Generative models and sequence modelling

Assignment 2 of Introduction to Deep Learning

Xiangyuan Tao (4256778)
x.tao@umail.leidenuniv.nl

Kartikey Ahlawat (4178262)
k.ahlawat@umail.leidenuniv.nl

S.K. Reddy Mulakkayala (4238206)
s.k.r.mulakkayala@umail.leidenuniv.nl

ACM Reference Format:
Xiangyuan Tao (4256778), Kartikey Ahlawat (4178262), and S.K. Reddy Mulakkayala (4238206). 2024. Generative models and sequence modelling: Assignment 2 of Introduction to Deep Learning. In *Proceedings of Introduction to Deep Learning, 2024 (IDL '24)*. ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

In this assignment, we used Keras to design and train Generative and Sequence models. 1st task involving finding a suitable image dataset for the training of Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs) to generate novel images and evaluate them.
2nd task addressed a sequence-to-sequence problem. Built an Encoder-Decoder Recurrent Neural Network (RNN) that learns the principle behind simple addition and subtraction arithmetic. Divided into three main parts: text-text, image-text, and text-image. Parameter tuning and the addition of Long Short-Term Memory (LSTM) layers are done to enhance the performance.

2 TASK 1

We covered two generative models for generating novel images:

- Variational Autoencoders (VAEs)
- Generative adversarial networks (GANs)

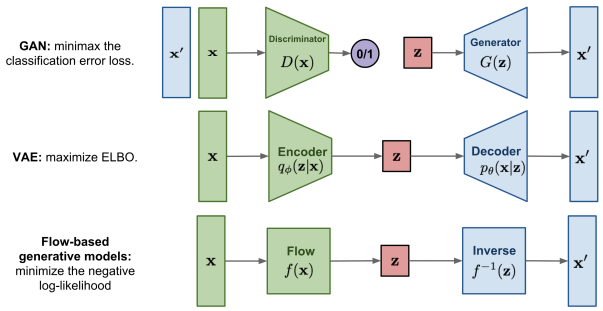


Figure 1: Flow-based Generative models

2.1 Dataset Description

We are using a dataset from Kaggle that contains anime faces. The dataset is stored in a .zip file, which we extract to access the images. After extraction, the images are saved as a .npy file, and we load it into memory for training. The dataset consists of 64x64 RGB images representing different anime characters. This dataset allows us to experiment with generative models like VAEs and GANs to generate new anime faces. This is the link to the dataset. Moreover, converting the dataset into .npy has its importance, like:

- Reduces the overall disk space required for storage compared to keeping a folder full of individual image files.
- With .npy, you can ensure the entire dataset is uniform.

After loading the dataset TensorFlow's ImageDataGenerator is used for augmentations such as random rotations, shifts, flips, and zooms. This will help to increase the diversity of the training dataset while improving the model's robustness.

2.2 Generative Model's Component

The generative models that we are going to cover have the following components:

(1) A downsampling architecture (encoder in case of VAE, and discriminator in case of GAN) to either extract features from the data or model its distribution.
(2) An upsampling architecture (decoder for VAE, generator for GAN) that will use the lower-dimensional latent vector to generate new samples that resemble the data on which it was trained.

Since we are dealing with images, we will use convolutional networks for up-sampling and down-sampling.

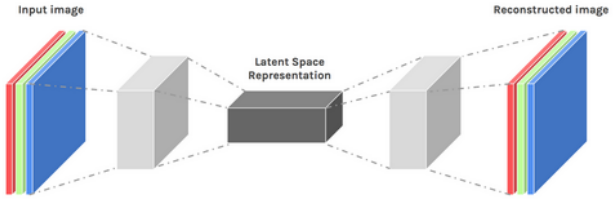


Figure 2: General encoder–decoder network

3 VARIATIONAL AUTOENCODERS (VAES):

A Variational Autoencoder (VAE) is a type of generative model that learns a latent representation of input data while simultaneously enabling the generation of new samples. Unlike traditional autoencoders, which directly encode and decode input data, VAEs impose a probabilistic structure on the latent space by modeling it as a continuous distribution. This allows the model to generate new samples by sampling from this latent distribution.
The VAE consists of three main components: the encoder network, the decoder network, and the reparameterization trick. Together, these components allow the model to learn a meaningful latent representation while enabling gradient-based optimization.

IDL '24, December 13, 2024, Leiden, the Netherlands
2024.

3.0.1 ***Encoder Network:*** The encoder network defines the approximate posterior distribution, which takes an observation as input and outputs a set of parameters that specify the conditional distribution of the latent representation. In this implementation, the posterior distribution is modeled as a diagonal Gaussian. The encoder outputs two key parameters: the **mean** (μ) and the **log-variance** ($\log(\sigma^2)$) of a factorized Gaussian distribution. Log-variance is used instead of variance directly for numerical stability during training. This allows the encoder to effectively learn a probabilistic mapping from input observations to the latent space.

3.0.2 ***Decoder Network:*** The decoder network defines the conditional distribution of the observation given the latent variable z. It takes a latent sample z as input and outputs the parameters for the conditional distribution of the reconstructed observation. In this implementation, the latent prior is modeled as a unit Gaussian distribution ($\mathcal{N}(0, I)$). The decoder's role is to reconstruct input observations by decoding the latent variables sampled from the learned posterior distribution.

3.0.3 ***Reparameterization Trick:*** A key challenge in training VAEs arises from the sampling operation within the latent space, which prevents backpropagation from flowing through a random node. To address this, the *reparameterization trick* is used. The latent variable z is reparameterized as:

$$z = \mu + \sigma \cdot \epsilon$$

where μ and σ are the mean and standard deviation of the Gaussian distribution derived from the encoder output, and ϵ is random noise sampled from a standard normal distribution ($\mathcal{N}(0, 1)$). This reparameterization ensures that the latent variable z is differentiable with respect to μ and σ, enabling gradients to propagate through the encoder while maintaining stochasticity through ϵ.

3.0.4 ***Implementation:*** The reparameterization trick can be implemented by creating a custom layer in TensorFlow by subclassing `tf.keras.layers.Layer`. This custom layer takes the mean (μ) and log-variance ($\log(\sigma^2)$) from the encoder and generates a sample from the latent distribution using the reparameterized formula. Additionally, a KL divergence loss term is introduced to regularize the posterior distribution to align with the unit Gaussian prior. The KL loss ensures that the learned latent space remains smooth and continuous, facilitating meaningful interpolation and sampling.

3.0.5 ***Training and Analysis:*** The Variational Autoencoder (VAE) was trained for 50 epochs with a latent dimension of 64 and a batch size of 8, using binary cross-entropy loss to minimize reconstruction error. During training, the loss steadily decreased, reflecting the model's progress in learning meaningful latent representations. Random latent vectors were sampled to generate and visualize images, showcasing the decoder's ability to produce realistic outputs. This demonstrates the VAE's capacity to encode input data into a structured latent space and generate diverse images.

3.0.6 ***Loss curve for VAE:.*** The loss curve for the VAE training reveals critical insights into the model's learning dynamics. During the initial epochs, there is a significant drop in loss, indicating that the model is rapidly learning to reconstruct input images and approximate the posterior distribution. Around epoch 10, the rate of loss reduction slows, suggesting that the model is nearing convergence. The flattened curve in the later epochs demonstrates stabilization, implying that further training provides diminishing improvements. This behavior reflects the VAE's ability to balance the reconstruction loss and KL divergence effectively, achieving a smooth latent space representation while maintaining the model's generative capacity.

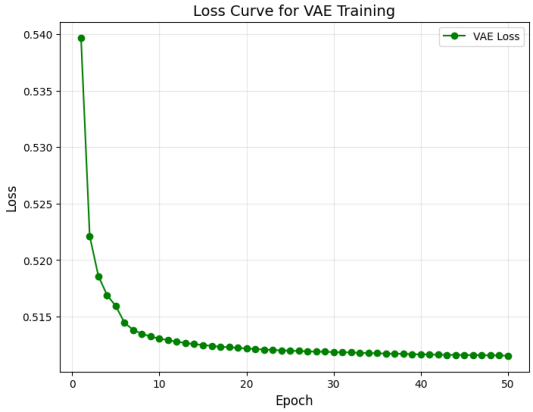


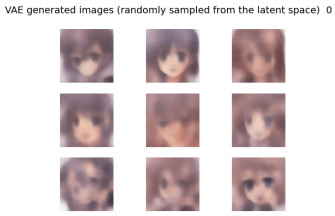
Figure 3: Loss Curve for VAE

3.0.7 ***Latent Space Visualization for VAE:.*** The t-SNE visualization of the VAE's latent space illustrates a structured and probabilistic organization of input data, driven by the model's ability to encode observations into a continuous latent distribution. Unlike deterministic models like CAEs, the VAE leverages its probabilistic framework to cluster images based on shared visual traits while preserving diversity through smooth transitions between clusters. This structure ensures that the latent space is not only informative but also generative, allowing meaningful sampling of new data points. The arrangement of clusters and their spread highlights the model's success in balancing reconstruction accuracy with latent space regularization via the KL divergence term.

Figure 4: Latent Space for VAE

3.0.8 ***Results of VAE Reconstruction Across Epochs:***

Randomly sampled from Latent space 1.



Randomly sampled from Latent space 25.

Randomly sampled from Latent space 50.

Analyzing the above images, we note that the generated images at epoch 1 are highly blurred and indistinct. The facial features, such as eyes and face contours, are vague and lack any meaningful structure.
By epoch 25, the images show substantial improvement in terms of structure and detail. Basic facial features, such as the shape of the face, eyes, and hair, are discernible, though still not fully sharp or realistic.
At epoch 50, the generated images exhibit high-quality reconstructions with clear and consistent facial features. The details of the eyes, hair, and face are sharper, and the images are visually closer to the original dataset.

The VAE loss, which combines reconstruction loss and KL divergence, shows improvement from 0.561 at epoch 1 to 0.512 at epoch 50, indicating effective but modest learning. To enhance performance, consider increasing the latent dimension, fine-tuning the learning rate, or balancing the KL divergence term to prioritize reconstruction during early training. Architectural improvements like deeper networks, residual connections, or advanced upsampling can capture more data complexity. Regularization (dropout or L2), data augmentation, and latent space exploration (e.g., t-SNE or interpolation) can further improve robustness and representation quality. Monitoring reconstruction quality and validation loss ensures the model generalizes well while maintaining a structured latent space.

4 GENERATIVE ADVERSARIAL NETWORKS

Generative Adversarial Networks (GANs) are a class of neural networks designed for generative modeling tasks. A GAN consists of two primary components: a generator and a discriminator. The generator learns to create realistic data samples (e.g., images) from random noise, while the discriminator classifies whether a given input is real (from the training dataset) or fake (produced by the generator). Both networks train simultaneously in a competitive manner: the generator tries to produce samples that fool the discriminator, while the discriminator tries to distinguish real from fake. This adversarial process allows the generator to improve its ability to produce data that closely resembles the training data.

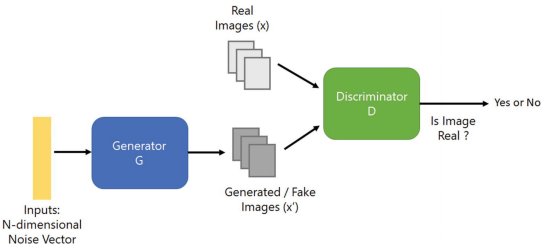


Figure 5: Generative Adversarial Networks GAN

4.0.1 ***GAN Architecture and Its Explanation:*** The architecture of a GAN involves two neural networks: the generator and the discriminator, each serving a distinct purpose.The discriminator model is a classifier that determines whether a given image looks like a real image from the dataset or like an artificially created image. This is basically a binary classifier that will take the form of a normal convolutional neural network (CNN). As an input this network will get samples both from the dataset that it is trained on and on the samples generated by the generator. The generator model takes random input values (noise) and transforms them into images through a deconvolutional neural network. Over the course of many training iterations, the weights and biases in the discriminator and the generator are trained through backpropagation. The discriminator learns to tell "real" images of handwritten digits apart from "fake" images created by the generator. At the same time, the generator uses feedback from the discriminator to learn how to produce convincing images that the discriminator can't distinguish from real images.

4.0.2 ***Latent Space:*** The latent space in GANs represents the input domain from which the generator samples random vectors. These vectors, typically drawn from a normal distribution, act as seeds for generating outputs. Each point in the latent space corresponds to a unique generated sample. By mapping random noise vectors to the data distribution, the generator learns to model complex patterns in the dataset. In the provided code, the latent space has 100 dimensions, allowing for a diverse range of generated outputs. The t-SNE visualization further illustrates how the latent space is organized: similar points in the latent space correspond to visually similar outputs, showcasing the generator's ability to structure the space meaningfully.

4.0.3 ***Training and Analysis:*** The GAN was trained over 50 epochs using TensorFlow's **fit** method for its efficiency and simplicity. The fit method manages epoch and batch iterations automatically, reducing code complexity and seamlessly integrating with callbacks like **LossesHistory** to track losses. This approach ensures optimized training and stable performance, enabling focus on model improvements rather than manual loop implementation.

4.0.4 ***Loss curve for GAN:.*** The loss curves for the generator and discriminator provide insights into the training dynamics. The discriminator's loss initially decreases as it learns to distinguish between real and fake samples. However, as the generator improves, the discriminator's task becomes more challenging, leading to slight fluctuations in its loss. Conversely, the generator's loss starts high and gradually decreases as it learns to produce more convincing samples. Balanced loss curves indicate successful adversarial training without mode collapse or overfitting. In this experiment, both losses converge, reflecting stable training.

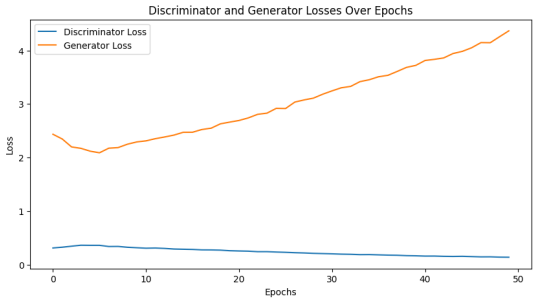


Figure 6: Loss over Epochs

Figure 7: Loss curve for GAN

4.0.5 ***t-SNE Visualization:*** The t-SNE visualization illustrates the structured latent space of the GAN's generator. Each blue dot represents a latent vector reduced to 2D for interpretability, with the corresponding overlay showing the generated image. Nearby points produce visually similar outputs, clustering features like hairstyles or colors. This demonstrates the generator's ability to map latent vectors to coherent, diverse, and realistic outputs, highlighting its understanding of the data's structure.

Figure 8: t-SNE Visulaization

4.0.6 ***Results:*** The GAN training results show significant improvements in the generator's ability to create realistic images over 50 epochs. At **epoch 1**, the discriminator loss (`d_loss: 0.31`) indicates it is relatively good at distinguishing real and fake images, while the generator loss (`g_loss: 2.5`) reflects the generator's initial struggle to produce realistic outputs. By **epoch 50**, the discriminator loss decreases (`d_loss: 0.13`), showing it remains confident, while the generator loss (`g_loss: 4.39`) increases slightly as it produces more convincing outputs that challenge the discriminator. The final images exhibit visually coherent anime-style faces with realistic features and diversity in hair color, facial structure, and expressions. This indicates that the generator effectively learned the data distribution and improved its mapping from the latent space to realistic outputs.
The final images exhibit visually coherent anime-style faces with realistic features and diversity in hair color, facial structure, and expressions, indicating that the generator effectively learned the data distribution and improved its mapping from the latent space to realistic outputs.

4.1 Evaluating the models

4.1.1 ***Evaluation of Convolutional Autoencoder (CAE):.***

- cae.predict(dataset) reconstructs the input images.
- We visualize both original and reconstructed images
- MSE is calculated between the original and reconstructed images to quantify reconstruction performance.

The Mean Squared Error (MSE) for the Convolutional Autoencoder (CAE) is 0.022, which quantifies the reconstruction error between the input and output images.

4.1.2 ***Evaluation of Variational Autoencoder (VAE):.***

- The resizing ensures that the images are compatible with InceptionV3, which expects images of at least 75x75 pixels.
- The FID score helps in assessing the quality of the generated images by comparing the distribution of features in the real and generated images.

The FID score is $1989424.07 - 0.003j$, which evaluates the similarity between the distributions of generated and real images.

4.1.3 ***Evaluation of Generative Adversarial Network (GAN):.***

Figure 9: (Original and Reconstructed Figures)

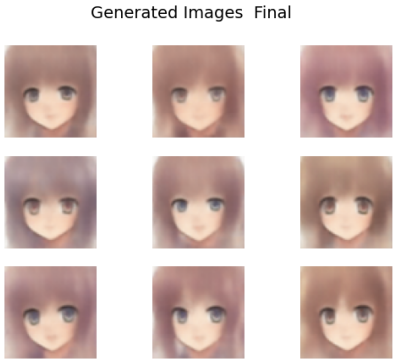


Figure 10: (Final Generated Image)

- Qualitative Evaluation: Visualize the GAN-generated images by feeding random noise into the generator.
- FID Score: Like the VAE, compute the FID score to quantify the quality of generated images by comparing real vs. generated distributions.

Figure 11: (Final Generated Images for GAN)

The FID score for the GAN is $1495560.58 - 0.03j$, measuring the similarity between the real and generated image distributions.

5 TASK 2

5.1 Background

This task focuses on building three different encoder-decoder recurrent models to handle the sequence-to-sequence problem of simple arithmetic ('-' & '+') operation applied between two 2-digit numbers. Each model is trained to handle its respective scenario: text-text, image-text, and text-image. A custom dataset was created to train the models. To generate image queries of simple arithmetic operations such as '29+1' or '3+3' MNIST handwritten digit dataset was used to represent the digit and OpenCV created images of '-' and '+' matching the dimensionality of MNIST (28x28).
The final dataset has 20,000 instances (addition and subtraction between all two integers). Having 4 sets, namely: X_text, X_image, y_text, and y_image. X_image having the shape of (5x28x28), representing a sequence of 5 images, each of shape (28x28), and y_image having the shape of (3x28x28), representing a sequence of 3 images, each of shape (28x28) as an answer to the query. Similarly, X_text is a string having the size 5, representing a sequence of 5 characters ('23-43'), and y_text having the size 3, representing a sequence of 3 characters ('-20') i.e., the answer from X_text.

Table 1: Overview of the Custom Dataset

Instances	X_text query size	X_image shape	y_text query size	y_image shape
20,000	5	5x28x28	3	3x28x28

5.2 Text-Text Scenario

The goal of this scenario is to develop an encoder-decoder recurrent neural network (RNN) model that converts arithmetic queries in text format (e.g., 23-43) into their corresponding text-based results (e.g., -20).

5.2.1 ***Data Preparation:*** In the text-text scenario, both inputs and outputs are represented as one-hot vectors, utilizing a dictionary of 13 unique characters (0-9, +, -, and whitespace). The encoder processes the input query, while the decoder generates the corresponding output sequence. Each query and result are transformed into one-hot encoded tensors to enable efficient processing by the RNN model.

Input Queries (X_text): A tensor of shape (20,000, 5, 13), where 5 is the query length, and 13 is the size of the character set.

Output Results (y_text): A tensor of shape (20,000, 3, 13), where 3 is the result length, and 13 is the size of the character set.

5.2.2 ***Model Architecture:*** As shown in Figure 1, the encoder employs LSTM layers to encode the input sequence into a fixed-length context vector. The decoder utilizes another LSTM layer to generate the output sequence from this context vector. A softmax layer is then applied to the decoder's outputs to predict the character probabilities at each timestep.

5.2.3 ***Training and Evaluation:*** The models were trained using the categorical cross-entropy loss function, optimized with the Adam optimizer to ensure efficient learning and convergence. Various training-test splits were explored to assess the model's generalization capability.

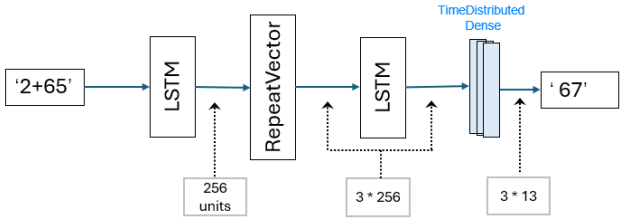
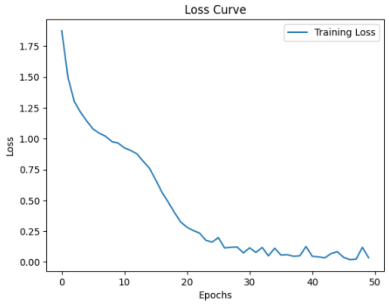
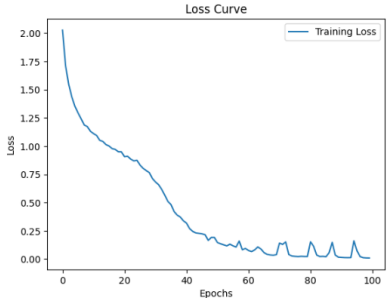


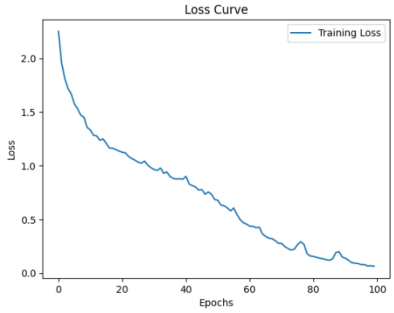
Figure 12: Text-Text Recurrent Model Structure



((a)) 50%-50% split



((b)) 25%-75% split



((c)) 10%-90% split

Figure 13: Training Loss Curves for Three Data Splits

Table 2: Train and Test Accuracy/Loss for Different Splits of Text-Text Model

Split	Train Accuracy	Train Loss	Test Accuracy	Test Loss
50-50	98.79%	0.0409	99.34%	0.0286
25-75	99.96%	0.0129	97.36%	0.0870
10-90	99.48%	0.0599	72.94%	0.9134

As shown in Table 2, the training and evaluation of the text-text model were conducted using three different training-test splits: 50%-50%, 25%-75%, and 10%-90%. The key observations are as follows:

50%-50% Split: The model achieved 98.79% train accuracy and 99.34% test accuracy, with low train loss (0.0409) and test loss (0.0286). The loss curve in Figure 2(a) is smooth and shows steady convergence, indicating stable training and excellent generalization. The high test accuracy reflects the model's ability to learn arithmetic principles effectively with sufficient training data.

25%-75% Split: The model achieved near-perfect 99.96% train accuracy and 97.36% test accuracy with train loss (0.0129) and test loss (0.0870). The loss curve in Figure 2(b) is similarly smooth, but the slightly higher test loss and lower test accuracy reflect the challenges of generalizing with reduced training data. Despite the smaller training set, the model demonstrated strong performance and stability.

10%-90% Split: With only 10% training data, the model still achieved 99.48% train accuracy, but the test accuracy dropped to 72.94%, with a much higher test loss (0.9134). The loss curve in Figure 2(c) is stable but reflects slower convergence due to the limited data. While the model performed perfectly on the training data, the test results indicate overfitting, as the model struggled to generalize to unseen queries.

5.2.4 ***Error Analysis:*** Since the 50%-50% split achieved excellent test accuracy and generalization, we use it to further analyze the mistakes made by the text-text model. Intuitively, we hypothesize that the model may find Borrowing and Carrying operations challenging to learn. Additionally, subtraction, which often involves handling negative results, might be harder for the model to master. To investigate further, we present a breakdown of error rates for different arithmetic operations and processes, as shown in Table 3.

Table 3: Error Analysis of Text-Text Model

Metric	Error Rate (%)	Error Count/Total Queries
Total Misclassifications	1.94	194
Character-Level Accuracy	99.34	–
Addition (+) Error Rate	1.68	84/5003
Subtraction (-) Error Rate	2.20	110/4997
Borrowing (→) Error Rate	1.78	58/3253
Carrying (←) Error Rate	0.83	29/3505
Borrowing + Carrying Errors/Total Errors	44.85	87/194

From Table 3, the addition (+) error rate is 1.68%, while the subtraction (-) error rate is slightly higher at 2.20%, likely due to the additional complexity of handling negative results and borrowing. Borrowing (→) and carrying (←) operations have error rates of 1.78% and 0.83%, respectively, indicating that these operations are

not disproportionately challenging for the model. Additionally, borrowing and carrying errors combined account for 44.85% of the total errors, aligning with their proportional presence in arithmetic tasks. These statistics suggest that no specific operation or process (e.g., borrowing or carrying) is the primary cause of the errors.

To gain deeper insights, we visualized (printed) the prediction errors to observe their patterns. The model frequently mispredicts results by 1 in all operations. For instance, the predicted value for the query 12-5 is 6, instead of the correct result, 7. The consistent occurrence of Off-by-One Errors across all operations suggests that the root cause lies in general numerical approximation and sequence decoding rather than challenges with specific arithmetic operations.

Adding another LSTM layer to improve the encoding and decoding mechanisms might help reduce the Off-by-One error to some extent. We plan to explore this enhancement in a later section.

5.3 Image-Text Scenario

The image-to-text model aims to solve arithmetic queries where inputs are represented as sequences of images and outputs are text-based results. This model processes queries like 23+45, represented as a sequence of 5 images (2, 3, +, 4, 5), and predicts the corresponding text-based result (68).

5.3.1 ***Data Preparation:*** For the image-to-text scenario, the dataset comprises 20,000 arithmetic queries represented as sequences of 5 images, each sized 28x28 pixels. Digits (0-9) were sourced from the MNIST dataset, while arithmetic operators (+, -) were created using OpenCV to match the MNIST dimensions. Each query (e.g., 23+45) was converted into a sequence of 5 images (2, 3, +, 4, 5), while the corresponding result (e.g., 68) remained in text format and was encoded using the same one-hot encoding scheme as the text-text model. The input data has a shape of (20,000, 5, 28, 28), and the output data has a shape of (20,000, 3, 13), ensuring compatibility with the model architecture. All images were normalized to the range [0, 1] to enhance training stability.

5.3.2 ***Model Architecture:*** The image-to-text model is designed to solve arithmetic queries represented as sequences of images, producing text-based results. As shown in Figure 3, the model employs an encoder-decoder architecture, where the encoder extracts spatial and sequential features from the input images, and the decoder generates the final result using these features. The model seamlessly bridges the gap between visual and textual data by integrating convolutional neural networks (CNNs) for image processing with recurrent neural networks (LSTMs) for sequence modeling. This design effectively manages the images' visual complexity and the arithmetic operations' temporal dependencies, enabling accurate translation of image-based queries into numerical results.

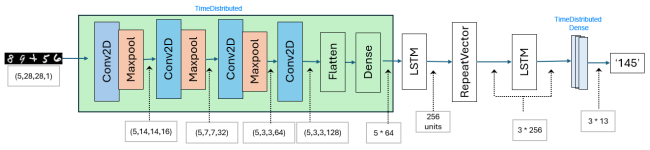


Figure 14: Image-Text Recurrent Model Structure

Table 4: Train and Test Accuracy/Loss of Image-Text Model

Split	Train Accuracy	Train Loss	Test Accuracy	Test Loss
50-50	98%	0.075	92.54%	0.334

5.3.3 ***Result:*** We used a 50%-50% split to train and test the image-to-text model. The model achieved a training accuracy of 93.89% with a loss of 0.3133. The accuracy curve (Figure 4) demonstrates steady improvement during training, with rapid gains in the initial epochs and convergence around 20 epochs. While the training curve remains stable in later epochs, indicating that the model successfully learned the task.

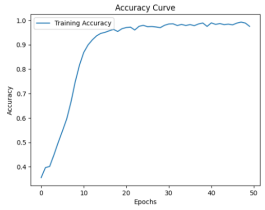


Figure 15: Training Accuracy for Image-Text Model

5.3.4 ***Text-Text model VS Image-Text model:*** The image-to-text model achieved a training accuracy of 92.54% with a loss of 0.334, while the text-to-text model significantly outperformed it with a test accuracy of 99.34% and a loss of 0.0286. The accuracy curve for the image-to-text model (Figure 4) demonstrates steady improvement, converging to near-optimal performance around 20 epochs. This is similar to the convergence speed of the text-to-text model, indicating efficient training dynamics despite the added complexity of processing image inputs. However, the image-to-text model's final performance remained lower, reflecting the challenges associated with visual feature extraction and sequence decoding.

The performance gap can be attributed to the inherent differences in input representation. The text-to-text model directly processes numerical queries, whereas the image-to-text model must interpret visual features from images before performing sequence modeling. This additional step introduces potential errors, such as digit misclassification (1 mistaken for 7) and operator confusion (+ vs. -). Both models exhibited Off-by-One Errors, but these were more frequent in the image-to-text model due to compounded inaccuracies during image interpretation and sequence decoding. Despite these challenges, the image-to-text model demonstrated stable convergence and reasonable accuracy, showcasing its ability to handle the added complexity of visual inputs.

In conclusion, while the text-to-text model outperforms the image-to-text model in terms of accuracy and loss, the latter effectively bridges the gap between image-based queries and text-based results. Improvements such as pretraining the CNN on digit classification, applying data augmentation, and incorporating attention mechanisms could further enhance the image-to-text model's robustness and accuracy.

5.4 Text-Image Scenario

For this task as well we used encoder-decoder architecture. The goal of the model is to output the answer to the text query ('11+19') i.e., sequence of length 5 into the sequence of 3 images each of shape (28x28) as a result ('3', '0', '').

Here, the encoder will take text as an input and encode it to a desirable state, which will be considered as an input for the decoder. Then decoder will formulate an image. As part of the encoder will apply input and LSTM layers. Input is taken in the shape of the 'number of timesteps of input sequence' (5) and the 'number of unique characters in the vocabulary' (13). Unlike in the 'Image-Text' scenario, will first apply the LSTM layer as now we already have the sequence in the desirable state for the LSTM. It will be set to return_sequence=False, as will only need the final. hidden state's latent vector of shape [batch_size, lstm_units] that summarises the entire input sequence (captures both structure and semantics of the input sequence).

Before providing the encoder-LSTM's output to the decoder, it needs to be passed through the RepeatVector layer. It ensures that the latent vector gets converted to a context vector having a latent vector across all timesteps in the output sequence. It is of shape [batch_size, image_shape[0], lstm_units]. Here, image_shape[0] refers to the number of digits/image in output, in our case, it would be 3. Now, in decoder will first pass it through an LSTM with return_sequence set to True. It will generate a hidden state for each timestep. It will also be of shape [batch_size, image_shape[0], lstm_units]. It is then passed to a Dense layer (fully connected) which converts hidden states into vectors of size 7x7x64, which acts as initial feature maps for the deconvolutional layer. Then this is passed through a Reshape layer that reshapes it to a 7x7x64 tensor that acts as a spatial image representation. The output from this is of the shape [batch_size, image_shape[0], 7, 7, 64].

Now, once we have spatial image representation in the form of tensors, will start applying deconvolution using convolutional transpose (Conv2DTranspose()). Will pass it through two Conv2DTranspose layers every time increasing the dimension by 2 ($7X7 \rightarrow 14X14 \rightarrow 28X28$). The upsampling increases the spatial resolution of the feature maps. Moreover, for interim Conv2DTranspose layers 'ReLU' is used to capture complexities and 'Sigmoid' for the final layer, to ensure pixels are normalized (0,1) for the grayscale image. The final output is of shape [batch_size, image_shape[0], 28, 28, 1]. One thing to note is that after the decoder's LSTM will apply TimeDistributed to all layers (Dense, Reshape, Conv2DTranspose, etc.) to ensure the sequence nature of the data so that operations are applied to each timestep individually. Please find below the architecture of what we just discussed, Figure 16.

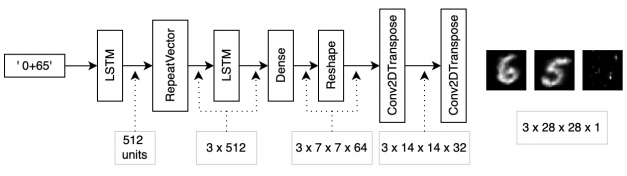


Figure 16: Text-Image Recurrent Model Structure

Table 5: Parameters of Text-Image Recurrent Model

Parameter	Value
LSTM_unit	256
Kernel_size	3x3
Strides	(2,2)
Padding	same
Epochs	30
Batch_size	32
Optimizer	Adam
Loss function	MSE

5.4.1 ***Hyperparameter Tuning:*** The initial result with the aforementioned parameters was satisfactory, but to improve it we performed experiments with different hyperparameter values.

It is hard to evaluate the text-image model as the training accuracy does not directly correlate with the generated image's quality. When I tried different hyperparameters receiving similar accuracies and loss but different image qualities. So, for this scenario, we chose visual analysis.

Continuing with hyperparameter tuning, it was observed that batch_size of 16 gives clearer image outputs with less pixel distortion than (8, 32, and 64), please find the below figures for the visual analysis.

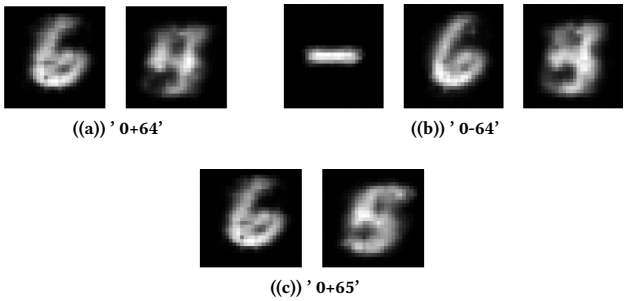


Figure 17: Output with Batch_size= 32

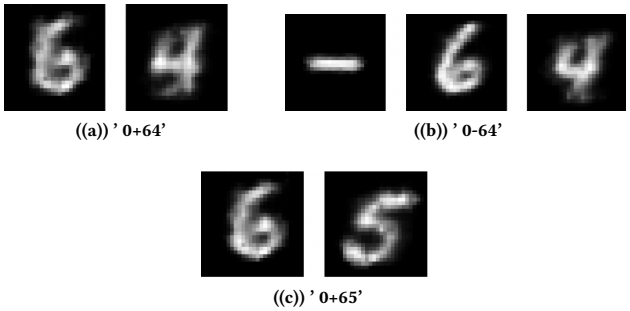
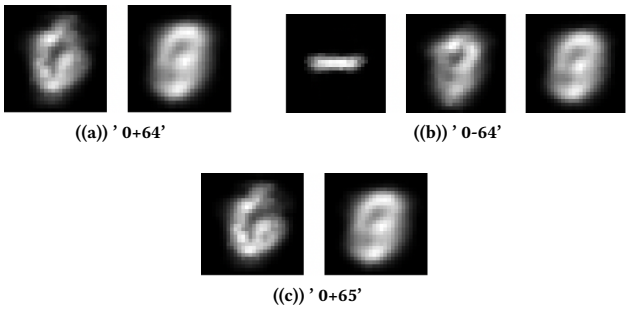


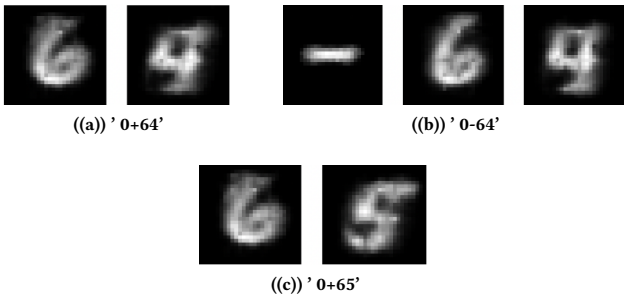
Figure 18: Output with Batch_size= 16



((a)) ' 0+64' ((b)) ' 0-64'

((c)) ' 0+65'

Figure 19: Output with Batch_size= 64

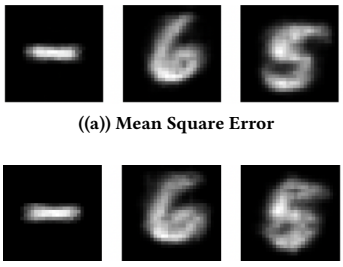


((a)) ' 0+64' ((b)) ' 0-64'

((c)) ' 0+65'

Figure 20: Output with Batch_size= 8

We also tested 'binary_crossentropy' as a loss function, but the pixels came out more distorted than in 'mse'.



((a)) Mean Square Error

((b)) Binary Cross Entropy

Figure 21: Output with Different Loss Functions

Moreover, as we used 'ReLU' in Dense and internal Conv2DTranspose layers, so initialized kernel_initializer with 'he_normal' as it is recommended for it [1]. But output images came out more blurry than it was by default ('glorot_uniform'). Experiments with LeakyReLU (alpha set to 0.1 & 0.01) also did not result fruitfully, giving blurred images. Also, optimizers like NAdam and AdamW did not exhibit any improvement over Adam. Coming on to epochs, 30 proved to be better than higher or lower epochs (15 & 50).

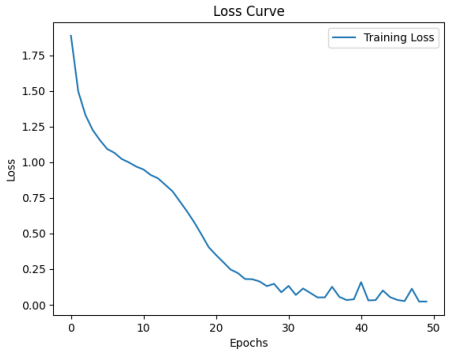
6 OBSERVATION ON ADDING ADDITIONAL LSTM

As directed in the assignment we tested all three models (text-text, image-text, and text-image) with an additional LSTM layer (return_sequence=True) in the encoder. Below are its outcomes:

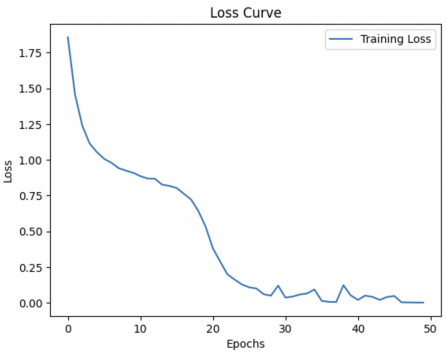
6.0.1 ***Text-Text Model:*** As evident from table 6 additional LSTM layer enhanced the performance of our text-text recurrent model. An increase in accuracies along with a decrease in error rate and loss were observed. Moreover, from figure 22 it can be seen that the loss curve is smoother with the additional LSTM layer. The additional layer helps in capturing complex patterns, evident from the 68.6% drop in misclassification.

Table 6: Text-Text: Evaluation of Additional LSTM (train-test: 50:50)

	Without additional LSTM	With additional LSTM	% change
Train_accuracy	98.79%	100%	1.22%
Train_loss	0.0409	0.002	95.1%
Test_accuracy	99.34%	99.81%	0.47%
Test_loss	0.0286	0.007	75.52%
Number of Misclassification	194	61	68.6%



((a)) Without Additional LSTM



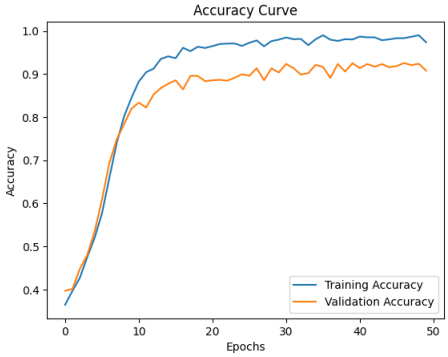
((b)) With Additional LSTM

Figure 22: Validation Loss across epochs

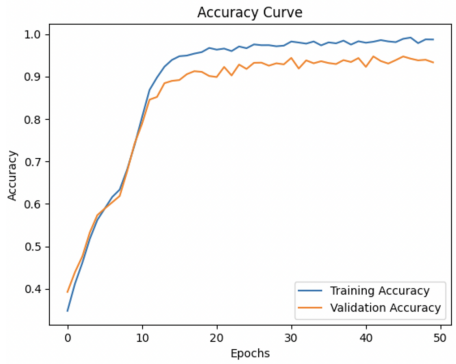
6.0.2 ***Image-Text Model:*** Similar to the text-text model, the image-text model also benefited from the additional LSTM layer in the encoder, as shown in the table 7 and figure 23.

Table 7: Image-Text: Evaluation of Additional LSTM

	Without additional LSTM	With additional LSTM	% change
Train_accuracy	98%	98.84%	0.86%
Train_loss	0.075	0.047	37.33%
Test_accuracy	92.54%	94.83%	2.48%
Test_loss	0.334	0.251	24.85



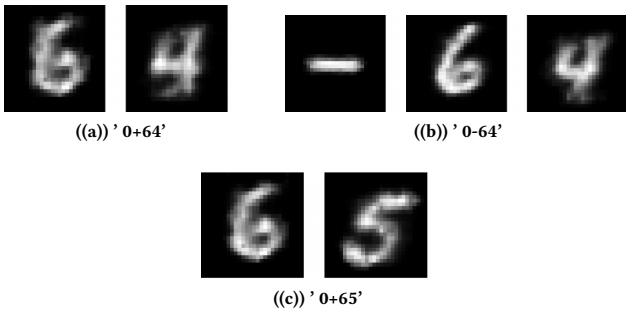
((a)) Without Additional LSTM



((b)) With Additional LSTM

Figure 23: Train/Validation Accuracy across epochs

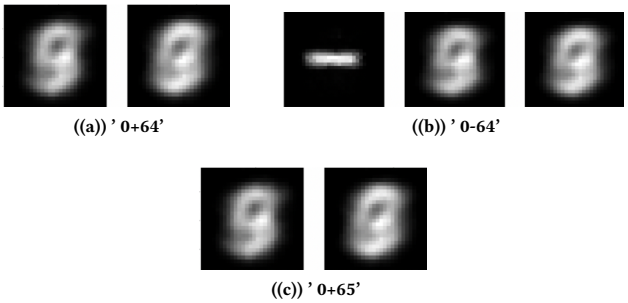
6.0.3 ***Text-Image Model:*** In contrast, the text-text and image-text models benefiting from the additional LSTM layer did not translate to the text-image model, as evident from the figures 24 & 25. The Model with additional LSTM generates images that are distorted enough to not be able to understand the digits.



((a)) ' 0+64' **((b)) ' 0-64'**

((c)) ' 0+65'

Figure 24: Text-Image: Output without additional LSTM



((a)) ' 0+64' **((b)) ' 0-64'**

((c)) ' 0+65'

Figure 25: Text-Image: Output with additional LSTM

REFERENCES

[1] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow.* O'Reilly Media, 2nd edition, 2019.

CONTRIBUTIONS

- **Tao:** Task 2 (2nd & 3rd) code+report and support in Task 1
- **Kartikey:** Task 2 (1st, 4th & 5th) code+report, Task 1 & Task 2 (2nd & 3rd) review, support in Task 1, and final report refinement.
- **Sai:** Task 1 code+report.