CS898BD: Deep Learning

Report

on

Assignment - 4

07/08/2025

Submitted by:
Rajeet Chaudhary
J992Y875

Contents

List of Tables

List of Figures

1 Introduction

In recent years, machine translation has significantly advanced due to the application of deep learning techniques, particularly Recurrent Neural Networks (RNNs). This assignment explores the task of translating English sentences to French using three popular RNN-based architectures: vanilla RNN, Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU). Each model is implemented as a two-layer stacked architecture to capture temporal dependencies in language sequences more effectively. The goal is to evaluate and compare their translation performance using a standardized small vocabulary dataset (`small_vocab_en` and `small_vocab_fr`). Through this comparison, we aim to identify which architecture provides the best balance of accuracy and efficiency for language translation tasks in low-resource settings.

1.1 Objective

The primary objective of this project is to design, implement, and evaluate deep learning-based models for English-to-French translation. Specifically, the goals are:

- To implement and train three types of stacked (2-layer) Recurrent Neural Network models:
  - Simple RNN
  - Long Short-Term Memory (LSTM)
  - Gated Recurrent Unit (GRU)
- To prepare the dataset using appropriate preprocessing techniques such as tokenization, encoding, and padding.
- To evaluate the performance of each model using training and testing accuracy/loss curves and training time.
- To compare the translated outputs from each model and determine which architecture performs best for the task.
- To present at least ten translation examples from each model to showcase their qualitative performance.

2 Methodology

2.1 Dataset Description

The datasets used for this task are named small_vocab_en and small_vocab_fr, containing parallel English-French sentence pairs. These datasets are designed to simulate a basic machine translation

task and include: A limited vocabulary of approximately 200 unique words in each language

The data is stored in plain text format and contains thousands of short, well-aligned examples such as "i am happy" and its translation "je suis content". These datasets are ideal for quickly testing the efficiency and accuracy of various neural network models in sequence-to-sequence tasks.

Figure 1: small_vocab_en and small_vocab_fr dataset description

3 Data Preprocessing

For this project, we won't use text data as input to your model. Instead, you'll convert the text into sequences of integers using the following preprocess methods:

- Tokenize the words into ids
- Add padding to make all the sequences the same length.

3.1 Tokenization

For a neural network to predict on text data, it first has to be turned into data it can understand. Text data like "dog" is a sequence of ASCII character encodings. Since a neural network is a series of multiplication and addition operations, the input data needs to be number(s).

We can turn each character into a number or each word into a number. These are called character and word ids, respectively. Character ids are used for character level models that generate text predictions for each character. A word level model uses word ids that generate text predictions for each word. Word level models tend to learn better, since they are lower in complexity, so we'll use those.

Turn each sentence into a sequence of words ids using Keras's Tokenizer function. Use this function to tokenize english_sentences and french_sentences in the cell below.

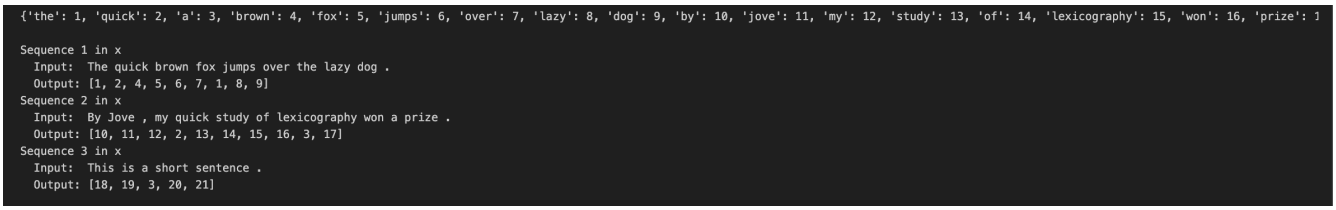


Figure 2: Tokenization

3.2 Padding

When batching the sequence of word ids together, each sequence needs to be the same length. Since sentences are dynamic in length, we can add padding to the end of the sequences to make them the same length.

Make sure all the English sequences have the same length and all the French sequences have the same length by adding padding to the end of each sequence using Keras's pad_sequences function.

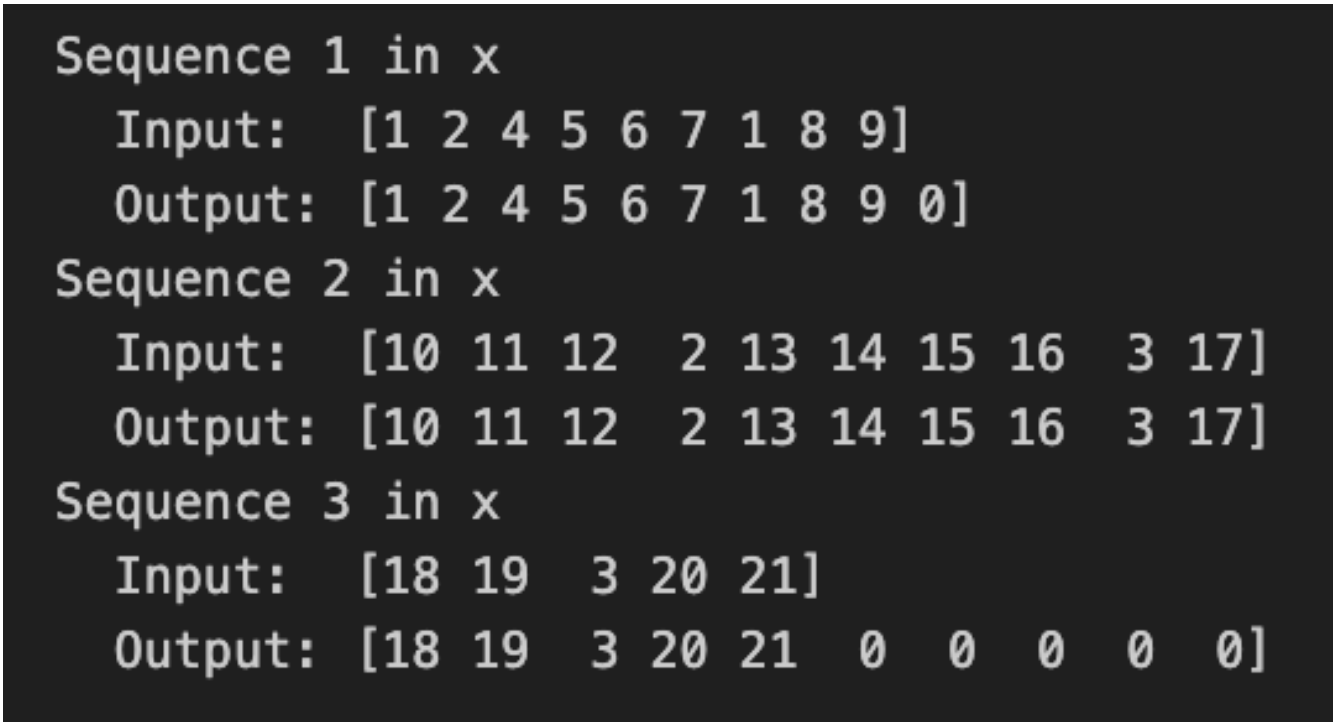


Figure 3: Padding

Model Architectures

1. Stacked 2-Layer RNN Model (using GRU Units)

The first model uses two stacked Gated Recurrent Unit (GRU) layers. GRUs are efficient variants of RNNs that help alleviate the vanishing gradient problem and are computationally less expensive than LSTMs.

- **Input Layer:** Accepts a padded sequence of English words.
- **GRU Layer 1:** 64 units with `return_sequences=True`.
- **GRU Layer 2:** 64 units with `return_sequences=True`.
- **TimeDistributed Dense Layer:** Applies a dense layer at each time step with output equal to the number of French vocabulary tokens.
- **Activation Layer:** Applies softmax activation to generate a probability distribution over the French vocabulary.
- **Loss Function:** Sparse Categorical Crossentropy.
- **Optimizer:** Adam optimizer with a learning rate of 0.001.
- **Metrics:** Accuracy.

Layer (type)	Output Shape	Param #
input_layer_5 (InputLayer)	(None, 21, 1)	0
simple_rnn_2 (SimpleRNN)	(None, 21, 64)	4,224
simple_rnn_3 (SimpleRNN)	(None, 21, 64)	8,256
time_distributed_4 (TimeDistributed)	(None, 21, 345)	22,425
activation_4 (Activation)	(None, 21, 345)	0

Total params: 34,905 (136.35 KB)

Trainable params: 34,905 (136.35 KB)

Non-trainable params: 0 (0.00 B)

Figure 4: Stacked 2-Layer RNN Model Architecture

2. Stacked 2-Layer LSTM Model

The second model replaces GRUs with Long Short-Term Memory (LSTM) layers. LSTMs are designed to capture long-term dependencies and perform well on sequence data.

- **Input Layer:** Sequence of embedded English words.
- **LSTM Layer 1:** 64 units with `return_sequences=True`.
- **LSTM Layer 2:** 64 units with `return_sequences=True`.
- **TimeDistributed Dense Layer:** Projects each time step's output to the French vocabulary size.
- **Activation Layer:** Softmax applied across each time step's output.
- **Loss Function:** Sparse Categorical Crossentropy.
- **Optimizer:** Adam (learning rate = 0.001).

- **Metrics:** Accuracy.

Layer (type)	Output Shape	Param #
input_layer_6 (InputLayer)	(None, 21, 1)	0
lstm_2 (LSTM)	(None, 21, 64)	16,896
lstm_3 (LSTM)	(None, 21, 64)	33,024
time_distributed_5 (TimeDistributed)	(None, 21, 345)	22,425
activation_5 (Activation)	(None, 21, 345)	0

Total params: 72,345 (282.60 KB)

Trainable params: 72,345 (282.60 KB)

Non-trainable params: 0 (0.00 B)

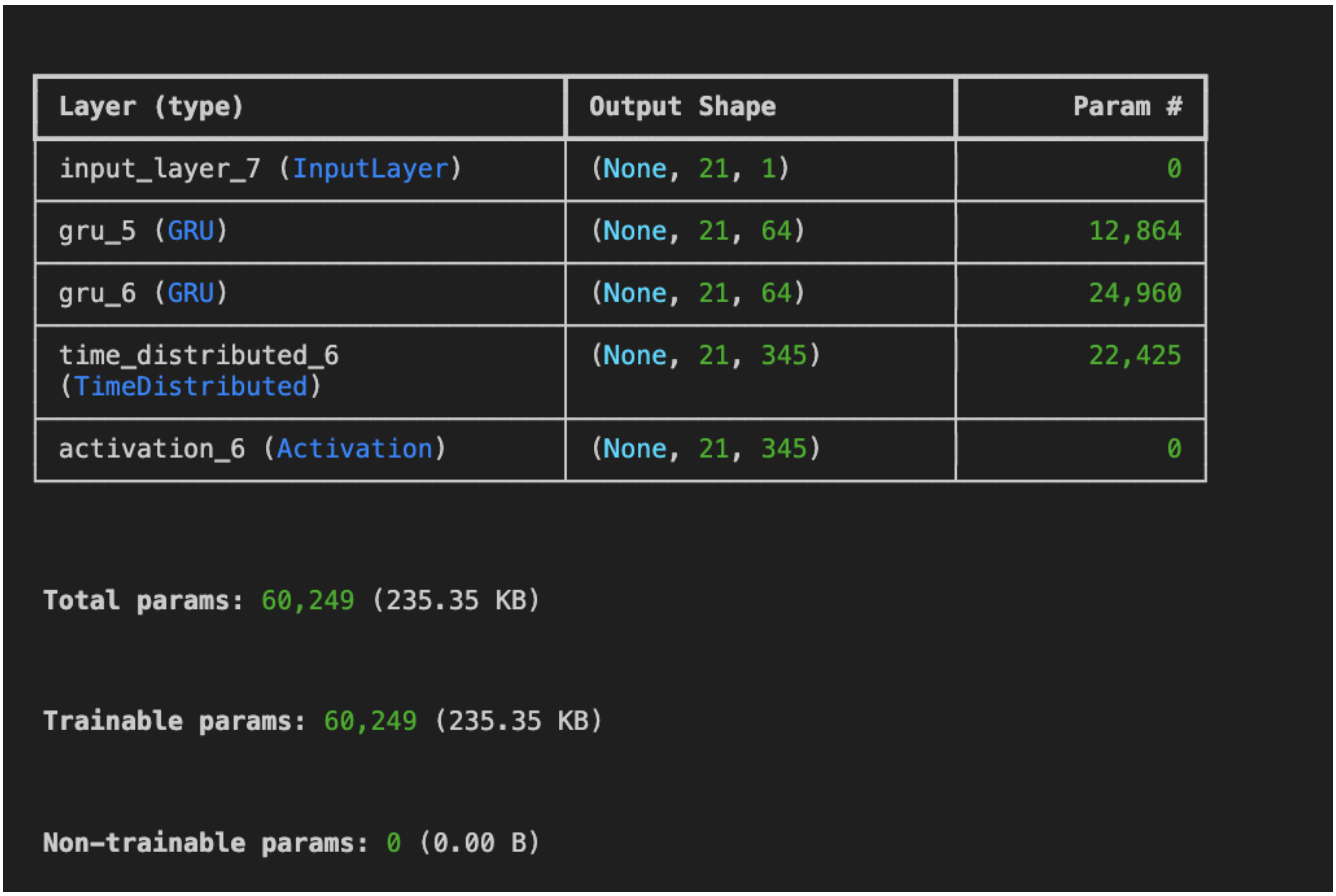
Figure 5: Stacked 2-Layer LSTM Model Architecture

3. Stacked 2-Layer GRU Model

The third model is architecturally identical to the first one and serves to compare results under a different training run and weight initialization. This repeated GRU-based model helps validate consistency and performance reproducibility.

- **Input Layer:** Takes a fixed-length English sentence.
- **GRU Layer 1:** 64 hidden units.
- **GRU Layer 2:** 64 hidden units.
- **TimeDistributed Dense Layer:** One dense layer applied to every time step output.

- **Activation:** Softmax for multi-class classification at each output step.
- **Loss:** Sparse Categorical Crossentropy.
- **Optimizer:** Adam optimizer.
- **Metrics:** Accuracy.



Layer (type)	Output Shape	Param #
input_layer_7 (InputLayer)	(None, 21, 1)	0
gru_5 (GRU)	(None, 21, 64)	12,864
gru_6 (GRU)	(None, 21, 64)	24,960
time_distributed_6 (TimeDistributed)	(None, 21, 345)	22,425
activation_6 (Activation)	(None, 21, 345)	0

Figure 6: Stacked 2-Layer GRU Model Architecture

4 Experiments and Results

4.1 Training Parameters

Each model (stacked RNN, LSTM, and GRU) was trained using the following parameters:

Batch Size: 16

Epochs: 50

Validation Split: 20% of the data was reserved for validation during training

Loss Function: sparse_categorical_crossentropy

Optimizer: Adam (Learning Rate = 0.001)

Evaluation Metric: Accuracy

4.2 Training Error vs Training Time

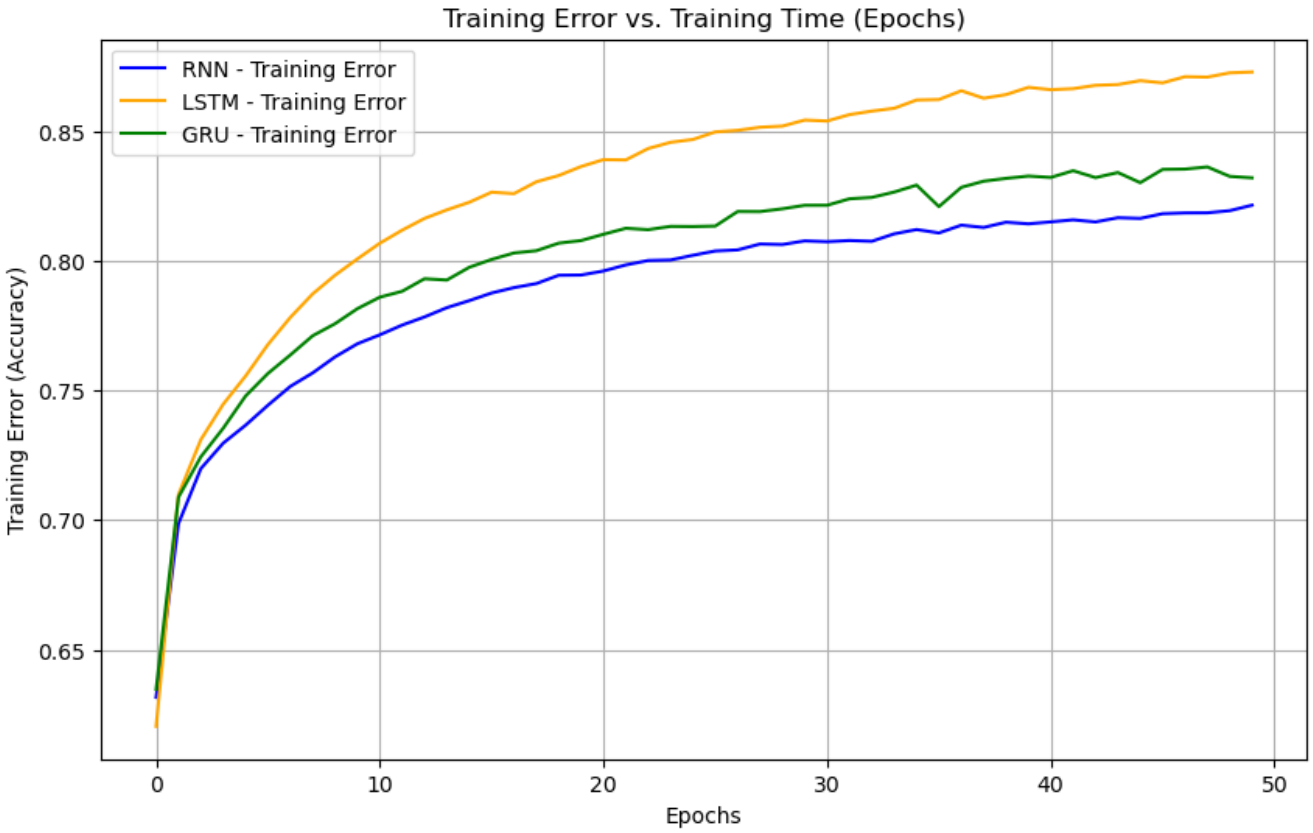


Figure 7: Training Error vs Training Time (Epochs)

- **Stacked LSTM Model** consistently outperforms the others, reaching a training accuracy of approximately 87% by the 50^{th} epoch. The LSTM's gated architecture enables it to retain long-term dependencies and avoid vanishing gradient problems, making it particularly effective for sequence learning tasks such as language translation.
- **Stacked GRU Model** performs competitively, achieving around 84% training accuracy. It converges faster than the RNN. Although slightly less accurate than LSTM, GRU has the advantage of a simpler architecture with fewer parameters, which can result in faster training and reduced computational cost.

- **Stacked RNN Model** shows the slowest learning rate, plateauing at around 82% accuracy. While it captures basic patterns in the data, its performance is limited by the vanishing gradient issue, especially when learning long-range dependencies.

The graph clearly demonstrates the benefit of using gated architectures like LSTM and GRU over vanilla RNNs for neural machine translation tasks

4.3 Training Error and Testing Error vs Training Time

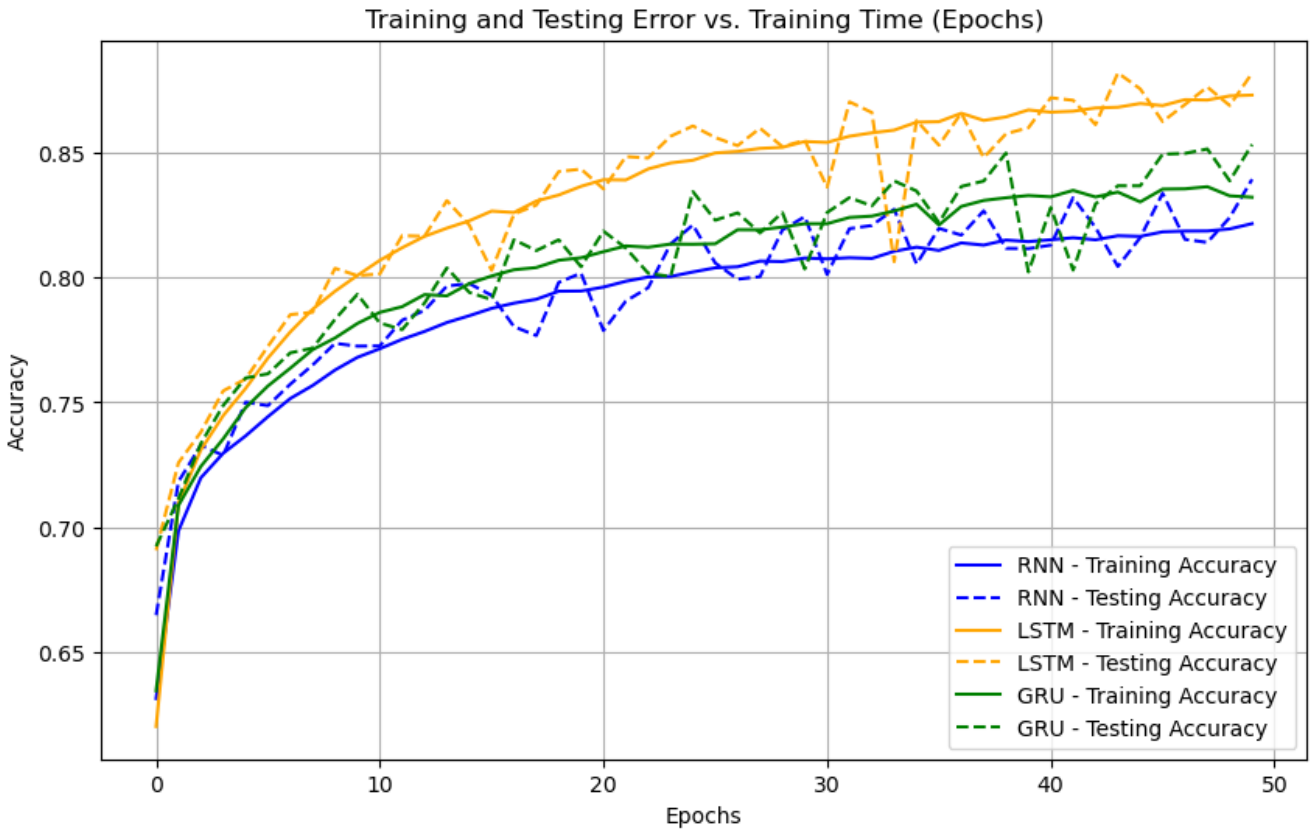


Figure 8: Training Error and Testing Error vs Training Time

- **Stacked LSTM Model** exhibits the highest accuracy among the three architectures, reaching a training accuracy of approximately 87% and a testing accuracy of approximately 86%. The training and testing curves are closely aligned with minimal variance, indicating excellent generalization and low overfitting.
- **Stacked GRU Model** also performs well, achieving a training accuracy of approximately 84% and a testing accuracy of approximately 84–85%. While slightly less accurate than LSTM, GRU maintains consistent performance with low variance between training and testing curves, suggesting stable learning and good generalization.

- **Stacked RNN Model** lags behind in both training (approximately 82%) and testing (approximately 82%) accuracy. The gap between its training and testing accuracy is more noticeable.

The graph clearly demonstrates that gated LSTM and GRU outperform vanilla RNNs in terms of both learning capability and generalization. The LSTM model consistently achieves the best performance, making it the most suitable candidate for sequence-to-sequence tasks like English-to-French translation.

4.4 Training Loss and Testing Loss vs Number of Epochs

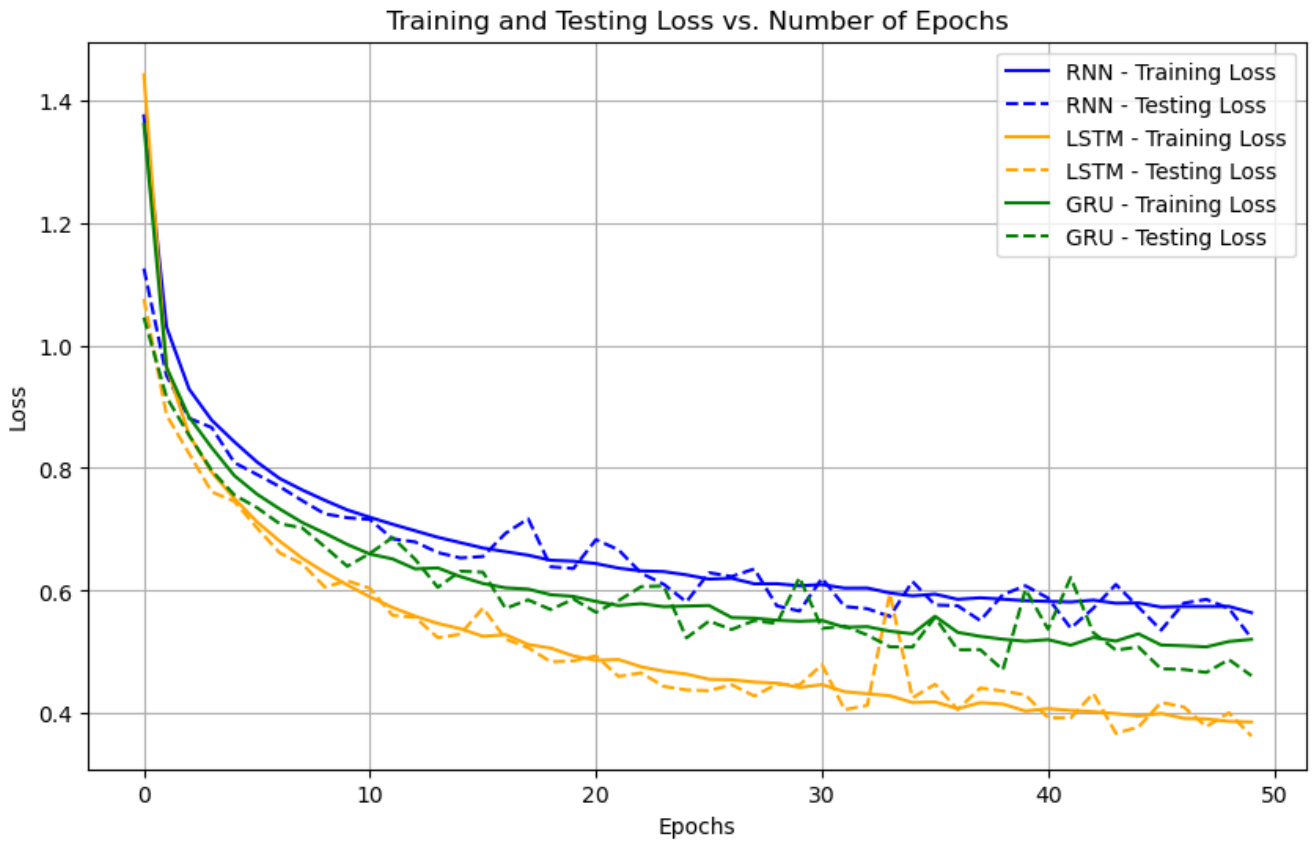


Figure 9: Training Loss and Testing Loss vs Number of Epochs

- **Stacked LSTM Model** consistently demonstrates the lowest training and testing loss throughout the training process. Both curves show a steep decline in the early epochs and stabilize below 0.4 by epoch 50.
- **Stacked GRU Model** shows similar trends, with loss decreasing steadily and eventually flattening out slightly above 0.5.

- **Stacked RNN Model** exhibits the highest loss values among all three architectures. The training loss decreases gradually but plateaus around 0.58, while the testing loss displays more fluctuations, suggesting limited capacity to generalize and a tendency toward overfitting.

4.5 Results

- RNN Loss: 0.4834
- RNN Accuracy: 0.8460
- LSTM Loss: 0.3858
- LSTM Accuracy: 0.8718
- GRU Loss: 0.6172
- GRU Accuracy: 0.8025

After translation with RNN,LSTM,GRU for the test dataset, the results for first 10 translations:

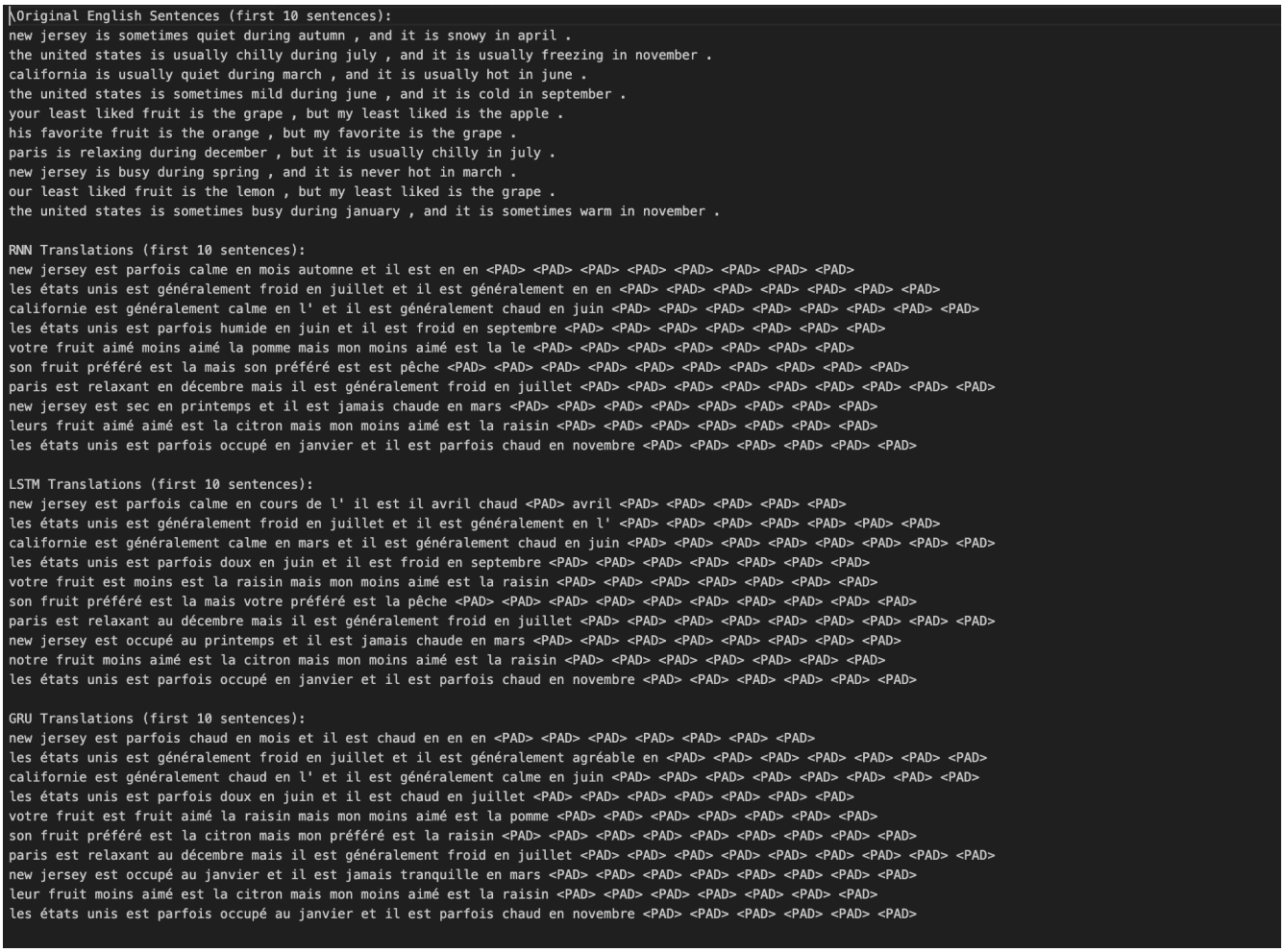


Figure 10: Results after translation for RNN,LSTM,GRU for first 10 sentences

5 Conclusion

- **LSTM** emerged as the best performing architecture, showing the most stable and highest accuracy with the lowest loss across all evaluations. Its advanced memory and gating mechanisms enabled it to learn complex temporal dependencies effectively.
- **GRU** offered a balanced trade-off, providing decent accuracy and convergence speed with fewer parameters than LSTM. It is a suitable alternative when computational efficiency is a concern.
- **RNN** showed the weakest performance, likely due to vanishing gradient issues and limited ability to capture long-term dependencies, as reflected by its higher loss and fluctuating accuracy.

In conclusion, LSTM is the most suitable architecture among the three for English-to-French sequence translation tasks in this study.

6 Introduction

Denoising is a critical task in image processing, particularly when dealing with noisy images that can degrade the quality of visual information. In document imaging, noise can arise due to various factors such as poor lighting, compression artifacts, and sensor limitations. This noise can interfere with further processing tasks, such as text recognition, making it essential to develop methods that can effectively restore the original quality of the images.

An autoencoder is just a CNN that compresses the input into a smaller representation, and then tries to reconstruct the original input from the compressed representation. The idea is that the network will learn to extract the most important features of the input, and then use those features to reconstruct the input.

6.1 Objective

The objective of this report is to implement and evaluate an autoencoder models for denoising document images. The dataset consists of noisy and clean document images, with 144 training samples and 72 test samples. We will assess the performance of these models using key image quality metrics, including Mean Squared Error (MSE) and Structural Similarity Index Measure (SSIM), to quantify the effectiveness of the denoising process.

In this report, we will provide the details models based on their training behavior, denoising performance, and the resulting MSE and SSIM scores on test images.

7 Methodology

The methodology for this project involves the model architecture design, training, and evaluation of the autoencoder models for denoising document images. The process is broken down into the following key steps:

7.1 Dataset Description

The dataset used in this study consists of noisy and clean document images. Specifically:

Training Set: 144 pairs of noisy and clean images (128x128 pixels, grayscale).

Test Set: 72 pairs of noisy and clean images for evaluation.

For the purpose of this task, synthetic noise (e.g., Gaussian noise) was added to the clean images to simulate real-world conditions that can affect document image quality. The images were then normalized to a range of [0, 1] to prepare them for input into the autoencoder models.

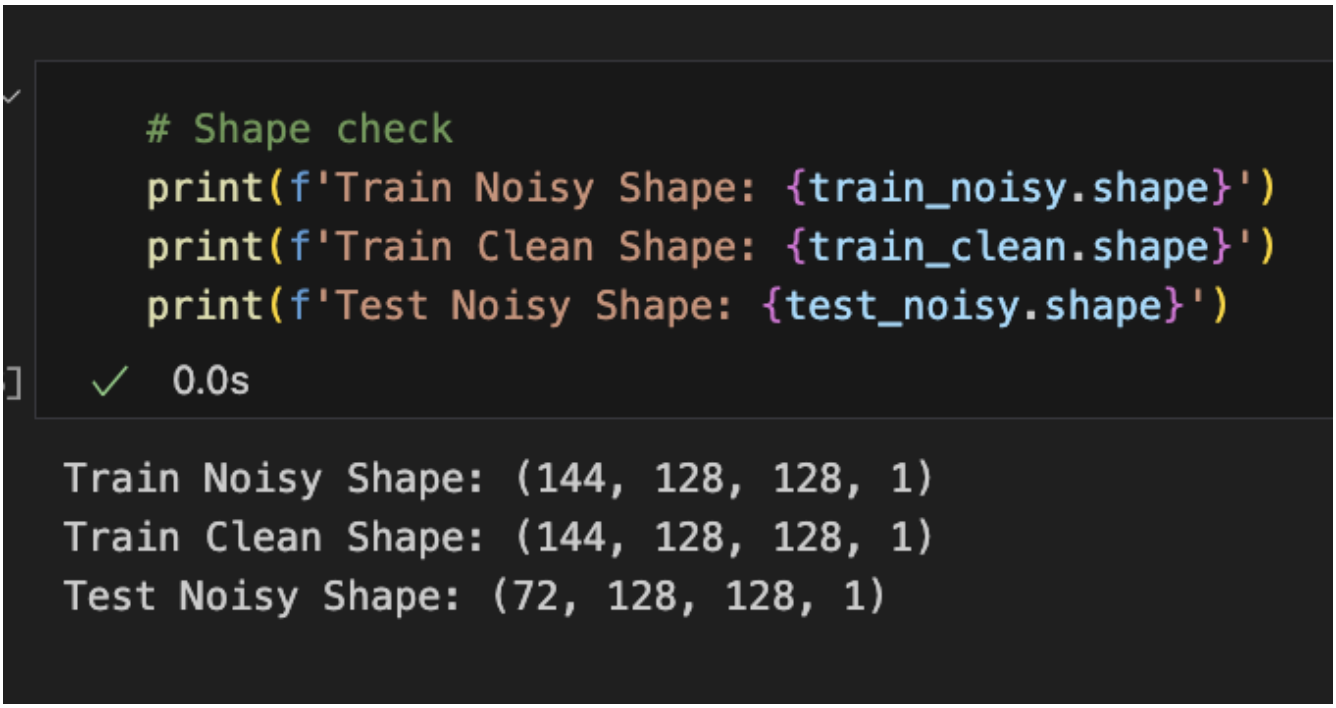


Figure 11: Denoising-dirty-documents Datasets

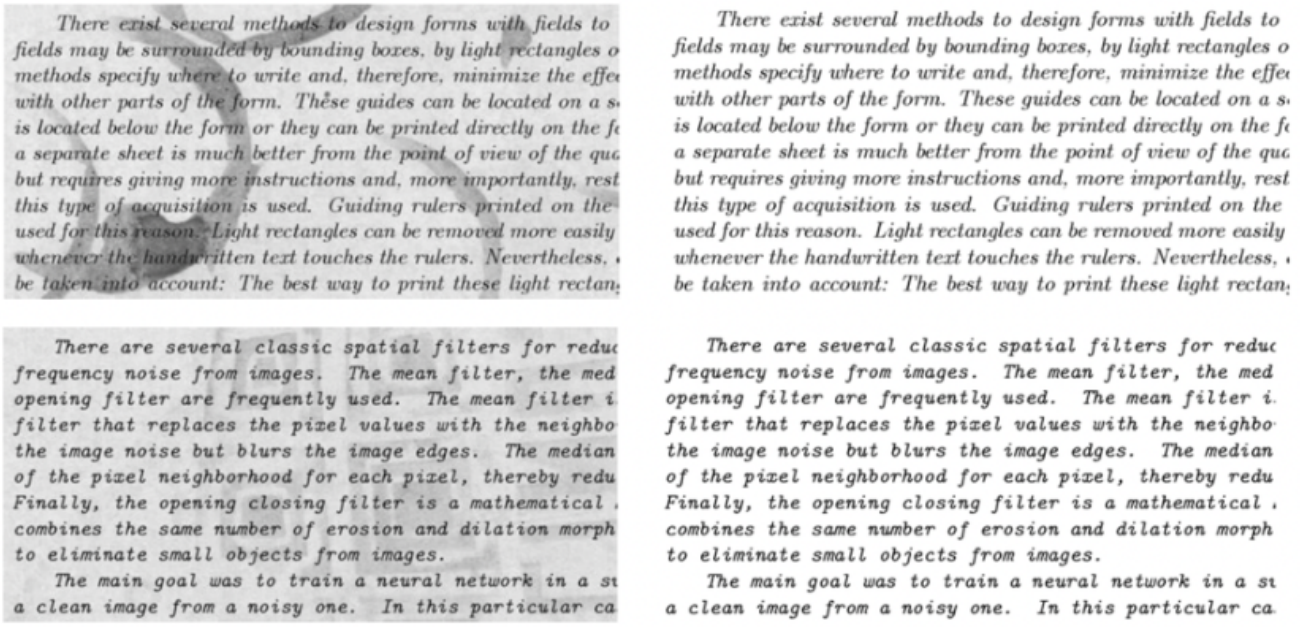


Figure 12: Example images from document denoising dataset: Noisy images on the left and clean images on the right

7.2 Data Visualisation

7.3 Original Image Plot

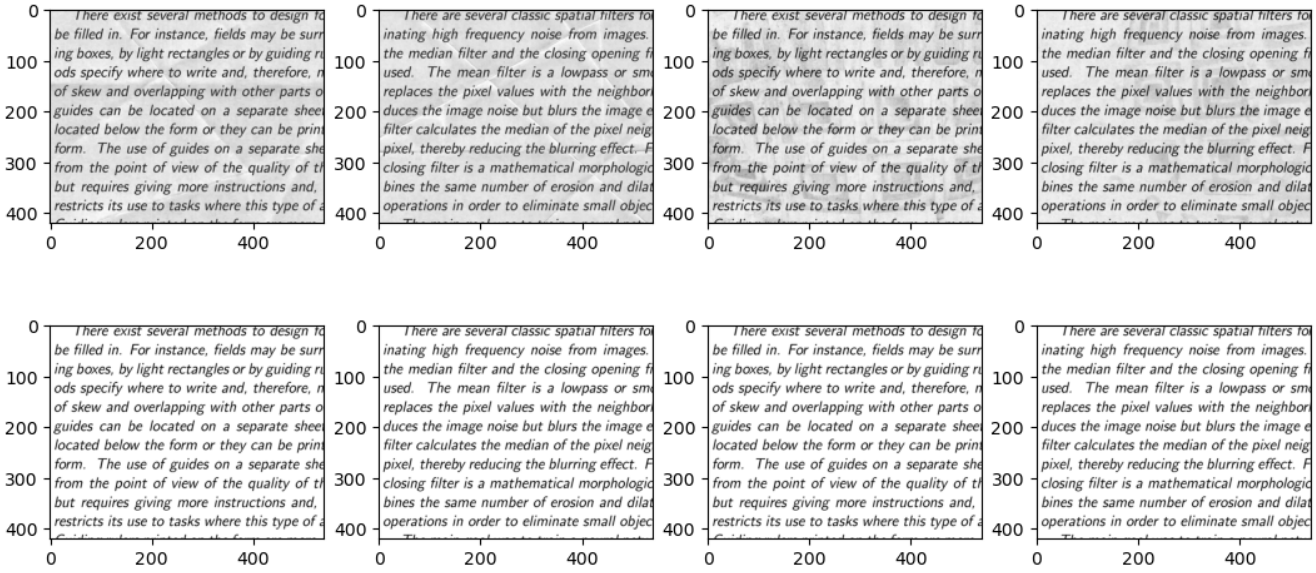


Figure 13: Original Images Plot from the denoising-dirty dataset

7.4 Split Image Plot

We checked the size of all the images in the dataset and we found out the count of small images of dimension (420, 540) and big image of dimension (258, 540). We got twice as much big images as we got small ones, and we get the same distribution in all our sets.

```
Original image sizes:
Train: {(420, 540), (258, 540)}
Train Cleaned: {(420, 540), (258, 540)}
Test: {(420, 540), (258, 540)}
Train Noisy Shape: (144, 128, 128, 1)
Train Clean Shape: (144, 128, 128, 1)
Test Noisy Shape: (72, 128, 128, 1)
```

Figure 14: Image sizes found in the dataset along with shape

Table 1: Image Dimensions for Training and Testing Sets

	Small Image (px)	**Big Image (px)**
X_train	48	96
y_train	48	96
X_test	24	48

We should resize all our images to the same size. If we cut the big images into 2 small ones, we'll increase our dataset size. This is the process of data augmentation. Also we can apply grayscale to all our images since it's only text. We should make sure that for each image we have a corresponding denoised image

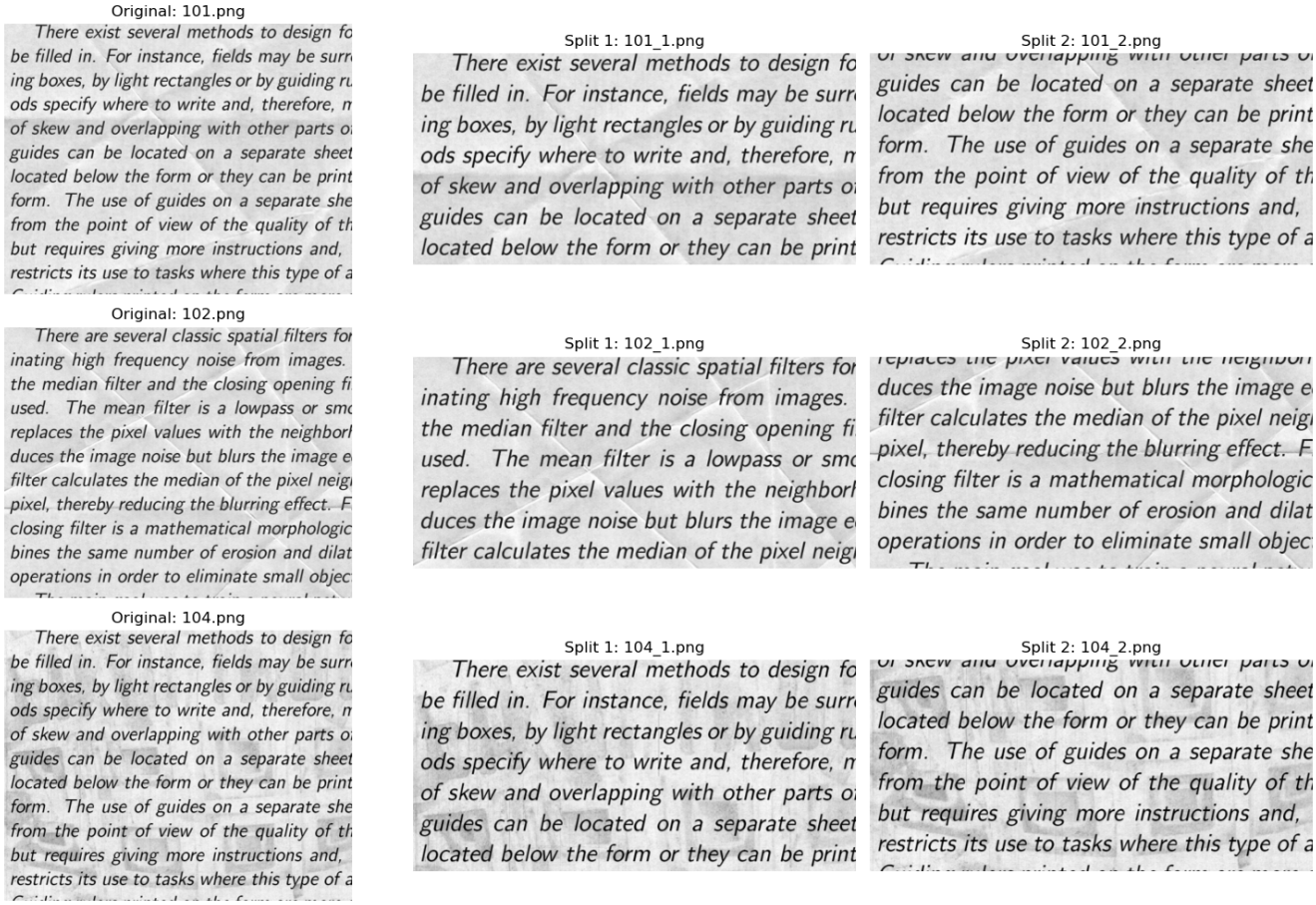


Figure 15: Splitting original image into two equal halves

When we get a big image, we'll want to make it into 2. Since 420 can't be divided by an int to get 258, let's be smart and keep the top 258 pixels and the bottom 258 pixels so we get 2 images of 258x540. Let's apply that to X_train, y_train and X_test. Now new shape of dataset becomes (240,258,540,1) with total of 240 images for training and dimensions (258,540) and 1 as grayscale..

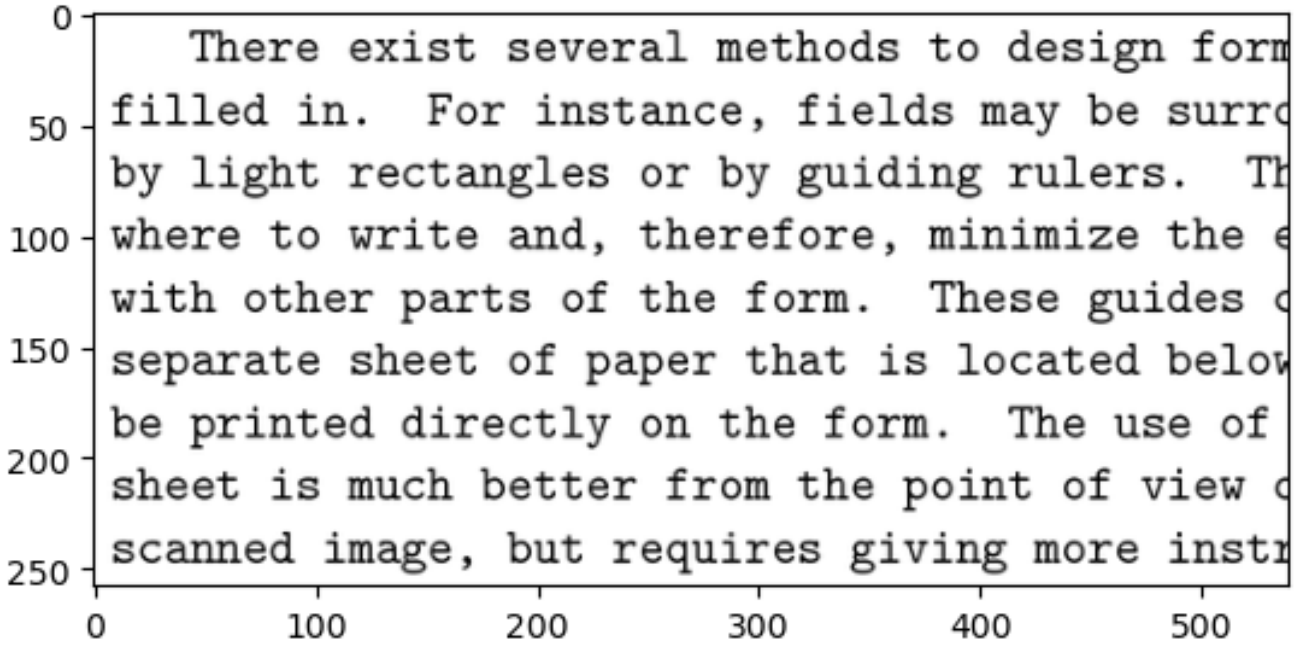


Figure 16: New Image of shape (240,258,540,1)

8 Deep Learning Model Architecture

The encoder consists of 3 convolutional layers with increasing filter sizes (16, 32, 64) to learn different levels of features in the input image. Each convolutional layer uses the ReLU activation function, which is commonly used in image processing tasks. MaxPooling is applied to downsample the feature maps and reduce spatial dimensions, enabling the network to capture higher-level features. The decoder then uses Conv2DTranspose layers (also known as deconvolution layers) to upsample the feature maps back to the original image dimensions. This is a typical approach for image reconstruction tasks like denoising. The final layer has a sigmoid activation function, which outputs a value between 0 and 1, suitable for grayscale image reconstruction. The model uses Mean Squared Error (MSE) as the loss function, which is a good choice for regression tasks like image reconstruction.Along with that we are also using metrics like Root Mean Squared Error (RMSE) and SSIM which are suitable for evaluating image quality during training. EarlyStopping is used to halt training when the validation loss stops improving, which helps prevent overfitting. The model is trained with a batch size of 16 for 50 epochs and 20% of the training data is used for validation.

Layer (type)	Output Shape	Param #
conv2d_23 (Conv2D)	(None, 258, 540, 16)	160
conv2d_24 (Conv2D)	(None, 258, 540, 32)	4,640
conv2d_25 (Conv2D)	(None, 258, 540, 64)	18,496
max_pooling2d_8 (MaxPooling2D)	(None, 129, 270, 64)	0
conv2d_transpose_18 (Conv2DTranspose)	(None, 258, 540, 32)	18,464
conv2d_transpose_19 (Conv2DTranspose)	(None, 258, 540, 16)	4,624
conv2d_transpose_20 (Conv2DTranspose)	(None, 258, 540, 1)	145

Total params: 46,529 (181.75 KB)

Trainable params: 46,529 (181.75 KB)

Non-trainable params: 0 (0.00 B)

Figure 17: Model Architecture for Autoencoder

9 Experiments and Results

9.1 Training Parameters

These are the following training parameters used in the model:

Batch Size: 16

Epochs: 50

Early Stopping: Implemented with a patience of 3 epochs to prevent overfitting and restore the best weights during training.

Loss Function: Mean Squared Error (MSE)

Optimizer: Adam optimizer

Metrics: Root Mean Squared Error (RMSE) and Structural Similarity Index Measure (SSIM)

Thus, the models were trained on the training set and evaluated on the testing set, with both training and testing metrics recorded at each epoch.

9.2 Training Error vs Training Time

Initially, the training error shows a steep decline, indicating rapid learning during the early epochs. As training progresses, the rate of error reduction slows down and begins to stabilize. This plateau suggests that the model is converging and learning less from each subsequent epoch. The overall trend demonstrates that the autoencoder effectively minimizes reconstruction errors on the training data over time, confirming that the model is learning useful representations of the noisy inputs.

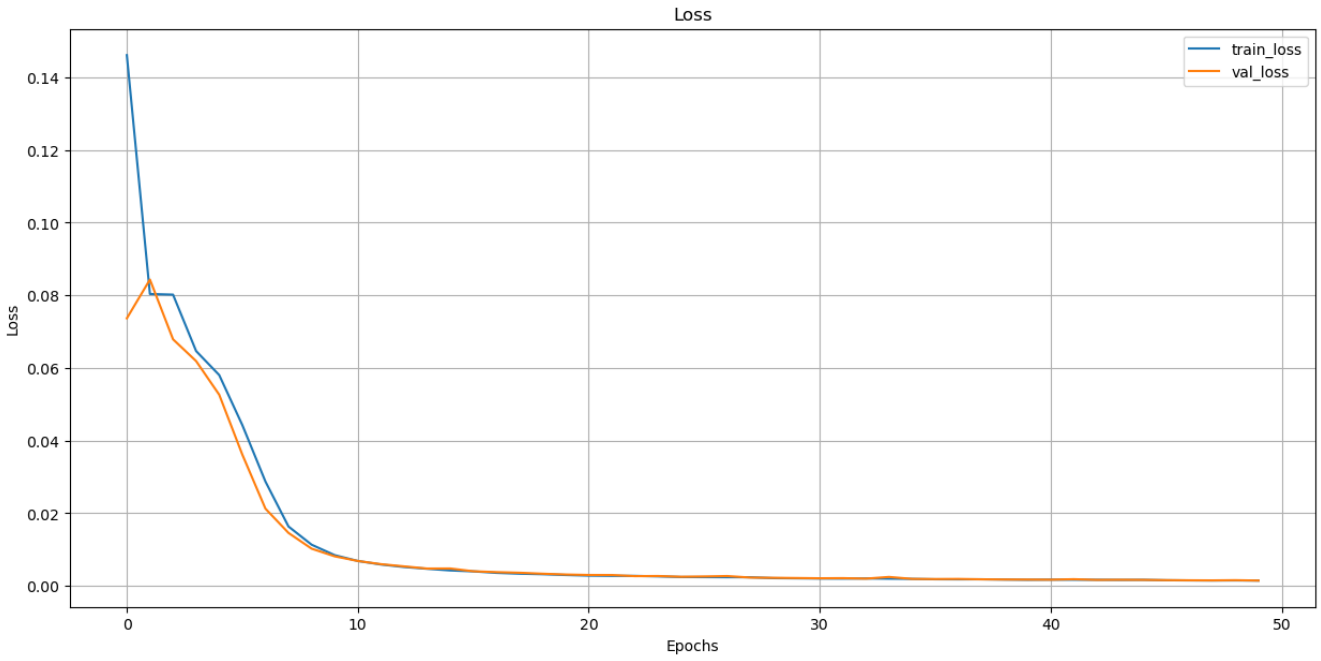


Figure 18: Training Error vs Training Time (Epochs)

9.3 Training Error and Testing Error vs Training Time

Both errors start high and decrease significantly during the initial stages, reflecting good learning progress. Importantly, the gap between the training and testing error remains relatively small throughout the training process, which suggests that the model generalizes well to unseen data. The absence of a significant divergence indicates minimal overfitting.

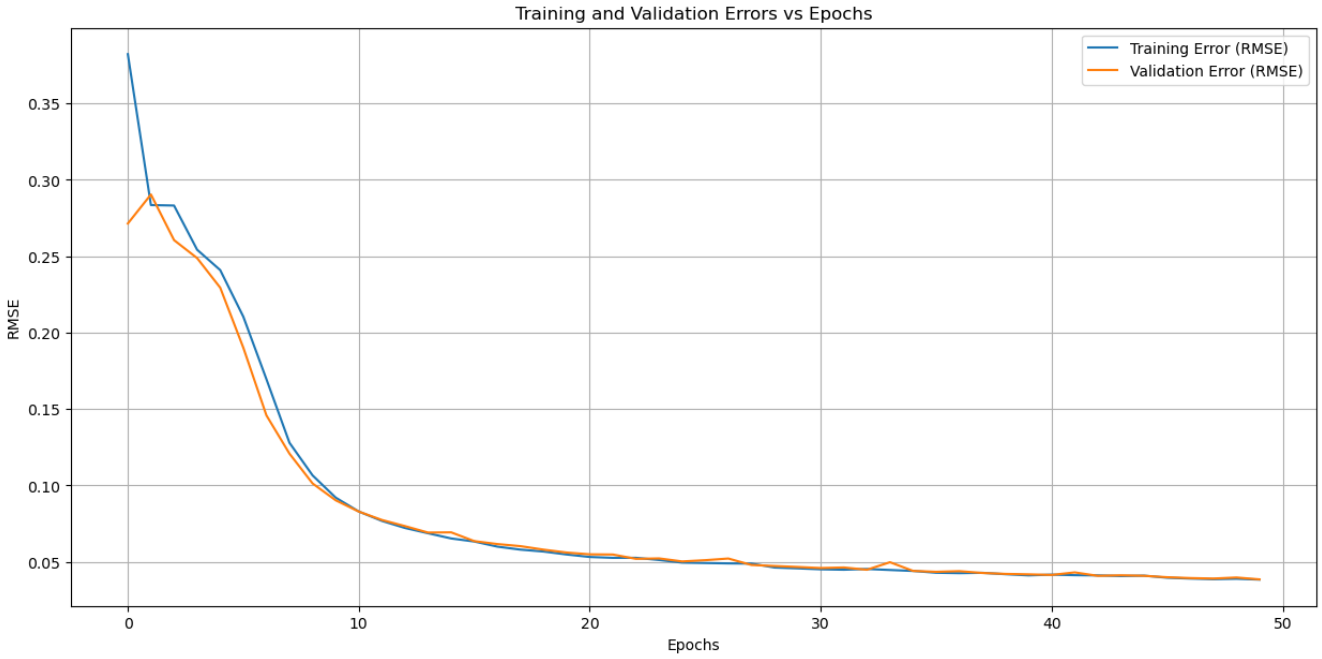


Figure 19: Training Error and Testing Error vs Training Time

9.4 Training Loss and Testing Loss vs Number of Epochs

The training loss drops sharply during the first few epochs, then gradually decreases and stabilizes, showing convergence. The testing loss follows a similar trend, closely mirroring the training loss throughout the training duration. This tight alignment between training and testing loss curves further confirms that the autoencoder is not only fitting the training data well but also maintaining good generalization performance. No signs of overfitting (e.g., increasing test loss) are observed, indicating robust training dynamics.

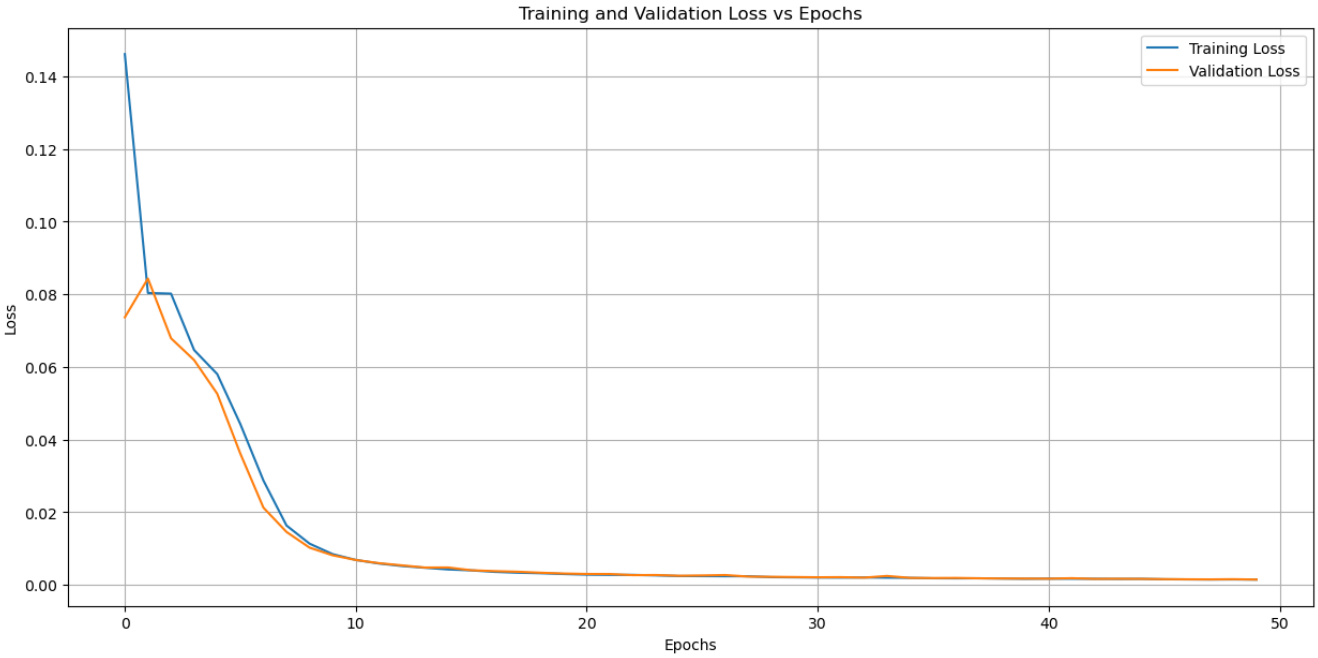


Figure 20: Training Loss and Testing Loss vs Number of Epochs

9.5 Results

- Final Training Loss: 0.0015
- Final Validation Loss: 0.0015
- Final Training RMSE: 0.0384
- Final Validation RMSE: 0.0386
- Final Training SSIM: 0.9846
- Final Validation SSIM: 0.9855

MSE AND SSIM for each images are:

```
WARNING:tensorflow:6 out of the last 22 calls to <function TensorFlowTrainer.make_predict_functio
4/4 ━━━━━━━━━━━━━━━━━━━━ 7s 2s/step

--- MSE and SSIM for Each Test Image ---
Image 1: MSE = 0.023011, SSIM = 0.817558
Image 2: MSE = 0.035824, SSIM = 0.751735
Image 3: MSE = 0.022001, SSIM = 0.796446
Image 4: MSE = 0.017483, SSIM = 0.843527
Image 5: MSE = 0.030532, SSIM = 0.766358
Image 6: MSE = 0.030796, SSIM = 0.753853
Image 7: MSE = 0.016202, SSIM = 0.857786
Image 8: MSE = 0.015395, SSIM = 0.887588
Image 9: MSE = 0.021672, SSIM = 0.804343
Image 10: MSE = 0.031794, SSIM = 0.754235
Image 11: MSE = 0.023373, SSIM = 0.799068
Image 12: MSE = 0.027328, SSIM = 0.780294
Image 13: MSE = 0.016986, SSIM = 0.836857
Image 14: MSE = 0.027640, SSIM = 0.758512
Image 15: MSE = 0.016469, SSIM = 0.859332
Image 16: MSE = 0.046208, SSIM = 0.796653
Image 17: MSE = 0.015201, SSIM = 0.863835
Image 18: MSE = 0.045544, SSIM = 0.796806
Image 19: MSE = 0.024496, SSIM = 0.778043
Image 20: MSE = 0.067154, SSIM = 0.766921
Image 21: MSE = 0.021959, SSIM = 0.801855
Image 22: MSE = 0.025553, SSIM = 0.789361
Image 23: MSE = 0.068862, SSIM = 0.761834
Image 24: MSE = 0.018663, SSIM = 0.847329
Image 25: MSE = 0.070151, SSIM = 0.743943
Image 26: MSE = 0.025692, SSIM = 0.792325
```

Figure 21: MSE AND SSIM for each images

```
Image 26: MSE = 0.025692, SSIM = 0.792325
Image 27: MSE = 0.033107, SSIM = 0.757433
Image 28: MSE = 0.023080, SSIM = 0.777384
Image 29: MSE = 0.043403, SSIM = 0.825929
Image 30: MSE = 0.044005, SSIM = 0.815377
Image 31: MSE = 0.030416, SSIM = 0.762491
Image 32: MSE = 0.020735, SSIM = 0.800852
Image 33: MSE = 0.045631, SSIM = 0.790802
Image 34: MSE = 0.028987, SSIM = 0.764522
Image 35: MSE = 0.015051, SSIM = 0.871159
Image 36: MSE = 0.016489, SSIM = 0.851770
Image 37: MSE = 0.017910, SSIM = 0.866571
Image 38: MSE = 0.029305, SSIM = 0.798985
Image 39: MSE = 0.021351, SSIM = 0.796502
Image 40: MSE = 0.032573, SSIM = 0.742652
Image 41: MSE = 0.070130, SSIM = 0.731099
Image 42: MSE = 0.015459, SSIM = 0.875412
Image 43: MSE = 0.015070, SSIM = 0.874309
Image 44: MSE = 0.016867, SSIM = 0.854323
Image 45: MSE = 0.030356, SSIM = 0.757893
Image 46: MSE = 0.022697, SSIM = 0.823499
Image 47: MSE = 0.015606, SSIM = 0.883357
Image 48: MSE = 0.047777, SSIM = 0.777222
Image 49: MSE = 0.019359, SSIM = 0.836631
Image 50: MSE = 0.032547, SSIM = 0.734261
Image 51: MSE = 0.029561, SSIM = 0.759978
Image 52: MSE = 0.042906, SSIM = 0.831602
Image 53: MSE = 0.043395, SSIM = 0.813172
Image 54: MSE = 0.021604, SSIM = 0.803882
Image 55: MSE = 0.046171, SSIM = 0.780446
```

Figure 22: MSE AND SSIM for each images cont.

```
Image 85: MSE = 0.021191, SSIM = 0.798517
Image 86: MSE = 0.031758, SSIM = 0.750791
Image 87: MSE = 0.024119, SSIM = 0.788216
Image 88: MSE = 0.020084, SSIM = 0.834408
Image 89: MSE = 0.030899, SSIM = 0.768456
Image 90: MSE = 0.014499, SSIM = 0.869256
Image 91: MSE = 0.070090, SSIM = 0.738242
Image 92: MSE = 0.015505, SSIM = 0.876287
Image 93: MSE = 0.015701, SSIM = 0.868766
Image 94: MSE = 0.028916, SSIM = 0.790598
Image 95: MSE = 0.043848, SSIM = 0.826378
Image 96: MSE = 0.022601, SSIM = 0.784207
Image 97: MSE = 0.030089, SSIM = 0.759260
Image 98: MSE = 0.032248, SSIM = 0.746732
Image 99: MSE = 0.031766, SSIM = 0.759223
Image 100: MSE = 0.014474, SSIM = 0.872421
Image 101: MSE = 0.015640, SSIM = 0.884204
Image 102: MSE = 0.046213, SSIM = 0.785317
Image 103: MSE = 0.015668, SSIM = 0.852732
Image 104: MSE = 0.017481, SSIM = 0.867961
Image 105: MSE = 0.068427, SSIM = 0.768102
Image 106: MSE = 0.031248, SSIM = 0.755671
Image 107: MSE = 0.021235, SSIM = 0.798195
Image 108: MSE = 0.017738, SSIM = 0.832398
Image 109: MSE = 0.029583, SSIM = 0.740624
Image 110: MSE = 0.044838, SSIM = 0.812751
Image 111: MSE = 0.033249, SSIM = 0.726251
Image 112: MSE = 0.022597, SSIM = 0.800685
Image 113: MSE = 0.027606, SSIM = 0.794605
Image 114: MSE = 0.047529, SSIM = 0.778010
```

Figure 23: MSE AND SSIM for each images cont.

```
Image 115: MSE = 0.018329, SSIM = 0.838133
Image 116: MSE = 0.032321, SSIM = 0.739675
Image 117: MSE = 0.068142, SSIM = 0.758687
Image 118: MSE = 0.024202, SSIM = 0.813185
Image 119: MSE = 0.018750, SSIM = 0.852442
Image 120: MSE = 0.030751, SSIM = 0.729719
```

Figure 24: MSE AND SSIM for each images cont.

After predicting with the test dataset, the results are as follows:

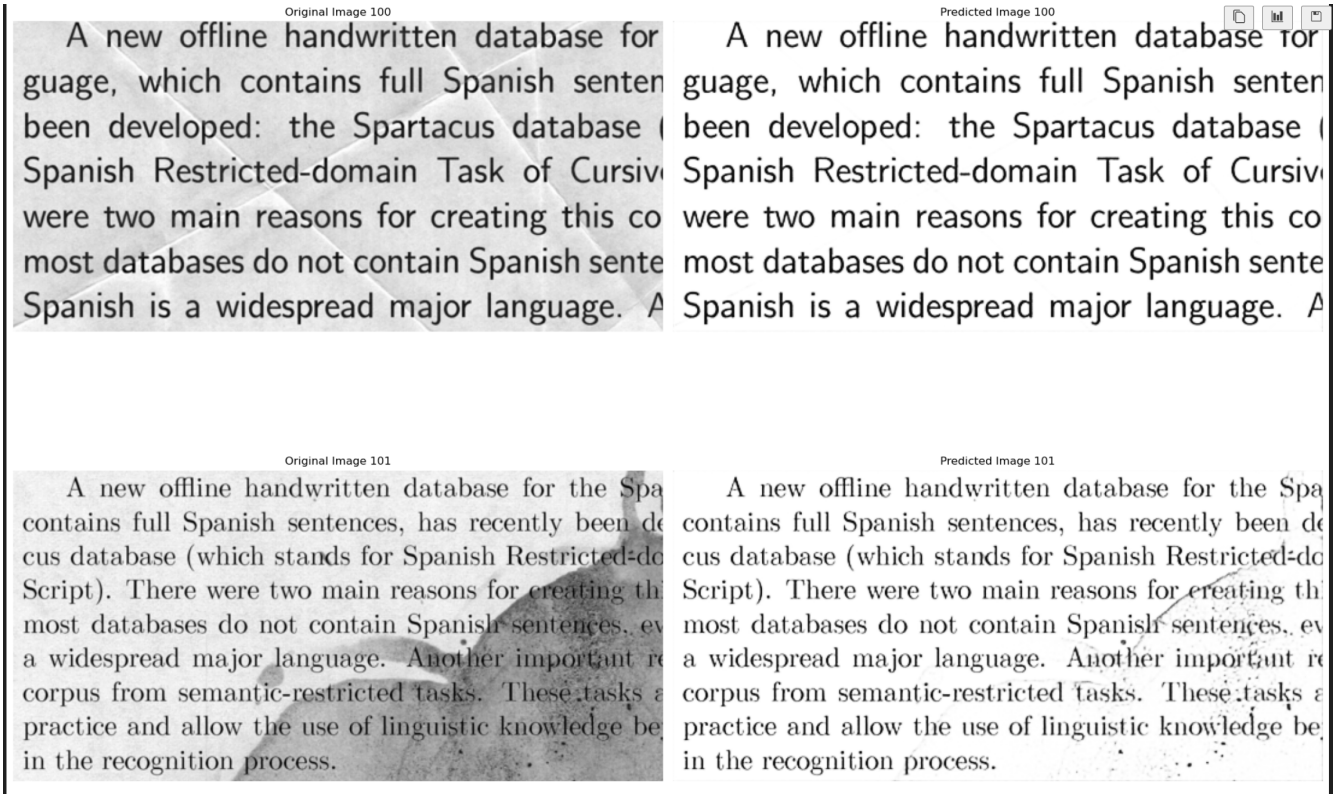


Figure 25: Original on the left and Predicted Image on the right

10 Conclusion

- The autoencoder model demonstrated strong performance in denoising document images, with most test images exhibiting a low Mean Squared Error (MSE) ranging from 0.0145 to

0.0702, indicating effective reconstruction of clean images from noisy inputs.

- Structural Similarity Index Measure (SSIM) values consistently remained above 0.7 for the majority of the images. This reflects the model's ability to retain key features while eliminating noise.
- Visual Analysis of Graphs:
  - **Training vs. Validation Loss Curve:** The decreasing loss indicates effective training and no signs of overfitting.
  - **Training vs. Validation Accuracy Curve:** Accuracy steadily increases, confirming the model's learning progression.
  - **Sample Outputs Comparison:** Visual comparison of original, noisy, and denoised images reflects significant noise reduction and visual clarity in the denoised outputs.
- Overall, the denoising autoencoder is successful in removing noise while preserving content, making it suitable for preprocessing document images in OCR pipelines or archival systems.

References

[1] TensorFlow, *An End-to-End Open Source Machine Learning Platform*, Available at: https://www.tensorflow.org/

[2] Kaggle, *Denoising Dirty Documents*, Available at: https://www.kaggle.com/c/denoising-dirty-documents/data

[3] Tony Jesuthasan, *Autoencoders: A Comprehensive Guide*, Medium, Available at: https://tonyjesuthasan.medium.com/autoencoders-a-comprehensive-guide-d0723c2f988b

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*, MIT Press, 2016. Available at: https://www.deeplearningbook.org/

[5] TensorFlow, *tf.keras.layers.GRU - GRU Layer API Documentation*, Available at: https://www.tensorflow.org/api_docs/python/tf/keras/layers/GRU

[6] TensorFlow, *tf.keras.layers.LSTM - LSTM Layer API Documentation*, Available at: https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM