Assignment 3

Introduction to Deep Learning

Peter Breslin s3228282

Louis Siebenaler s3211126

December 2021

Task 1: Recurrent Neural Networks

A Recurrent Neural Network (RNN) uses sequential or time series data to mainly solve sequence classification, labeling, and generation problems. This class of artificial neural networks uses the output from the previous step as an input to the current step while retaining the ability to have hidden layers in its architecture. RNNs were used to explore sequence to sequence models in Task 1. These models are trained to convert sequences of data from one domain to sequences in another domain. A primary goal of Task 1 was to better understand these models by learning how to use encoder-decoder recurrent models.

Sequence to sequence models consist of an encoder-decoder architecture. This begins with the encoder block, an RNN layer which acts to process each data item, or token, from the input sequence of data. The information from each token is stored into the encoder vector which is then passed onto the decoder. This block attempts to predict the target sequence based on the token information encapsulated by the encoder vector. This is designed such that the output at a given time-step will correspond to the specific item of the target sequence at that time-step (e.g. a specific word if the target were a sentence).

The encoder and decoder blocks are often initialised as Long Short-Term Memory (LSTM) cells. These type of RNNs are capable of learning long-term dependencies through a 3 step process. Firstly, information from the input tokens is assessed and either passed on or forgotten based on its relevancy - this is known as the *Forget Gate*. Secondly, the input of new information is then learned - this is known as the *Input Gate*. Finally, the updated information is propagated forward as the input to the following time-step - this is known as the *Output Gate*. This type of encoder-decoder architecture for RNN LSTM models was used to investigate a range of sequence to sequence problems in this task.

The first task encouraged us to explore the Jupyter Notebook provided in the assignment. This contained functions for creating image and text datasets along with sample code for building a text-to-text RNN model. The open-source *OpenCV* library was used to create image queries of arithmetic operations using the MNIST dataset. However, the '+' and '-' operand signs needed to be created as these are not included in the MNIST data. This was achieved by creating blank images and populating certain entries with point values corresponding to coordinates for variations of the +/- signs. These operands could then be visually generated by drawing line segments between those coordinates and plotting the resulting image array. The MNIST images were then incorporated to create 80000 samples of arithmetic operations that could be queried as images or numerically queried as strings.

The architecture of the text-to-text RNN model was inspected to better understand the encoder-decoder process. Before the network can be constructed, the input data must be encoded correctly so that they can be processed by the RNN correctly. The code for generating the queries and answers have the text queries/answers represented by strings of the label val-

ues, which formats the data categorically. Categorical data is problematic as RNNs require that the input and output variables are numbers. Label (*or* integer) encoding provides a way to handle categorical value by assigning each label with a unique integer based on its ordering. However, because this specifically orders the data, this type of encoding introduces the chance that the model may misunderstand the data and derive false correlations. One-hot encoding allows us to avoid this problem by converting the categorical data to binary. This works by splitting the categorical data into separate vector arrays and replacing the values with 1s or 0s depending on the value of a given array (these are known as one-hot vectors). This eliminates the ordering bias while converting the data into something the RNN can process correctly. If one wanted to convert the one-hot values back to categorical data, the values would have to be decoded. This is simple to do as the 1-D one-hot vectors contain 0s and a single 1 corresponding to the value of the previously categorical data. The 1 values are therefore the largest element contained in each one-hot vector which allows for their indices to be easily found. The original label string can then be determined by joining together the unique characters corresponding to these indices.

The text-to-text RNN model could now be implemented using the correctly formatted inputs and outputs. This configure uses similar architecture to that of the multilayer perceptron (MLP) and convolutional neural network (CNN) we've seen before, with some changes to the individual layers. A *sequential model* is initialized with an *LSTM* to implement the encoder-decoder model, and the input shape is known as the vectors are not of variable length. A *RepeatVector* is added so that the output contains the previous input of the RNN for each time step. However, RNNs require the collective output to be passed on (i.e. the outputs from all previous steps), which is simply achieved by setting *return_sequences* to True. Dense layers are then added to each temporal slice of an input and the model is compiled with hyperparameters akin to those used in MLPs/CNNs. Initially, this text-to-text RNN model performed quite badly on the validation set (accuracy of $\sim$50%) after training. This did not fluctuate by much after each epoch, indicating that the model was not learning well.

However, after some exploration into the model architecture and inputs, it was realized that the input data was ordered and not randomized. Since the input data is split for training and testing, there will be images in the test set that the model would never have seen before because of the ordered sequence of input data. This leads to the model becoming poorly generalizable, and hence is why the accuracy on the validation set was so bad. This was remedied by randomly shuffling the input images such that the ordering was erased. The network was re-trained with this optimization and the model's accuracy on the validation set was significantly improved after each epoch, reaching an accuracy of $\sim$99%. The effect of different ratios of the training and test set splitting on the model performance were investigated. The results are summarized as training histories for the different cases in Figure

1. For each ratio (5%, 10%, 20%, 30%) the model converges very fast, after $\sim$ 10 epochs, and reaches an accuracy of $\sim$99% on both the training and validation set, thus indicating that the network generalizes extremely well and has learned the general principles behind arithmetic operations. Different architectures for the RNN model were tested by varying the number of memory cells in the LSTM layers and more dense layers to the model. No significant changes in the model performance could be discerned, the accuracy on both validation and test set remained $\sim$99%. The only difference consisted in an increased training time per epoch for larger numbers of trainable parameters. This analysis suggests that the task of understanding arithmetic operations itself is very easy for modern RNN's.

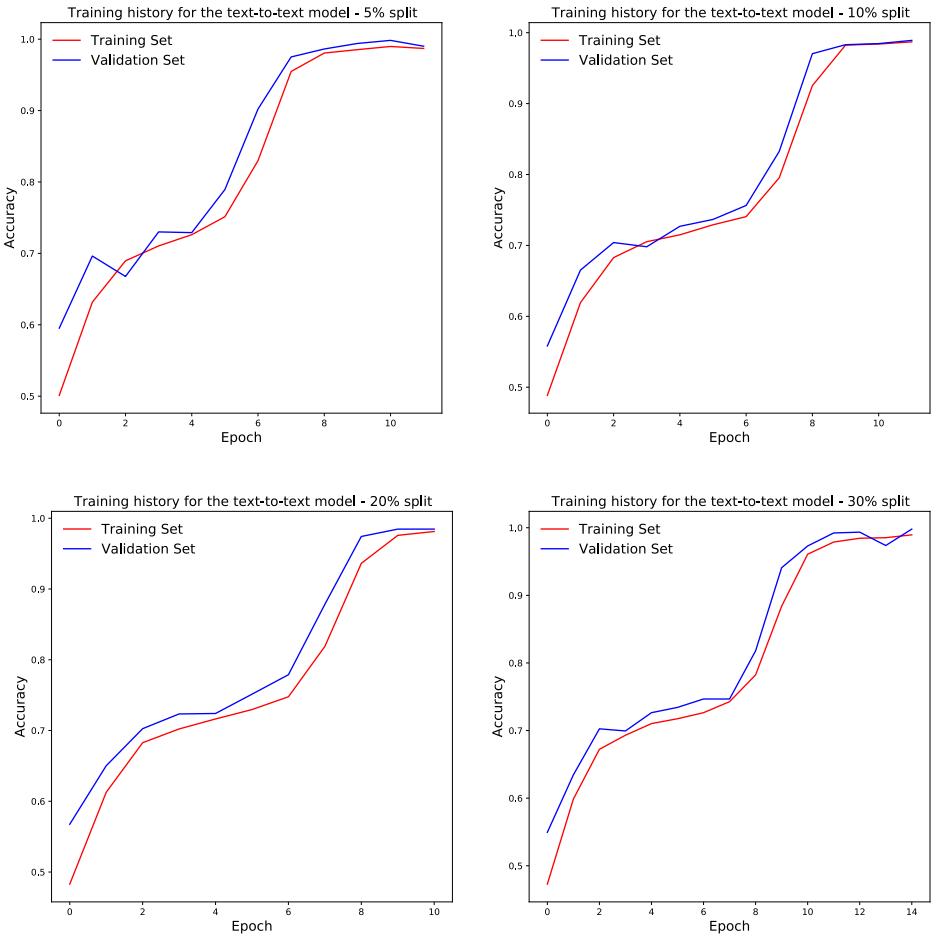

Figure 1: Training history of RNN's involving different splits of the training and test set.

An image-to-text RNN model was implemented next, where the input consisted in a sequence of MNIST images representing a query of arithmetic operations and the output corresponds to the result in the form of a string. Given that the input involves sequences of images, it was attempted to solve this problem using recurrent convolutional layers (*ConvLSTM2D*). This works similarly to a regular LSTM layer, but internal matrix multiplications are exchanged with convolution operations which ensures that spatial patters in the image are

more efficiently understood and the data flowing through the ConvLSTM cells keeps the input dimension. The *ConvLSTM2D* layer is implemented in the encoding part of the network and has an input shape of dimensions [7, 28, 28, 1], since the sequence consists of 7 images which are grey-scale. The decoding part consists of a regular LSTM layer. Poor generalization capabilities were identified when training RNN's with this general architecture.
Implementing different batchsizes, dropout rates or batch normalization layers did not improve the results. However, changing the architecture in such a way that the encoding part includes a regular LSTM layer instead improved the performance of the RNN immensely. For this the input images of the sequence need to be flattened into one-dimensional vectors of $28 \times 28 = 756$ entries. The training history for a RNN of this kind of architecture with a total of 319373 trainable parameters is shown in Figure 2.

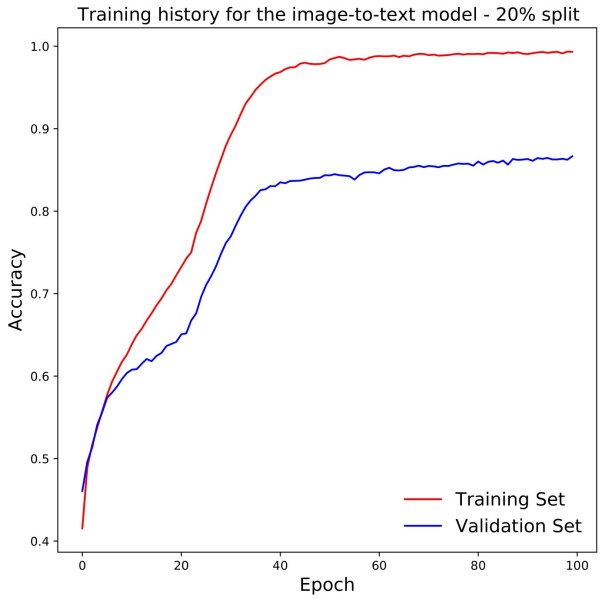

Figure 2: Training history of the RNN for the image to training task.

At the end of training the model reaches an accuracy on the validation set of ∼86%. Although this is a high accuracy, it is significantly lower than the accuracy on the test set for text-to-text network. The reason for this can be explained by realizing that the task of understanding digits based on an image involves fitting a much more complex function of many coefficients compared to understanding a string corresponding to the digit. Furthermore, a digit string does not involve any noise, every string corresponding to '0' looks identical. Conversely, the images in the MNIST dataset involve noise and thus the same digit will always look slightly different. As a result, the image-to-text task is much more difficult for a RNN compared to the text-text task. Nevertheless, a validation accuracy of ∼86% implies that the network has been able learn the general principles behind arithmetic operations.

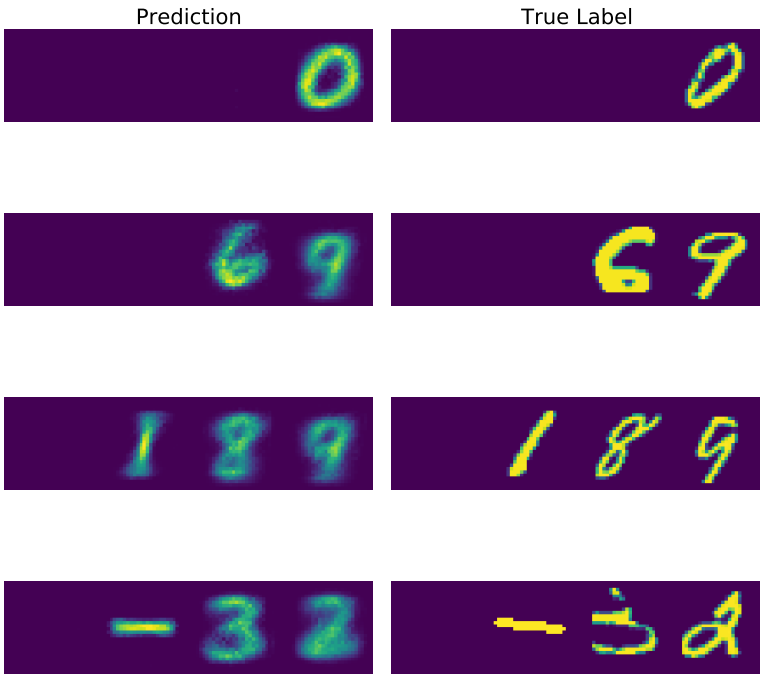

Figure 3: Examples of predictions on the test of the text-to-image RNN. The LHS shows the predictions while RHS correspond to the actual images in the test set.

Lastly a text-to-image RNN was created which takes a text query as an input and generates a sequence of images corresponding to the correct answer. In order to produce images, the model requires deconvolutional layers which perform the transposed operations of convolutional layers. Furthermore, given that the encoder produces a flat latent vector based on the text query input, a reshape layer is necessary to transform the latent vector into a series of matrices on which deconvolutional operations can be applied. Assessing the accuracy of this model is achieved by first generating a image-to-text RNN which takes as an input a series of digit images and outputs them as their corresponding string.

The architecture of this model is identical to the text-to-image RNN model described above, however this task is significantly simpler given that it does not involve learning arithmetic operations. As a result, the new text-to-image RNN reaches an accuracy of $\sim 99\%$ on the test set. This model is used to transform series of images predicted by the text-to-image network into strings. This way, the images are transformed into a text query which we can directly compare to the expected label. This direct comparison is not possible using images only. Using this approach of using the text-to-image and image-to-text networks in series, an accuracy of $\sim 68\%$ on the test set was determined with an average error of ~ 3. Given that the image-to-text network is extremely accurate, most of the misprediction arises from the text-to-image network. Figure 3 illustrates some example predictions of the of the text-to-image model on the test set. The predictions look astonishing good and one can argue that in some cases (ex. -32 and 189) the output generated by the RNN looks much better than the

actual handwritten number. Although, the predicted digit images appear blurred compared to the actual images, the digits can be clearly identified by a human being. Furthermore, it is interesting to note the consistency in how the network draws digits. In Figure 3 both 9's look identical which is not the case for the true dataset. Importantly, these four examples reflect the ability of the text-to-image network to perform simple arithmetic operations which yet again underlines the power of modern RNN's.

Task 2: Generative Models

In deep learning, generative modeling is a learning method that involves automatically discovering the patterns in input data such that the model can generate new examples which could have been drawn from the original dataset. Two of the most popular generative models in machine learning are variational autoenconders and generative adversarial networks. While both rely on supervised learning methods, their underlying architectures and training methods differ significantly.

Variational Autoencoders (VAEs)

To understand VAEs, it is beneficial to review convolutional autoencoders (CAEs) as these are constructed with very similar architectures. Autoencoders employ the encoder-decoder structure described in Task 1 in which high-dimensional data is learned through a lower-dimensional representation. This process of dimensionality reduction summarizes the data in fewer parameters and encourages the network to learn the most important parts of the input image during training.

For more complex images with finer details, CNNs are much better suited than dense layers. Including CNNs with the encoder-decoder architecture gives us the so-called CAEs, where the encoder takes the shape of a regular CNN composed of convolutional and pooling layers. The spatial dimensionality of the inputs are reduced while its depth is extended, meaning that the number of feature maps is increased. As the decoder must perform the opposite to this in order to decompress the image, the convolutional layers are generally transposed, or combined with layers that 'upsample' the downsampled data.

Just like standard autoencoders, VAEs learn to represent the input in a compressed, downsampled form known as the latent space. VAEs differ from standard autoencoders in that they express the attributes from this latent space (*i.e. the latent vector*) as a probability distribution. This is called the latent distribution, and the vectors sampled from this are given as input into the decoder module. This differs from the autoencoders we have previously seen which outputs a single value described by each latent vector. Moreover, these are generative autoencoders as the data related to the original source must be newly created.

Modeling the latent vectors as probability distributions is done so to address the limitations associated with standard autoencoders for generating data. Standard autoencoders perform well when attempting to replicate images because the encoding of the input data into the latent space reveals distinct groupings which the decoder finds easy to decode. However, this means that the latent vectors may not be continuous which can make interpolation difficult when the network is given input data it has not seen before - since the inputs given to the decoder may come from a discontinuous region of latent space. This is important for the generation of new features when you don't want to simply replicate the input image.
Representing the latent vectors as probabilistic Gaussian distributions allows for a continuous latent space to be created which provides the decoder the ability to understand an input given from any region of space. Random samples of the latent space distribution are the inputs fed into the decoder which can then be decoded to generate variations for a given input image. Although the VAE architecture is very alike to that for CAEs and other autoencoders, there are of course some differences.
Rather than only directly encoding a given input, the encoder defines a mean and log-variance to go along with it. This approximates a posterior distribution which takes an observation as its input and outputs a parameter set for specifying the conditional distribution of the latent space. The directly encoded input is then randomly sampled from a Gaussian distribution with the previously defined mean and variance. Following this, one would normally feed the sample to the decoder for decoding.
However, the backpropagation process can not be achieved in this way due to the inherently non-deterministic sampling that is done from a random node in the latent space distribution. In other words, backpropagation cannot flow through random nodes. However, a solution to this can be found by introducing a new parameter which allows us to reparameterize the random node such that backpropagation can flow in a deterministic way, while maintaining the stochastic nature of the latent sample. This is known as the "reparameterization trick" and it is implemented into the VAE by introducing a custom layer into its architecture that connects to the output of the encoder. Furthermore, an additional KL loss term is included to characterize how well the probability distribution at the output approximates that of the latent space. Finally, the model can then be compiled and trained.

Generative Adversarial Networks (GANs)

GANs are composed of two neural networks, a generator and a discriminator. In short, the purpose of the generator is to generate data that looks similar to the training data and the discriminator tries to distinguish real data from fake data. The training method is known as adversarial training in which the two networks compete against each other in the hope that this pushes them to excel in their respective task.

In the case of an image-related task, the generator takes a random latent representation of an image as an input and outputs the corresponding image. Thus, the generator has the same functionality and architecture as the decoder in a VAE. The input of the discriminator is either a fake output image from the generator or a real image from the training set. The output is its guess as in whether the input is a real or fake image. As a result, the discriminator acts as a regular binary classifier.

Given that GANs are composed of two networks with different objectives, each training iteration is divided into two phases. In the first phase the discriminator is trained. A batch of real images from the training set and fake images produced by the generator are sampled and used as the training data for the first phase. In general, Gaussian noise is fed to the generator to produce the fake images. The labels of the training set are defined as 0 or 1 for fake or real images respectively. The discriminator is trained on this labeled batch for one iteration using binary-cross entropy as a loss function. It is important to emphasize that during this phase only the weights of the discriminator are optimized using backpropagation, the weights of the generator are said to be frozen.

In the second phase, the generator is trained. A new training set is produced which corresponds to a batch of fake images produced by the generator. No real images are included in the training data which implies that the labels for all of the data is 1 (i.e fake). During this phase, the generator learns to produce images which the discriminator will wrongly believe to be real. Note that at this point the weights of the discriminator are frozen and backpropagation only affects the weights of the generator.

From this, it is obvious that the training method is very different to the one applied in VAE. One of the fundamental differences is that the generator is never provided with any real images, yet it continuously learns to produce convincing fake images. All it gets is the gradients flowing back through the discriminator. The fact that GANs never see the training data makes them very robust to overfitting, which is one of their main advantages over VAEs.

Task 2 Results

To train the generative models, the UTKFace dataset was used, which can be downloaded from here. This is a large face-dataset consisting of 20000 faces of all sorts of ethnicities with an age span ranging from 0 to 116 years. Furthermore, the images cover a large variation in facial expressions, illuminations and resolution. All of these properties suggests that the dataset samples the latent space of human faces very well, thus making it ideal for the usage of generative models. For training the models, cropped images of the faces were used. This simplifies training significantly and improves the final performance of the models given that the images focus solely on the face and the background is minimized as much as possible.

The original dataset is provided in RGB images of 200×200 pixel size. The resolution of the images was downsampled to 64×64 to reduce memory which was necessary due to the limited RAM on Google Colab. The provided architectures were used to train VAEs and GANs on this face-dataset.

VAEs

Figure 4 shows the results of a random sampling of points in latent space by the VAE after $\sim$2 hours of training. Although limited GPU and RAM resources were to our disposal, the generated faces look very realistic. Humans of different age groups, genders and ethnicities are present which reflects the rich coverage capabilities of the VAE when sampling from its latent space. The fact that some of the faces and backgrounds are blurry suggests that the model can be trained further. A peculiar observed property in the results of the VAE is that it appears that no faces with glasses or facial hair are generated by the model.

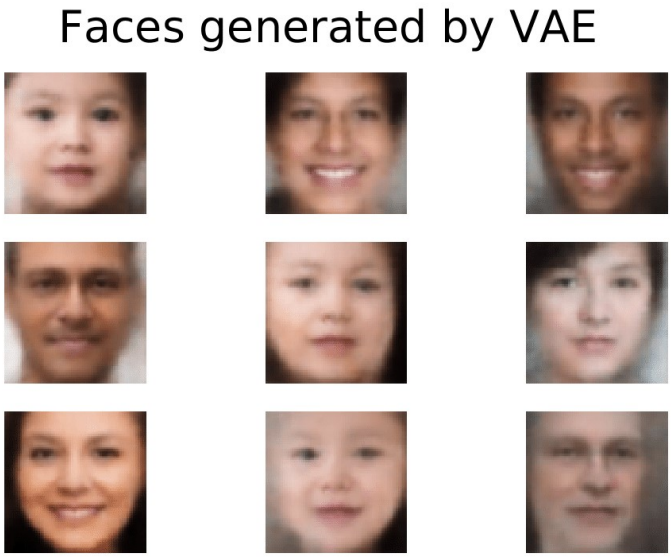

Figure 4: Fake human faces produced by VAE.

Figure 5 illustrates an interpolation between two points in latent space. The baby on the right-hand-side and adult on the left-hand-side correspond to the randomly sampled latent vector. The middle face is the result of interpolating between the two faces and agrees with the expected outcome. The skin has an average color between white and dark and the face could be assigned to an early 20's age group, right in the middle of a baby and an adult of age 40-50 years old. Furthermore, the smile on the interpolated face can be regarded as an average between the other two.

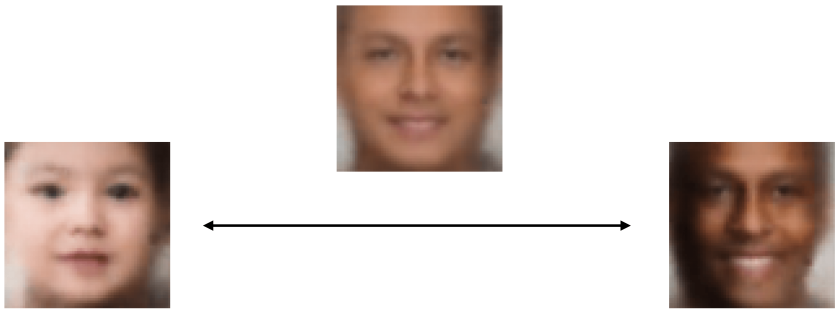

Figure 5: Illustration of interpolation between two points in latent space (rhs and lhs face) perofrmed by the VAE. The middle face corresponds to the interpolation.

GANs

Figure 6 show the results of a random sampling of points in latent space by the GAN after ∼2 hours of training. Despite our GPU and RAM limitations, we were able to generate quite realistic faces once more. Similar to our VAE results, humans of different age, genders and ethnicities were generated, demonstrating the broad sampling range of its latent space. Likewise to our VAE result, the blurriness seen throughout the images generated by our GAN can be attributed to a lack of training. However, unlike the VAE we observe that our GAN model has generated faces with facial hair, glasses, and more defined facial features such as wrinkles, illustrating the capabilities of these network for generating finer details.

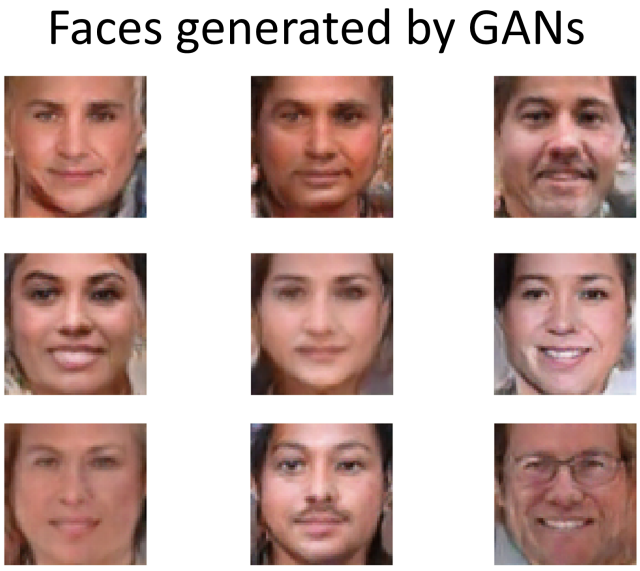

Figure 6: Fake human faces produced by GAN.

Figure 7 illustrates an interpolation between two points in the latent space of our GAN model. The older man with glasses on the left-hand-side and the young female on the right-hand-side

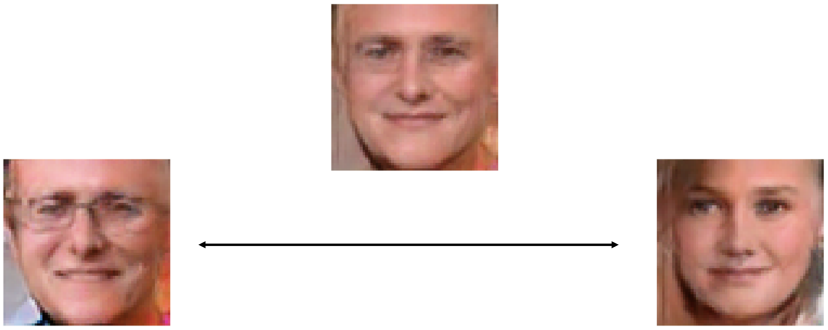

Figure 7: Illustration of interpolation between two points in latent space (rhs and lhs face) perofrmed by the GAN. The middle face corresponds to the interpolation.

correspond to the randomly sampled latent vector from the generator of the network. The middle face is the result of interpolating between these two faces, and the resulting image agrees well with our intuition of what should be expected. If we compare the outcome to the input image of the man, a younger and more feminine person is generated, which is due to the presence of the woman as the second input. Interestingly, the glasses from the man can be somewhat seen in the output image, although they are very faint due to the fact that the woman does not have a similar attribute like this. Finally, a combination of finer details such as wrinkles from the two input images can be seen in the output, where a somewhat average intensity of their prominence can be observed.

Conclusion

After completion of this assignment, a deep understanding of RNNs and generative models was developed. The power of modern RNNs was illustrated through the task of learning the general principles of arithmetic operations. A text-to-text scenario turned out to be a very easy problem and convergence to $\sim$99% was achieved after only 10 epochs. The text-to-image and image-to-text tasks proved to be more challenging which was associated to the underlying noise in the MNIST digit dataset and the increased complexity of the parameter space of a series of images compared to a simple sequence of strings. Nevertheless, the results were still encouraging as illustrated by Figure 3.

The underlying principles of both VAEs and GANs were explored and applied to generate fake faces. Examples of interpolation between randomly sampled points in latent space illustrated the good fitting accuracy of both the VAE and GAN. Comparing the results of the VAE and GAN, it was found that the VAE generates cleaner faces which are absent of any facial defects. On the other hand, the GAN proved more powerful in generating fine details such as glasses, facial hair and wrinkles. This suggests that the VAE and GAN are more sensitive to different patterns in the data.

It can be concluded that the interpolation approach for generation can be employed to produce an infinite array of faces and facial features given the continuous behavior of the latent space. Hence, there exists a large number of use cases for these type of models. A prime modern-day example of this would be the so-called 'Deep Fakes' in which these models are used to generate facial alternations. Lastly, it should be emphasized that both generative models can be further improved through longer periods of training.