion MNIST data and subsequently the CIFAR-10 data.

1.3.1 Fashion MNIST

We built an MLP first with two hidden layers with 300 and 100 nodes and ReLU activations. From this, we find that the best-performing model is one with the Adam optimizer, $\eta = 0.0005$, a batch size of 64, and 10 epochs yielding a test accuracy of 0.877. However, we found that removing the hidden layer with 100 nodes improved the model, now reaching a test accuracy of 0.883 after hyperparameter optimization. Adding a dropout layer after each hidden layer in the MLP architecture with two hidden layers, each with a rate of 0.5, also improved the model with it now reaching a test accuracy of 0.884. Removing one of these dropout layers and keeping just one of them after the first hidden layer turned out to be even better, reaching a test accuracy of 0.888 with the Nadam optimizer, $\eta = 0.0005$, a batch size of 128 and 30 epochs. Combining these experiments into one, by creating an MLP with just one hidden layer of 300 nodes and one dropout layer with a rate of 0.5, yielded slightly inferior results with the optimal found model yielding a test accuracy of 0.885. Moreover, we experimented with L1 regularization, adding this to the 300 node layer in our best model architecture so far. Performing hyperparameter optimization on this yielded that the best model we found had a test accuracy of 0.769 with L1 regularization, suggesting that L1 regularization is not fruitful to better results here. All in all the best MLP we found was one with two hidden layers, one with 300 and the next with 100 nodes, with a dropout layer between them with a rate of 0.5. This model reached a test accuracy of 0.888.

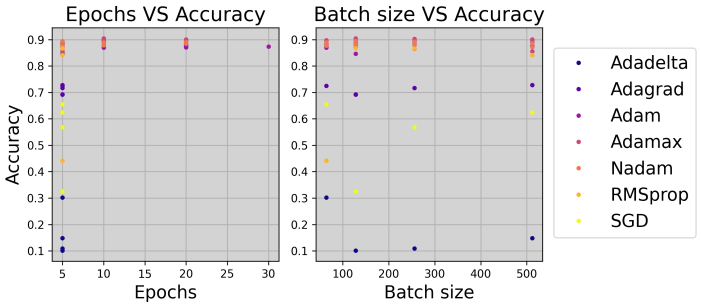

Figure 1: The evolution of the accuracy for various CNNs (discriminated by their optimizer, no batch normalization) as a function of epochs (left) and batch size (right).

However, the top three models we found were all CNNs, with some of the more mediocre CNNs outperforming even the best MLP. We made the CNNs with an architecture as follows. First, we begin with an input layer with ReLU activation and 'same' padding, followed by a MaxPooling layer which reduces the dimensionals by a factor 2 (this factor 2 will be the standard throughout this assignment unless specifically stated otherwise). Then there are two blocks of two layers of convolution layers and MaxPooling. All convolutional layers use ReLU and 'same' padding. The output is then flattened and put through two blocks of a dense layer (with ReLU) and a dropout layer with a rate 0.5. Finally, there is an output-dense layer with just 10 nodes - one for each possible output of the data. The top results of CNN runs (with various hyperparameters) are visible in table 2, with the full list of runs visible in Appendix A. The effect of specifically the number of epochs and batch size on our CNN is visible in figure 1. Here we can see that for most optimizers, the accuracy grows with both epochs and batch size and eventually seems to converge. This is in accordance with expectations - if you train more often you should get better results and a larger batch size usually yields better results per the reasoning given before. In order to reduce the runtime we experimented with batch normalization (BN)[4], which is the process of re-centering and re-scaling batches of data. It is shown to help speed up training, as well as increase the performance as we apply a transformation that maintains the mean output close to 0 and the output standard deviation close to 1. When applying this after a convolution layer we make details more apparent as opposed to the total picture. In addition, batch normalization acts as a regularizer by reducing the possibility of overfitting. Results of various runs using a BN layer right before flattening the output of the CNN are shown in table 3. As shown by the results of table 2 and 3 CNNs with BN do not outperform those without, meaning our top three networks are those with the architecture described previously (thus without BN) and the hyperparameters as specified in table 2.

Optimizer	η	Batch size	Epochs	Accuracy
Adamax	.0005	128	10	0.905
Adamax	0.001	256	10	0.903
Adamax	0.001	512	20	0.900
Adamax	.0005	64	10	0.898
Adam	0.001	128	10	0.896
Adam	0.001	256	10	0.896
Adamax	0.001	64	20	0.896
Nadam	0.001	128	5	0.895
Adamax	.0005	512	10	0.894
Adam	.0005	512	10	0.893

Table 2: Truncated view (10 highest accuracies) of tried hyperparameters for the CNN without batch normalization ordered on highest achieved test accuracies. For full table see appendix A.

Optimizer	η	Batch size	Epochs	Accuracy
Nadam	0.001	256	10	0.872
Nadam	0.001	128	10	0.862
Nadam	0.001	512	10	0.862
Adamax	0.001	64	10	0.851
Adam	0.001	64	10	0.850
Adam	0.001	256	10	0.838
Nadam	0.001	64	10	0.838
Adam	0.001	512	10	0.828
Adam	0.001	128	10	0.822
Adamax	0.001	128	10	0.820
Adamax	0.001	512	10	0.801
Adamax	0.001	256	10	0.795

Table 3: Various hyperparameters used with batch normalization on the CNN for Fashion MNIST data, ordered on highest achieved test accuracy.

[4] Ioffe, S. & Szegedy, C.. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.

1.3.2 CIFAR-10

After finding the best-performing models on the Fashion MNIST data we experiment with different image data to see how they perform. We do have to make a slight alteration in the input layer of the architecture, as the CIFAR-10 data does not consist of 28x28x1 images but 32x32x3 - slightly larger and RGB instead of gray-scale. Other than this, the architecture of the CNNs given the CIFAR-10 data is the same as described previously for the Fashion MNIST data. The results can be found in table 4.

Optimizer	η	Batch size	Epochs	Accuracy
Adamax	0.0005	128	10	0.4854
Adamax	0.001	256	10	0.4563
Adamax	0.001	512	20	0.5249

Table 4: Hyperparameters and test accuracy of the three best-performing models on the Fashion MNIST data on the CIFAR-10 data, ordered on best performing on Fashion MNIST data.

It can be seen here that the models perform terribly on this new data. The main reason for this is the fact that the CIFAR-10 data is quite different from the Fashion MNIST data in various ways. First off, quite simply put, articles of clothing are fundamentally different from various vehicles and various animals. The image classes are not only fundamentally different but also more complex - animals and vehicles have more and more detailed defining features than clothing does. Not only that, but the model now also needs to be able to distinguish between vehicle and animal. Finally, the images are more complex now being 32x32x3 instead of 28x28x1 - thus drastically increasing the dimensionality. This may lead to the models not picking out enough differences and nuance in the RGB values and the details of the image with the given hyperparameters. All these reasons contribute to the model not performing well on this data. The hyperparameters were tuned to much less complex data. As we can see in table 4, the model that performs best is one with a higher batch size and the number of epochs (the higher learning rate can not be confidently said to improve results, as the second model performs worst than the first with a higher learning rate and the same amount of epochs). However, finding the exact optimal hyperparameters is beyond the scope of this assignment.

1.4 Conclusions

We can conclude that CNNs outperform MLPs on the Fashion MNIST data by a significant margin after hyperparameter optimization. However, these chosen hyperparameters do not translate well to new, more complex data (CIFAR-10) yielding test accuracies of $\lesssim 0.5$ after training CNNs.

2 Task 2 - Tell-the-time network

For the second task of this assignment, we aim to create a CNN that can predict the time as correctly as possible. To do this, we have a big collection of images presented. These images contain analogue clocks and corresponding labels that tell the time presented in the images. In Figure 2, an example of an image is presented.

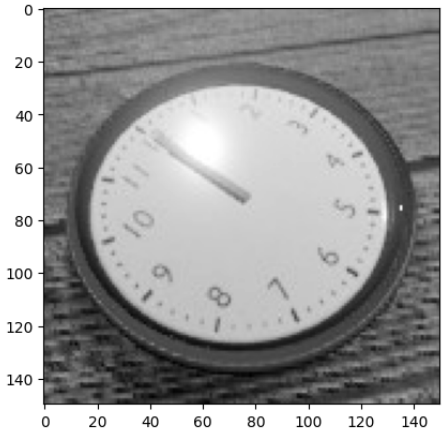

Figure 2: Example of an instance from the dataset. The corresponding label is (0,1), implying 1 past 12 or 00:01.

From Figure 2, we can see that the instance also has some flash or reflection on the clock. In addition, the clock is also rotated, and the image is not taken directly above the clock. All instances in the dataset have some rotation and flash as the example in Figure 2, making it more challenging to tackle the problem of predicting the time, as we use most of these samples to train our CNN. We use 80% of the dataset for training and 20% for testing/validation. In order to have reproducible results we use a random state, this is used during the splitting of training and test sets. The entire dataset consists of 18000 gray-scale images which have the shape of 150x150.

The problem of correctly predicting to tell the time can be formulated in many ways, i.e. multi-class classification, regression, or multi-head. To make such models we must adapt our labelled set of instances. We do this by changing the representation of the labels.

2.1 Common sense time

To account for the common sense error between predicting the time we implemented a function that converts the true and predicted time into minutes at first. After this, it calculates the minimal difference between $y_{true} - y_{pred}$ and $y_{pred} - y_{true}$. This is in units of minutes and uses the '% 720' operator to ensure that the result is within a 12-hour cycle (720 minutes) by accounting for time loops, i.e. the difference between 11:00 PM and 1:00 AM is 120 minutes, not 10 hours.

2.2 Classification

2.2.1 Label representation

The first approach is to treat our problem as an n-class classification problem. This implies treating all unique instances as single classes. Therefore, we have 720 classes in total, since there are 60 minutes in an hour and 12 hours in total ($12 \times 60 = 720$). In order to create the 720 classes we transform our labels using the // operator, which is used for integer division. It divides one number by another and returns the largest whole number that is less than or equal to the result. The result of 'a // b' is an integer, and any fractional part of the division is truncated. We use it to convert the hours of an instance to minutes, i.e. 60//h. In addition, we add the

minute of a label to it. In Figure 3 we present the result of transforming our labels to 720 unique classes. We can see that each class is equally distributed as the others. Each minute has an equal amount of representation.

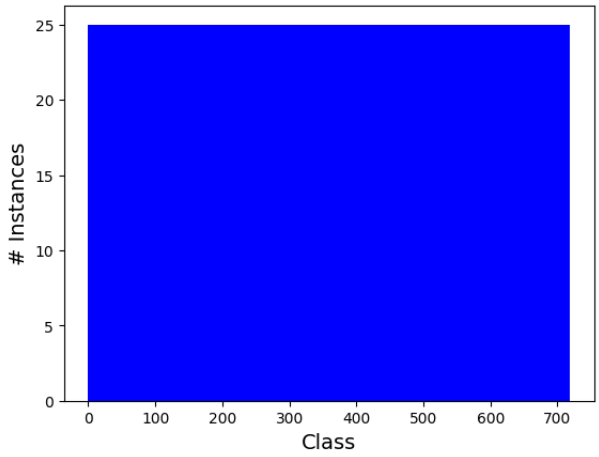

Figure 3: Representation of classes using 720 unique labels for all available minutes on an analogue clock. Each class has 25 instances associated with it.

However, when using such a label representation, we have very few instances per class, in total only 25. This makes it very difficult to predict all classes correctly for our CNN since the representation is rather low. Moreover, it is also computationally expensive as we have 720 unique classes to which an instance can be labelled. We split our data into a training and test set using 'train_test_split' from the Scikit-learn library. Furthermore, we use a random state to allow for reproducibility.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 150, 150, 1)]	0
conv2d (Conv2D)	(None, 150, 150, 16)	160
activation (Activation)	(None, 150, 150, 16)	0
batch_normalization (BatchNormalization)	(None, 150, 150, 16)	64
max_pooling2d (MaxPooling2D)	(None, 50, 50, 16)	0
dropout (Dropout)	(None, 50, 50, 16)	0
conv2d_1 (Conv2D)	(None, 50, 50, 32)	4640
activation_1 (Activation)	(None, 50, 50, 32)	0
batch_normalization_1 (BatchNormalization)	(None, 50, 50, 32)	128
max_pooling2d_1 (MaxPooling2D)	(None, 25, 25, 32)	0
dropout_1 (Dropout)	(None, 25, 25, 32)	0
conv2d_2 (Conv2D)	(None, 25, 25, 32)	9248
activation_2 (Activation)	(None, 25, 25, 32)	0
batch_normalization_2 (BatchNormalization)	(None, 25, 25, 32)	128
max_pooling2d_2 (MaxPooling2D)	(None, 12, 12, 32)	0
dropout_2 (Dropout)	(None, 12, 12, 32)	0

Figure 4: Summary of our CNN before flattening the classification model.

2.2.2 Model

We begin the construction of our classification model with an input layer. This layer is then followed by blocks of a convolution, activation (ReLU), batch normalization, and MaxPooling layer, which is then followed by a dropout layer. We use this block structure to obtain high performance as well as keeping the number of parameters relatively low. We first take a convolution such that neurons are not connected to every single pixel of our input image, but to only pixels in their receptive fields. This allows our network to concentrate on small low-level features. We use a padding of 'same' for all convolutional layers. Furthermore, we use BN to speed up training.

After batch normalization we use a MaxPooling layer to reduce the dimensionality of our image, the goal is to shrink the input image to reduce the computational load and number of parameters needed throughout the model. MaxPooling downsamples the input along its spatial dimensions, height and width. It does so by taking the maximum value over an input window for each channel of the input. Finally, we end a block using dropout to reduce overfitting.

In Figure 4 we present the structure of our model before the flattening layer. It can be seen that the first block of convolution, batch normalization, MaxPooling, and dropout has a convolution of 3x3 with 16 filters, a MaxPooling of 3x3, and a dropout of 0.25. Furthermore, the second and third blocks have 32 filters with size 3x3 for convolution, MaxPooling of 2x2, and a dropout of 0.25.

After three of the above described blocks we flatten our network, leading us to the fully connected layers of our network. We have 2 blocks in this layer. Each block has a structure of first a dense layer with a ReLU activation function, followed by a batch normalization layer and a dropout layer of 0.25. The dense layer in block 1 has 144 neurons and the one in the second block has 60 neurons. The final dense layer consists of 720 neurons, one per classification class. This final layer has a Softmax activation function, such that each class has some probability per instance (adding up to 1). The class with the highest probability is the class to which our image will be labelled to. In Figure 5 we present the final structure of our classification model after flattening the structure of Figure 4. The total number of parameters is 731500, of which 730932 are trainable. We compile our model using a sparse categorical cross-entropy as the loss function, since we have a multi-class classification problem. We use the Adam optimizer using a learning rate of 0.001 and have 100 steps per execution. Adam has a particularly high performance and fast convergence, allowing us to fit our model relatively quickly. As a metric we use accuracy. In the training phase, we use a batch size of 32 to prevent a loss of detail, which we can afford when using Adam. Furthermore, we re-scale the input images to a 0 to 1 scale, this allows our model to converge faster, as well as make the input images less complicated. In total, we run 100 epochs. After each epoch, we shuffle our training set to increase generality.

```
flatten (Flatten)            (None, 4608)              0

dense (Dense)                (None, 144)               663696

activation_3 (Activation)    (None, 144)               0

batch_normalization_3 (Batc  (None, 144)               576
hNormalization)

dropout_3 (Dropout)          (None, 144)               0

dense_1 (Dense)              (None, 60)                8700

activation_4 (Activation)    (None, 60)                0

batch_normalization_4 (Batc  (None, 60)                240
hNormalization)

dropout_4 (Dropout)          (None, 60)                0

dense_2 (Dense)              (None, 720)               43920

activation_5 (Activation)    (None, 720)               0

=================================================================
Total params: 731,500
Trainable params: 730,932
Non-trainable params: 568
```

Figure 5: Summary of our CNN after flattening the classification model.

2.2.3 Results

We fitted the model described in the previous section. The amount of time required for this was 1 hour and 2 minutes. The first epoch has a loss value of 6.640, accuracy of 0.0016 and validation loss and accuracy of 15.64 and 0.0014 resp. The final epoch (100) has a loss value of 1.556, accuracy of 0.5132 and validation loss and accuracy of 2.761 and 0.2275 resp. In Figure 6 we present the growth of accuracy for the training and testing sets over all epochs. It can be seen that both cases grow with respect to the number of epochs, implying our model is stepwise increasing its predictability of telling the time.

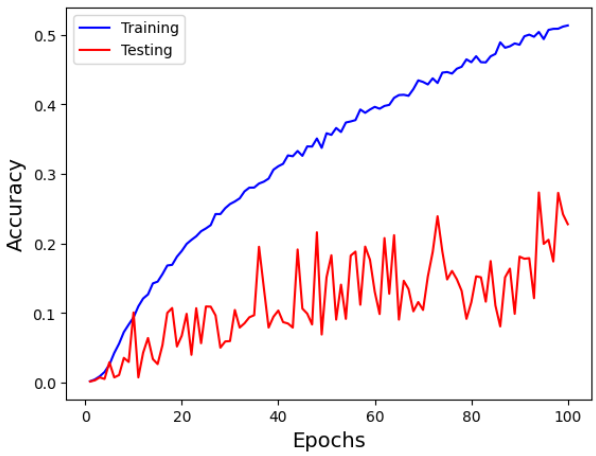

Figure 6: The growth of training and testing accuracy of our classification CNN over each epoch.

When we predict all instances of our dataset we correctly classify 11204 instances, implying 62.2% correctly classified instances. Lastly, in Figure 7 we present the common sense error distribution. We report an average common sense error of 24 minutes.

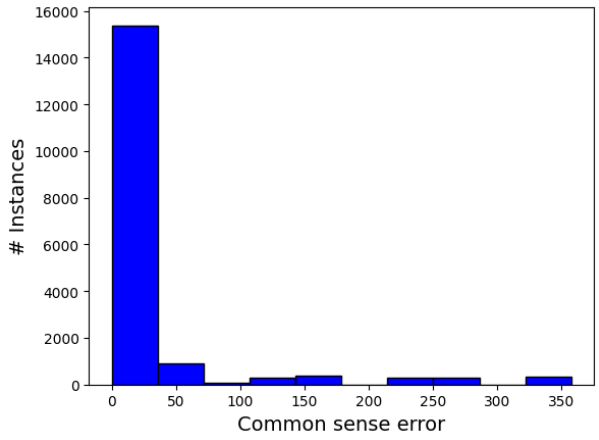

Figure 7: Common sense error distribution of our classification model for telling the time.

2.3 Regression

2.3.1 Label representation

For our regression model, we need to represent our labels in a different way as to how they were handed out. We aim to create continuous labels between 0 and 12. For example (5,30) has a continuous value of 5.5, which shows that the hours are unchanged, but the minutes are converted to units of hours. To do this for all labels we simply divide the minute of each label by 60. In addition, we use a round of 3 as we do not need to use all decimals for our model, this improves the convergence of our model as we do not over-specify the labels. In total this results in 720 unique labels. We present the histogram of our label transformation in Figure 8, and we can clearly see that each hour is equally distributed. Each hour has a total of 1475 instances, i.e. between 0 and 1 there are 1475 clock images.

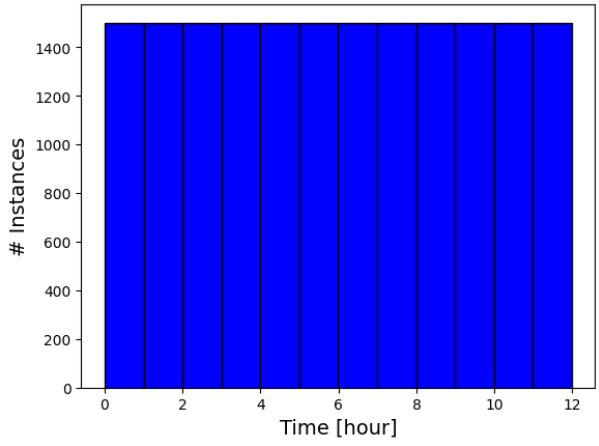

Figure 8: Represenation of classes using a continuous value.

2.3.2 Model

flatten_1 (Flatten)	(None, 4608)	0
dense_3 (Dense)	(None, 128)	589952
activation_9 (Activation)	(None, 128)	0
batch_normalization_8 (BatchNormalization)	(None, 128)	512
dropout_8 (Dropout)	(None, 128)	0
dense_4 (Dense)	(None, 1)	129
regression_output (Activation)	(None, 1)	0

Figure 9: Summary of our regression CNN after flattening.

As our regression model, we create a similar model to that described in section 2.2.2. We create three blocks of convolution, batch normalization, maxpooling, and dropout. The variables are the same as in the classification model. The only difference is found after flattening our model. After flattening, we continue in descending order of nodes of 128 to 1 for our dense layers. The dense layers each have a ReLU activation function. The final layer only has 1 node as we have a regression model, therefore we only want one output value. In addition, the final layer has a linear activation function. This does not change the output but is required for the model to compile. Furthermore, between the layers, we have a batch normalization and dropout layer of 0.5 to reduce overfitting. The summary of our model after flattening is presented in Figure 9. In total, the regression model has 604961 parameters, of which 604545 are trainable. Lastly, we use a mean squared error (MSE) loss function. From Figure 8 we can see that our distribution of labels is uniform, therefore an MSE loss function is applicable. In addition, we utilize the optimizer Adam with a learning rate of 0.0005.

2.3.3 Results

Below we present the results of our regression CNN. In Figure 10 we see the mean absolute error per epoch for the training and testing set. It can be seen that both follow a downward trend. However, the testing set does behave rather chaotic over each epoch. Nevertheless, it does seem to follow the training set. This is expected as the testing set only contains 20% of the dataset, as well as that we are looking at the mean absolute error here. This is defined as

$$\mathrm{MAE} = \frac{1}{N}\sum_{i=1}^{N} |y_{\mathrm{pred,i}} - y_{\mathrm{true,i}}|. \quad (1)$$

Consequently, as the testing set is smaller, it is more difficult to have a lower MAE as we are dividing by a smaller sample as opposed to the training set. In addition, since the test set is not used during training we expect the performance to be worse than on the training set.

In Figure 11 we present the absolute errors in a histogram for better visualization. For this, we use the entire dataset but use separate colours for the training and test sets respectively. It can be clearly seen that the majority of both sets lay between 0 and 2. Moreover, it can be seen that the test distribution varies somewhat from the training distribution, this indicates that the model had difficulty predicting the test set.

Furthermore, the mean value of all residuals is 1.63 seen in Figure 11, indicating that on average the error is about 98 minutes. Lastly, in Figure 12 we present the common sense distribution of our regression model. We report an average common sense error of 97 minutes.

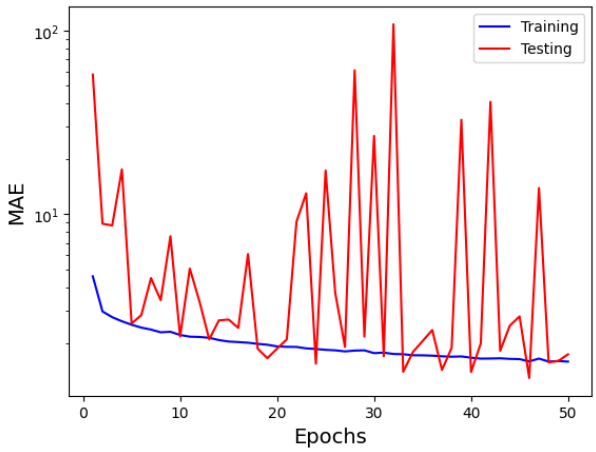

Figure 10: Performance of our regression CNN presented by the mean absolute error (MAE) per epoch.

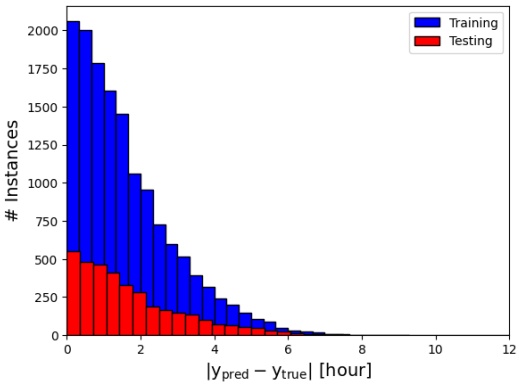

Figure 11: Mean absolute value of all instances for our regression CNN of task 2. In blue and red the training set and test set instances are indicated respectively.

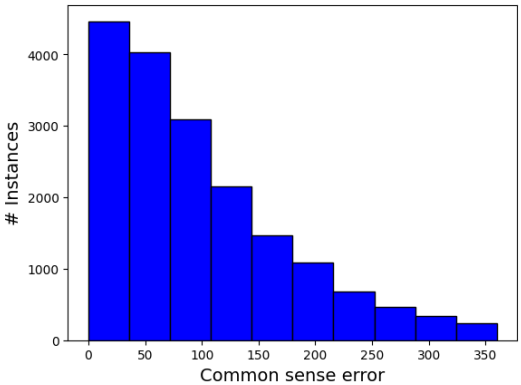

Figure 12: Common sense error distribution of our regression model for telling the time.

2.4 Multi-head

2.4.1 Label representation

In the case of a multi-head CNN, we need to have two sets of labels. In our model, we have chosen to predict minutes using regression and hours using classification. We have seen from the previous models that the combination of full classification and regression can be difficult for the models implemented. We only have 12 classes to predict hours for classification, as opposed to 720 in the classification case. In addition, when predicting minutes, we now have predictions between 0 and 60, instead of between 0 and 720 (the total amount of minutes between 00:00 and 11:59) as in the full regression model. The aim of using a multi-head model is to reduce the complexity by creating two branches that solve a task separately. When compiling the model, the two branches are connected through the input layer.

To have a correct label representation, we use the labels presented as they are in the dataset. We can do this as we want the hours to be integers from 0 to 11, and we want minutes to be between 0 and 59. By default, we already have these labels. However, we do need to create two different training and testing sets for our model, as we aim to have two separate solvers. To achieve this we split our data and labels as done in the previous models, only this time having an additional labelled set.

2.4.2 Model

We define our multi-head CNN using a class in Python. This allows us to build two separate branches at first, which we can later merge when defining the whole model. We create two branches, minutes and hours.

As inspiration for the minute branch, we use the model from the full regression, described in subsubsection 2.3.2. The branch is composed of convolutions, followed by MaxPooling and batch normalization. The first block has a 10x10 convolution layer with 64 filters and a 2x2 MaxPooling layer. The following layers before flattening only differ in the size of the convolutional layer, which is 5x5 instead of 10x10. After this, we flatten our model and define a sequence of dense and dropout layers. The neurons go from 128 to 64 to finally 1 neuron. In between these steps, we have added a dropout of 0.5 to account for overfitting.

For our hour branch we start with 3 blocks of convolution, batch normalization, MaxPooling and dropout. We use the same structure of the classification model described in subsubsection 2.2.2, but differ in the final dense layer, where we use 12 neurons instead of 720. In our case, we aim to predict the hour classes instead of all minutes available between 00:00 and 11:59.

We compile our model using sparse categorical cross-entropy for our hour branch. For our minute branch, we use the mean squared error. We use optimizer Adam with a learning rate of 0.001. Furthermore, we use the accuracy and mean absolute error as metrics for the hour and minute branches respectively. We run for 50 epochs, together with a batch size of 32 to prevent detail loss. Finally, our model has a total of 1675193 parameters, of which 1674113 are trainable. Due to the size of our model, we present the full image in Figure 19, found in the appendix.

2.4.3 Results

While fitting our multi-head model, the hour loss went down from 2.7477 to 0.4509 and the minute loss went down from 415 to 35. In addition, the training accuracy of hours increased from 0.0930 to 0.8299. The mean absolute error of minutes decreased from 16.8981 to 4.2592. The total training time was roughly 3.75 hours. We present the metric evolution in Figure 13 for the training and test sets of the classification branch.

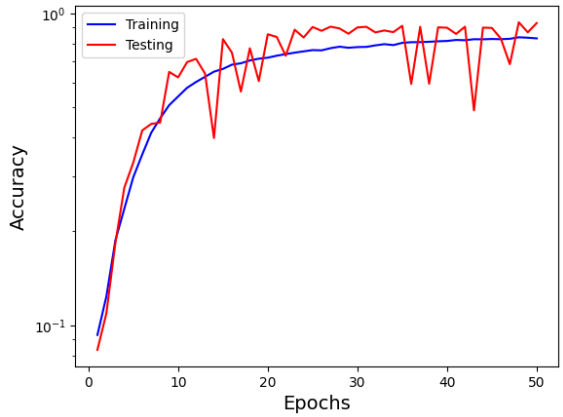

Figure 13: Evolution of the accuracy metric for the hour class.

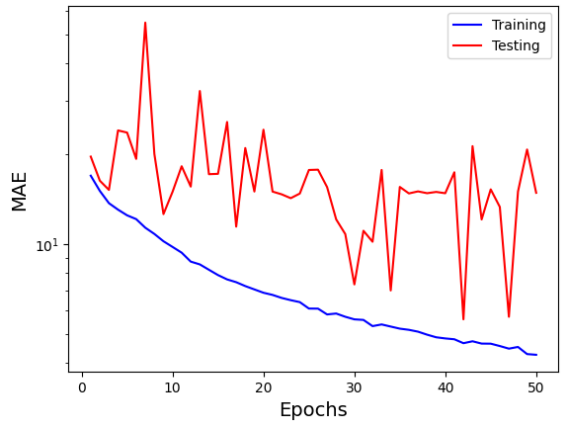

Figure 14: Evolution of the mean absolute error (MAE) metric for the minute branch.

It can be seen that the classification branch has significant performance on both the training and test sets. From our classification task, we find that out of all 18000 instances, 17455 were correctly classified, of which 3345 belonged to the test set. However, as seen in Figure 14, the regression branch has difficulty obtaining good performance on the test set. A possible explanation is that the loss function had not converged yet, however, considering the running time, it is not recommended to push the number of epochs further. Furthermore, it should be taken into account that the test set only contains 3600 instances, while the metric used for our regression case is the mean absolute error, defined in Equation 1. Consequently, we divide by a smaller population number (N) than the training set's size.
In Figure 15 we present the histogram of the evaluation of our regression of the multi-head model. We see that the absolute error is between 0 and 45. In addition, the training and testing sets both

seem to follow the same distribution roughly, despite the difference in the number of instances. Therefore, we find an explanation for the result in Figure 14. The mean value of the residuals is 14.6 minutes and the number of residuals between 0 and 10 is 6573. Lastly, in Figure 16 we present the common sense error distribution for our multi-head model. We report an average common sense error of 16 minutes.

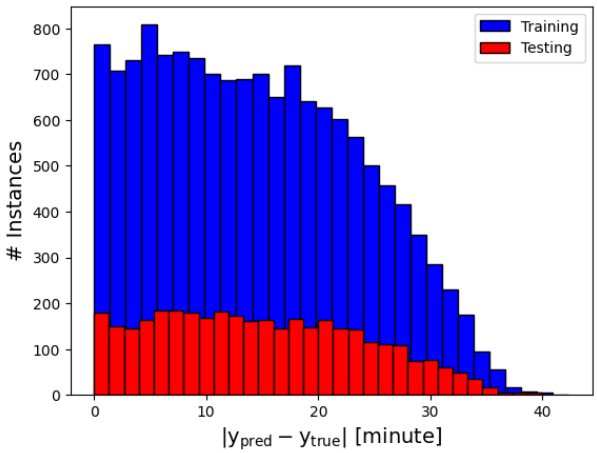

Figure 15: Mean absolute value of all instances for our regression task in the multi-head model. In blue and red the training set and test set instances are indicated respectively.

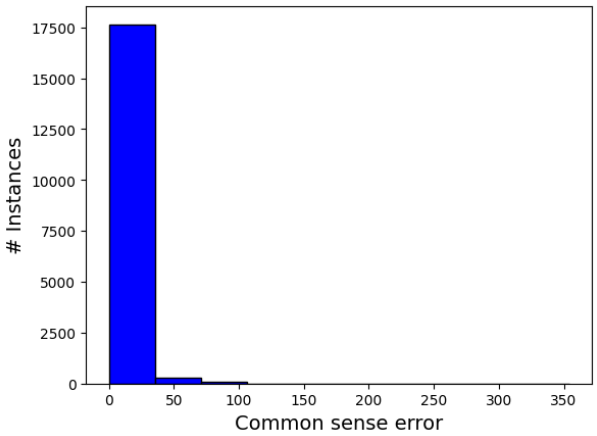

Figure 16: Common sense error distribution of our multi-head model for telling the time.

2.5 Conclusion

From the various models applied to this task, we see that in the multi-head model, the classification task is executed to great performance, having a performance of 83% and 93% on the training and test sets respectively. When we compare this to the sole classification model of 720 classes we see that it finds difficulty in assigning the right classes to instances. It shows that there is an upper limit to the number of classes we can use when applying classification to solve the problem of telling the time. In addition, regression seems to be struggling throughout the problem in comparison to the performance of the other models. A potential reason for this is that we were using units of hours, therefore the number of minutes for an instance indirectly became less important for the regression prediction. By using the multi-head problem we were able to shrink down the average error of 98 minutes in the sole regression model to 14.6 minutes. This is a decrease of 85% in residuals ($|y_{pred} - y_{true}|$). Consequently, the multi-head model seemed to solve the problem of telling the time the best. Potentially a double classification model could solve the task with greater performance, but this is beyond the scope of this report. Lastly, the common sense performance of our models shows that indeed the multi-head model is the best at predicting the time throughout the 3 proposed methods. We found an average common sense error of 16 minutes for the multi-head model, as opposed to the 24 and 97 minutes for classification and regression respectively.

3 Task 3

In this task, we train both a Variational Auto-Encoder and a Generative adversarial network (GAN) on found data and use them to generate novel data. First, we must go over the data and how the networks work.

3.1 Data & pre-processing

The dataset we used to generate new images is called the 'New Plant Diseases Dataset', which we found on the website Kaggle. This is a website where users can attempt machine learning problems, like building the perfect model for a (real-world) classification problem. The dataset[5] consists of more than 80.000 images (JPG) of both healthy and diseased leaves. These images are categorized into 38 different classes and given labels (e.g. 'Apple___healthy' and 'Tomato___Bacterial_spot'). We chose to use this dataset because, like the dataset of faces, it is relatively uniform and it can easily be downloaded from Kaggle. The entire dataset is too big for our purposes and the images are in JPG format rather than the npy format that we are used to. Thus, we decided to reduce the size of the dataset and change the format by pre-processing the data.

3.1.1 Pre-processing

The data that we start with has 38 classes, which is a subdivision of the 14 crops that are used in the dataset. We chose to use all crops, but limit the number of classes per crop to 5, which gave us 34 different total classes. We wanted a dataset of roughly 10.000 images, which would give us a large enough sample to test the image generation while not being too large for the colab environment that we used. We chose 300 images per class, which gave us a total of 10.200 images. Each of these images is a 256 by 256 JPG with RGB colours, which gives a total of $256 \cdot 256 \cdot 3 = 196608$ components per image. Due to the constraints of the environment in which we run our code, we had to reduce this to 32 by 32 images with RGB colours, which has only 3072 components per image. This was done using the `image` module of the `matplotlib` library. The method `imread` takes a JPG and turns it into a correctly shaped array, which was sliced to reduce the images to the 32 by 32 RGB format that we wanted. Afterwards, we divide the

[5] Which may be found here.

components of the images by 255, to reduce the range of the components from [0..255] to [0..1]. Finally, all images are put together in `images.npy` and all labels are put together in `labels.npy`.

3.2 Applying the decoder/generator

First, we must go into the workings of both VAEs and GANs before showing their results.

3.2.1 Variational Auto-Encoders

The variational auto-encoder, or VAE, is an extension of the classical convolutional auto-encoder, or CAE. In essence, the VAE is comprised of an encoder, a sampler, and a decoder. The encoder takes an input (an image in our case) and outputs a vector with twice the dimensions of the latent space. The first half of the values represent the mean vector of the latent space vectors and the second half of the values represent the log of the variance of the latent space vectors. These two vectors, $\vec{\mu}$ and $\vec{\sigma}$ are combined to form the distribution of the latent space by $\vec{z} = \vec{\mu} + \vec{\sigma} \odot \vec{\varepsilon}$. The $\vec{\varepsilon}$ can be thought of as random noise used to maintain the stochasticity of the latent space vector. We take this to be the normal distribution with unit variance and zero mean. In the end, this results in a latent space vector given by a normal distribution with variance $\vec{\sigma}^2$ and mean $\vec{\mu}$. The decoder takes the latent space vector and outputs, what should be, the exact input image that we put into the VAE. The model is trained by setting the inputs and the expected outputs of the VAE to the same thing. This will train the model to find a latent space representation that will represent our images.
The VAE can be used as a generative model and will be used to generate new images from the dataset that we have processed. We can make new images by sampling from the distribution of the latent space and decoding this new latent space vector.

3.2.2 Generative Adversarial Networks

Given a dataset of images, GANs work as a combination of two models: the generative model and the discriminative model. The discriminator works to distinguish real images from fake - generated - ones. This means it is effectively a binary classifier answering the question: is the image from the original data? The generator in turn works by taking N-dimensional random "noise" vectors as input and, through a deconvolutional network, transforming them into images. While iterating over the training set, the weights for the discriminator are updated through backpropagation in order to successfully distinguish real and fake images. The weights in the generator are simultaneously updated similarly through feedback from the discriminator, such that the generator becomes better at generating images that the discriminator cannot properly classify. In this way, the learning process becomes essentially a zero-sum game where the discriminator and the generator try to outclass each other at every epoch - one player's gain is the other's loss.

In more detail, the discriminator is a basic CNN that has an input layer of Conv2D with 128 filters. Subsequently, there are 3 layers of Conv2D with each a kernel size of (3,3), stride of (2,2), 'same' padding and ReLU activation. This final output is then flattened and put through one more Dense layer that outputs a single number with sigmoid activation which predicts whether the input was real or generated.
The generator in turn works via a deconvolutional network that takes in a latent vector of a certain dimensionality, which we will call N or the latent dimensionality. The structure for this network was as supplied in the assignment, with first a Dense layer to take in the latent vectors. This is then reshaped to (4,4,32) to match the output of the downsampling architecture. Then 3 upsampling layers consist of Conv2DTranspose layers in the Keras API, with a (3,3) kernel size, a stride of (2,2), a padding of 'same' and ReLU activation. Finally, to convert back to a 3-channel RGB image, a Conv2D layer with 3 filters a (3,3) kernel size, 'same' padding and sigmoid activation is added as the output layer.

From these descriptions we find a few distinct differences in the workings of VAEs and GANs in image generation. First off, they use wildly different loss functions with the GAN looking to balance the loss of the discriminator and the generator and the VAE calculates just one loss, usually (and in our case) the KL-loss. Furthermore training a VAE is an unsupervised method while the GAN is trained in a supervised manner. Finally, generally GANs outperform VAEs when it comes to generating realistic new images since it is built strictly to generate new images and autoencoders are not meant to do this in essence. This finding is in line with what we find as can be seen in the next section.

3.3 Results

3.3.1 VAE

We can generate random vectors in the latent space for the decoder to generate novel images. These are as presented in figure 20. As we can see, some general features - such as white spots or overall more yellow colours - are somewhat reconstructed by the VAE. However, the details are clearly smoothed out in the reconstructed image giving it a blurred look. To generate these images, we experimented slightly with a few hyperparameters. First, we upped the amount of epochs in order to let the VAE train more. However, we do not find that the VAE improves significantly up to 100 epochs and due to time constraints we do not experiment with a larger amount. We have therefore settled on 50 epochs. Furthermore, we experimented with 32,64 and 128 latent dimensions and again find no visible improvement in the blurriness or detail of the generated images. For various generated images with various latent dimemsionality, see appendix C. Here we present now a (linear) interpolation between two random vectors sampled from the latent space in figure 17. Here we see in the top-left one random vector from the latent space translated into an image by the VAE and another in the bottom-right. All in between is linearly interpolated vectors between them as per equation (2):

$$\alpha \cdot \vec{\mathbf{v}_1} + (1-\alpha) \cdot \vec{\mathbf{v}_2} \qquad 0 \leq \alpha \leq 1, \tag{2}$$

for two N-dimensional vectors $\vec{\mathbf{v}_1}$ and $\vec{\mathbf{v}_2}$ and various values of α within the given range.

From this figure we can see the earlier described blurriness. We can also somewhat see how the image how the image in the top-left is slowly turning into the one in the bottom-right - though it is hard to distinguish details with the blurriness of the image. Still, it can clearly be seen that the image smoothly transforms into the other through interpolation.

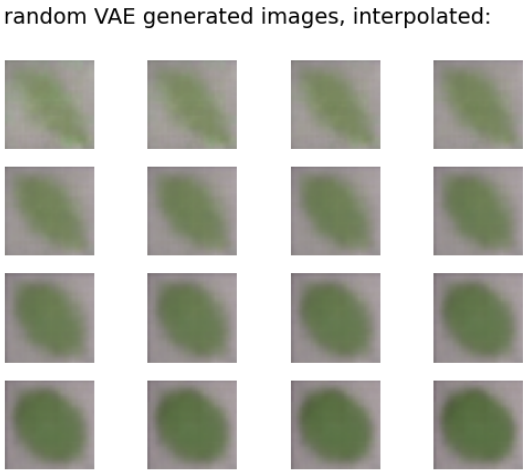

Figure 17: Images generated from interpolating two randomly sampled vectors from the latent space of the VAE (top-left and bottom-right). Interpolation is done linearly. The figure may be read from left to right, top to bottom.

3.3.2 GAN

The GAN shows much more detailed generated pictures than the VAE does. Again, various examples may be found in appendix C, though we did not feel the need to experiment much with epochs or latent dimensions as the results with the given parameters already yielded good results. These are 50 epochs of training and 256 latent dimensions. Now again, following equation (2), we interpolate two random vectors in identical fashion to figure 17 in figure 18. In contrast to the VAE, we can very clearly see how the image changes shape and colour from the first randomly sampled vector into the second. The image quality is far superior to that of the VAE; they are less blurry and have more defined colour contrast. However the images, especially the interpolated ones, do not necessarily look like leaves; though this may just be an artefact of compressing the images to 32x32 versions of themselves.

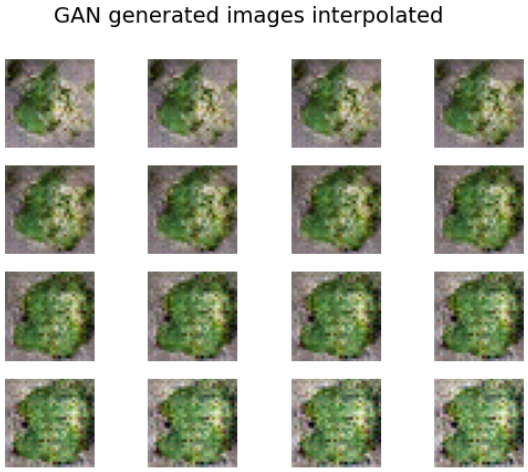

Figure 18: Images generated from interpolating two randomly sampled vectors from the latent space of the GAN (top-left and bottom-right). Interpolation is done linearly. The figure may be read from left to right, top to bottom.

3.4 Conclusion

In conclusion, it is clear to see that the GAN performs more optimally than the VAE in generating novel images from random latent vectors, though neither succeed at generating images that a human would distinctly classify as a leaf.

4 Division of work

Table 5: An overview of who worked on the different sections

Task	Code	Report
1	Menno & Colin	Tim
2	Colin	Colin
3	Menno & Tim	Menno & Tim

Appendices

A CNN results (task 1)

	Optimizer	Learning rate	Batch size	Epochs	Accuracy
0	Adamax	.0005	128	10	0.905
1	Adamax	0.001	256	10	0.903
2	Adamax	0.001	512	20	0.9
3	Adamax	.0005	64	10	0.898
4	Adam	0.001	128	10	0.896
5	Adam	0.001	256	10	0.896
6	Adamax	0.001	64	20	0.896
7	Nadam	0.001	128	5	0.895
8	Adamax	.0005	512	10	0.894
9	Adam	.0005	512	10	0.893
10	Nadam	0.001	256	5	0.892
11	Nadam	0.001	256	20	0.891
12	Adam	.0005	256	10	0.891
13	Adamax	.0005	256	10	0.89
14	Adamax	0.001	256	20	0.89
15	Adamax	0.001	128	10	0.89
16	Nadam	.0005	64	10	0.889
17	Nadam	0.001	64	20	0.889
18	Adam	0.001	256	5	0.887
19	Nadam	0.001	256	10	0.887
20	Adam	.0005	128	10	0.887
21	Adamax	0.001	128	20	0.887
22	Adam	0.001	512	5	0.887
23	Nadam	.0005	512	10	0.886
24	Adamax	0.001	64	5	0.886
25	Adam	.0005	64	10	0.885
26	Nadam	0.001	128	20	0.885
27	Adam	0.001	64	20	0.885
28	Nadam	.0005	128	10	0.883
29	Adam	0.001	256	20	0.883
30	Nadam	0.001	128	10	0.879
31	Adam	0.001	64	20	0.879
32	Adamax	0.001	128	5	0.879
33	Nadam	.0005	256	10	0.879
34	Nadam	0.001	64	5	0.877
35	Adam	0.001	512	20	0.875
36	Adam	0.001	64	5	0.873
37	Adam	0.001	64	30	0.873
38	Nadam	0.001	512	5	0.872
39	Adam	0.001	128	20	0.871
40	Adam	0.001	64	5	0.871
41	Adam	0.001	64	10	0.87
42	RMSprop	0.001	128	5	0.868
43	RMSprop	0.001	256	5	0.866
44	Adamax	0.001	256	5	0.866
45	Adamax	0.001	512	5	0.855
46	Adam	0.001	128	5	0.847
47	RMSprop	0.001	512	5	0.842
48	Adagrad	0.001	512	5	0.728
49	Adagrad	0.001	64	5	0.725
50	Adagrad	0.001	256	5	0.717
51	Adagrad	0.001	128	5	0.693
52	Adagrad	0.001	128	5	0.692
53	SGD	0.001	64	5	0.655
54	SGD	0.001	512	5	0.624
55	SGD	0.001	256	5	0.568
56	RMSprop	0.001	64	5	0.441
57	SGD	0.001	128	5	0.325
58	Adadelta	0.001	64	5	0.301
59	Adadelta	0.001	512	5	0.149
60	Adadelta	0.001	256	5	0.109
61	Adadelta	0.001	128	5	0.101

Table 6: Full version of table 2.

B Multi-head model (task 2)

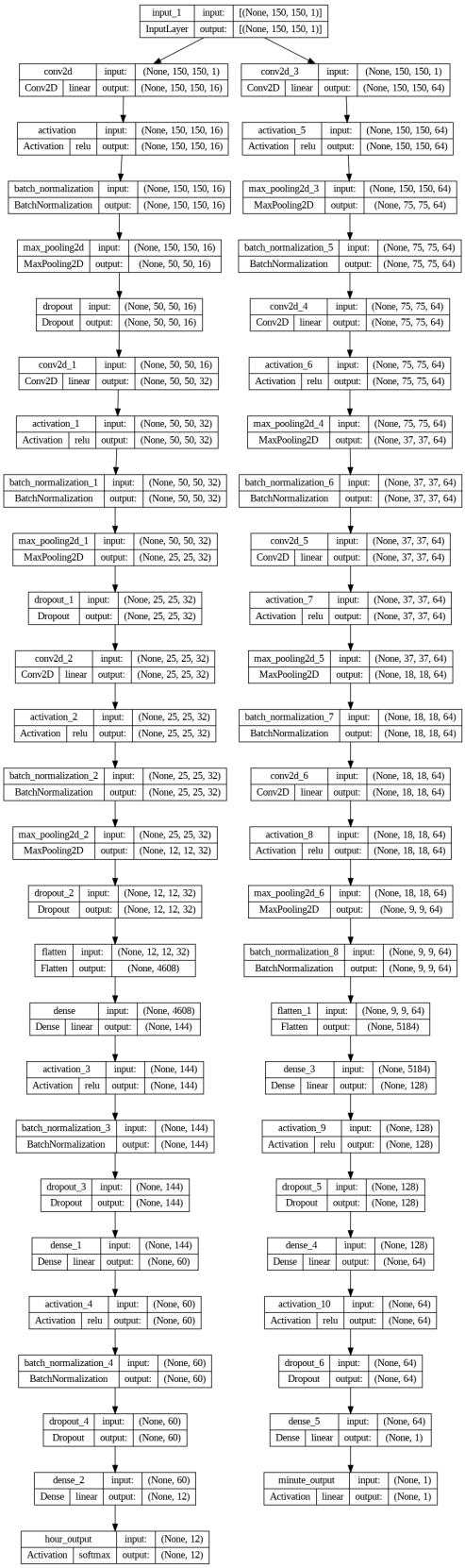

Figure 19: Multi-head CNN used in task 2, subsubsection 2.4.2, for the telling-the-time network. Two branches are present, on the left the hour (classification) branch and on the right the minute (regression) branch. The sizes of each input and output per layer are indicated by the blocks on the right of each layer.

C Generated images

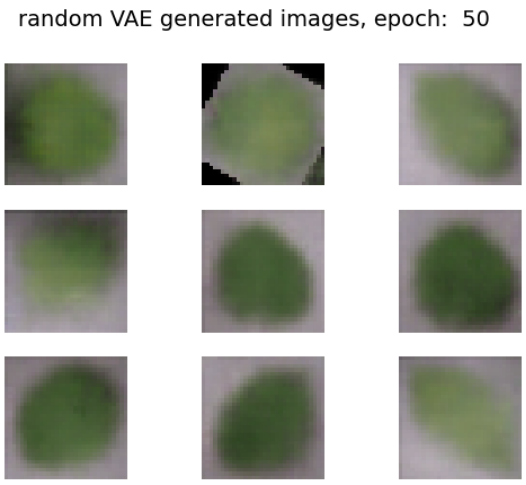

Figure 20: Images generated by the VAE by decoding various random latent vectors. Latent dimensionality is 32.

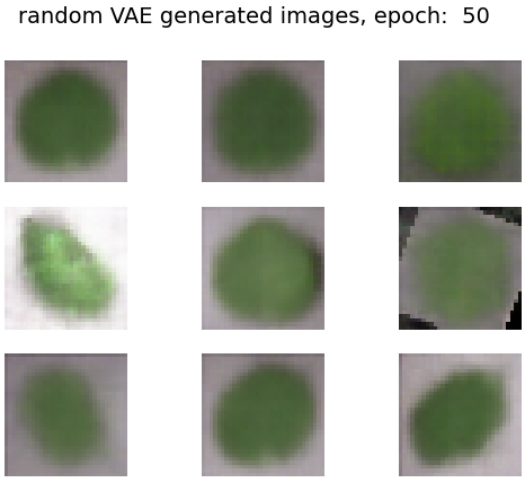

Figure 21: Images generated by the VAE by decoding various random latent vectors. Latent dimensionality is 64.

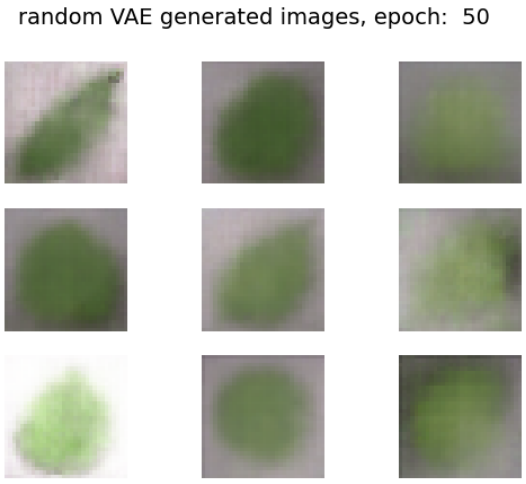

Figure 22: Images generated by the VAE by decoding various random latent vectors. Latent dimensionality is 128.

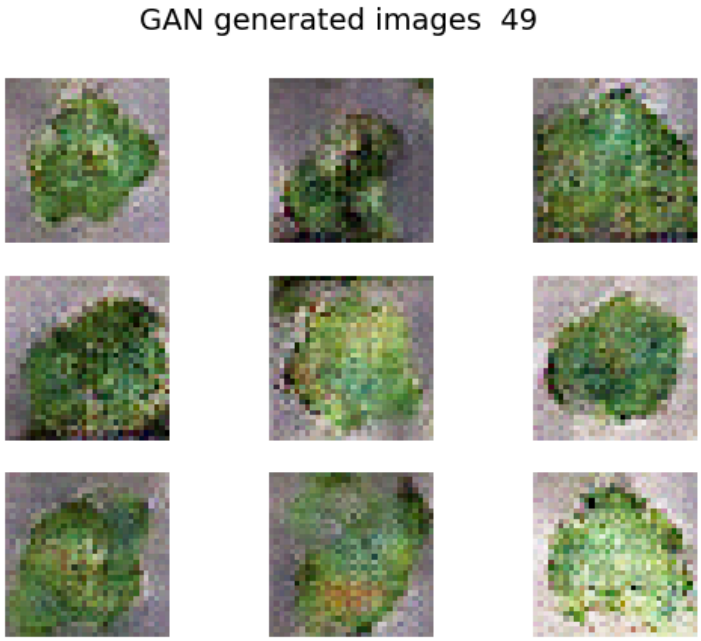

Figure 23: Images generated by the GAN from random latent vectors. Latent dimensionality is 256. The number refers to the epoch index, therefore this is the 50th (and last) epoch of training.