Introduction to Deep Learning - Assignment 2

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

December 17, 2023

Introduction

In this assignment, we are tasked to build a recurrent neural network using TensorFlow and Keras capable of evaluating a sequence of text or images. These sequences are simple mathematical expressions such as addition ('13+20'), subtraction ('23-50'), or multiplication ('91*4 ') of digits up to and including 99. These sequences can be encoded text or provided as a stack of images, which are picked from the MNIST [1] handwritten digit dataset.
First, we will discuss the data that was used for this assignment. We will explain where the data was taken from and how it was modified to suit our models.
In part 1, we will be building different models to tackle addition and subtraction. We try 3 different models: a text-to-text model, an image-to-text model, and a text-to-image model. These correspond to the different manners in which we supply the inputs and outputs, which are either encoded text or a stack of images. We will explain the architecture of the models, how they perform on their own, and how they compare to the other models.
In part 2, we will be building 2 of the earlier mentioned types of models, specifically the text-to-text and image-to-text models, to tackle the multiplication task. Again, we will explain the architecture of the models, how they perform on their own, and how they compare to the other models. Furthermore, we will discuss how the performance of the models differed between the addition/subtraction problem versus the multiplication problem.

Data

The data used for this assignment consists of expressions (inputs) and their evaluations (outputs). As we mentioned previously, we have simple mathematical expressions which we wish to teach our RNN how to evaluate. For this assignment, we generated 2 different datasets consisting of strings of expressions and strings of their answers (e.g. "24-3 ", with answer "21 "). For part 1, our expressions include the operations of addition and subtraction. For part 2, our expressions include only multiplication. As we wish to train the RNN for at most two-digit numbers, we have 2*100*100=20000 strings in our first and 10000 in the second datasets.
The expressions can be seen as a sequence of characters. First, we have a digit, which is followed by an operation and another digit. Finally, all expressions are padded with whitespaces to be of equal length, which in our case is 5. The text expressions are encoded using one-hot encoding. This method assigns a vector to each character in the text sequence, where the vector has a single 1 at the index that corresponds to the position of the character in some lookup table. Since our characters are digits, operations, or potentially whitespaces, our lookup table is a string of these characters (e.g. '0123456789+- ' for addition and subtraction). The encoded text therefore has shape (5, n), where n is the number of characters in the lookup table.
To be able to process images, we generated a dataset with a stack of images, instead of a sequence of characters. The shape of the stack of images is (5, 28, 28), since we provide 5 grayscale 28 by 28 pixel images. To generate these stacks, we used the MNIST dataset, which consists of 70000 images of handwritten digits. We process the text sequence by looking up a random image of the current character in the MNIST dataset and appending it to the stack. For the non-digit characters, we generate some random images using Python's `open cv` library.
In this way, we create the following datasets: **X_text_onehot**, **y_text_onehot**, **X_img**, and **y_img**. Which are used for the text in/outputs and the image in/outputs respectively.

1 Part 1: Addition and subtraction

In this section, we look at data constructed from the MNIST set; namely addition and subtraction operations. We take addition and subtraction from numbers between 0 and 99. The final output will be 3 characters for this type of equation. An example image in Figure 1 may be found.

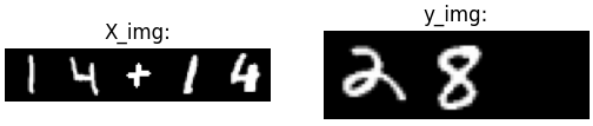

Figure 1: Example input data ("X_{img}") and output data ("y_{img}") for created multiplication data. The corresponding text values are X_{text}: "14+14" and y_{text}: "28 ".

1.1 Text-to-text model

1.1.1 Architecture

The text-to-text (T2T) model that we have built takes the one-hot encoded text, with shape (5, 13), as input and produces one-hot encoded text, with shape (3, 13), as output. We build the T2T model by starting with an LSTM layer. This will encode the data into a 256-dimensional vector. In essence, this combines all the "temporal" data of the input into a single representation. This single vector is repeated 3 times - to accommodate the length largest possible output (-99) - and is put through another LSTM layer, again with 256 nodes. However, this time we do not compress the "temporal" part of the data, but we return the entirety of the (3, 256) tensor. Each temporal slice of this tensor is put through its Dense layer using the TimeDistributed keras layer. The Dense layers each have as many nodes as the number of unique characters (13 in this case) and a softmax activation. This ensures that we will have a single one-hot encoded vector for each character in the output sequence. We compile the model with categorical cross-entropy as

[1] Downloaded using keras

the loss function and Adam as the optimizer. In total, the model has 805,133 parameters of which all are trainable. The architecture of our text-to-text model can be found in Table 5.

1.1.2 Performance

To map the performance of the text-to-text model, we performed a hyperparameter test. In this test, we check for the accuracy of training and testing sets using various combinations of epochs, test set sizes and batch sizes. We do this because the performance of a model can heavily depend on its hyperparameters. For instance, a large batch size limits the amount of detail used from our data in the training phase, whereas a small batch size doesn't. Furthermore, as we want a network that generalizes rather than memorizes our data, we test for various sizes of test and training sets. A small training set leads to limiting our network to only a few cases in our data, while a large training set allows our model to see the majority of them. Commonly used training and testing splits are 80/20 or 90/10. It can be seen that in general, the training set is larger than the testing set. However, this doesn't necessarily mean that it generalizes better, therefore it is important to look at both the accuracy of train and test sets. For our hyperparameter test, we use training splits of 0.25, 0.5, 0.75 and 0.9. We vary between batch sizes of 32, 64, 128 and 256. Lastly, we run for 20, 30 and 40 epochs. From our test, we find the following best hyperparameters:

Batch size	Epochs	Split size
32	40	0.9

Table 1: Best found hyperparameters for the text-to-text addition and subtraction model.

Once plotted, it was seen that after about 20 epochs the accuracy did not increase. Therefore, we conclude that the number of epochs should be 20, rather than 40 since this saves computation time. We did one final training using the new hyperparameters, resulting in Figure 2, where we show the growth of accuracy per epoch. It can be seen that in the early epochs (1 to 3) the increase in accuracy is large, which is then followed by a period of small increase (3 to 8). Furthermore, at around epoch 10, the accuracy increases dramatically to about 0.9 at epoch 13, where it then flattens to about 0.95 accuracy on training and testing sets.

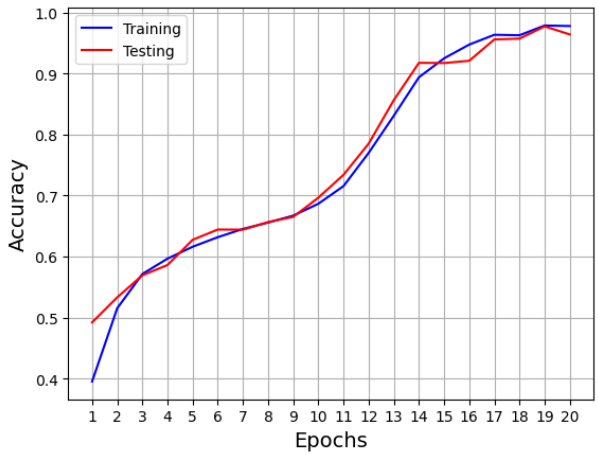

Figure 2: Evolution of accuracy per epoch for our text-to-text model using the optimized hyperparameters. Batch size = 32, number of epochs = 20, splitting size = 0.9.

Using the model described in subsubsection 1.1.1, we evaluated it using the entire training data. To visualize our results, we decoded our predictions using the decode function presented in the notebook from the course. The decoded predictions then needed to be compared to the true labels. In addition, we needed to convert our predictions and true labels into integers to allow for visualization. To do so, we need to account for misclassification that only contains whitespace, or contains a minus sign followed by a whitespace. If such predictions were made, we converted these to a new value of 999, which is much larger than the typical prediction or true label. In total, we had 15 troublesome predictions. Furthermore, to plot our results, we split the true labels between a negative and positive selection. This allows us to better visualize differences between + and - equations. The results can be found in Figure 3 and Figure 4. We present in each figure a subplot containing 250 instances, filtered on their index, to improve visibility. In red are the (wrongly) predicted values marked with a cross ('x'). In blue the true values are shown. In Figure 3, we present the positive true labelled instances, and in Figure 4, we present the negative true labelled instances.

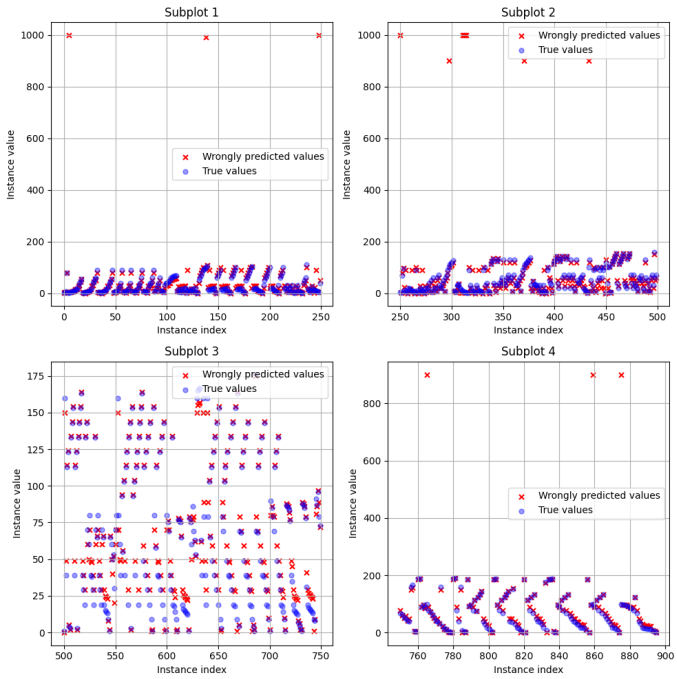

Figure 3: Positional difference between predicted (wrong) values and true labels of '+' equation instances. Outliers from whitespaces or a combination of a minus sign and whitespaces are indicated by a value of 999. Each subplot presents a range of 250 instances, given by the instance index.

From Figure 3 and Figure 4, we can see that the majority of the incorrect predictions are very close to their correct true values. This implies that the wrong predictions are not off by a large margin. However, some outliers exist, indicated by a very large number. Some of which, are the assigned 999 values from our earlier discussion in this section. Furthermore, when comparing Figure 3 to Figure 4, we see that the scatter between correct and predicted values is largest for '+' equations. This is especially apparent in subplot 3 from Figure 3.

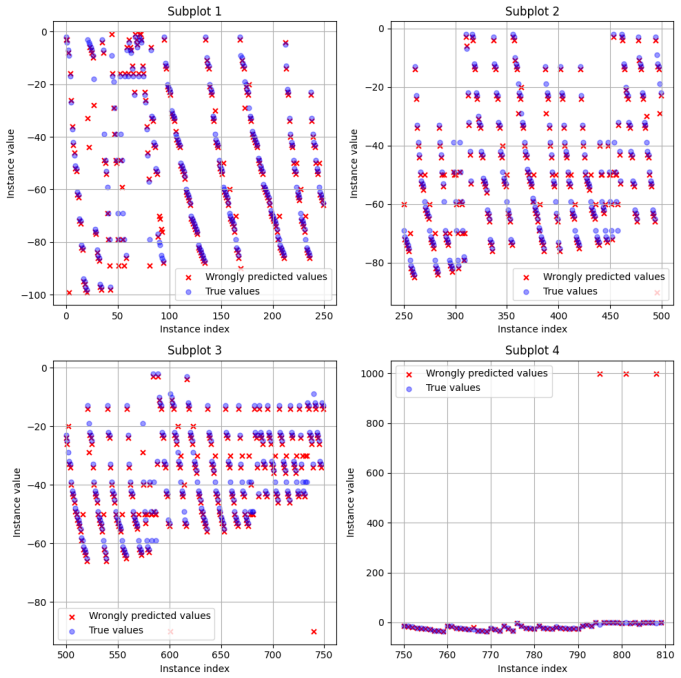

Figure 4: Positional difference between predicted (wrong) values and true labels of '-' equation instances. Outliers from whitespaces or a combination of a minus sign and whitespaces are indicated by a value of 999. Each subplot presents a range of 250 instances, given by the instance index.

1.2 Image-to-text model

1.2.1 Architecture with LSTM

The image-to-text (I2T) model that we have built takes the stack of images from **X_img**, with shape (5, 28, 28), as input and produces one-hot encoded text, with shape (3, 13), as output. The model looks at a sequence of 5 images that create an equation, just like the text-to-text model described in subsubsection 1.1.1. For the I2T model with long-short term memory layers (LSTM), we first reshape the input of (5, 28, 28) into a 5 by 28*28 shape, which then leads into the encoder, which uses LSTM. We have the option to have multiple LSTM layers for the encoder and decoder if needed. As default, we use 1 of each. Each LSTM layer has 400 filters. After the encoding layer, we have a RepeatVector which uses the maximum answer length as input, making it applicable to multiple tasks, such as multiplication for task 2. After the RepeatVector, we have our decoder, composed of an LSTM layer of 400 filters, and a sequence of TimeDistributed dense layers with RELU as activation functions. The layers have 400, 200, 100 and 50 resp. neurons. After the dense layers, we find the output layer. This layer has a softmax output as we aim to classify the characters per output position. The final layer has an output for each unique character. We optimize the model using Adam, together with categorical cross-entropy as the loss function. In total, the model has 6,007,162 parameters, all of which are trainable. We present the architecture in Table 6.

1.2.2 Architecture with ConvLSTM2D

For the I2T model with convolution long-short term memory, we did not have to encode the stack of images into a single representation, because we use convolutions. This was done using Keras' ConvLSTM2D layer, which acts as the LSTM layer, but both the input transformations and recurrent transformations are convolutional. The ConvLSTM2D layer has 400 filters and a kernel size of 3 by 3. After this layer we add another ConvLSTM2D layer with only 1 filter, to obtain an output shape of (None, 28, 28, 1). We need this so that we can reshape it into 28x28 images. After reshaping, we flatten, followed by a RepeatVector of our maximal answer length, i.e. for our '+' and '-' equations it is equal to 3. After this, we have 4 TimeDistributed dense layers which have a number of neurons which go in descending order. Starting with 400 neurons and ending with 50. The activation functions for these layers are all RELU. After these layers, we have one final dense layer with a number of neurons equal to the number of unique characters. This final layer has a softmax activation function. We compile the model using a categorical cross-entropy loss function and Adam as its optimizer. We allow for the option of using 1 or more encoders and decoders, but by default, we use 1 for each to allow for better comparison between models. In total, our ConvLSTM2D model has 7,952,853 parameters, of which all are trainable. In Table 7 we present the architecture of our model using ConvLSTM2D layers instead of LSTM layers.

1.2.3 Performance

For both models, we train using a training test set ratio of 90/10%, batch size of 32 and a total of 50 epochs. The number of epochs chosen is to see if the performance would increase over a large number of training epochs. The number needed for the optimal model can be lower, therefore we stay alert to the training and testing accuracy given a certain amount of epochs.
Using the LSTM model described in subsubsection 1.2.1, we obtain an accuracy of 0.97 and 0.64 on training and test sets respectively at epoch 50. It can be seen that there is quite a large gap between the training and testing set accuracies. An explanation is that our model went into overfitting on the training set, leading to no further increase in accuracy on the testing set. We present the accuracy per epoch on both training and test sets in Figure 5. It can be seen that the discrepancy between the two increases over the number of epochs. Furthermore, it shows that after about epoch 30, the increase in accuracy for testing is negligible. Consequently, this model could run up to epoch 30 and stop. The corresponding training and testing accuracies for epoch 30 are 0.895 and 0.61 respectively. Nevertheless, the network fitting went very rapidly, taking only 5 seconds per epoch, consequently, we could afford to have a large number of epochs as the computation time was small.

When using the model which utilizes ConvLSTM2D in its encoder, the performance was significantly lower than in the LSTM-utilizing model. A potential explanation for this is the complexity that arises when using convolutions. Therefore, the encoder became very good, making it extremely hard for the decoder to find weights optimal given the true labels. However, when experimenting with the model, it seemed to help to add additional dense layers, allowing it to capture greater complexity in the network for the decoder. In Figure 6 we present the training and testing accuracies per epoch.

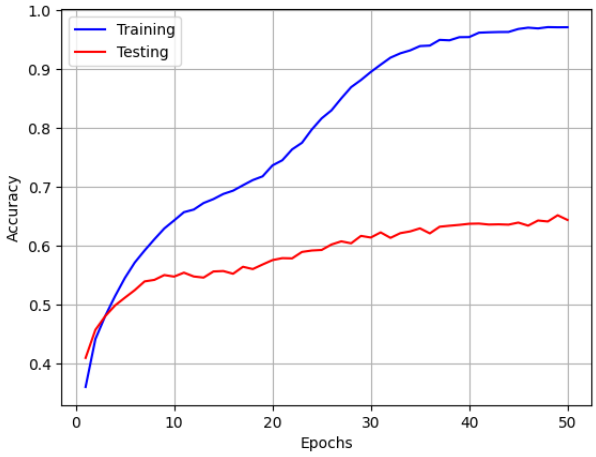

Figure 5: Image-to-text model using LSTM layers in the encoding and decoding parts. In this model, we use 1 encoder and 1 decoder. The evolution of training and testing set accuracies are presented in blue and red curves respectively.

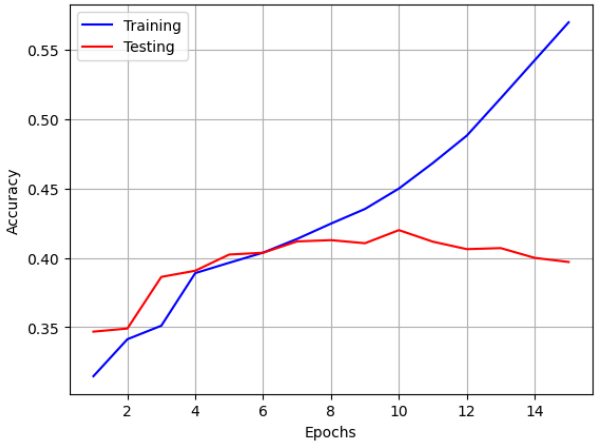

Figure 6: Image-to-text model using ConvLSTM2D layers in the encoding and decoding parts. In this model, we use 1 encoder and 1 decoder. The evolution of training and testing set accuracies are presented in blue and red curves respectively.

From Figure 6, we see that the performance on the test set never exceeded an accuracy of 0.43, whereas for the training set it was possible to go beyond this point quite comfortably. The model was able to overfit the training set when increasing the number of epochs. It was seen that the gains made on the training sets did not transfer to the accuracy of the testing set. Finally, the model using ConvLSTM2D also faced another difficulty in the time needed for training. Each epoch took about a minute. Whereas the regular LSTM utilizing model had epochs of about 5 seconds. This shows that the convolutional layers increase the computational time required, as well as the increase of complexity.
If we compare the results of both models to the text-to-text model, we see that the performance is lower. Over 20 epochs neither of the models gets close to the performance of the text-to-text model test set accuracy (shown in Figure 2). We can expect this outcome by understanding the two methods. In the text-to-text model, our input is not as complicated as in the image-to-text model. In the latter, we put in images and return text, whereas for the former we have an input which is of equal type as the output. Consequently, it is less difficult for the text-to-text model to understand the relationship between the data of input and output. Nevertheless, the text-to-text model performs better than our image-to-text models using 1 encoder and 1 decoder.

1.3 Text-to-image model

1.3.1 Architecture

The text-to-image (T2I) model that we have built takes the one-hot encoded text, with shape (5, 13), as input and produces a stack of images, with shape (3, 28, 28), as output. The start of the T2I model is similar to the T2T model. We start with an LSTM layer, which will encode all of them into a n-dimensional vector, where n is the number of units in the LSTM. This vector is repeated 3 times and passed to the next section of the model. The goal of this section is to turn the (3,n) tensor into a (3, 28, 28) tensor, which represents the output stack. For this, we tried various architectures. First, tried using the Conv2DTranspose layer to upscale the output of the decoder with $n = 256$. To do this, we reshape the output to (16,16,1), where '1' represents the single greyscale channel. We then use a Conv2DTranpose layer with 512 filters to upscale this to (32,32,512). Then, using two more Conv2D layers - first with 64 filters and kernel size (5,5) to downscale to (28,28,64) and second with kernel size (1,1) but just 1 filter to downscale to (28,28,1) which is subsequently reshaped to (28,28) - we create the desired output shape. Of course, we do this all in a TimeDistributed sense, meaning that the final output is (3,28,28) as desired. We varied on this architecture slightly, trying $n = 49$ and upscaling with 3 Conv2DTranspose layers and downscaling once with Conv2D. We moreover tried $n = 9*9*4 = 326$, reshaping to (9,9,4) and up- and subsequently downscaling from there. However, contrary to prior expectations, we found these architectures failed to perform as well as a more simple one that does not use convolution of which we will present the results in this section. The architecture of this model consists of an LSTM unit encoder and decoder with $n = 1024$ (more on this in section 1.4), the output of which is subsequently fed to a Dense layer with 256 nodes in a TimeDistributed sense. Then, a Dropout layer with a rate of 0.5 is put in before the output layer which is another Dense layer with $28*3 = 84$ nodes and sigmoid activation. This output is then, after all TimeDistributed operations are complete, reshaped to the desired (3,28,28) output. In Table 9 we present the architecture of our image-to-text model. The total number of parameters is 21,321,044.

1.3.2 Performance

To describe the performance of this model is not an easily quantifiable task. One must identify if the produced image indeed represents the correct number. For this purpose, we show a few randomly drawn examples from the results of this model in figure 7. From this we can see that the prediction does not perfectly match the true output, but it does get close. In previous models, we found that the predicted image was somewhat of a glorified average of all symbols - meaning that every single prediction looked the same with differences only in the order 10^{-4} in pixel values. However, with this model, we see that it manages to capture some numbers that can be identified as such. Some numbers we can see the model still struggles with - e.g. 5,6,7, 8 and 9. We presume that this is because of their graphical similarity, it is harder to capture the difference between the numbers. The others, including the minus

sign, the model captures well evidently. We can still see that the digits look quite blurry and each instance of a digit looks roughly the same - this is most evident in the case of a displayed "1" in Figure 7. It appears that the model learns which digit should be presented at which slot (first, second or third reading from the left) and subsequently outputs its found "average" for that digit. In Figure 8 we see the average of the digit "1" in the MNIST data. From this, we can see how our model very much approximates this average - the slight tilt to the right and the hourglass shape are present in both.

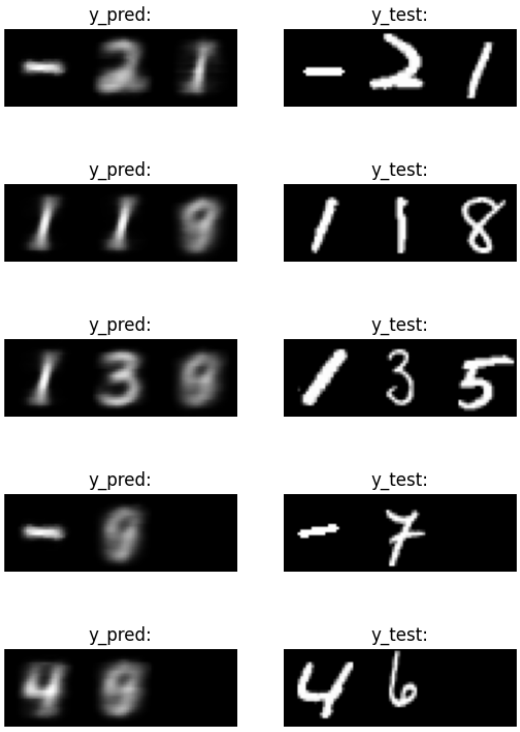

Figure 7: Various examples of the output from the text-to-image model and the true answer compiled from MNIST data.

In an attempt to quantify the accuracy of our model, we trained a fairly simple Multi-layer Perceptron (MLP) that can take in such an image of 3 stacked MNIST digits and output the corresponding one-hot encoded vectors. With this, we can try to have the model identify all the predictions from the text-to-image model and as such quantify its accuracy. We train an MLP with two hidden layers (details may be found in appendix A). This achieved a test accuracy of 0.9860 (and a train accuracy of 0.9703). If we then take all the predictions from the text-to-image model on its test set and run it through this evaluation model we see that we get an accuracy of 0.216. Now since the evaluation model is not 100% accurate this must be taken with a grain of salt, since certain instances may be misclassified and thus either accidentally right or accidentally wrong. What may be noticed in Figure 7 however is that the model tends to get the first digit right but performs worse on the second and third. If we evaluate the model differently - each of the three digits in a prediction separately - and count the average as being the amount of correctly identified digits divided by 3 (so that the accuracy of one prediction may be either 0, 0.33, 0.67 or 1.0) we find a mean accuracy of 0.699 on the test set, which again must be taken with a grain of salt. More insightful maybe, is that both the median and mode of the accuracies are 0.667 which means that the T2I is most likely to get two out of the three digits right. We find that the first digit is correctly predicted 88.05% of the time, the second digit 51.65% of the time and the third digit 70.10% of the time - again it must be taken into account that the evaluator is not perfectly accurate. However, we still see that the text-to-image model is quite good at the first digit, quite poor at the second digit and somewhat decent at the third digit.

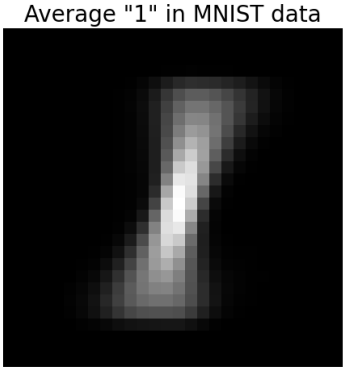

Figure 8: The average of all instances in the MNIST data labelled as "1".

We observe that the text-to-image model improves steadily with the number of nodes in the LSTM (e.g. the value of n). However, increasing n also drastically increases the number of tunable parameters and as such the time needed to train. For this reason, we settled on $n = 1024$, this may however be upscaled to improve the results. Another way to increase the reliability of the evaluator is to tune that model more finely, which was not done here due to time constraints. A convolutional model may be able to more accurately predict the right digit at the right spot, but that is outside the scope of this report. More investigation is needed as to why this model tends to correctly predict the first digit but perform significantly worse on the second and third digit.

1.4 Multiple LSTM layers

We will test how the number of encoding/decoding LSTM layers influences the performance of our models by setting either the number of encoding (resp. decoding) LSTM layers to 1, while we vary the decoding (resp. encoding) LSTM layers. We find how the model performed in terms of the final accuracy on both the training and testing set after a certain number of epochs. We will also compare the loss of the models, but only for the text-to-text model and the image-to-text model, since the text-to-image model has no sensible measure of loss.

1.4.1 Text-to-text model

For the text-to-text model we tried incrementally increasing first the amount of LSTM nodes in the encoder part of the model, keeping just one decoder, and vice versa. The results of varying the number of encoders are displayed in figure 10, the result of varying the number of decoders is shown in figure 9.

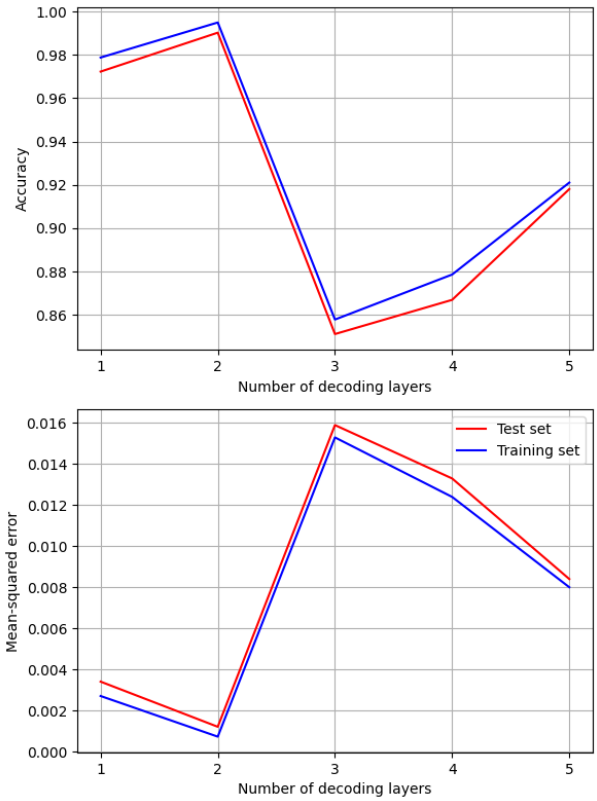

Figure 9: Results of varying the number of LSTM units in the decoder of the text-to-text model. Shown are the accuracy (above) and the MSE (below).

First it must be noted that the first point in both figures are for the same model architecture: 1 layer in both the encoder and the decoder. However we observe a distinct difference in achieved accuracy and MSE. The model seems to perform better in figure 9. This is explicable through randomness in initialization of weights and the fitting process however. The model was rebuilt properly for each data-point so there is no case of fitting more than 30 epochs which may explain the higher accuracy. It shows that this model is quite volatile, since an increase in accuracy of 92% to 97% is substantial in this context. The presented results are still relevant since we care here about the relative increase or decrease in performance after adding more layers in either the encoder.

We can see an increase in performance after adding a layer in the decoder as well as adding a layer in the encoder. This makes some intuitive sense. If the model has a second encoder, this extra layer of abstraction may filter out some of the mistakes that the first encoder LSTM puts into the data. Similarly, if data is decoded through two LSTM layers the decoder can correct a decoder mistake and more accurately find the details that were encoded before. However, there can be such a thing as overcompensating which we can see happening if we add more than 2 layers of LSTM in either the encoder or decoder. The accuracy takes a substantial dive at this point, which is indicative of the fact that the model is overcompensating and as such reducing the accuracy of it prediction. To make sure that this is not a result of random increase in accuracy (like explained before), this process was ran a few times. Each time yielded the same results that both two encoder and decoder layers improved the accuracy of the model. Thus we find that two encoders and two decoders works the best for the text-to-text model.

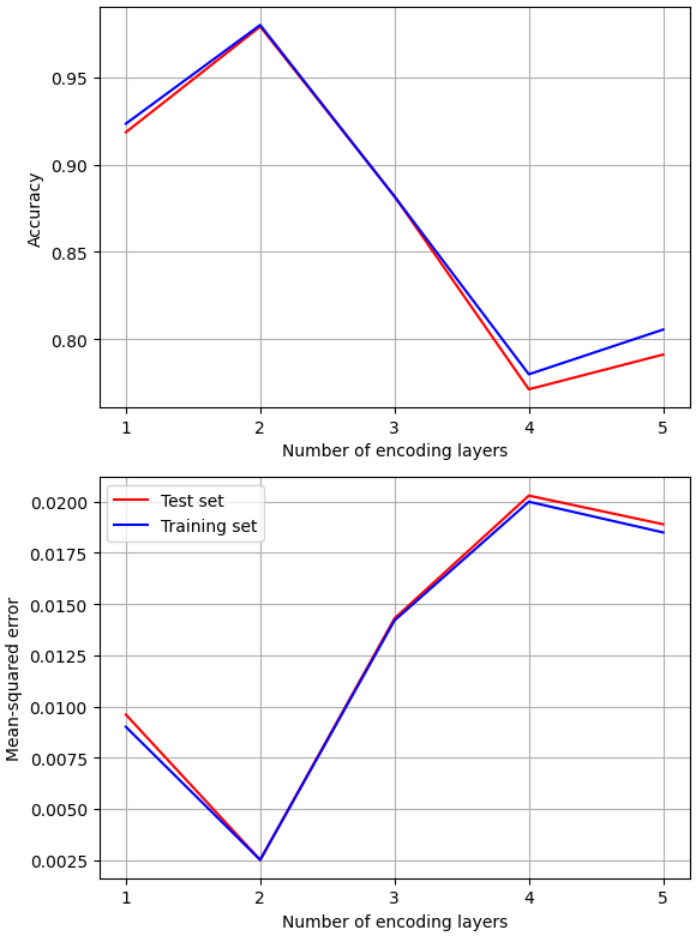

Figure 10: Results of varying the number of LSTM units in the encoder of the text-to-text model. Shown are the accuracy (above) and the MSE (below).

1.4.2 Image-to-text model

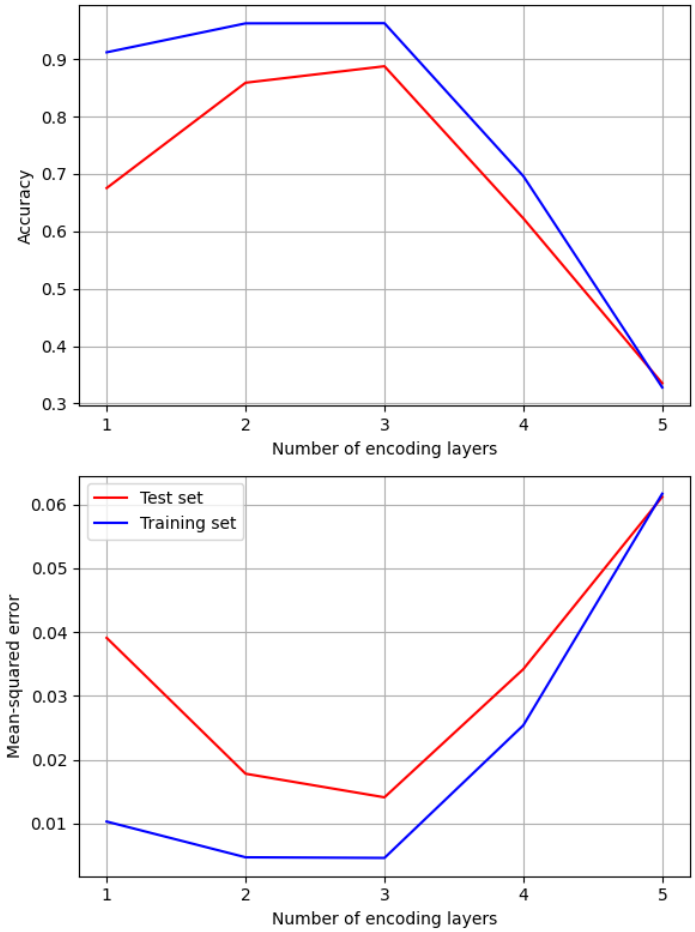

Figure 11: Results of varying the number of LSTM units in the encoder of the image-to-text model. Shown are the accuracy (above) and the MSE (below).

The results of varying the encoders and decoders in much the same way as before but now for the image-to-text model are shown in figures 11 and 12. We see here that the quality of the model increases and reaches an optimum for 3 encoders, 1 decoder and that adding more decoders does not improve the model at all. This increase in quality by adding more encoders follows the same reasoning as before for the text-to-text model. It may be the case that the encoding makes mistakes even more so than for the text-to-text model since the input data is substantially more complex than before (being a (28,28,1) array instead of a vector of shape 13). Still, overcompensation may happen which we see happening for more than three encoder layers. Adding decoder layers only worsens the quality. This might be due to the relative difference in complexity between the input data and the output data - only one decoder LSTM layer is needed to generate the important text from the image.

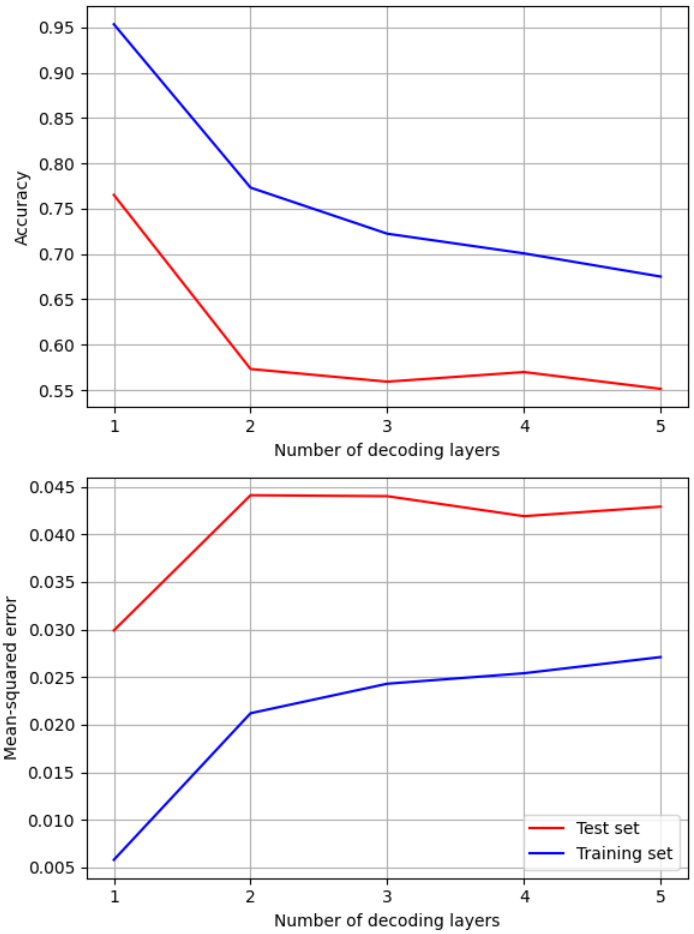

Figure 12: Results of varying the number of LSTM units in the decoder of the image-to-text model. Shown are the accuracy (above) and the MSE (below).

1.4.3 Text-to-image model

Again, the results are displayed in figures **??**, this time only once since we measure the quality of the model quite differently here. Due to time constraints we show here only to a maximum of 3 layers for both the encoder and decoder. However this is sufficient for the purposes of this report, since it is obvious what the effect is. The displayed accuracy here is both the full accuracy (how many instances were completely correctly displayed and identified as such by the MLP evaluator) and the mean individual accuracy (as explained in section 1.3; checking for each of the three digits if it is right according to the MLP evaluator and getting an accuracy of 0, 0.333, 0.667 or 1.0 for each instance). This decrease in quality might be explained through the fact that the model is quite complex in its transformations and what it needs to learn - the input data is quite small compared to what the output should be. Therefore, opposite to the reasoning presented in section 1.4.2, more the encoder may already start overcompensating after just two LSTM layers. If the encoder is not so complex, adding more decoders that lay the foundation for the upscaling process, might introduce new features and relations in the data that simply should not exist thus worsening the performance of the model. All in all, we find that just one LSTM layer in both the encoder and decoder part of the text-to-image model works best with our used architecture.

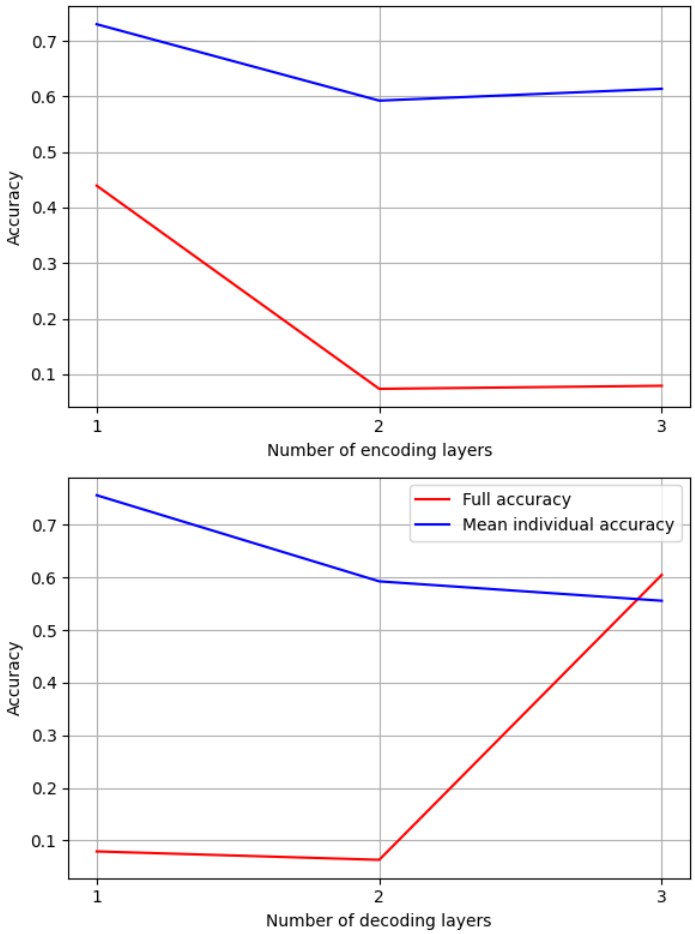

Figure 13: Results of varying the number of LSTM units in both the encoder and decoder, while keeping the other constant at 1, of the text-to-image model. Shown are the full accuracy, the total fraction of correctly classified cases, and the mean individual accuracy, as explained in section 1.3.

2.1 Text-to-text model

We build the text-to-text model with the same architecture as for addition and subtraction. The data is now more complicated and may have 5 output slots instead of 3, but this is easily remedied. We expect beforehand that this operation will be much more difficult for the model to learn because of the larger output as well as the fact that multiplication is an inherently more complicated operation than addition or subtraction. First, we take a model with just one encoder LSTM unit and one decoder LSTM unit and try to find optimal hyperparameters. The best hyperparameters we found are presented in table 2.

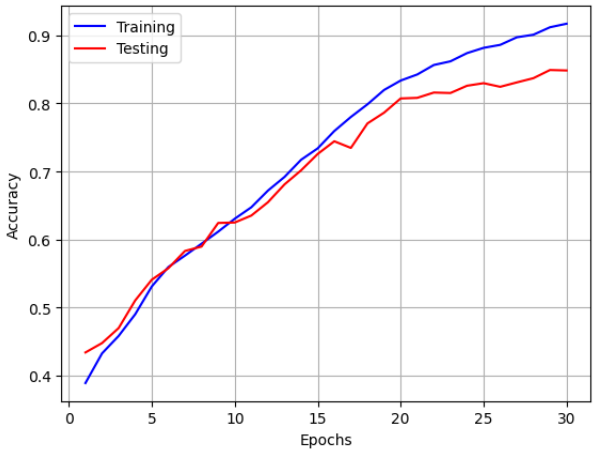

Figure 15: Text-to-text model using LSTM layers in the encoding and decoding parts for the multiplication task. In this model, we use 1 encoder and 1 decoder. The evolution of training and testing set accuracies are presented in blue and red curves respectively.

2 Part 2: Multiplication

In this section we look at at slightly different data constructed as text or from the MNIST set; namely multiplication operations. Likewise, we take multiplications from numbers between 0 and 99. Since we are now dealing with multiplications, the final output will now be 5 characters with the possibility of padding. An example image may be found in Figure 14. We train a text-to-text model very similar to in section 1.1 and an image-to-text model very similar to section 1.2.

Figure 14: Example input data ("X_{img}") and output data ("y_{img}") for created multiplication data. The corresponding text values are X_{text}: "30*26" and y_{text}: "780 ".

We see that our model performs best on a small batch size, a large number of epochs and a large split size. However, it should be noted that the optimal parameters are not the found values because they are boundary values from our hyperparameter testing, meaning we suspect that the real optimum values lay somewhere outside the bounds we used. Furthermore, it should be noted that a large number of epochs most certainly yields a higher performance on most models. However, a larger split size doesn't necessarily give a larger testing accuracy, therefore showing the model prefers more training instances, to have higher generality when applied to a set of unknown instances, the test set. In Figure 15 we present the evolution of the accuracy of our text-to-text for the training and test set. From Figure 15, we see that at the final epoch, the accuracy for the testing set is about 0.8382 and for the training set it is 0.9237.

Batch size	Epochs	Split size
32	30	0.9

Table 2: Best found hyperparameters for text-to-text multiplication model.

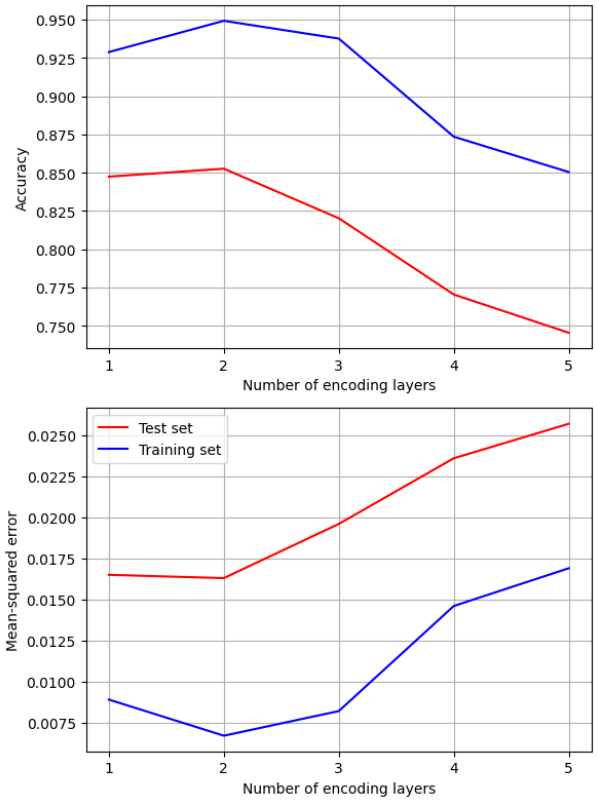

Figure 16: Text-to-text modelling for a higher number of encoders and a single decoder. Shown here are the accuracy and the mean squared error on the test set.

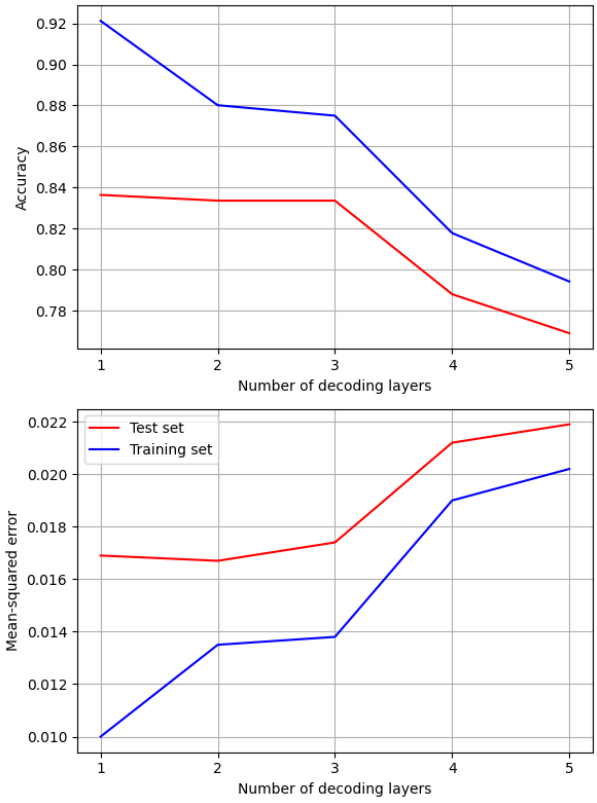

Figure 17: Text-to-text modelling for a higher number of decoders and a single encoder. Shown here are the accuracy and the mean squared error on the test set.

Using the best hyperparameters, we go over combinations of encoder and decoder layers, similarly to section 1.4. The results of this procedure are presented in figures 16 and 17. Here we can see that an optimum is reached for 2 encoders (paired with just one decoder) reaching a test accuracy of 0.8510 and a training accuracy of 0.9498. These values are just marginally better than for just 1 encoder and 1 decoder. After this optimum, the accuracy for both training and test starts to steadily decline. For various decoders (paired with just one encoder) we find an optimum at just one decoder. Again, adding more decoders only seems to worsen the performance of the model. Therefore the best improvement to this model, given the hyperparameters, is two encoders and one decoder yielding a test accuracy of 0.8510 and a training accuracy of 0.9498.

2.2 Image-to-text model

We build the image-to-text model using almost the same architecture as the first image-to-text model that we used in the addition and subtraction part. We chose to use the first model over the second model since the performance of the convolutional model was a lot worse than the other model. Like the text-to-text model that we used in the previous subsection, the only difference is that the model can have 5 output slots instead of the 3 that we needed for addition and subtraction. Again, we believe that the performance of the model will be worse for multiplication than for addition and subtraction, due to the added complexity.
To tune our hyperparameters and architecture, we apply the same method discussed in subsubsection 1.1.1 and subsection 1.4 respectively. We let our hyperparameters vary in training/testing split, batch size and epochs. As opposed to before, we now let the batch sizes be 32, 64 or 128, therefore 256 is no longer an option. We rule 256 out since it was never in the top 5 of our previous analysis, as well as because it gets rid of too much detail from our data. In addition, we pick epochs of 10, 20 or 30. From this analysis, we find that the best image-to-text model uses the following hyperparameters.

Batch size	Epochs	Split size
32	30	0.9

Table 3: Best found hyperparameters for image-to-text multiplication model.

It shows that once again, we see that our model performs best when the batch size is small and the number of epochs and split size are large. In Figure 18, we present the evolution of the training and testing set accuracies over the number of epochs. We can see that the testing accuracy levels off after epoch 25. The best-performing model has an accuracy of 0.68 on the training set and an accuracy of 0.57 on the testing set. Similar to how the text-to-text model outperformed the image-to-text model in the addition and subtraction assignment, we see that the text-to-text model performs significantly better than our image-to-text model for multiplication. Again, we believe that the explanation for this is that the input and output of the text-to-text model are less complicated than those of the image-to-text model. Understanding the input images for our model creates an additional difficulty in grasping the right features to be used when training.

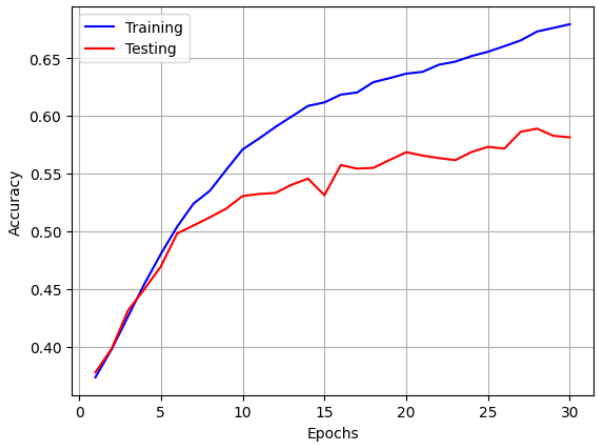

Figure 18: Image-to-text model using LSTM layers in the encoding and decoding parts for the multiplication task. In this model, we use 1 encoder and 1 decoder. The evolution of training and testing set accuracies are presented in blue and red curves respectively.

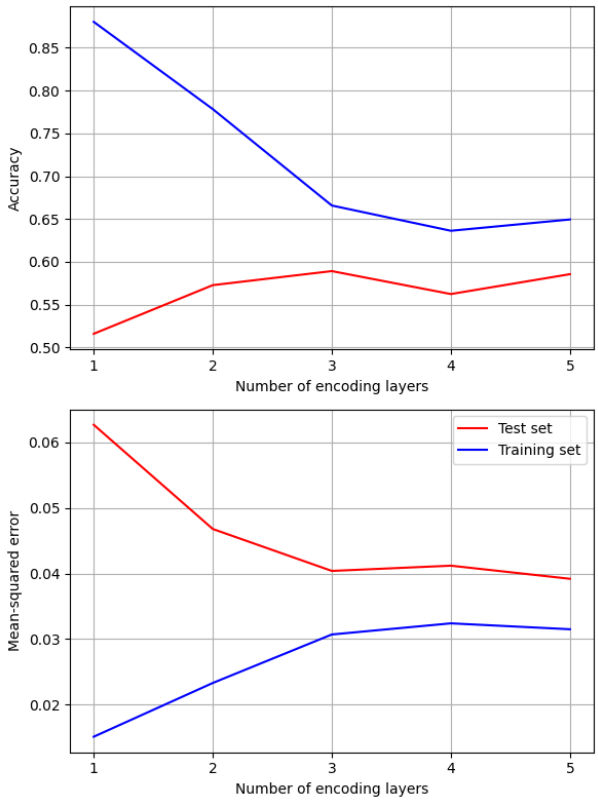

Figure 19: Image-to-text modelling for a higher number of encoders and a single decoder. Shown here are the accuracy and the mean squared error on the test set.

If we compare the performance of the addition/subtraction image-to-text model (cut off after 30 epochs) to the multiplication image-to-text model, which are in figures 5 and 18 respectively, we notice that both graphs start from roughly the same accuracy at epoch 1. Afterwards, we see that the behaviour of both graphs matches the different assignments. The main difference is that the accuracy on the training set in Figure 5 continues to grow quickly after epoch 20, whereas the accuracy on the training set in Figure 18 grows linearly after epoch 15. The accuracy of the testing set is similar in both graphs. They grow linearly after epoch 10. The final accuracies that are reached (at epoch 30) are different, however. We see that the model for the addition/subtraction task got an accuracy of roughly 0.9 on the training set and 0.62 on the testing set, whereas the model for the multiplication task got an accuracy of roughly 0.68 on the training set and 0.57 on the testing set. This tells us that both models could perform roughly equally well on new data, but that the addition/subtraction task was easier to learn for the model.

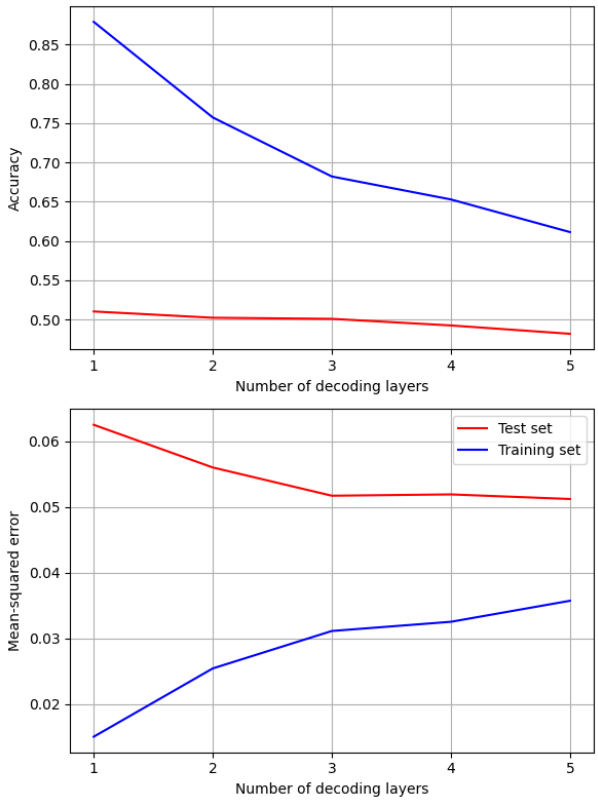

Figure 20: Image-to-text modelling for a higher number of decoders and a single encoder. Shown here are the accuracy and the mean squared error on the test set.

Again, we tested how the number of encoding and decoding LSTM layers influenced the accuracy of the model. Similarly to how we did this in section 1.4, we set the number of decoding (resp. encoding) layers to 1 and varied the number of encoding (resp. decoding) layers. This resulted in figures 19 and 20. We can see from both the accuracy and the mean squared error that we have an optimal performance at 1 encoding layer(s), where we reach an approximate accuracy of 0.51 on the testing set and 0.875 on the training set. This story is similar to the number of decoding layers. Here we see that we have optimal performance for 1 decoding layer(s), where we reach an approximate accuracy of 0.536 on the test set and 0.875 on the training set.

3 Division of work

Table 4: An overview of who worked on the different sections.
* *integrated into sections 2.1 and 2.2 for better report flow.*

Part	**Code**	**Report**
Introduction + Data	-	Menno
1.1	All	Colin & Menno
1.2	All, mostly Colin	Colin
1.3	All, mostly Colin	Colin
1.4	Menno & Tim	Tim
1.5	Colin & Tim	Tim
2.1	All	Tim
2.2	All	Menno
2.3	Colin	Colin*

Appendices

A Architectures

Name	Type	Shape
input	Input	(None, 5, 13)
lstm	LSTM	(None, 256)
repeat_vector	RepeatVector	(None, 3, 256)
lstm_1	LSTM	(None, 3, 256)
time_distributed	TimeDistributed	(None, 3, 13)

Table 5: Text-to-text model architecture for the '+' and '-' task. Total number of parameters is 805,133, of which all are trainable.

Name	Type	Shape
input	Input	(None, 5, 28, 28)
reshape	Reshape	(None, 5, 784)
lstm_2	LSTM	(None, 5, 400)
lstm_3	LSTM	(None, 5, 400)
lstm_4	LSTM	(None, 400)
repeat_vector_1	RepeatVector	(None, 3, 400)
lstm_5	LSTM	(None, 3, 400)
time_distributed_1	TimeDistributed	(None, 3, 400)
time_distributed_2	TimeDistributed	(None, 3, 200)
time_distributed_3	TimeDistributed	(None, 3, 100)
time_distributed_4	TimeDistributed	(None, 3, 50)
time_distributed_5	TimeDistributed	(None, 3, 13)

Table 6: Image-to-text model architecture for the '+' and '-' task using LSTM layers. The total number of parameters is 6,007,162. All of which are trainable.

Name	Type	Shape
input	Input	(None, 5, 28, 28, 1)
conv_lstm2d	ConvLSTM2D	(None, 5, 28, 28, 400)
conv_lstm2d_1	ConvLSTM2D	(None, 28, 28, 1)
reshape_1	Reshape	(None, 28, 28)
flatten	Flatten	(None, 784)
repeat_vector_2	RepeatVector	(None, 3, 784)
lstm_6	LSTM	(None, 3, 400)
time_distributed_6	TimeDistributed	(None, 3, 400)
time_distributed_7	TimeDistributed	(None, 3, 200)
time_distributed_8	TimeDistributed	(None, 3, 100)
time_distributed_9	TimeDistributed	(None, 3, 50)
time_distributed_10	TimeDistributed	(None, 3, 13)

Table 7: Image-to-text model architecture for the '+' and '-' task using ConvLSTM2D layers instead of regular LSTM layers. The total number of parameters is 7,952,853. All of which are trainable.

Name	Type	Shape
input	Input	(None, 5, 13)
lstm_7	LSTM	(None, 5, 1024)
lstm_8	LSTM	(None, 1024)
repeat_vector_3	RepeatVector	(None, 28, 1024)
lstm_9	LSTM	(None, 28, 1024)
time_distributed_11	TimeDistributed	(None, 28, 256)
time_distributed_12	TimeDistributed	(None, 28, 256)
time_distributed_13	TimeDistributed	(None, 28, 84)
reshape_2	Reshape	(None, 3, 28, 28)

Table 8: Text-to-image model architecture for the '+' and '-' task. The total number of parameters is 21,321,044. All of which are trainable.

Name	Type	Shape
input	Input	(None, 3, 28, 28)
reshape_2	Reshape	(None, 3, 784)
dense_4	Dense	(None, 3, 364)
batch_normalization	BatchNormalization	(None, 3, 364)
dropout_2	Dropout	(None, 3, 364)
dense_5	Dense	(None, 3, 52)
batch_normalization	BatchNormalization	(None, 3, 52)
dropout_3	Dropout	(None, 3, 52)
dense_6	Dense	(None, 3, 13)

Table 9: MLP for evaluating the output of the text-to-image model. The total number of parameters is 307,073 of which 832 are not trainable.