

#### **The Idea**

#### The idea for this project came from the personal need to convert my handwritten notes into a digital format, making them easier to read and carry around. Handwritten notes, while valuable for their immediacy and convenience, often suffer from issues such as legibility and physical storage constraints. By using deep learning techniques, we aim to solve this problem by developing a system that can accurately recognize and convert handwritten text into digital text.

#### The core of this project involves designing an effective model architecture capable of accurately recognizing and converting handwritten text into machine-encoded text. This task, known as Optical Character Recognition (OCR), poses unique challenges due to the inherent variability in individual handwriting styles. To address these challenges, we used advanced neural networks and thorough training methods to create our model.

#### This report details the development and training of our model, outlining the key components and methodologies employed. In the subsequent sections, we will discuss the specific design choices, the dataset preparation, and the rigorous training process that underpins our OCR system. In the final section we will also show the results and discuss the impact on the performance by using different hyperparameters and model architectures.

#### By the end of this report, we aim to provide a clear understanding of the intricate processes involved in creating a robust handwriting recognition system and showcase its practical utility in converting handwritten notes into a digital format.

#### **Goal**

#### The goal of the Handwriting Recognizer project is to develop a robust and accurate software application that converts handwritten text into digital, machine-readable typed text. This application aims to support English language and Latin character sets, providing a reliable solution for digitizing handwritten notes, documents, and other handwritten materials.

#### **Expected Outcomes**

#### - Enhanced efficiency in converting handwritten notes to digital text.

#### - Improved accessibility for diverse handwriting styles.

#### Dataset Utilization

#### To implement the handwriting recognition model, we utilized the dataset from the IAM Handwriting Database. Given our supervised approach, the dataset includes images of handwriting along with their corresponding labels. This comprehensive dataset provided a rich collection of labeled handwritten text samples, which were essential for training and evaluating the performance of our recognition system.

#### **Preprocessing**

The following libraries were utilized for data preprocessing, each serving specific purposes:

- random: Used for generating pseudo-random numbers, which is essential for operations such as shuffling the data to ensure randomness during training and testing.

- cv2 (OpenCV): A powerful library for image processing tasks. It was used for reading, writing, and manipulating image data, including resizing, normalization, and other image transformations.

- numpy (NumPy): A fundamental library for numerical computations in Python. It was employed for handling arrays and performing operations on the image data, such as pixel value normalization and array manipulation.

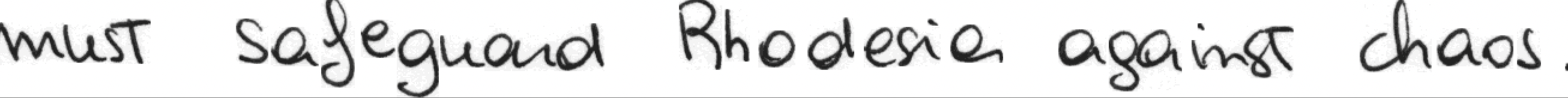
- torchvision.transforms (Torchvision Transforms): Part of the PyTorch ecosystem, this module provides common image transformations for preprocessing. It was used to apply transformations like converting images to tensors, normalizing pixel values, and augmenting images to improve model robustness.

- torch (PyTorch): An open-source machine learning library used for deep learning applications. It provides support for tensor computations and automatic differentiation, essential for machine learning.

The \_\_init\_\_ method in the Preprocessor class is a constructor method, which is called when an instance of the class is created. This method initializes the attributes of the Preprocessor object with the provided arguments or default values.

The \_\_call\_\_ method in the Preprocessor class preprocesses an input image and its corresponding label. It resizes the image, converts the label into an indexed format based on a given vocabulary, pads the label to a specified maximum length, and optionally applies data augmentation techniques to the image. It then returns the preprocessed image and label, ready for use in training or evaluating a handwriting recognition model.

Difference in data before and after augmentation part:





Preprocessing of the input taken from user interface

The single\_image\_preprocessing method preprocesses an input image received from a user interface to ensure it is suitable for use in a machine learning model. The method performs the following steps:

- Resizes the image to fit within specified target dimensions while maintaining the aspect ratio.

- Pads the image to ensure it matches the target size, using a padding color of white (255).

- Converts the image from a NumPy array to a PyTorch tensor.

- Normalizes the image pixel values to the range [0, 1].

- This preprocessing ensures that the input image is in a consistent format and size, ready for accurate and reliable processing by the machine learning model.

**Data loader**

The HandwritingDataset class is a custom dataset class for handling handwritten text images and their corresponding labels. It inherits from PyTorch's Dataset class and is designed to facilitate the preprocessing and augmentation of handwriting data for use in machine learning models.

The \_\_init\_\_ method initializes the dataset with the given data, vocabulary, maximum label length, optional transformations, and augmentation settings.

The getitem method retrieves a data sample (image and label) at the specified index, applies preprocessing and optional transformations, and returns the processed image and label.

#### **Model architecture**

Model Design:

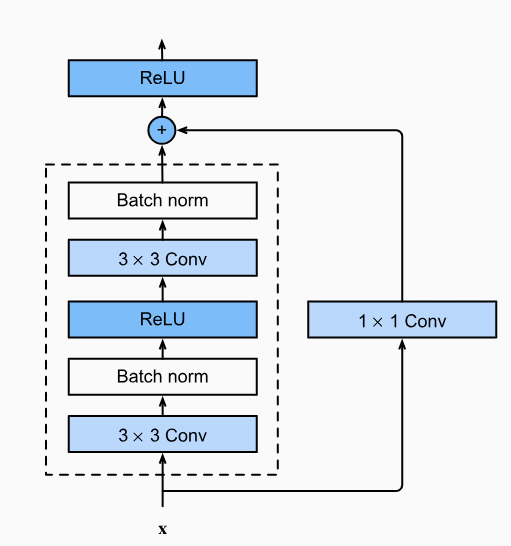
The architecture of our model is designed to effectively recognize and transcribe handwritten text. We built a Convolutional Recurrent Neural Network (CRNN) that combines convolutional layers for feature extraction and recurrent layers for sequence modeling. The model is implemented using PyTorch, and the key components are detailed below.

Residual Block:

The model employs a series of Residual Blocks, inspired by the ResNet architecture. These blocks help in training deep networks by allowing gradients to flow through the network more effectively. Each Residual Block consists of the following components:

1. Convolutional Layers: Two convolutional layers with batch normalization and ReLU activations perform feature extraction. The first convolutional layer may reduce the spatial dimensions of the input, while the second maintains them.

2. Skip Connections: Skip connections (or identity shortcuts) bypass the convolutional layers and directly add the input to the output. If the dimensions of the input and output do not match, a skip connection with an additional convolutional layer is used to match them. This ensures that the network can learn identity functions easily, facilitating the learning of residuals.



from the book: dive into deep learning Fig. 8.6.3 ResNet block with 1×1 convolution, which transforms the input into the desired shape for the addition operation.

3. Dropout: Applied to the output of each block to prevent overfitting, 0.2 by default.

Overall Model:

The overall model integrates multiple Residual Blocks followed by bidirectional Long Short-Term Memory (BiLSTM) layers to capture

the sequential dependencies in handwritten text. The architecture is as follows:

1. Residual Blocks: Nine Residual Blocks progressively increase the depth and complexity of the feature maps.

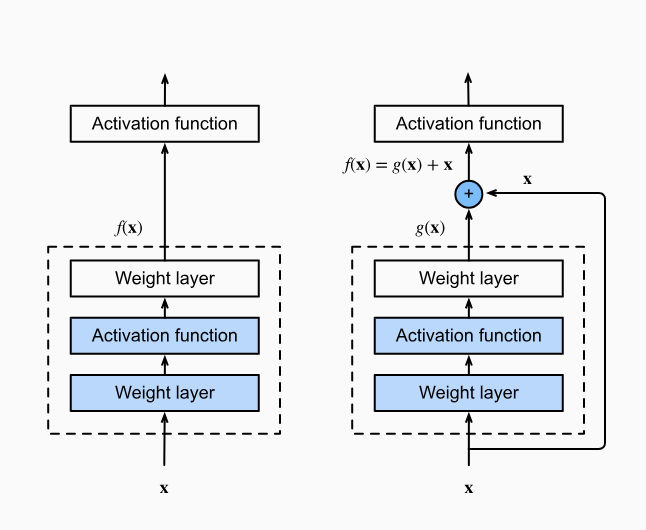


Fig. 8.6.2 from dive into deep learning,block (left), the portion within the dotted-line box must directly learn the mapping 𝑓(𝑥). In a residual block

(right), the portion within the dotted-line box needs tolearn the residual mapping 𝑔(𝑥)=𝑓(𝑥)−𝑥, making the identity mapping 𝑓(𝑥)=𝑥 easier to learn.

2. Bidirectional LSTMs: Two BiLSTM layers process the sequential features extracted by the convolutional layers. These layers capture the temporal dependencies in the sequence data.

3. Fully Connected Layer: A final fully connected layer transforms the LSTM outputs into unnormalized character probabilities.

4. Log-Softmax: Applied to the final output to produce log probabilities for each character class.

#### **Training Process**

The training process for the handwriting recognition model involves several key steps: data preparation, model initialization, training, validation, and checkpointing. Here's a detailed breakdown of each step:

1. Data Preparation:

Load Data: The dataset is loaded from a specified directory containing image paths and corresponding labels. The labels are translated from characters to integers representing their corresponding index in the vocabulary and then padded to the maximum length using the blank token (length of vocabulary + 1) to compute ctc loss.

2. Vocabulary and Maximum Length Calculation: A vocabulary of unique characters and the maximum length of labels are calculated to define the model's output dimensions.

3. Data Splitting: The dataset is split into training,

validation, and test sets using train\_test\_split from sklearn. Optionally, k-fold cross-validation can be used by setting the cross\_validation variable to True.

Model Initialization:

1. Model Selection: A model is chosen from a predefined set of models. If a previously trained model exists, it is loaded and continued training from there.

2. Loss Function: The Connectionist Temporal Classification (CTC) loss is used for sequence prediction tasks. At each time step the model outputs a probability distribution over all possible labels + the blank symbol, the CTC loss represents the probability of getting an output sequence that collapses into the ground truth sequence. Below is an example of how CTC decoder ‘collapses’ repeated outputs and the blank symbol, a nice explanation of CTC loss can be found in reference [9].



model prediction: \_\_\_A\_ABB\_B\_AA\_CC\_ Collapsed: AABBAC  
Ground truth: ABCC

Optimizer: AdamW optimizer is used, with a learning rate scheduler to adjust the learning rate based on the validation performance, a slightly different learning rate or learning rate scheduling algorithm will infect the final

training result significantly, we will discuss it in the next section.

Training Loop:

1. Epoch Loop: The model is trained for a specified number of epochs. For each epoch:

Training Phase: The model is set to training mode. For each batch, inputs and targets are processed, and the loss is computed and backpropagated.

Logging: Metrics from the library torchmetrics such as Character Error Rate (CER) and Word Error Rate (WER) are computed and logged together with the loss using TensorBoard.

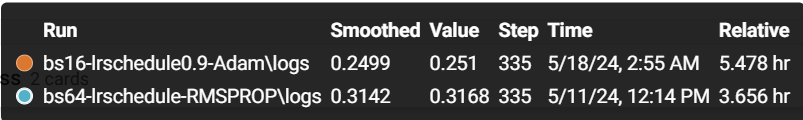
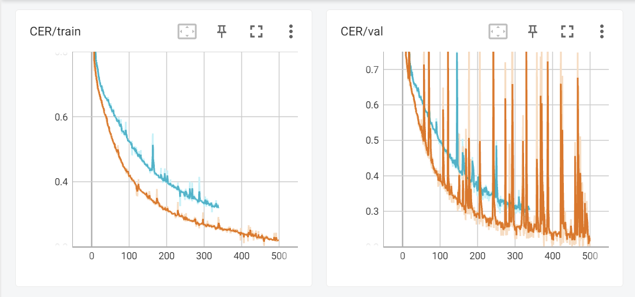
2. Validation Phase: The model is set to evaluation mode. Validation data is processed similarly to training data, but without backpropagation. Metrics are computed and logged.

3. Learning rate scheduler: When the model is not improving on validation metrics for several epochs, the learning rate will decrease by a factor of gamma.

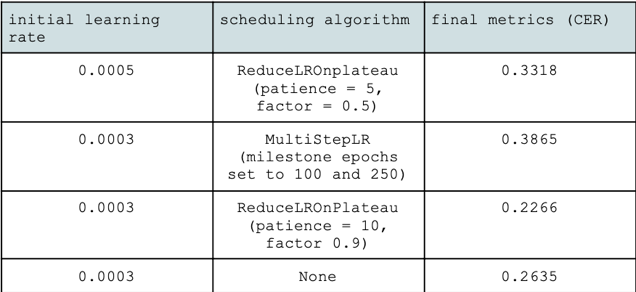
4. Checkpoints and Early Stopping: Checkpoints are saved based on validation metrics to prevent overfitting, it steps after every epoch and saves the model that has the best validation cer so far. Early stopping halts training if no improvement is seen for a defined number of epochs, in our case the patience is set to 20.

#### **Results visualization**

Optimizer: RMSProp Vs Adam, RMSProp is an optimizer that is mentioned in different ResNet papers to perform well, however, in our specific case, Adam outperformed RMSProp.

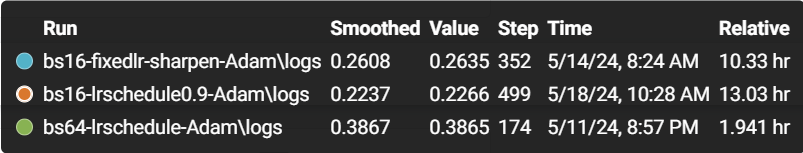
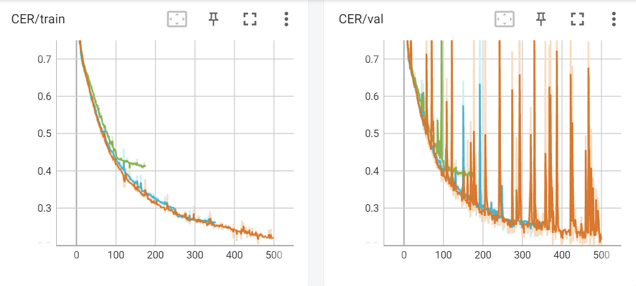


Learning rate:

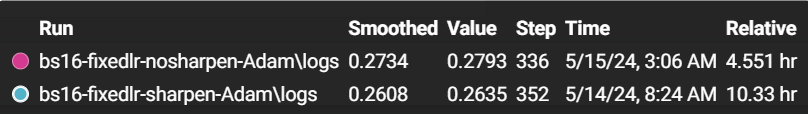
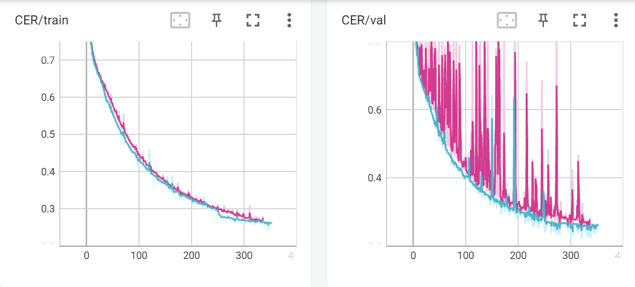


We noticed that even when using the same scheduling algorithm, changing its parameter such as patience or multiplicative factor changes quite significantly the final model, we found that a larger factor works empirically better since decreasing the learning rate too much will make the model not able to learning anything in a reasonable number of timesteps. Also 0.003 was the best learning rate to start with, even when it is fixed it

outperforms almost all other models that are not listed in the above table.



Augmentation: we tried to train the model with and without random sharpening, one of the augmentations that we apply to the model, to see the impact of data augmentation on the results:



As we can observe from the plot, there is no big difference during the training process, however in the validation the one without the random sharpening seems to overfit the data.

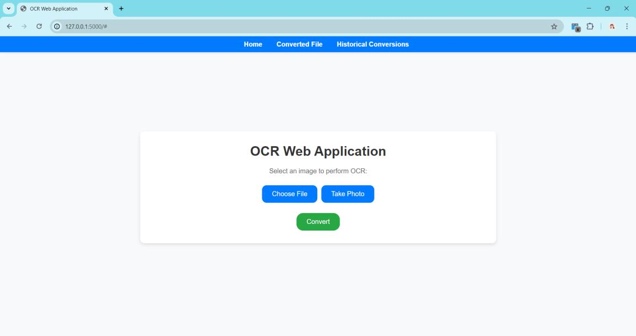
#### **Introduction GUI**

This report details the development of the front-end components of the application, including the HTML structure, CSS styling for responsiveness, and integration with Flask for backend functionality.

Method

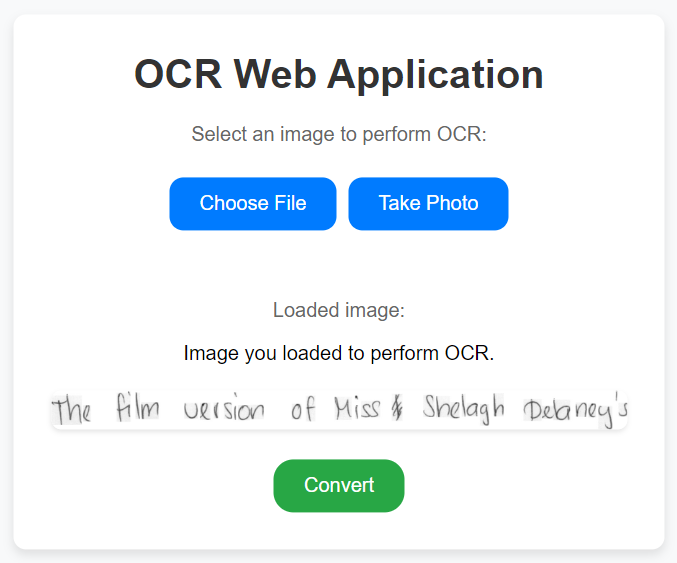
HTML Structure:

1. Navigation Bar: We created a fixed navigation bar at the top of the page, which includes links to different sections of the application (Home, Converted File, Historical Conversions). This ensures easy navigation for the user.



2. Containers: The main content is divided into three containers: `home-container`: This is the default view where users can upload an image or capture a photo using their device’s camera. `converted-file-container`: This displays the result after the OCR processing.

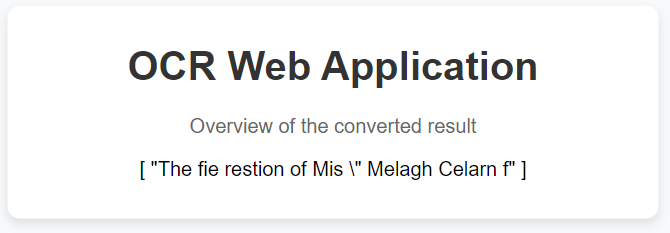
`historical-conversions-container`: This shows previously converted texts.

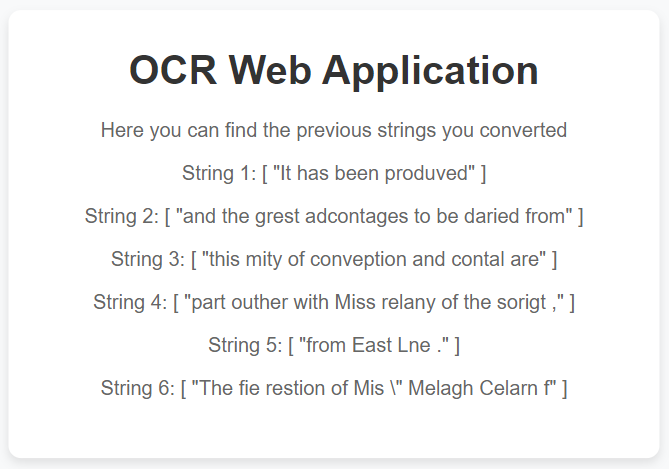
page1image19956960

CSS Styling:

We employed CSS to enhance the visual appeal and ensure the responsiveness of the web application. Key CSS features include: ″ Responsive Design: Objects of the form are sized according to screen size of the device used.

Button Styling: Buttons were styled to be visually consistent and included hover effects to improve interactivity.  
Mirroring Camera Feed: When capturing an image using the device’s camera, the video feed is mirrored horizontally to make pre-processing feature easier.





JavaScript Functionality:

JavaScript was used to add interactivity to the web page. Key functions include:

Camera Access: Functions to access the user's camera by creating a video element, capturing images, displaying the captured image, and removing the video element from

the DOM9. Functions involved: openCamera(), capturePhoto(), stopCamera()

File Upload: Handles image uploads, including format conversion and image delivery to the server via a POST request. It displays the uploaded image, waits for the server

response, and then switches the view to show the converted string results.

Functions involved: convertImage()  
Navigation: Switching between different containers based on user actions.

Functions involved: showHome(), showConvertedFile,

showHistoricalConversions

EventListener //Event listener for file input change event

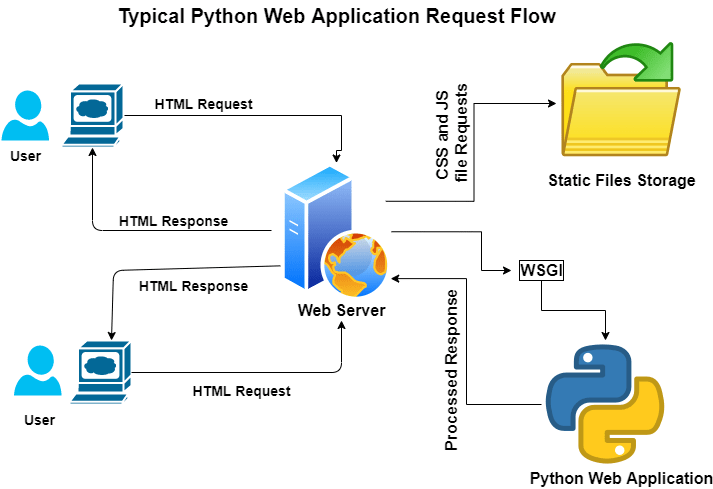
**Integration with Flask10**

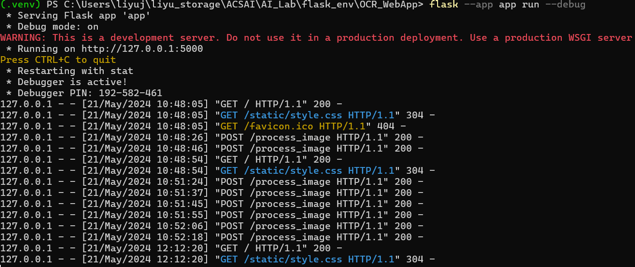
Flask is a WSGI application. A WSGI server is used to run the application, converting incoming HTTP requests to the standard WSGI environ, and converting outgoing WSGI responses to HTTP responses.

In our project, Flask:  
1. Renders the main HTML page: Displays the GUI interface for users.

2. Handles image processing requests: Receives the base64-encoded image data from the client, processes it using the OCR model, and returns the extracted text.

3. Manages different views: Switches between the home view, converted file view, and historical conversions view based on user interactions.





Results:

The application allows users to easily upload images or capture photos directly from their devices. The responsive design ensures that the application is accessible on various devices, providing a consistent user experience.

Challenges Encountered:

Camera Access and Mirroring: Implementing the camera functionality and ensuring the video feed was mirrored correctly required careful handling of video streams and CSS transformations.

Responsive Design: Ensuring the layout was fully responsive involved extensive testing and adjustments, particularly for mobile devices with different screen sizes.

Integration with Flask: Managing the communication between the front end and the Flask backend required a clear understanding of both client-side and server-side programming.

**References:**

[1] Zelun Wang, Jyh-Charn Liu(2019) Translating Math Formula Images to LaTeX Sequences Using Deep Neural Networks with Sequence-level Training https://arxiv.org/abs/1908.11415  
[2] How do you find a good learning rate https://sgugger.github.io/how-do-you-find-a-good-learning-rat e.html

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun(2015) Deep Residual Learning for Image Recognition

https://arxiv.org/abs/1512.03385  
[4] Aston Zhang, Zachary C. Lipton, Mu Li, Alexander J. Smola(2021) Dive into Deep Learning https://d2l.ai/  
[5] Dami Choi, Christopher J. Shallue, Zachary Nado, Jaehoon Lee Chris J. Maddison, George E. Dahl(2019)On Empirical Comparisons of Optimizers for Deep Learning https://arxiv.org/pdf/1910.05446  
[6] Handwritten sentence recognition with TensorFlow https://pylessons.com/handwritten-sentence-recognition  
[7] meijiru crnn.pytorch

https://github.com/meijieru/crnn.pytorch

[8] Carnegie Mellon University Ryan Brigden https://www.cs.cmu.edu/afs/cs/user/bhiksha/WWW/courses/deeple arning/Fall.2018/www/recitations/recitation8.pdf

[9] Document Object Model

[10]http://flask.palletsprojects.com/en/2.3x/deploying/