Introduction to Deep Learning: Assignment 2

Ioannis Koutalios (s3365530)
Daan Planken (s1838547)
Jelle Pleunes (s2443341)

October 9, 2024

Introduction

For this assignment we considered convolutional neural networks. We started by getting familiar with the Keras API for Tensorflow and training multilayer perceptrons (MLP) and convolutional neural networks (CNN). Then, we tackled a new problem for telling the time from a picture of an analogue clock, using CNNs. Finally, we trained different generative models, and generated new images using these models. We started by using the given Flickr-Faces-HQ dataset, and after that we found a new dataset, in our case the Simpsons Faces dataset. The following generative models were used: Convolutional Autoencoders (CAE), Variational Autoencoders (VAE), and Generative Adversarial Networks (GAN).

Task 1: Learn the basics of Keras API for TensorFlow

1.1

We were able to run *mnist_mlp.py* with a final test accuracy of 98.26%. *mnist_cnn.py* was also run, with a final test accuracy of 83.74%, which is significantly lower than the test accuracy of 99.25% that is mentioned in the header of the file. A possible cause for this is the fact that in the notebook the test data are passed in where validation data should be passed in, which may lead to overfitting. However, if we ran with the RMSProp optimizer, instead of the Adadelta optimizer, we got an accuracy of 99.14%, which is close to the 99.25% that was advertised.

1.2

Fashion MNIST

When using both reference networks from the book [1] we see that the MLP achieves a test accuracy of 84.44% and that the CNN achieves a test accuracy of 47.74%. Here the MLP does not have any modifications, and the CNN uses the RMSprop optimizer instead of the given Adadelta optimizer.

Using the `HeNormal` weights initializer for all dense layers results in a test accuracy of 84.90% for the MLP. Doing the same for the CNN increases the accuracy to 63.51%.

Using the `sigmoid` activation instead of `relu` for all dense (non-output) layers results in a test accuracy of 76.75% for the MLP, and merely 10.00% for the CNN.

Using the `RMSprop` optimizer instead of `SGD` for the MLP gives an 87.05% test accuracy, which is a good improvement. Using `SGD` instead of `RMSprop` for the CNN gives a lower test accuracy of again only 10.00%. Using `RMSprop` with an increased learning rate of 0.01 (from 0.001) gives a test accuracy of 57.59% for MLP, and 18.06% for CNN. Using the default learning rate (0.001), but decreasing the discounting factor rho to 0.7 (from 0.9), gives 83,98% test accuracy for MLP and 17.81% for CNN.

Adding `L1` regularization with a value of 0.01 to all (non-output) dense layers leads to a test accuracy of 43.53% for MLP, and 84.46% for CNN. Changing from `L1` to `L2` (keeping the 0.01 value) gives 64.87% for MLP and 88.67% for CNN.

Adding a dropout layer with value 0.5 right before the output layer of the MLP gives a test accuracy of 83.96%. For the CNN, removing all dropout layers result in 77.80% test accuracy.

For the MLP, adding another dense hidden layer (which gives a total of three hidden layers, with 300, 200, and 100 nodes, respectively) results in a test accuracy of 85.31%.

For the CNN, increasing the number of filters for the first (input) `Conv2D` layer from 64 to 128 results in a test accuracy of 25.74%. If we instead decrease this number to 32, we see a test accuracy of 48.24%.

CIFAR-10

For the MLP network on fashion MNIST, the best hyperparameter sets were as follows: `RMSprop` optimizer (87.05%), added hidden layer with 200 nodes (85.31%), and `HeNormal` weights initializer (84.90%). For the CNN on fashion MNIST, the best hyperparameter sets were as follows: `L2` regularizer (88.67%), `L1` regularizer (84.46%), and `HeNormal` weights initializer (63.51%). We will now train these six network configurations on the CIFAR-10 data set.

We start with the MLP. Using the `RMSprop` optimizer gives a test accuracy of 34.05% on this data set. Using the added dense hidden layer with 200 nodes gives a test accuracy of 42.03%. Using the `HeNormal` weights initializer the test accuracy is 38.33%.

Now we train the CNN. Using the `L2` regularizer gives a test accuracy of 61.07%. The `L1` regularizer gives 31.76%. Using the `HeNormal` weights initializer gives 54.49% test accuracy.

From the resulting test accuracies that we measured, we can conclude that the effect on prediction performance caused by some hyperparameter value is dependent on the data set. For instance, the CNN with the `L1` regularizer has a similar accuracy to the CNN with the `L2` regularizer on fashion MNIST (84.46% vs. 88.67%), but on CIFAR-10 the difference in accuracy is quite large (31.76% vs. 61.07%). Also, the hyperparameter set that gave the highest accuracy on fashion MNIST does not necessarily give the highest accruacy on CIFAR-10. For the MLP, the configuration with the `RMSprop` optimizer had the best performance on fashion MNIST, but on CIFAR-10 this was not the case. However, for the CNN, the configuration with the `L2` regularizer gave the best accuracy on both data sets. Generally, the networks have worse accuracy on the CIFAR-10 data sets. For the CNN, the accuracy between different hyperparameter sets seems to have a larger variance, with both higher and lower values.

Task 2: Develop a "Tell-the-time" network

For this part we are working with a data set containing images of clocks which are labeled with the time they are showing. The goal is to train a convolutional network that will be able to accurately give us the correct time when given a picture.

After downloading, the data needs to be randomized because they are sorted by default. We are also using a RandomState in order to ensure that the code is reproducible. After that we implement an 80:20 splitting for training and testing/validating.

Regression

For the regression problem we convert each time label to a float number. This way we have a single output node for our model. We also define a "common sense" accuracy that measures the mean absolute error but takes into consideration the absolute value of the time difference. This means that the difference between 10:00 and 02:00 is not 8 hours but just 4 hours.

In order to measure the common sense accuracy we take the module of the predicted time value and 12 (the number of hours). We are then taking the absolute value of the difference between the true and the modified predicted value. After that we keep the minimum between this value and 12 minus this value. By doing this we are essentially checking if the calculated difference is indeed the one that "makes sense" and if it's not we correct the error. After all this we take the average for all the different predictions and we have our common sense Mean Absolute Error. The function calling this algorithm is named "clock_loss_np".

For the purpose of training the network using this approach we create a class named "ClockLossAccuracy" which has the same algorithm as "clock_loss_np" with the only difference being that before taking the average we square the result of all our previous calculations. By doing this we are essentially taking the Mean Squared Error. Taking the square root of that would give us the Root Mean Squared Error, that is another "common sense" accuracy. We find that using the Mean Squared Error is a better way to train the network using "common sense".

The model that we implement is a convolutional one. The exact architecture that we used was based on experience from working with such networks in our previous assignment as well as general instructions on how to successfully build one from the book [1] and other literature sources. We also tried many different layouts before reaching to a final one that correctly trains the network.

First of all we have three convolutional layers (Conv2D) with kernel size (3, 3) mixed with three MaxPooling2D layers of pooling size (2,2). Of the three convolutional layers the first one uses 16 filters while the other two are using 32. We then add a single Flatten layer and then three Dense layers. The last Dense layer represents the output so it has only 1 unit while the other two have 128. The activation function for the first two is the "ReLU" function while the output dense layer has a "linear" activation function. Separating the three Dense layers are two Dropout layers with a 0.1 rate in order to not over-fit the network to the training set.

We train the network twice. For both cases we use the same optimizer, RMSprop, and the same metric, mae. The number of epochs we use is 60 for both cases. For the first training we use the MeanSquaredError loss function, while for the second one we use our own "common sense" accuracy algorithm that we implemented in our class "ClockLossAccuracy".

After training with each of the two loss functions we calculate the "common sense" Mean Absolute Error by using the function "clock_loss_np". The results are shown in Table 1.

Loss function	training MAE(float)	training MAE(minutes)	testing MAE(float)	testing MAE(minutes)
MeanSquaredError	0.224	13.4	0.794	47.7
ClockLossAccuracy	0.282	16.9	0.555	33.3

Table 1: The "common sense" Mean Absolute Error for training the network with each of the two different loss functions using regression.

We find that there is a small improvement when training with a "common sense" Mean Squared Error loss function compared to the regular MeanSquaredError loss function. There is also an improvement on the overfitting as the MAE for the testing set improved while the one for the training increased. Even though the improvement is not huge, it is significant enough to make this way of training more preferable.

Classification

The problem of training the neural network can be treated as an n-class classification problem. We create 24 different classes on our data set. This means that every half-hour has its own label. We then train our model using these labels. After that we generate the predictions for our testing set and assign it the value for the label that has the best score. The final accuracy is measured by using our "common sense" Mean Absolute Error function.

The architecture of our network is very similar to the one we used for our Regression model. One difference is that the number of filters for our first convolutional layer (Conv2D) is 32 instead of 16. The other difference is for the output layer where the Dense layer uses a "softmax" activation function instead of "linear" and the number of units corresponds to the number of classes we have.

We repeat the process while increasing the number of classes. From 24 different classes we go to 48 meaning that each quarter of an hour has its own classifier. After that we increase it to 120, 240, 480 and finally 720 which means that each minute has its own class.

As we increase the number of classifiers we expect our model to not be able to accurately predict the correct label. However the trade off should be that the classifiers are now smaller, so by predicting a close classifier to the actual one we can still get a value close to the true time.

Number of classes	training MAE(float)	training MAE(minutes)	testing MAE(float)	testing MAE(minutes)
24	0.243	14.6	0.413	24.8
48	0.118	7.1	0.382	22.9
120	0.059	3.5	0.597	35.8
240	0.032	2.0	0.685	41.1
480	0.028	1.7	0.939	56.3
720	0.025	1.5	1.389	83.4

Table 2: The "common sense" Mean Absolute Error for training the network as a classification problem using different number of classes.

In Table 2 we see the errors we calculated after training the network for the different number of classes. We see that our intuition of decreasing the errors when increasing the number of classes was correct but only applied to the training set. The errors for the testing set were increasing as the number of classes increased. This is a clear case of over-fitting the data. This becomes even more clear by looking at Figure 1, where we plot the errors for both the training and testing set as a function of the number of classes.

Multi-head models

For this approach, we gave the network two different outputs: we train the minutes with regression (using one output node), and the hours with classification (using twelve output nodes). The first part of our network is the same as for the first two approaches: we have three convolutional layers (Conv2D) with kernel size (3, 3)

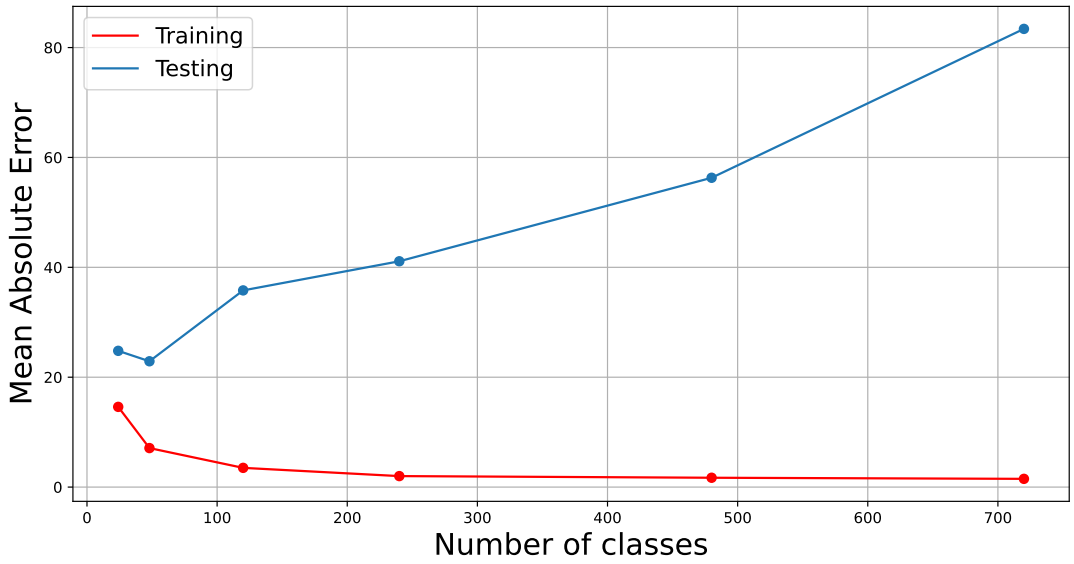

Figure 1: The "common sense" Mean Absolute Error for the training and testing dataset for the classification problem. We can see that increasing the labels leads to overfitting.

mixed with three Max-Pooling2D layers of pooling size (2,2). Of the three convolutional layers the first one uses 16 filters while the other two are using 32. We then add a single Flatten layer.

After this, we connect to different parts of the network to this layer. The first part consists of two Dense layers with 128 nodes each and the ReLu activation function, followed by 1 output node with the linear activation function. This part corresponds to the part of the network that learns the minutes. The second part consists of two Dense layers, one with 256 nodes, and one with 128 nodes, both with ReLu activation function, followed by 12 output nodes, each with the Softmax activation function. After each of the Dense layers we have a Dropout with parameter 0.1, and the first Dense layers has a L2 regularizer with parameter 0.1. The loss function for the hours is the categorical cross-entropy, and the loss function for the minutes is the mean squared error.

Using this configuration, we got a loss of 0.082 for the training set and 0.195 for the testing set. This corresponds to 11.7 minutes for the testing set.

The reason this network outperforms the other networks is because it consists of two parts which are learning two more-or-less independent problems. On top of that, classification is more suited for learning the hours, while regression is much better for learning the minutes, since there are so many possible values. On top of that, the multi-head model had more nodes, so more "power", which could also be a reason it performed better.

Task 3: Generative Models

3.1

We were able to run the given notebook.

3.2

We used the Simpsons Faces dataset[1], which consists of cropped frames of characters from The Simpsons. The images are relatively uniform, with all faces being at a similar (sideways) angle. The resolution of each image is 200x200 pixels. Figure 2 shows 9 random images from the original dataset.

3.3

We used `cv2.resize` to rescale the images to 64x64x3.

Generated images at different epochs for the VAE (with the default architecture) can be seen in figures 3, 4 and 5. The VAE does not improve much more after around epoch 19, and overall the images are slightly blurry, but acceptable looking.

[1] `https://www.kaggle.com/datasets/kostastokis/simpsons-faces`

Figure 2: Original images from the Simpsons Faces dataset.

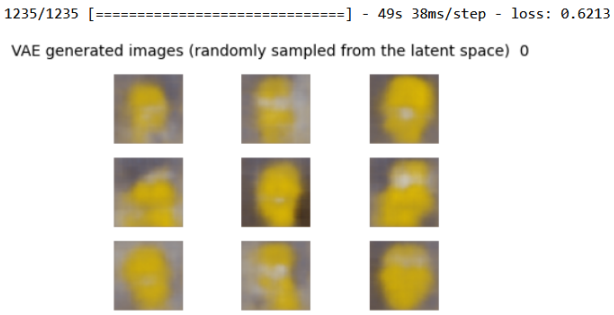

Figure 3: Images generated by the default VAE at epoch 0.

Figure 4: Images generated by the default VAE at epoch 19.

Figure 5: Images generated by the VAE at epoch 39.

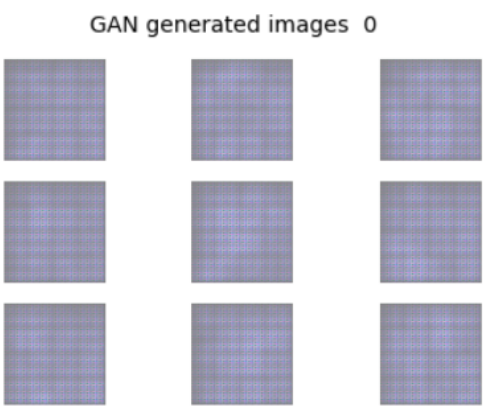

Figure 6: Images generated by the default GAN at
epoch 0.

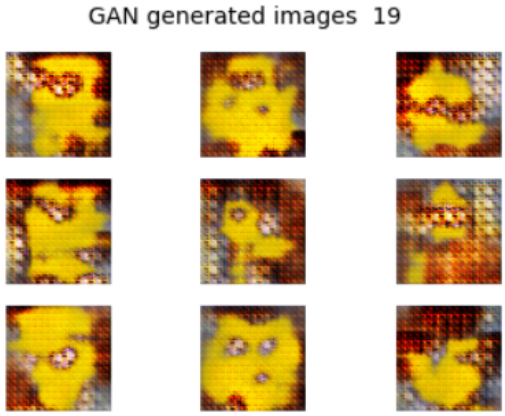

Figure 7: Images generated by the default GAN at
epoch 19.

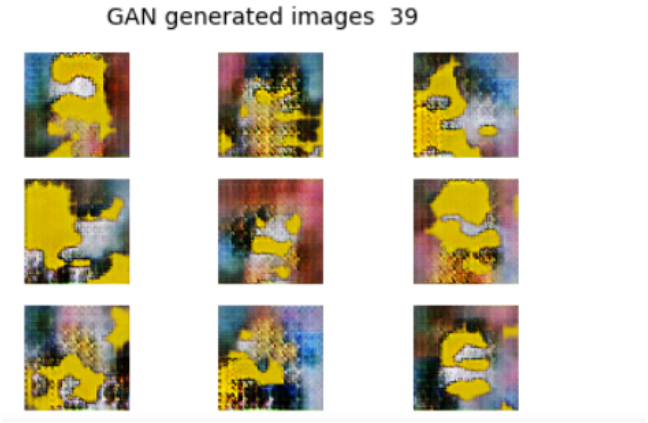

Figure 8: Images generated by the GAN at epoch 39.

Generated images at different epochs for the GAN (with the default architecture) can be seen in figures 6, 7 and 8. Here we see that the generated images do not visually resemble the original data at epoch 0. At epoch 19 we start seeing yellow shapes, and at epoch 39 we see sharper yellow shapes, sometimes with clear white dots for the eyes.

For the VAE, we reduced the complexity of the model by reducing the number of downsampling layers from 4 to 3. In our opinion, the results (figure 9) are slightly lower quality compared to the default architecture.

For the GAN we also reduced the complexity of the model by reducing the number of downsampling layers for the discriminator from 4 to 3. The generated images are visually similar to the ones generated by the GAN with the default architecture.

3.4

Two random points were chosen in the latent space, and images were generated for 9 points interpolated between these points. For the VAE the latent space is 32-dimensional and for the GAN the latent space is 256-dimensional. Figure 11 shows the resulting visualisation for the VAE, and figures 12 and 13 show the resulting visualisations for the GAN (for two training runs). For the VAE, we can see a 'smooth' transition from one generated face to another. The transition is smooth, because the latent space is continuous. For the GAN we can also see a smooth transition. We can also see that the quality of the generated images varies between training runs. For figure 12 the generated images are very noisy, while for 13 the generated images contain clearer yellow shapes.

3.5

The generative models work by downsampling the input data using convolutional layers, in order to capture the most important features. New data that resemble the original training data can then be generated using

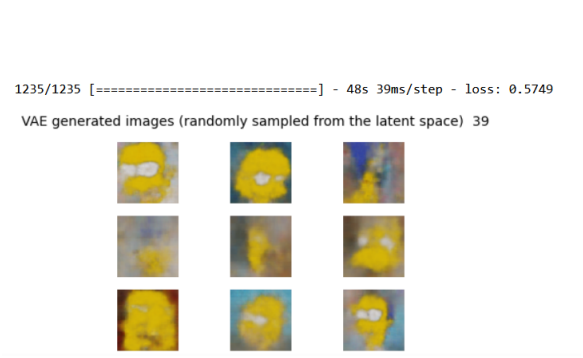

Figure 9: Images generated by the VAE with `n_downsampling_layers=3` at epoch 39.

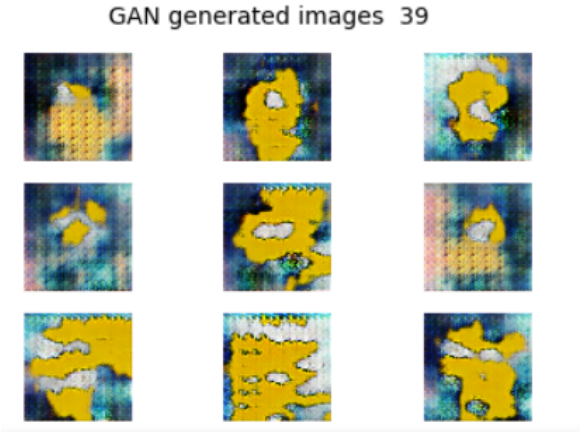

Figure 10: Images generated by the GAN with `n_downsampling_layers=3` at epoch 39

Figure 11: Images generated by the VAE (trained with 40 epochs), using interpolation between two random points in the latent space.

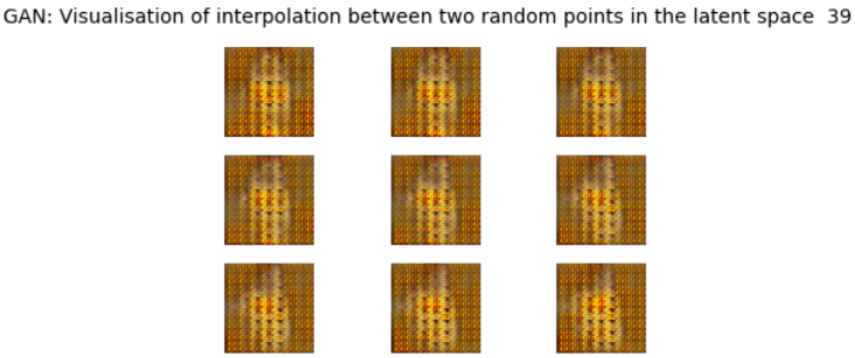

Figure 12: Images generated by the GAN (trained with 40 epochs), using interpolation between two random points in the latent space (run 1).

Figure 13: Images generated by the GAN (trained with 40 epochs), using interpolation between two random points in the latent space (run 2).

an upsampling architecture.

For the VAE, downsampling is done by approximating conditional probabilities, which results in parameter values specifying the conditional distribution. Generating images is then done by taking a vector from the latent space, and outputting the estimated distribution parameters for this vector.

The GAN also uses downsampling, but in this case it is used for the discriminator. The discriminator receives input images from both the original dataset and from the generator, and tries to classify which images are original and which are generated (binary classification). The generator uses a deconvolutional network to turn vectors of random noise into output images. Weights for the discriminator and the generator are adjusted by the training process. The generator and discriminator can be seen as 'adversaries' (as implied by the name "Generative Adversarial Network"), since the generator tries to 'trick' the discriminator, while the discriminator tries to pick out images made by the generator.

Conclusions

We learned how to use the Keras API for training different kinds of deep neural networks. Additionally, we learned how to construct an architecture for a CNN to perform classification/regression for a more difficult problem of telling the time from images of clocks. We were able to achieve a "common sense" accuracy of 11.7 minutes, which is similar to the value mentioned in the assignment as being achievable for simple CNN architectures (10 minutes). Finally, we learned how to train different generative models, and how to generate images using these trained models.

Contributions

Report task 1: Jelle Pleunes
Report task 2: Daan Planken and Ioannis Koutalios
Report task 3: Jelle Pleunes
Code task 1: everyone
Code task 2: Daan Planken and Ioannis Koutalios
Code task 3: Jelle Pleunes

References

[1] A. Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition.* O'Reilly Media, Inc., 2019. ISBN: 9781492032649.