Arsen Ignatosyan (s4034538)
Andrew Caruana (s4014049)
Van Nguyen (s3726266)

Assignment 2: Recurrent Neural Networks

Introduction to Deep Learning

December 19, 2023

1 Introduction

The goal of this report is to showcase our results of designing RNN models capable of performing arithmetic operations with focus on addition, subtraction and multiplication. Our models were tasked not to memorize each possible operation but rather to discern and generalize behind these mathematical operations. RNN network in this assignment follow a usual pattern of having a encoder and decoder architecture to solve this sequence to sequence task.

1.1 Problem statement

Given input of strings e.g '22+13' or '33-44' the networks are supposed to reproduce results of '35' and '-11' respectively. The given input of strings defined as queries are 13 unique characters consisting of 0-9 digits and mathematical operands '+' and '-' and white spaces for padding.

Additionally, either input or output of model can be represented as MNIST handwritten dataset depending on specific model. MNIST dataset is a widely used benchmark dataset in the field of machine learning. The dataset consists of 28x28 grey-scaled digits from 0-9. In total there are 70 000 images which 60 0000 of them are divided into training examples and 10 000 of them are divided into test examples.

2 Default Jupyter notebook

This section will briefly go over the Jupyter notebook that was given to us to complete this assignment.

2.1 generate_data

This function serve a purpose of creating a synthetic images representing subtraction. The function takes two argument first **number_of_images_** and then **sign**. By default it creates images of '-' by generating random pair of points (x, y) and then drawing a line between them to resemble a subtraction sign. The coordinates of points are randomized to introduce a variability.

The function also has option to generate addition sign. Similarly to the minus sign it first draw two points and connects them and additionally to make a addition sign the function draws a perpendicular line. To introduce even more variety the second pair of points are also randomized.

2.2 create_data

The create_data function serves to generate dataset for training a text to image model. The function limits the dataset by using arguments such as **highest_integer**, **num_addends** and list of operands ('+','-') by default. The highest allowed integer in this case is 99 and default value for number of addends is 2 (1+1 are 2 addends and 1+2+3 are three addends). It creates four datasets:

1. X_text is a sequence of text as query formatted as '22+33'.
2. X_img a sequence of MNIST images corresponding to the each character in the querry with dimensionsof [5, 28, 28]
3. y_text a sequence of string representing the correct answer '55' with max of length 3: ['2', '156']
4. y_img a sequence of MNIST images representing the answer with dimensions of [3, 28, 28]

This function basically maps the characters and corresponding images to represent the images for both input and output.

2.3 encode_labels and decode_labels

These two functions are made to one-shot encode and decode text labels to suitable format for processing in RNN.

1. **encode_labels**: Takes a list of strings, each with a maximum length of 3, and performs one-hot encoding on each character. It returns a 3D NumPy array, **one_hot**, where each element corresponds to a one-hot encoded representation of a character in a label.
2. **decode_labels**: This function takes one-hot encoded labels and decodes them back into their original string format.

3 Addition & Subtraction

3.1 Text to Text Model

The text to text model is created that should take text of two integers with addition or subtraction sign and outputs the value that should be achieved in this logical operation in a form of text. The model consists of encoder and decoder RNNs with a base architecture presented in Figure 1.

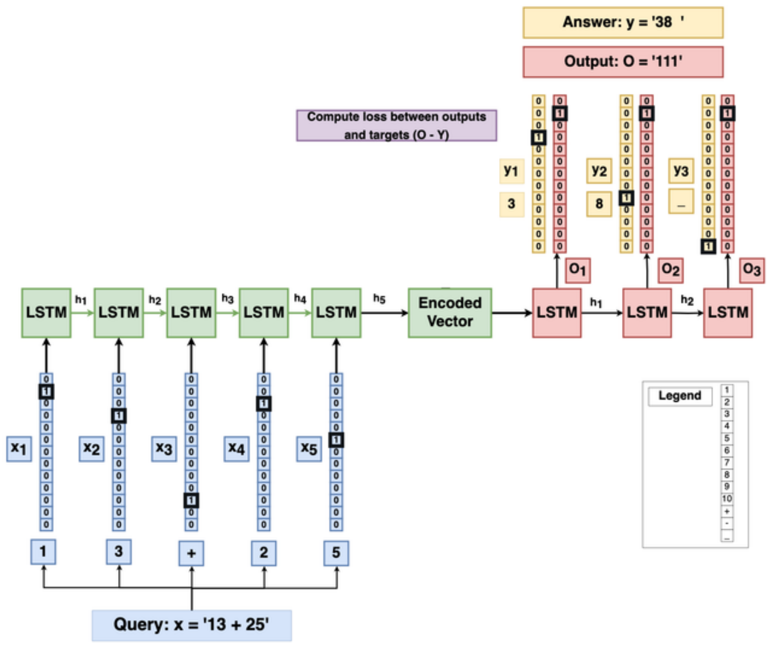

Figure 1: Base RNN architecture

The first task is to try different splits for training and test sets, and for this purpose the following values were chosen for the proportion of test data: 0.25, 0.5, 0.75, 0.9. The results of building the RNN model around these train test split datasets are shown in Figure 6. The leftmost figure of subplots is the boxplot of differences of true values and prediction, the center figure presents the histogram of differences of true values and prediction after the outliers have been removed (to make the plotting easier and results more visible), and its corresponding boxplot on the right side. Because of the seen results of generalization, the test split size of 0.25 is chosen for further experimentations, as the model with the most training data had much fewer errors with predictions that were really far away from the true value.

Experimentation with the architecture is done by stacking different number of LSTM layers to the encoder part. For the first experiment one additional LSTM layer is stacked on the encoder, and for the second one two additional layers are stacked. The results can be seen in Table 1. It can be seen that the heavier model performs better, but the trade-off between training time and performance increase is not worth it, as the RNN with only one additional LSTM performs almost perfectly.

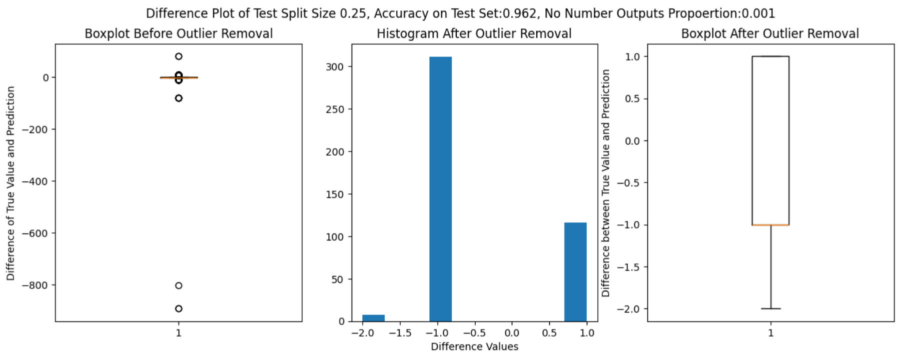

Figure 2: Test Ratio 0.25

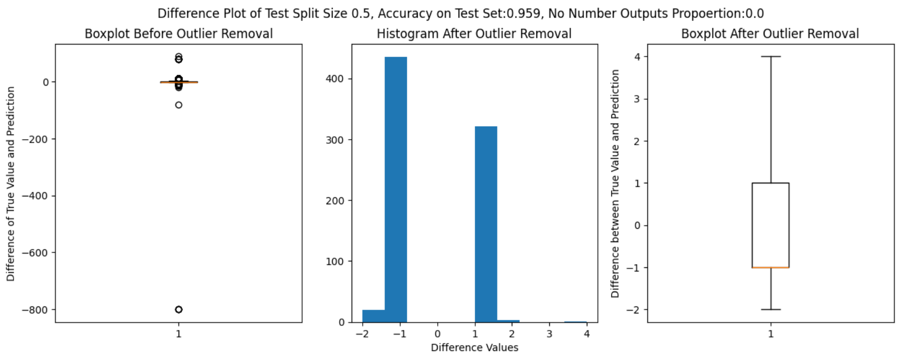

Figure 3: Test Ratio 0.5

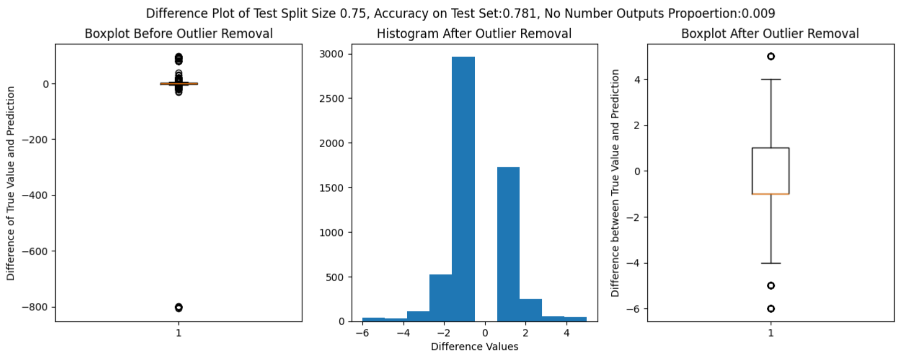

Figure 4: Test Ratio 0.75

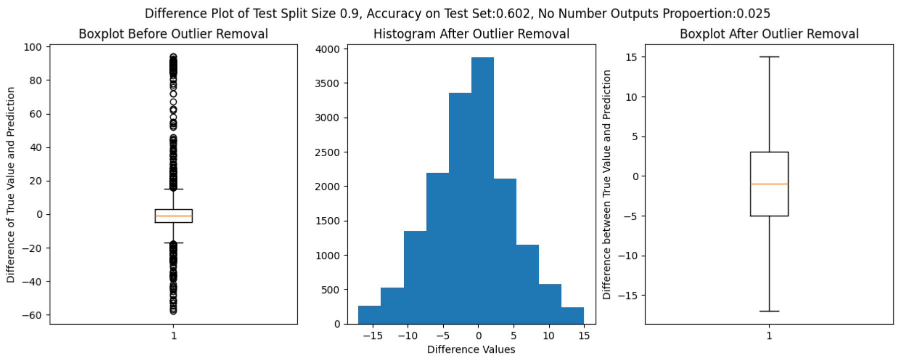

Figure 5: Test Ratio 0.9

Figure 6: Errors Displayed for Model's Trained with Different Split Sizes

Additional LSTM Layers	#Trainable Parameters	Train Accuracy	Test Accuracy
1	1,330,445	0.9963	0.9925
2	1,855,757	0.9745	0.9947

Table 1: LSTM Stacking Experiment

3.2 Image to Text Model

After the creation of the Text to Text model, we created an Image to Text model that takes images of the sums as input, and outputs the predicted result in text. It should be noted that the labels had the be one hot encoded in order for them to be used in the network.

Different parameters of the model were experimented with, and at first, CONVLSTM layers were used, however, we then switched to using separate time differentiated convoluted layers and LSTM layers. The results of our experimentations can be seen in the tables below.

Test Accuracy	Train Accuracy	LSTM Layers	Epochs	Learning Rate	Batch Size	Test Percentage
28	94	1	25	0.001	128	20
30	82	1	100	0.0001	128	20
30	84	1	100	0.001	128	20

Table 2: CONVLSTM Experimentation

Test Accuracy	Train Accuracy	LSTM Layers	Epochs	Learning Rate	Batch Size	Test Percentage	Kernel Size	LSTM Size
51	61	2	20	0.001	128	20	32	128
51	71	2	20	0.001	128	20	64	128
54	66	3	20	0.001	128	20	32	128
51	64	5	20	0.001	128	20	32	128
54	68	3	20	0.001	128	25	32	128/256
61	98	3	100	0.001	128	25	32	128/256
58	68	5	20	0.001	128	20	32	128/256
62	72	5	50	0.001	128	20	32	128/256
82	92	5	100	0.001	128	20	32	128/256
89	93	5	100	0.001	16	20	32	128/256

Table 3: Time Differentiated Conv Layers + LSTM Experimentation

The best results obtained can be seen in the final line of Table 2. We found that adding additional LSTM layers reduced overfitting, which was quite a big issue towards the start of the experimentation phase. This could be due to the increased memory capacity attributed to these extra LSTM layers, which allow the model to determine the context of the input more accurately, which for summation, could be handy. Furthermore, we found that decreasing the batch size made a decent difference as well. Adjusting the train/test ratio did not seem to have much of an effect on the results.

3.3 Text to Image Model

3.3.1 Early stages

We first set up model that was using Conv2DTranspose layers (Figure 7).

The accuracy of this model was around 40% which looked promising from the beginning but after training it, the model could not improve and the images do not look like expected results at all (Figure 8 and 9). Our guess is that the model tries to fill up the image with random pixels therefore making correct guess of pixel positions at 40% of the time. We abandoned this model and moved to a different structure.

3.3.2 Base model

Next model we tried was without Conv2DTranspoe decoder layers, instead we used TimeDistributed Dense layers.

Layer (type)	Output Shape	Param #
InputLayer	$(None, 5, 13)$	0
LSTM	$(None, 1024)$	4251648
RepeatVector	$(None, 49, 1024)$	0
Reshape	$(None, 7, 7, 1024)$	0
Sequential	$(None, 28, 28, 3)$	2731907

Figure 7: Conv2DTranspose model

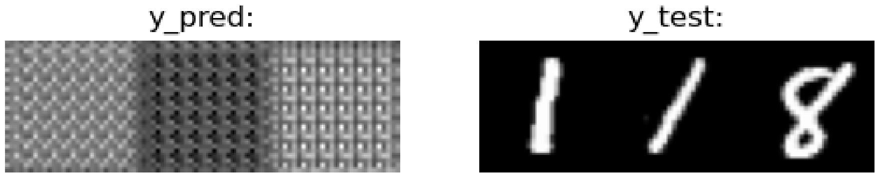

Figure 8: Example 1 picture from Conv2DTranspose

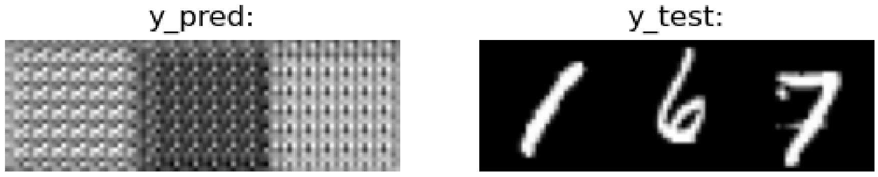

Figure 9: Example 2 picture from Conv2DTranspose

Layer (type)	Output Shape	Param #
LSTM	$(None, 256)$	276480
RepeatVector	$(None, 3, 256)$	0
LSTM	$(None, 3, 256)$	525312
TimeDistributed	$(None, 3, 128)$	32896
TimeDistributed	$(None, 3, 784)$	101136
Reshape	$(None, 3, 28, 28, 1)$	0

Figure 10: Time distributed dense layers model

First layer is encoding part where LSTM layer with 256 units. The input shape is set to (None, len(unique_characters)) which indicates variable lenght of input where each character get one-hot encoded.

The RepeatVector repeats the output of LSTM to match the desired sequence length for image length. The decoder part has another LSTM layer with 256 units as well. The flag **return_sequences=True** which is for producing a sequence outputs. TimeDistributed layers are applied to a fully connected Dense layers with 128 units and ReLU activation. Then another similar layer with Dense having 28*28 inputs and sigmoid activation function. At the end there's a Reshape layer that reshapes the output to the correct format (max_asnwer_lenght, 28, 28, 1)

The accuracy of the model stays around **86%** right from the first epoch (MSE loss function) but the goal of training it over more epochs is not to raise accuracy but more to improve the quality of the images.

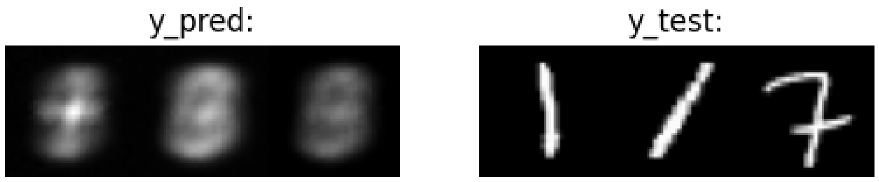

Figure 11: Model trained over 5 epochs

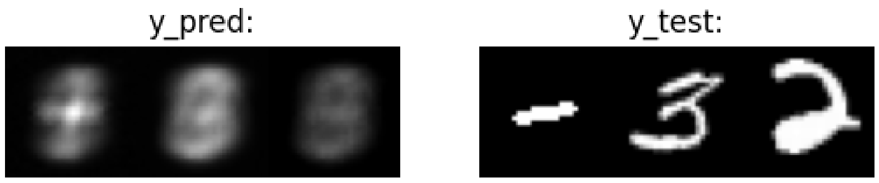

Figure 12: Model trained over 5 epochs

As seen in Figure 11 and 12 the model predicts basically same pictures for any scenario while still having high accuracy. The model gets much better over greater number of epochs and stabilizes around 100 epochs.

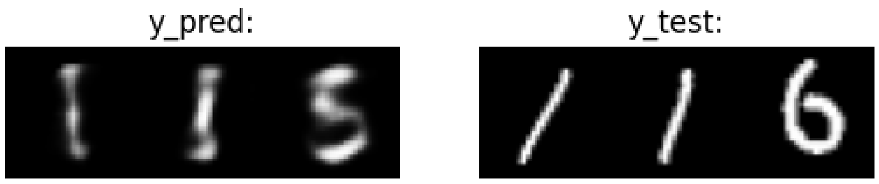

Figure 13: Model trained over 200 epochs

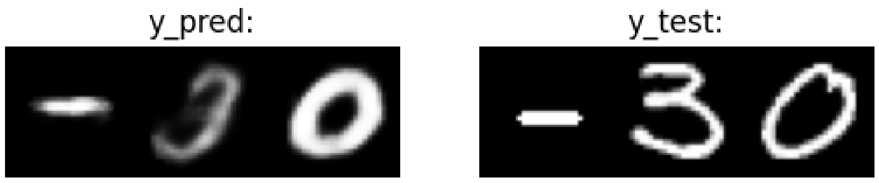

Figure 14: Model trained over 200 epochs

With 200 epochs the model becomes much more consistent and the images are much clearer but there are some more inaccuracies as shown in Figure 13.

3.3.3 Improving the quality

After on setting the number of epochs (quality peaks around 100 but to make sure we set 200) we decided to further improve the images. The accuracy using MSE loss function did not yield any improvement so we focused solely on looking at images.

The first thing we tried was introducing **Bidirectional LSTMs**.The hypothesis is that as it provides more benefits such as capturing context from both directions the image quality should improve at very least. The results are as follow:

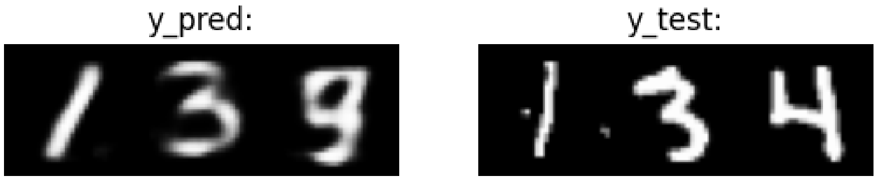

Figure 15: Enter Caption

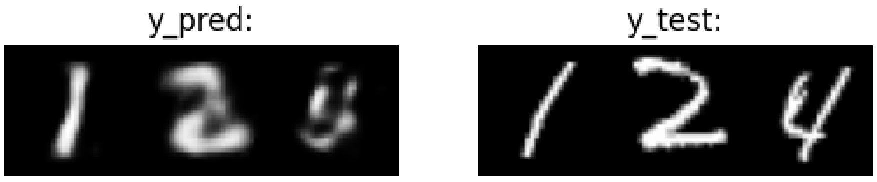

Figure 16: Enter Caption

There was no significant improvement nor decrease in quality so we decided to keep it.

Another experiment we tried to do is to set a **generate_binary_images** to be activated at lower threshold (0.4). The script makes prediction using model's specification and then using binary predictions generates the images with custom threshold. This will function should activate pixel even when they should not be activated resulting in potentially sharper images.

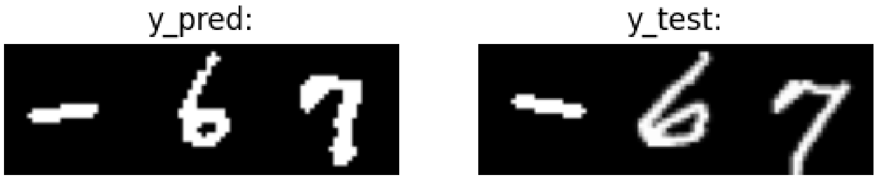

Figure 17: Threshold 0.4

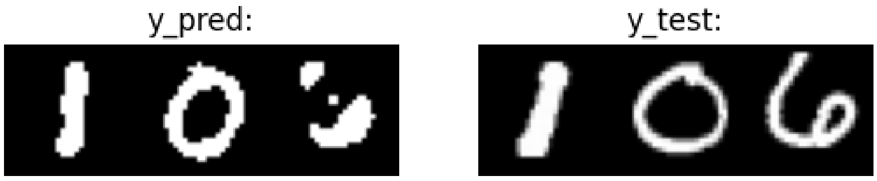

Figure 18: Threshold 0.4

As seen from figures 11 and 12 the images do improve in clarity but not in consistency.

Finally we introduced batch normalization and dropout to the model to see how regularization would affect the quality of the images and we found the best results so far.

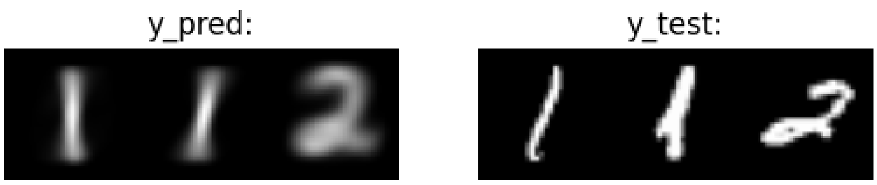

Figure 19: Model with regularization

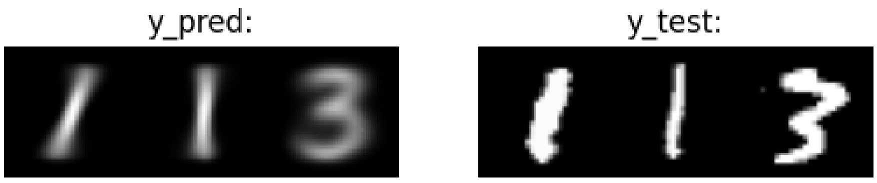

Figure 20: Model with regularization 2

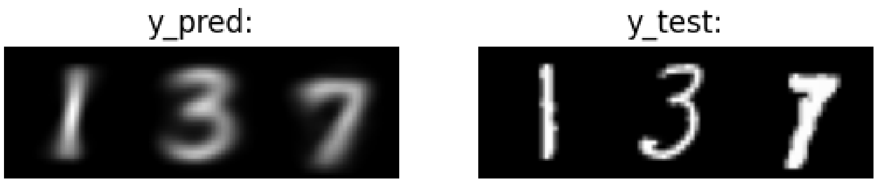

Figure 21: Model with regularization 3

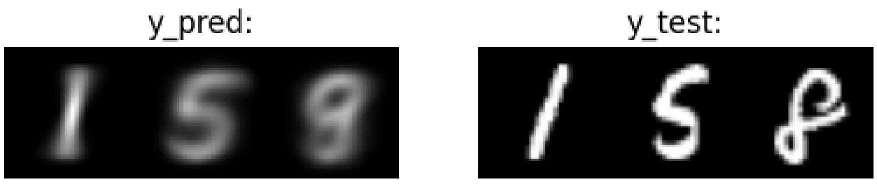

Figure 22: Model with regularization 4

As we can see from Figures 19, 20, 21 the images look much more consistent but at the conses of "sharpness". Using above mentioned threshold does not yield any significant improvement here. From Figure 22 we can still see some inconsistency in the model but that is to be expected.

3.3.4 Additional LSTM layer to encoder

Adding the additional encoder layer to text to image model did not yield any significant improvement either and therefore we stopped investigating after adding first additional layer.

To conclude the best and most consistent model we achieved was the base model with regularization introduced to it. Further experiment can be done such as sharpening the images or evaluating the images using models. Another thing that would be interesting is to build training model and instead of using loss functions to evaluate the model we could use the model evaluating the images to decide the accuracy.

4 Multiplication

4.1 Text to Text Model

Compared to the addition and subtraction task, multiplication task was more challenging for text-to-text RNN, as the output sequence is longer, and the differences between true values and predictions are larger. The experimentation is done in the same fashion as for the addition subtraction task, the test size ratio is again chosen to be 0.25.

Additional LSTM Layers	#Trainable Parameters	Train Accuracy	Test Accuracy
0	803,852	0.9137	0.8243
1	1,330,445	0.9309	0.8330
2	1,855,757	0.9066	0.8065
2	2,379,788	0.8535	0.7605

Table 4: LSTM Stacking for Multiplication Task

It can be seen from the results in Table 4 that adding additional too many LSTM layers to the encoder model results in worsening the model's performance. The differences between the true values and predictions can be seen in Figure 25.

4.2 Image to Text Model

Compared to the previous Image to Text Model for addition and subtraction, this model struggled with overfitting even more, and was tougher to get decent accuracies with. The model had to be tweaked and experimented with quite a bit. These experimentations can be seen in the table below. It should also be noted that the learning rate (0.001) and the amount of epochs (100) were kept constant throughout these experiments.

Test Accuracy	Train Accuracy	LSTM Layers	Batch Size	Kernel Size	LSTM Size	Conv Layers	Test Percentage
59	90	5	128	32	128/256	1	20
54	80	10	128	32	128/256	1	20
70	90	5	128	32/64	128/256	2	25
67	87	5	128	32/64	128/256	2	30
73	57	6	128	64	256	2	20
60	79	6	256	64	256	2	20
76	87	6	64	64	256	2	20
77	86	6	64	64/128	256	2	20
77	91	6	64	32/64	128/256	2	20
81	95	6	32	32	128/256	2	20
88	97	6	16	32	128/256	2	20

Table 5: Image to Text

As can be seen from the results, increasing the amount of LSTM layers and reducing the batch size seemed to have the best effect on the accuracy of the model. This could be attributed to increased context awareness brought about by the LSTM layers, along with the low batch size, which can add more noise into the dataset,

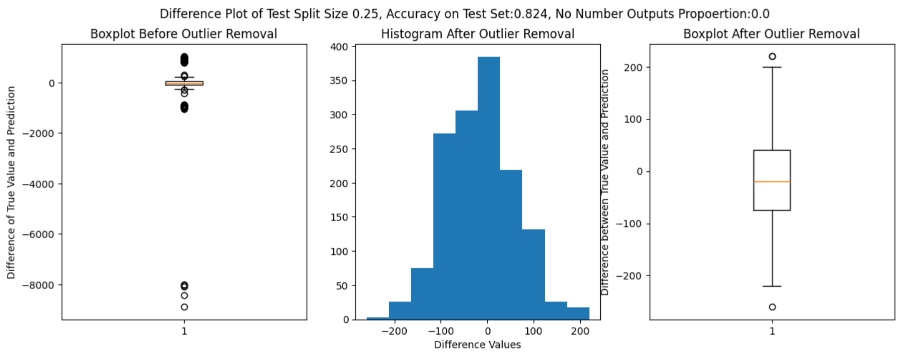

Figure 23: No Additional LSTM Layer

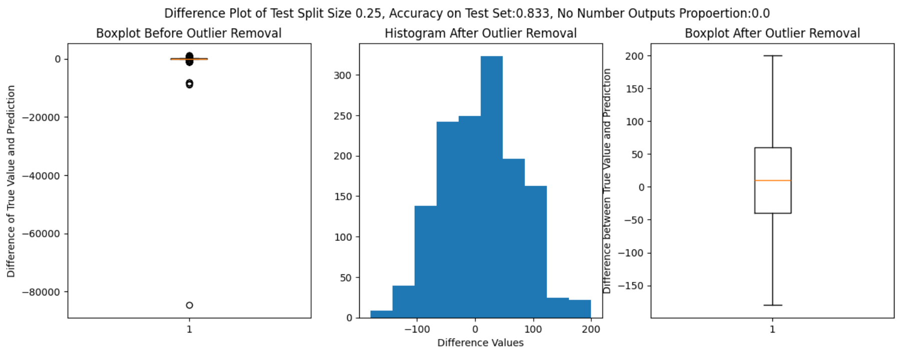

Figure 24: One Additional LSTM Layer

Figure 25: Errors Displayed for Model's Trained with Different Architectures

thus reducing chances of overfitting, which, paired with a low learning rate of 0.001 seemed to work quite well.