Introduction to Deep Learning: Assignment 2

Group 6: Hedieh Pourghasem, Mojhde Hamidi, Chiesa Serena

November 2023

Introduction

In this assignment we are asked to create some encoder-decoder models for sequence to sequence problems like adding/subtracting/multiplying 2-digit integers, using LSTM(Long Short-Term Memory) network which is a type of RNN architecture.

We are given a math query datasets which consists of text and images of math queries as the input data and their results (also in text and image format) as the target data. Then it is asked to use this data for training and testing "text-to-text", "image-to-text" and "text-to-image" models and trying to learn how an encoder-decoder model works and connects with other neural network architectures.

1 Part 1: Addition and subtraction

Adding and subtracting are two simple operands of math world. In this section, the goal of our models is to output the result of applying these operands on two two-digit integers.

1.1 Analysis of Query Generation and Model Architecture

The **generate_images** function creates images of arithmetic operands ('+', '-', '*') in a form to be suitable to use alongside MNIST digit images(28x28 pixels). For minus and plus, lines are drawn with random coordinates for each image. For the asterisk, the process is similar to a plus, with an additional step of rotating the image.

The **create_data** function creates datasets for arithmetic operations by generating combinations of two integers (up to a specified maximum) along with an operand. This dataset consists of four parts: X_text with shape (20000,) (Text representation of arithmetic queries (e.g., '22+13'), X_img with shape (20000, 5, 28, 28)(Corresponding images, a sequence of MNIST digits and operand images), Y_text with shape (20000,) (arithmetic results in form of text), and Y_img with shape (20000, 3, 28, 28) (results in form of image).

For using X_text and Y_text as input or output of our models we need a way to convert them to one-hot encoding and also a way to do the reverse. In the one-hot format we have a 13-character vector that represents for 0-9 digits, +/- signs and a space as padding.The encode_labels and decode_labels are designed for this purpose. The **encode_labels** function converts text data into a one-hot format. Each character is represented as a unique vector (13-character long). This encoding is needed for training and evaluating the neural network. The **decode_labels** function transforms the one-hot encoded format (usually prediction from the model) into a string(in form of three-digit integer), with help of np.argmax. This will help us in the readability of the output.

Model Structure: The text-to-text model begins with an LSTM layer with 256 units which acts as an encoder and encode the input sequence. The input should be in a one-hot encoding format creating an input vector of shape [5, 13] for queries of length 5 with 13 unique characters. A RepeatVector takes the output from the LSTM and replicates it three times. This is because the maximum length of the result for our arithmetic operations is three. Using RepeatVector will result in equality between the length of the output the LSTM and the length of the sequence we want to generate. For the decoder part, another LSTM with 256 units is used. The difference from the encoder is that the return_sequences parameter is set to True. This is needed for generating a sequence of characters as output. At the end, a TimeDistributed wrapper is implemented which wraps a dense (fully connected) layer with a softmax activation to each time step of the output from the LSTM layer. In this way, in each time step the most probable output for that time step will be determined. Since the softmax activation outputs a probability distribution over possible characters.

The following models are different approaches to detect the underlying principles behind the "+" and "-" operands and associated operations.

1.2 Text-to-Text Model

We investigated the generalization capability of the provided RNN model in performing arithmetic operations. Different train-test splits (50-50, 25-75, and 10-90) were used to evaluate the model's learning and generalization efficiency. For all the experiments we trained the network for 20 epochs with batch size of 8. The analysis focuses on model accuracy, character-level misclassifications, and the nature of the mistakes made by the model. For the purpose of misclassification analysis we wrote a function **misclassified_samples_analysis(predictions, true_labels)**. The function iterates through the list of predictions and true labels, comparing them to find mismatches. It tracks the total number of misclassified strings (where the entire string is incorrect) and the total number of misclassified characters (where one or more characters in the string are incorrect). For a subset of misclassified samples, the function prints detailed information, including the index of the sample, the true label, and the predicted label. The **plot_difference** function is called, which computes the absolute differences between the true and predicted values for each sample where the prediction is different from the true value. These differences are counted and plotted to visualize the distribution of prediction errors. This is done using a log-scaled frequency plot, showing the frequency of each unique difference between predicted and true values (Some outlier such as small number of labels with 800 difference were ignored to have an understandable visualization), it also prints out unique difference with their frequency.

1.2.1 Model Performance and Generalization

As shown in Table 10, the model exhibits varying degrees of accuracy across different train/test splits. This variation indicates how the proportion of training data affects the model's ability to generalize to unseen data. In the 50% / 50% split, the model achieves high accuracies of **96.81%** and **97.24%** for training and testing, respectively, indicating effective learning and generalization. However, as the proportion of training data decreases to 25% and 10%, there is a notable decline in performance on both train and test sets also the generalization of the model decreases as we see test accuracies are about 3% lower than train accuracies. This trend suggests a strong dependency on the quantity of training data for optimal learning and generalization, with reduced training data leading to lower model performance and potential overfitting.

Train/Test Split	Train Accuracy	Test Accuracy
50% / 50%	**96.81%**	**97.24%**
25% / 75%	84.71%	81.15%
10% / 90%	64.65%	60.18%

Table 1: Text-to-text Model accuracies for different train/test splits

1.2.2 Misclassification Analysis

One most common type was one unit of difference between one digits for example the when the true label was 45, the network predicts is 46, or in some cases 35. Also, the model struggle with carry-over operations, as seen in predictions like '88-8' being '70', and '65+62' being '126'. Another rare mistake was that it predicts "- " for some samples which does not have any mathematical interpretation. In figure 1, we observe that for the 50% split which had the best generalization, most of differences are either 1 or 10 or some value near them. The errors in this network generally involve small deviations from the true label, often by ± 1 or ± 10. As the split increase these missclassified differnces become larger and more frequent which indicate that the model hasn't learn patterns effectively.

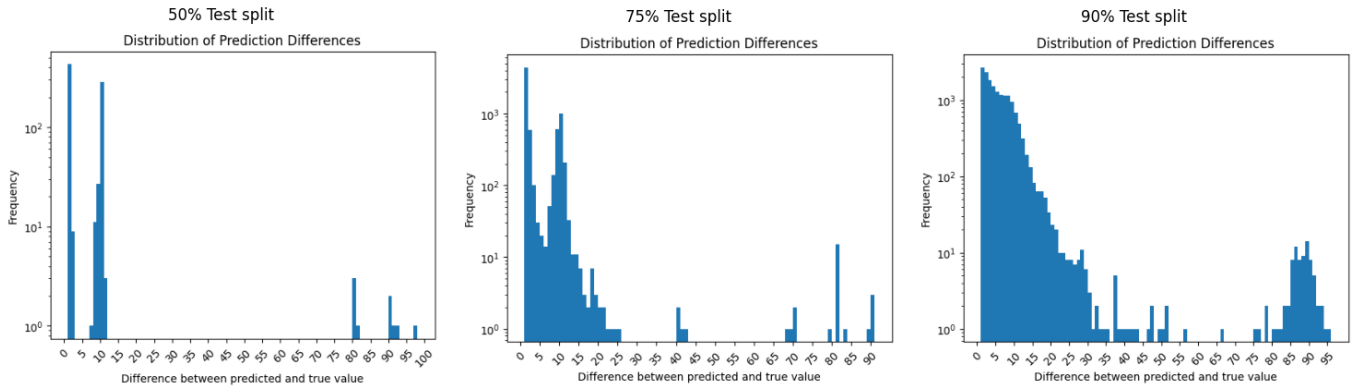

Figure 1: Distribution of number of digit differences in different trian/test splits

1.3 Image-to-Text Model

This model is considered to get five-channel images as input query and predict query results in text (one-hot) format. As the model needs to detect the features of the images to learn better, we decided to use CNN architecture. Because in RNNs, the output of each time step is connected to the next time step within the same layer, we used **TimeDistributed** layer to apply CNN layers to each time step independently. TimeDistributed layer acts as a wrapper for CNN and enables it to be applied across each time step of the input sequence independently.

At first, we implemented a combination of a single convolutional and a MaxPooling and a DropOut layer. MaxPooling layer helps our model robustness with downsampling the feature maps obtained from CNN layer, so defined it with default size (2,2). Dropout layer can play a significant roll to prevent overfitting, in our experiment its best value can be in the range 0.2 - 0.5. So we tried some values randomly from this range and 0.25 had best result. Although our input data was not a set of complex images with many features and the +/- operands didn't have complicated mathematical calculations, we noticed that using less filters didn't output desirable results, so we decided to increase the number of convolution filters to 512 with default kernel size. We flattened the output of DropOut layer to be compatible with the input of following dense layer. We tried different number of neurons for the Dense layer from 10 to 250. Having less neurons showed a better result in accuracy and we decided to keep it 80.

Now it's the time to pass the output of the Dense layer to RNN. With the help of predefined text-to-text model, we implemented our LSTM simply with 256 memory cells as our encoder. So as we need a decoder, a RepeatVector layer (which is a bridge to decoder) is defined with three timesteps (because the output length is 3). It repeats the vector from the LSTM encoder multiple times and creates a sequence of vectors that the decoder can utilize. It is the input of next layer which an LSTM plays as a decoder roll and processes these repeated vectors to learn to use this information for generating an output sequence.

We fitted this model on our dataset with test split rate 0.35, which was best choice based on our experiments. The accuracy of this model on test dataset was 0.92 which was almost good for this complex architecture. However we tried to optimize it with adding more layers or changing number of neurons, But the best result obtained from the first version of our model.

1.3.1 Model Performance and Generalization

Table 2 shows some of our experiments in optimizing model hyperparameters. The best accuracy of our model is related to first configuration which has the accuracy of **97.24%** for test dataset. Then to show how changes a parameter can make, we kept other parameters constant with its best value and changed the considered parameter. For example in the last configuration, we changed the number of neurons of the Dense layer (the Dense layer after flatten layer) to 200 then we saw a drop in the train accuracy in comparison with the first configuration. Moreover except for best configuration, we see a drop in test accuracy in other settings which indicates a fall in the generalization of our model.

Train/Test Split	Dropout	# Dense	# Convolutional	Train Accuracy	Test Accuracy
65% / 35%	0.25	80	512	**99.16%**	**92.97%**
75% / 25%	0.25	80	512	97.94%	92.73%
65% / 35%	0.25	80	128	98.47%	91.43%
65% / 35%	0.45	80	512	98.72%	92.00%
65% / 35%	0.25	200	512	98.78%	91.37%

Table 2: Comparison between the accuracy of different configurations of the Image-to-Text model for +/- dataset

1.3.2 Misclassification Analysis

As we achieved a high accuracy for our model, it's expected to have low missclassified predictions. So we checked the results which didn't match with the true targets. As it shows in Tabel 10, a small error happens in some cases which just one digit is predicted to its close value. It seems that we have gotten good results from our model, but in comparison with text-to-text model we encounter more errors, because text-to-text model is a simple model with the same format as input and target, so it's reasonable that our image-to-text model makes a little more mistakes.

True Label	Predicted Label
45	55
-20	-10
-32	-42
171	181
178	188

Table 3: Examples of Image-to-Text Model Misclassifications for +/- Dataset

1.4 Text-to-Image Model

Similar to our models in text-to-text and image-to-text, the structure of our text-to-image model follows the basic principles of encoder-decoder RNN. For our encoder, we opt for an LSTM layer featuring 512 units. We choose LSTM units because they work well with sequences, which is essential for our situation where the input is a series of characters representing arithmetic operations. It's worth noting that the LSTM layer doesn't give outputs one by one; it does so only after handling the entire input sequence, which represents the arithmetic operation in text form.

Following the LSTM layer, a dropout layer is introduced with a dropout rate of 0.2. Our reason behind using dropout is to help prevent overfitting. Subsequently, we utilized the RepeatVector layer to ensure that the output sequence matches our desired length of 3, corresponding to a sequence of 3 images.

Moving to the decoder, we use another LSTM layer with 256 units that indeed returns sequences, providing an output at each time step. Following this, a TimeDistributed wrapper is applied to a Dense layer. The TimeDistributed layer allows the Dense layer to be used for every part of the input sequence over time. Our selected setup for the Dense layer involves 7 * 7 * 128 units and uses the ReLU (Rectified Linear Unit) activation function.

The output from the Dense layer goes through a reshaping process to ensure it works well with the Conv2DTranspose layers that come next. These Conv2DTranspose layers play a crucial role in increasing the size of the feature maps and are commonly used in generative models. Our Conv2DTranspose layers have 64 and 1 filters, respectively, both with a kernel size of 3 and a stride of 2. The activation functions for these layers are different; the first uses ReLU, and the final layer uses the sigmoid function. We choose the sigmoid function in the last layer because it helps keep the output within the range [0, 1], which suits our grayscale image output. Before each Conv2DTranspose layer, we introduce BatchNormalization layers. These layers help speed up training and improve overall stability during the training process.

During model fitting, **X_text_hot** represents input text with character sequences encoding mathematical operations involving two two-digit numbers and an operator ('+', '-'). The output, **y_img**, signifies a sequence of images reflecting the accurate results of these calculations. For training and testing, we split the dataset into 80% for training and 20% for testing, ensuring reproducibility with a random seed set to 42.

1.4.1 Model Performance and Predictions

Train/Test Split	Test Accuracy	Test Loss
90% / 10%	85.87%	3.93%
80% / 20%	**86.17%**	**3.99%**
75% / 25%	85.48%	4.13%
60% / 40%	85.94%	4.29%
50% / 50%	85.18%	4.21%

Table 4: Text-to-text Model accuracies for different train/test splits

The model does well with a final loss of 3.99% and an accuracy of 86.17% on the test set with an 80:20 training-testing split. Initially, our accuracy was lower at 45.11%. But as we kept working on it, making changes by adding and removing layers, we aimed to find the best setup for better results. We enhanced the architecture by increasing LSTM units in the encoder to 512, introducing dropout for regularization, and adjusting the decoder with BatchNormalization before Conv2DTranspose layers. These changes allowed our model to capture intricate patterns, stabilize training, and better reconstruct the desired output sequence, leading to the significant accuracy boost.

We use our trained model to predict image sequences based on specific mathematical problems given as input texts. Our **predict_model** function encodes the input text, predicts the corresponding image sequence, and adjusts its format for display. We show this using three examples: '12-75', '42+15', and '48+80' to demonstrate how well our model creates meaningful visual representations. The **display_images** function helps us visualize these predicted image sequences in a simple grid format.

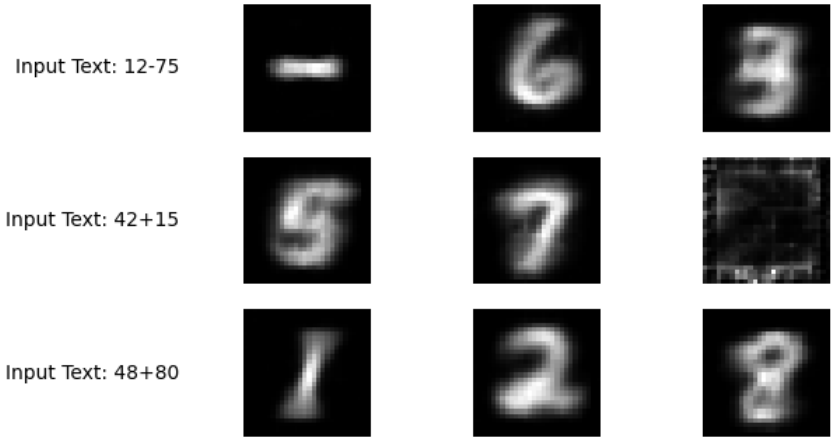

Figure 2: Some Examples of Text-To-Image Model Prediction Results

From Figure 3, the output images produced by our model display a slight blurriness, yet all the numbers remain distinguishable and recognizable. Despite this minor blurring, the images still have clear details, making it easy to recognize the numbers. The model is also good at getting the important details of handwritten numbers, showing it learned arithmetic operations well. While there's room for improvement in making the images sharper, the overall look of the generated images is promising.

1.4.2 Misclassification Analysis

From our examination of the images produced, we see that they generally do well in showing the numbers in the mathematical problems. But as we look closely, we notice some interesting things that tell us more about how well our model is doing:

1. When we input specific mathematical operations, like '48+80', our model generates images with clear representations of the intended numbers. However, when we make a slight change in the input, as in '49+80', we observe a decrease in the clarity of certain digits, particularly the number '2'. This change is likely because our model is sensitive to small differences in the input information.

2. Some digits, such as '8' and '3', may sometimes appear similar. This similarity between certain numbers might be because our model has a hard time telling apart certain shapes of numbers.

3. The differences in clarity we observe between similar inputs could be attributed to our model's sensitivity to certain numerical configurations. Certain combinations of digits might be tricky for our model, affecting how clear each number appears in the images it generates.

1.5 Adding Additional LSTM Layers to Encoder Networks

In this section we will investigate the implementation and results of adding one and two additional LSTM layers to our three different models.

1.5.1 Text-to-Text Model

Implementation: The first LSTM layer is set with return_sequences=True to return the full sequence of outputs. The additional LSTM layers are added, also with return_sequences=True. The final LSTM layer before the RepeatVector doesn't use return_sequences=True, as the RepeatVector layer expects a single output from the preceding layer and the rest of the model remains the same. We split the data to 50% train and 50% test sets and train the models for 20 epochs with batch size of 8.

Results: The results, as summarized in Table 5, indicate interesting trends in model performance with the addition of LSTM layers.

The train and test accuracies without any additional LSTM layer are quite close, indicating that the model generalizes well to unseen data. This balance suggests that a single LSTM layer is capable of capturing the essential features of the dataset without overfitting. We observe that by adding one additional LSTM layer both the training and test accuracies are notably higher than the model without the additional layer. The improved test accuracy, in particular, suggests that the additional LSTM layer enhances the model's ability to capture more complex patterns

Model Description	Training Accuracy	Test Accuracy
Without Additional LSTM	96.81%	97.24%
With One Additional LSTM	**98.08%**	**98.61%**
With Two Additional LSTMs	73.56%	73.01%

Table 5: Comparison of Model Accuracies with Different Numbers of additional LSTM Layers

and relationships in the data, leading to better generalization. But when two additional LSTM layers are added we have a huge decrease in performance which suggests that the model is becoming too complex, making it challenging to train effectively. The similar low performance on both training and test sets indicate underfitting or difficulty in optimization due to the increased complexity.

Performance differences in the context of the mistakes: In the network without additional LSTM, the errors mostly involve small deviations from the true label, often by ±1 or ±10. These examples suggest that the network has a basic understanding of the arithmetic but struggles with precise calculations, leading to consistent but small errors. It also seemed to struggle a bit with carry-over operations, as seen in predictions like '65+62' being '126' instead of '127'. These errors indicate a limitation in complexity that a single LSTM layer can capture, especially in operations involving more complex calculations.

After adding one additional LSTM layer to the network, the nature of errors changed a bit. There are still mistakes involving a deviation of ±1 or ±10, but the overall some more complex calculations such as '65+62' are being done correctly. And deviations are really close to ±1 and ±10. Interestingly, there are also errors in simple calculations like '3+3' being predicted as '7'. This suggests that while the additional LSTM layer has enhanced the model's ability to capture complex patterns, it may also introduce some instability or overfitting.

Adding two LSTM layers resulted in a network that still struggles with precise arithmetic calculations, both in simple addition/subtraction and in more complex carry-over scenarios. It is notable that the number of these missclasificaitons increase in this model. This might suggest a trade-off where increased model complexity does not necessarily translate to improved accuracy in this context and it lost its generalizing ability on unseen data.

1.5.2 Image-to-Text Model

We tried some experiments of adding more LSTM layers to our model. So we kept the model with best accuracy and then added two LSTM layers one by one to the encoder. We added return_sequences=True to our first LSTM layer of Image-to-Text model to output its layer sequences. Then we added another LSTM layer after the first one with 256 neurons and return_sequences=False because RepeatVector needs only one input. As it is shown in Table 6, adding this LSTM layer has a good effect on the accuracy of our model on test dataset in comparison with its first version, however we see a small drop in the accuracy of training dataset. After that, we tried adding another LSTM layer while the return_sequences of the last one was set to True. So the result is adding another LSTM layer doesn't guarantee an increase in model accuracy as it is clear in last row of Table 6. In addition adding LSTM layer increased the runtime of our model, so the number of layers of an encoder is related to model architecture and its dataset. In some cases a complex encoder with multi LSTM layers can help to have a well-performing model and in some cases it doesn't work.

Model Description	Train Accuracy	Test Accuracy
Without Additional LSTM	**99.16%**	92.97%
With One Additional LSTM	98.54%	**95.25%**
With Two Additional LSTMs	96.12%	93.18%

Table 6: Comparison of Model Accuracies with Different Numbers of additional LSTM Layers

As we experienced a model with a higher accuracy (one additional LSTM layer) on unseen data, we saw less mistakes in the prediction. The mistakes were too close to its true classification label. For example if we expected to have "123" as the output of out model, we got "133" instead and it repeated few times in comparison with the predictions of our first model. The accuracy of our model dropped a little but we still got a good results in predictions, however the missclassifications happened more than the last model. So in conclusion, we experienced best results from having an encoder with two LSTM layers.

1.5.3 Text-to-Image Model

In our exploration of text-to-image models, we explored three different configurations to understand their impact on performance. The baseline model, which featured a single LSTM layer in the encoder with 512 units, served as the starting point for our analysis. Subsequently, we iteratively changed the model architecture by introducing one and two additional LSTM layers to the encoder.

Model	Accuracy	Loss
Baseline Model	86.17%	3.99%
One Additional LSTM	86.22%	4.21%
Two Additional LSTMs	86.25%	4.3%

Table 7: Performance Metrics for Different Text-to-Image Models On Test Dataset

Our baseline model performed reasonably well with an accuracy of 86.17%, indicating its ability to generate images based on the given mathematical operations. However, the addition of more LSTM parts to the encoder did not result in a significant increase in accuracy.

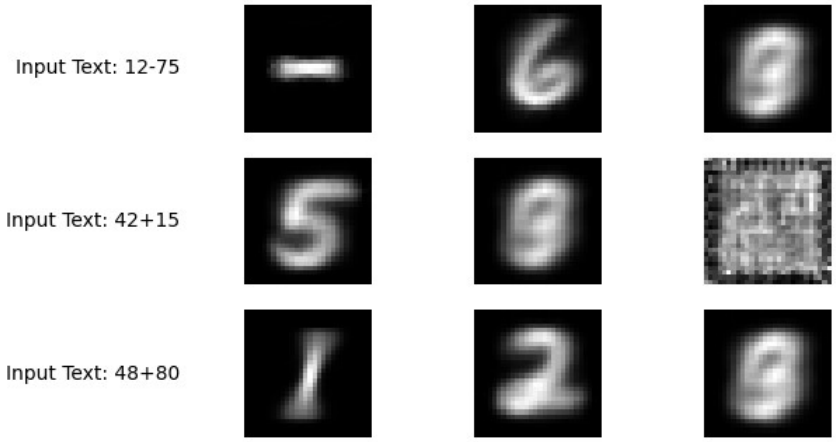

Figure 3: Some Examples of Text-To-Image Model Prediction Results

Despite the modest changes in accuracy, our analysis of the generated images revealed interesting character-specific nuances. For certain numeric characters, such as '-', '6', '5', '1', and '2', the introduction of two additional LSTM layers resulted in images with enhanced clarity and better-defined representations. In contrast, other characters, including '3', '8', and '7', displayed a degradation in image quality, with some becoming nearly unrecognizable.

The clearer and more defined images for specific characters with the addition of two LSTM layers can be explained by our model's ability to understand intricate patterns and details unique to those characters. These extra layers help the model grasp more nuanced information, leading to better visual quality for particular numeric representations. However, the varied effects on different characters imply that more adjustments and exploration of our model's structure may be needed to consistently improve all numeric characters.

In summary, our attempts to enhance the model yielded mixed results. While the overall accuracy did not show a substantial improvement, there were variations in the clarity of generated images for different numbers.

2 Part 2: Multiplication

For the purpose of training networks to learn multiplications, the overall dataset generation and encode and decode functions were mostly the same as part1. Multiplication operand is a more complex operation and we want to see how our models perform on this complicated training dataset. In this case we have unsigned answers with maximum length of 5 for each query and our one-hot vectors consist of 12 unique characters; 0-9 numbers, * sign and space.

2.1 Text to Text model:

We defined our model exactly the same as it was defined in task1. The only things that we changed were len(unique_characters) which was 12 for multiplication (it effects the input shape of LSTM encoder), and we set max_answer_length = 5 that we used in RepeatVector().

2.1.1 Improvements and Results:

We tried diffetent train/test splits (50% test, 40% test, and 60% test) near to the best split value that we had in task1. We also tried to see if we can increase the generalization and accuracy of the model by adding one additional LSTM layer or not. Furthermore, we experimented with increasing the output size of LSTM layers. In all experiments models were trained for 20 epochs and with batch size of 8. The results of this experiments are shown in table 8.

We assume the first row of the table as our baseline setting. After three experiments on train/test splits we conclude that 40% test split worked the best for our problem in terms of accuracy on both train and test set. This indicate that having more data for training helps the model to learn the multiplication task more effectively, improving its ability to generalize to the test set. The diminished performance with a higher test split (40%/60%) could be due to the model not having enough training data to capture the complexity of two-digit multiplication.

We can observe that adding an additional LSTM layer resulted in lower accuracies on train and test set in compare to the baseline setting. The decrease in accuracy suggests that the increased model complexity might not be beneficial for this specific task. The additional layer can lead to overfitting (as we observe 10% difference on train and test accuracy), where the model becomes too tuned to the training data and loses its generalization capability.

Finally, by increasing the output size of LSTM layers and having 40% test split we achieved the best result in both train and test accuracy. This suggests that larger LSTM units (more neurons) in the layers enhance the model's capacity to learn and generalize the multiplication task without the need for additional layers.

Train/Test Split	**Additional LSTM Layer**	**Output Size of LSTMs**	**Train Accuracy**	**Test Accuracy**
50%/50%	No	256	85.89%	78.59%
40%/60%	No	256	84.98%	75.57%
60%/40%	No	256	88.08%	79.33%
50%/50%	Yes	256	85.09%	75.72%
60%/40%	No	512	**93.86%**	**83.96%**

Table 8: Results of Configurations for Text to Text Model for Multiplication

2.2 Image to Text model:

The goal of this model is as the same as mentioned image-to-text model but its input is a query of multiplying two integers and the target length is extended to 4 consequently.

Like what we did in the last part, we chose to implement convolutional architecture but in this case we don't have a simple operation and our model needs to detect more features from input data. So we tried having one set of convolutional architecture with 512 filters just as the same as last model but didn't show a good result. So we increased the number of convolutional layers until we get a better result.

We tried to tune the architecture and hyperparameters and the final model that performed better that others is described as the following:
As multiplying is a more complex operation, we have to be sure that our CNN model can detect important features of input images. So we increased the number of convolutional layers to three and made some changes in each layer hyperparameters. First convolutional layer has 256 filters with kernel size = (5, 5), due to decrease the risk of overfitting and reduce computational cost, we decided to increase kernel size in first two layers because in this case we have a more complex model and therefor a longer running time. As the same as part 1, we defined MaxPooling and dropout layers after the convolutional layer. We repeated this set for our second convolutional layer with a change in the number of filters which was set to 512. In our experiments, adding another CNN layer increased the model accuracy. So in sum, we have three combination of convolutional and MaxPooling and DropOut layers before the flattening layer. We tried several number of neurons in the range of 10 to 250 for the Dense layer and 100 had the best result.

So we left our encoder-decoder architecture like part 1 but, as we needed to tune its hyperparameters, we tried 512, 256, 128 and 64 units for our decoder and we encountered 128 performed better than others. One of its reasons can be that we have a 12-character long string now so in this case, a decoder with a less units can be more appropriate.

We built this model, tried different test splits for training. Although with split ratio 0.5 we had a faster training time, it doesn't have a good effect on the model accuracy. In contrary, decreasing this ratio helped increasing accuracy a lot. Even though we had gotten good results with ratio= 0.35 in part1, the best result was obtained with 0.25 in this part. So this means that, in this scenario, the more data the model is trained from, the better predictions we can

get. Some of our experiment with different configurations can be shown as Table 9. It is clear that a small change in any hyperparameter value can make a huge drop in model accuracy. For example, we tried test split ratio = 0.35 and the test accuracy dropped to 38.31% which is too low.

Train/Test Split	**Decoder LSTM neurons**	**Dense neurons**	**Filters of 1st CNN**	**Test Accuracy**
75% / 25%	128	100	256	**76.34%**
65% / 35%	128	100	256	38.31%
75% / 25%	256	100	256	74.83%
75% / 25%	128	200	256	37.21%
75% / 25%	128	100	64	73.43%

Table 9: Comparison between the accuracy of different configurations of the Image-to-Text model for * dataset

So we investigated the model predictions of test dataset (which we converted them to integers with given "decoder" function) and compared them with their true values. We see more mistakes in predictions in comparison with the prediction of same model of part1. It's reasonable because it's a more complex problem and a more complicated model than part1 and besides that, the accuracy of the new model is less than it (Although it's more than 0.75, still is way less than 0.92). However, negligible mistakes are more common than significant ones. For example, Tabel 10 shows some missclassified predictions randomly. As it seems, in many cases, the prediction of whole number is true except for one digit (and in uncommon cases two digits) which its value is close to its actual value.

True Label	**Predicted Label**
2080	2220
1408	1488
2016	1976
2790	2990
1653	1533

Table 10: Examples of Image-to-Text Model Misclassifications for Multiplication Dataset

Conclusion

In this assignment, we learned about RNNs and one of its derivations as encoder-decoder. We tried to work with LSTM layers to create models that are able to add/subtract/multiply two integers in different format. We experienced some challenging ideas by having different format of output from the input and we combined our previous knowledge of other neural networks and architecture to achieve best results in models prediction.

Moreover, our exploration of adding one and two additional LSTM layers to text-to-text, image-to-text, and text-to-image models also reveals nuanced impacts. While a single LSTM layer enhances our text-to-text model performance, adding two layers results in potential underfitting. In our image-to-text model, the addition of two extra LSTM layers results in better outcomes compared to a model with only one additional LSTM encoder. Our text-to-image shows mixed results, with modest changes in accuracy but noticeable improvements in image clarity for specific numeric characters.