REPORT FOR TASK - 'Can you break the CAPTCHA?'

Shourya Sarkar

1 Introduction

Optical Character Recognition (OCR) has traditionally relied on handcrafted features. With the advent of deep learning, neural network–based methods have greatly improved performance on challenging problems such as CAPTCHA-breaking and reading text from noisy images. This project is structured into several tasks that progressively build an OCR pipeline:

- **Task 0: Dataset synthesis** (creating images with rendered words under different conditions).
- **Task 1: Classification** (training a neural network to classify word images into one of 100 classes).
- **Task 2: Generation** (designing a model that extracts the text from an image, handling variable-length sequences using a CNN+RNN+CTC architecture).
- **Task 3 (Bonus): Incorporating a background-based transformation rule**—if the background is red, the word is rendered reversed, but the target output is always the forward (correct) word. This requires special handling during both training and inference.

Each task builds on the previous ones and requires careful preprocessing, model design, training, and evaluation.

2 Task 0: Dataset Synthesis

2.1 Methodology

Objective: Synthesize a large dataset of (input=image, output=text) pairs where each image is a rendered word. The dataset is designed in three levels of difficulty:

2.1.1 Easy Set

- **Text Appearance:** The word is rendered in a fixed font with only the first letter capitalized (e.g., "Hello").
- **Background:** A plain white background with black text.

2.1.2 Hard Set

- **Text Appearance:** The word is rendered with random per-letter capitalization (e.g., "HeLLo" or "hElLo") to introduce variability.
- **Background:** Multiple fonts are used; the background is textured or noisy, and Gaussian noise is added to simulate real-world conditions.

2.1.3 Bonus Set (not detailed here)

Conditional Rendering: The bonus set follows all conditions of the hard set with an extra twist: if the background is green, the word is rendered normally; if the background is red, the word is rendered in reverse. However, the ground truth label remains in its forward (correct) form.

2.2 Implementation Details

Libraries Used:

- PIL (Pillow): To create and manipulate images.
- Matplotlib's font_manager: To access system fonts.
- NLTK: To select random words from a corpus.
- OpenCV: To add noise to the images.

Image Generation: The code renders text onto a blank RGB image (for example, of size 2048×64 or a reduced resolution such as 128×64 for faster experimentation). The font size is automatically adjusted to ensure that the entire word fits within the image boundaries, leaving a specified padding.

Output: Each generated image is saved with a filename that encodes the word and sample index. A CSV file is generated that contains the mapping of each image file to its corresponding word label.

3 Task 1: Word Classification

3.1 Methodology

Objective: Train a classifier that can assign a given word image to one of 100 classes (distinct words).

3.1.1 Data Preparation

Word Selection and Sample Generation:

- **Distinct Words:** First, 100 distinct words are selected from the corpus.
- **Sample Generation:** For each selected word, 30 samples are generated using the synthesis functions from Task 0. For the easy set, each word is rendered with only the first letter capitalized.

Data Splitting: The total dataset contains 100 words $\times$ 30 samples = 3000 images. For each class, the samples are split into:

- **Training Set:** 25 samples per word.
- **Test Set:** 5 samples per word.

(For the hard set version, these numbers were modified to 50 samples per word with a 40/10 split.)

Preprocessing: Images are loaded in grayscale, resized to a standard size (64×128), and normalized using mean and standard deviation. A mapping from word (label) to an integer is built so that each class is represented by a unique number.

3.2 Model Architecture

ResNet-18–Like Classifier
BasicBlock: A building block (defined as `BasicBlock`) implements two convolutional layers with batch normalization and ReLU activation. A skip (shortcut) connection is used to add the input to the output of the convolutional layers. This residual connection helps in training deeper networks by mitigating the vanishing gradient problem.
ResNetClassifier: The overall classifier (defined as `ResNetClassifier`) consists of:

3.2.1 Initial Layers

A convolutional layer with a large kernel (7×7), batch normalization, and ReLU activation, followed by max pooling. This quickly reduces the spatial dimensions and extracts low-level features.

3.2.2 Residual Layers

Four layers (each built by `_make_layer`) that stack multiple BasicBlocks. Each layer increases the number of feature channels and reduces the spatial dimensions further.

3.2.3 Global Pooling and Dropout

An Adaptive Average Pooling layer reduces the feature map to a fixed spatial size (1×1), and dropout (with $p = 0.5$) is applied for regularization.

3.2.4 Fully Connected Layer

A linear layer maps the flattened feature vector to logits over the 100 classes.

3.2.5 Instantiation

The helper function `ResNet18(num_classes)` instantiates the network with the standard ResNet-18 configuration ({[2, 2, 2, 2]} layers).

3.3 Training Procedure

3.3.1 Loss Function and Optimizer

Loss Function: The model is trained using `CrossEntropyLoss`, which is appropriate for multi-class classification.

Optimizer: The Adam optimizer is used with a learning rate of 0.001 and a weight decay (L2 regularization) of 1×10^{-4}. The weight decay helps reduce overfitting by penalizing large weights.

3.3.2 Training Loop

Batch Processing: The training loop iterates over the training DataLoader. For each batch:

- Images and labels are moved to the target device (GPU if available).
- A forward pass through the model produces logits.
- The loss is computed over the batch.
- Backpropagation is performed, and the optimizer updates the model parameters.
- The running loss is accumulated and the average loss per epoch is printed.

Epochs: The model is trained for a fixed number of epochs (in this code, 40 epochs are used).

3.4 Evaluation

3.4.1 Test Evaluation

Accuracy Computation: After training, the model is evaluated on the test subset. For each batch in the test DataLoader:

- The model's output is computed.
- The predicted class is determined by taking the index of the maximum logit.
- The predictions are compared with the true labels to compute the overall accuracy, which is printed.

3.4.2 Training Evaluation

Optional Evaluation on the Training Set: The code also contains an evaluation function that can be run on the training data to monitor how well the model fits the training samples.

Task - 2 : Sequence Generation (OCR)

Overview

Objective: The goal of Task-2 is to build a neural network that, given an input image containing a word, extracts the text embedded in the image. Since words can have variable lengths and there is no explicit segmentation provided, the model is trained with CTC loss. This loss function allows the network to learn an alignment between the predicted sequence and the target text automatically.

Key Components:

- Data Preparation: Load and preprocess images and labels.
- Model Architecture: A CNN (with residual blocks) followed by a bidirectional LSTM to produce a sequence of character probabilities.
- CTC Loss: To train the model without needing pre-segmented characters.
- Decoding: A greedy decoder to convert the network's output into predicted text.
- Evaluation: Compute metrics such as word accuracy, Levenshtein distance (edit distance), and Character Error Rate (CER).

2. Data Preparation

a. Data Loading and Transformation

CSV File: The dataset is assumed to be generated in a previous task and stored in a CSV file. The CSV contains at least two columns:

- `image_path`: The file path to each image.
- `label`: The ground-truth word (in lowercase).

`OCRSequenceDataset` **Class:** The custom dataset class performs the following:

- Image Loading: For each sample, it opens the image (using PIL) in grayscale mode (via `convert("L")`).
- Transformation: The image is resized to a fixed size (32×128, matching the network input) and normalized (mean 0.5, std 0.5). This is done using a `torchvision` transformation pipeline.

- Label Encoding: The ground-truth word (a string) is converted into a list of integer indices using a mapping (`char_to_idx`).
  - The vocabulary is built by concatenating all labels, converting them to lowercase, and extracting unique characters.
  - Index 0 is reserved for the blank token (required by CTC loss).

Custom Collate Function: Since the labels are variable-length sequences, a custom collate function is defined to:

- Stack image tensors.
- Concatenate all encoded labels into one long 1D tensor.
- Record the length of each label sequence.

This collate function is passed to the `DataLoader` so that batches are assembled correctly for CTC loss.

b. Data Splitting

Train/Test Split: The dataset is split into training and testing subsets (typically 80% training and 20% testing) using `random_split` so that the model can be trained and then evaluated on unseen data.

3. Model Architecture

a. Convolutional Feature Extractor with Residual Blocks

`ResidualBlock` **Class:** Each residual block contains:

- Two 3×3 convolutional layers with batch normalization and ReLU activation.
- A skip (shortcut) connection that may use a 1×1 convolution (with batch normalization) if the input and output dimensions differ.
- Dropout: After the addition of the identity (skip) connection and the activation, dropout is applied to regularize the network.

`CRNNResidual` **Class:** This class implements the overall network:

- Input: The network expects grayscale images of shape (1, 32, 128).
- Residual Blocks: A sequence of seven residual blocks is applied to the input. As the blocks proceed, spatial dimensions are reduced (using `stride=2`) and feature channels are increased.
- Reshaping: After the convolutional layers, the feature map has dimensions (B, 64, H', W'). This is reshaped into a sequence with shape (B, H'*W', 64) so that the spatial dimensions become the time (sequence) dimension.

- Bidirectional LSTM: The reshaped features are fed into a bidirectional LSTM with one layer (or more, if specified). The LSTM outputs a sequence of feature vectors, capturing context in both forward and backward directions.
- Fully Connected Layer: A final linear layer maps the LSTM outputs to logits over `output_dim + 1` classes (the extra class is for the CTC blank token).
- Output Permutation: The final output is rearranged into the shape (T, B, `num_classes`) to be compatible with the CTC loss function.

b. Dropout Integration Dropout is applied within each residual block to reduce overfitting by randomly zeroing out a fraction (here 20%) of the activations during training.

4. Training Procedure

a. Loss Function

CTC Loss: The Connectionist Temporal Classification (CTC) loss is used because the network predicts sequences (of variable lengths) without explicit segmentation of individual characters. The loss function aligns the network's output sequence with the target label sequence. The blank token index is set to 0. The output of the model is passed through a `log-softmax` before computing the CTC loss.

b. Optimizer

Adam Optimizer: Adam is used with a specified learning rate (1e-4). Adam's adaptive learning rate and momentum help optimize the network effectively even when gradients are noisy.

c. Training Loop

Epochs: The model is trained for a fixed number of epochs (here, 100 epochs).

Batch Processing: For each batch, images and target label sequences (and their lengths) are moved to the GPU (if available).

Input Lengths: A tensor is constructed to represent the number of time steps in the network's output (which is consistent for each sample in the batch).

Backpropagation: The CTC loss is computed, gradients are backpropagated, and the optimizer updates the model's weights.

Monitoring: The average loss per epoch is printed to monitor training progress.

5. Decoding and Evaluation

a. Greedy Decoder After training, the model's output (logits) is decoded using a greedy decoder that:

1. Permutes the output tensor to shape (batch, T, `num_classes`).

2. At each time step, selects the index with the highest probability.
3. Collapses consecutive duplicate indices and removes blank tokens.
4. Maps the remaining indices back to characters using the `idx_to_char` mapping.

This results in a list of predicted text strings, one per sample.

b. Evaluation Metrics The evaluation procedure calculates:

- Word Accuracy: The percentage of test samples where the predicted word exactly matches the ground-truth word.
- Levenshtein Distance: The average edit distance (number of insertions, deletions, or substitutions needed) between the predicted and true words.
- Character Error Rate (CER): The total edit distance divided by the total number of characters in the ground-truth texts.

These metrics provide insight into both overall and fine-grained transcription performance.

6. Visualization

For qualitative evaluation, the code includes a section that:

1. Denormalizes the Images: Converts the normalized tensor back to a displayable image.
2. Displays a Batch of Images: Using Matplotlib, the code displays a batch of test images alongside their predicted and ground-truth labels.

This visualization helps in assessing the OCR system's performance visually.

Task 3: Bonus – Handling Reversed Text Based on Background Color

Overview and Objectives

Objective: Task 3 aims to build an end-to-end OCR system that can correctly output the forward (canonical) text even when the input images have their characters rendered in reverse. This occurs in the bonus set where the rendering condition depends on the background color:

- Red background: The text in the image is rendered in reverse.
- Green background: The text is rendered normally.

The ground truth always remains the forward (correct) word. To help the model learn this behavior and to improve generalization, the network is built using a CNN+RNN (specifically, a CRNN with residual blocks) architecture with dropout layers integrated.

2. Data Preparation and Preprocessing

a. Dataset Generation and Labeling

Dataset Generation: The bonus set images are synthesized by a separate process (Task 0) that is not fully shown here. In the bonus set, the image generation function:

- Renders text on a background with a color chosen randomly from a red-ish or green-ish palette.
- Uses random per-letter capitalization to increase appearance variability.
- Optionally adds Gaussian noise (via a noisy background) to simulate realistic conditions.

Labeling and Background Analysis: For each image, the CSV file contains three columns:

- `set`: Indicates that the sample belongs to the bonus set.
- `image_path`: The file path where the image is stored.
- `label`: The ground-truth word (stored in lowercase).

When the dataset is loaded (via the `OCRBonusDataset` class), the code:

1. Opens the image in RGB to compute the average red and green channel intensities.
2. Uses a simple heuristic (if the average red is higher than green, the background is considered red) to determine the background color.

Based on the background color, the effective label for training is set:

- If red, the effective label is reversed (i.e. the string is reversed).
- If green, the label is left as is.

This preprocessing ensures that during training the model is provided with targets that are "flipped" when the image has a red background. Later, during inference, the model's output is post-processed (i.e. reversed again if necessary) to obtain the forward (correct) word.

b. Data Transformation

Transformation Pipeline: The images (originally generated in RGB) are loaded in grayscale. They are then resized to a fixed dimension (64×128, where the size is specified as (height, width)) and normalized using a mean of 0.5 and standard deviation of 0.5. This transformation ensures consistency in input to the model.

Custom Collate Function: Because the labels for the bonus set are variable-length sequences (when training with CTC loss, the labels are sequences of integers representing characters), a custom collate function is used. In this task, however, the classification is performed at the sequence level (with a greedy decoder) so that the network output is post-processed based on the background color.

3. Model Architecture

a. CNN + RNN (`CRNNResidual`) The network architecture is designed to handle sequence transcription (for OCR) using a combination of convolutional layers and recurrent layers, along with dropout for regularization:

Residual Blocks: The core building block is a residual block (the `ResidualBlock` class), which consists of two 3×3 convolutional layers with batch normalization and ReLU activations. A skip connection allows the network to add the input (possibly after a 1×1 convolution if needed) to the output of the two layers. Dropout: After the residual addition and activation, dropout is applied. This helps reduce overfitting by randomly zeroing out activations during training.

`CRNNResidual` **Class:** The overall network (`CRNNResidual`) is composed of several residual blocks arranged sequentially:

1. **Convolutional Feature Extraction:** The image (of size (1, 32, 128)) is processed through a series of seven residual blocks. Each block extracts increasingly abstract features.

2. **Reshaping for Sequence Modeling:** After the residual blocks, the feature map is reshaped so that one spatial dimension is interpreted as the sequence (time) dimension. For example, if the output of the convolutional part is of shape (B, 64, H, W), it is reshaped into (B, H*W, 64).

3. **Bidirectional LSTM:** The reshaped feature sequence is then processed by a two-layer bidirectional LSTM. This RNN layer models sequential dependencies and outputs a sequence of feature vectors.

4. **Fully Connected Layer:** A final linear layer maps the RNN outputs to logits for each character class (the number of classes is `output_dim + 1` to include the CTC blank token).

b. Dropout Integration

Where Dropout is Applied: In each residual block, dropout is applied after the addition of the residual connection and activation. This means that at every block, some of the feature activations are randomly dropped during training, which helps prevent the network from overfitting to the training data.

Overall Impact: Dropout helps the model generalize better to unseen data by introducing randomness and preventing reliance on specific activations. This is especially important in OCR tasks where variations in fonts, noise, and distortions occur.

4. Training and Loss

a. Loss Function

CTC Loss: The model is trained using Connectionist Temporal Classification (CTC) loss. CTC loss is designed for sequence-to-sequence problems where the alignment between the input sequence (the features) and the target sequence (the characters) is not known in advance. Blank Token: A blank token (with index 0) is reserved. Log Softmax: The outputs of the network are passed through a log softmax function before computing the CTC loss.

b. Optimizer

Adam Optimizer: The network is optimized using the Adam optimizer with a specified learning rate. Adam is popular for its adaptive learning rate and robustness to noisy gradients.

c. Training Loop

Epochs and Batching: The training loop iterates over the training `DataLoader` for a fixed number of epochs (`NUM_EPOCHS = 100`).

Batch Processing: For each batch, images and target labels (which are sequences encoded as integers) are moved to the appropriate device.

Input Lengths: A tensor containing the length (i.e. number of time steps) for each sample in the batch is constructed (assuming that all samples have the same length from the CNN feature extraction).

Backpropagation: The loss is computed, gradients are backpropagated, and the optimizer updates the model weights.

Monitoring: The average loss per epoch is printed to monitor training progress.

5. Evaluation and Post-Processing

a. Greedy Decoder

Decoding: After training, the model's output (a sequence of logits) is decoded using a greedy decoder. The decoder:

1. Selects the most likely character at each time step.
2. Collapses consecutive repeated predictions.
3. Ignores the blank token.

b. Background-Based Post-Processing

Why Post-Process? Since in the bonus set the effective label during training might have been reversed (if the background was red), the raw decoded output may be in the same "flipped" order.

Adjustment: For each sample, the background color is checked (as provided by the dataset).

- If the background color is red, the decoded text is reversed so that the final prediction matches the forward (correct) word.
- If the background color is green, the decoded text is used as is.

c. Evaluation Metrics

Word Accuracy: The proportion of samples where the final (adjusted) prediction exactly matches the ground truth word.

Levenshtein Distance: The average edit distance between the prediction and the ground truth, indicating how "off" the predictions are.

Character Error Rate (CER): The total edit distance divided by the total number of characters across all samples.

PERFORMANCE

Table 1: Training and Test Accuracy

	Train Accuracy	Test Accuracy
Task 1 (Easy Set)	99.9%	99.6%
Task 1 (Hard Set)	98%	88.8%
Task 2 (Easy Set)	100%	99.5%
Task 2 (Hard Set)	87.1%	78.6%
Task 3	89.9%	78.4%

INSIGHTS

1. I observed that increasing the dataset size for the Task 2 hard set improved accuracy. Specifically, increasing the dataset size from 2500 to 5000 and then to 10000 samples resulted in accuracy improvements from 65% to 76% and then to 87%, respectively.

2. A similar trend was observed for Task 3 (Bonus set) where increasing the dataset size led to improved accuracy.

3. Increasing the variety of fonts used in the dataset negatively impacted accuracy. For Task 2 and Task 3 (hard set and bonus set, respectively), increasing the number of fonts from 10 to 1000 resulted in a decrease in accuracy from 98% to 87%.

FINDINGS

Task 1: Word Classification – Methodology and Findings

1. Overview

The goal of Task 1 is to classify word images into one of 100 distinct classes. In this project, I implemented two different neural network architectures:

1. Simple CNN Classifier (Model 1)

2. ResNet-18-Like Classifier (Model 2)

Both models were trained using the same dataset and similar preprocessing steps. In the following sections i will describe the methodology behind each model and then analyze why the second model produced better results.

2. Model Architectures

Model 1: Simple CNN Classifier
Architecture Overview:

- **Convolutional Layers:**
  - The model starts with a 2D convolution layer that maps the single-channel input (grayscale) to 32 feature maps using a 3×3 kernel with stride 1 and padding 1.
  - A ReLU activation follows.
  - A MaxPooling layer with a kernel size of 2 reduces the spatial dimensions by half.
  - A second convolution layer increases the depth from 32 to 64 feature maps (again with a 3×3 kernel, stride 1, padding 1), followed by another ReLU.
  - A second MaxPooling layer further reduces the spatial dimensions.
- **Fully Connected Layers:**
  - After the convolutional layers, the feature maps are flattened.
  - A fully connected (dense) layer reduces the flattened features to 256 hidden units.
  - A ReLU activation is applied.
  - A final fully connected layer maps the 256 units to the number of classes (100).

Key Points of the Simple CNN:

- **Simplicity:** The architecture is straightforward, with only two convolutional layers and two fully connected layers.
- **Feature Extraction:** The two convolutional layers capture low-to-medium level features (edges, textures) but may not capture more complex invariances.
- **Limitations:** With a relatively shallow network, this model may have limited capacity to handle the variability in hard-set images (which may include noise, variations in font, and inconsistent text appearances).

Model 2: ResNet-18-Like Classifier
Architecture Overview:

- **Residual Blocks:** The core building unit is the `BasicBlock`, which consists of two 3×3 convolutional layers each followed by batch normalization and a ReLU activation.

- **Skip Connections:** The block uses a skip (shortcut) connection (possibly with a 1×1 convolution if dimensions do not match) to add the input directly to the output of the block. This alleviates the vanishing gradient problem and enables training of deeper networks.
- **Dropout:** A dropout layer is applied within the block (after the residual addition and activation), which improves regularization.
- **ResNet-18-Like Architecture:**
  - **Initial Layers:** The network begins with a 7×7 convolution layer with stride 2, followed by batch normalization, ReLU, and max pooling.
  - **Residual Layers:** Four sequential layers (each constructed using `_make_layer`) stack multiple `BasicBlock`s. As the network goes deeper, the number of feature channels increases (from 64 to 512), and the spatial dimensions are reduced via stride-2 convolutions.
  - **Global Average Pooling and Dropout:** An adaptive average pooling layer reduces the spatial dimensions to a single value per feature map. Dropout (with probability 0.5) is applied before the final classification layer.
  - **Fully Connected Layer:** A linear layer maps the final features to the output classes (100).

Key Points of the ResNet-18-Like Model:

- **Deeper Architecture:** The residual blocks allow for a much deeper network compared to the simple CNN. This extra depth helps capture more complex and abstract features that are necessary to distinguish between 100 classes, especially in challenging, noisy images.
- **Residual Connections:** The skip connections improve gradient flow, reduce the risk of vanishing gradients, and allow the network to learn identity mappings, which is beneficial when deeper layers are not needed.
- **Regularization with Dropout:** Dropout is explicitly applied before the fully connected layer. This regularizes the network by randomly zeroing out activations, reducing overfitting.
- **Robust Feature Extraction:** The combination of convolutional layers, batch normalization, and residual connections enables the network to learn robust and invariant features, which is particularly useful for the variability in hard-set images.

3. Comparison and Insights

Why the ResNet-18-Like Classifier (Model 2) Outperforms the Simple CNN Classifier (Model 1):

- **Depth and Representational Power:**
  - **Model 1:** The simple CNN consists of only two convolutional layers. This limits the complexity of features that can be extracted. While it may be sufficient for relatively clean, simple images, it might struggle with the noisy, varied conditions of hard-set images.
  - **Model 2:** The ResNet-18-like model is much deeper, using several residual blocks. The depth allows it to learn higher-level abstractions and more robust features, which are critical for distinguishing between 100 classes in challenging conditions.
- **Residual Connections:** The skip connections in Model 2 enable the training of deeper networks by preserving gradients. This architecture helps the network learn identity functions and simplifies the learning process when additional layers are not needed for transformation. This leads to better generalization and performance.
- **Regularization:** Model 2 integrates dropout and batch normalization in multiple layers. Dropout in particular prevents overfitting by randomly deactivating neurons during training, forcing the network to learn redundant representations. Batch normalization stabilizes and speeds up training. The simple CNN lacks these advanced regularization techniques.
- **Feature Extraction Robustness:** The deeper network in Model 2 can extract more complex features. Hard-set images often have variations such as noise, inconsistent text appearance, and background interference. The ResNet-18-like model's robust feature extractor is better suited to handle such variability.
- **Empirical Findings:** Experiments indicated that the second model performs better. This is likely because its increased capacity, better regularization, and more sophisticated architecture allow it to learn the nuances of the hard-set data more effectively than the shallow simple CNN.

Task 2: OCR Generation – Methodology and Comparative Analysis

Overview

The goal of Task 2 is to build a model that extracts text from an image. Because words have variable lengths and no explicit segmentation is provided, the network is designed to output a sequence of predictions that is later decoded using Connectionist Temporal Classification (CTC) loss. Two different architectures were implemented:

Model 1 – CRNN A straightforward CRNN model built with a series of convolutional blocks (organized sequentially), followed by bidirectional LSTM layers and a final fully connected (FC) layer. This architecture was adapted directly from a Keras implementation.

Model 2 – CRNNResidual An enhanced CRNN model that replaces the plain convolutional blocks with residual blocks (featuring skip connections and dropout) followed by a bidirectional LSTM and a fully connected layer. This design is inspired by ResNet architectures and is intended to ease training and improve feature extraction.

Below we detail the methodology behind each model and then explain why the second model outperforms the first.

1. Model 1: CRNN

Methodology

Convolutional Feature Extractor: The network begins with a series of convolutional blocks:

Block 1: A convolutional layer ($1 \rightarrow 16$ channels) with a 3×3 kernel (padding 1), followed by a ReLU activation and dropout (0.25). A second convolutional layer ($16 \rightarrow 32$ channels) with a 3×3 kernel, another ReLU activation, and finally a max pooling layer that halves the spatial dimensions.

Block 2: Another convolutional block processes the features further (from 32 channels), applies ReLU and dropout, then max pooling.

Block 3 and Block 4: More convolutional layers are applied (with batch normalization added in Block 3) to refine the features. Block 4 also includes a pooling layer (with a pool size of (2, 1)) to change the feature dimensions.

Block 5 & Block 6: Additional convolutional layers increase the depth to 256 channels and then process these features further with batch normalization and max pooling.

Block 7: A final convolutional block (with a 2×2 kernel) increases the channel depth to 512.

Sequence Preparation: After Block 7, the feature tensor has a shape of (batch, 512, 1, 31). The height dimension is squeezed so that the data is reshaped to (batch, 512, 31) and then transposed to (batch, 31, 512). This step transforms the spatial feature maps into a sequential representation.

Recurrent Layers: Two bidirectional LSTM layers are used:

LSTM 1: Takes the sequential features as input and outputs features of dimension 256 (since it is bidirectional with hidden size 128 in each direction).

LSTM 2: Further processes the sequence, also outputting features of dimension 256.

Final Classification: A fully connected (FC) layer is applied at every time step to project the features to the desired number of classes. The output is then permuted to meet the requirements of the CTC loss function (with shape (time_steps, batch, num_classes)).

Strengths and Limitations

Strengths:

- **Simplicity:** The architecture is straightforward and directly follows the Keras design.
- **Ease of Implementation:** Fewer layers and a direct sequential structure make it simpler to implement and debug.

Limitations:

- **Limited Depth:** With only a few convolutional blocks, the model's capacity to capture high-level features is limited.
- **Training Challenges:** Without skip connections, the network might face issues such as vanishing gradients when deeper layers are added.
- **Generalization:** The simple architecture may struggle to generalize well on noisy or highly variable data, which is common in OCR tasks.

2. Model 2: CRNNResidual

Methodology

Residual Blocks for Convolutional Feature Extraction:
ResidualBlock: Each residual block in this model consists of:

- Two 3×3 convolutional layers, each followed by batch normalization and a ReLU activation.
- A skip connection that adds the input (or a transformed version of it, if dimensions differ) directly to the output of the block.
- Dropout is applied at the end of the block to improve regularization.

Stacking Residual Blocks: The network stacks a series of residual blocks to form its convolutional feature extractor. In this implementation:

- The first block processes the input with 16 filters.
- Subsequent blocks increase the number of filters ($16 \rightarrow 32 \rightarrow 64$) while reducing spatial dimensions via strided convolutions.

Sequence Preparation and Recurrent Layers: After processing through the residual blocks, the feature map has a shape (batch, 64, H', W'). The spatial dimensions are flattened into a sequence (reshaped to (batch, seq_len, 64), where seq_len = H'×W'). A bidirectional LSTM is used to process the sequential data, capturing dependencies in both forward and backward directions. This layer outputs a feature vector for each time step.

Final Projection: A fully connected layer is applied at each time step to project the LSTM output to a vector of logits for each character class (including the blank token required for CTC loss). The output tensor is then permuted to match the expected input shape for CTC loss (time_steps, batch, num_classes).

Strengths and Limitations

Strengths:

- **Deeper and More Robust:** The residual architecture allows for a deeper network by using skip connections, which help preserve gradients and enable the learning of identity functions.
- **Better Feature Extraction:** The residual blocks capture more complex and hierarchical features, which is critical for recognizing text in noisy images.
- **Improved Regularization:** Dropout integrated within the residual blocks helps reduce overfitting, making the model more robust.
- **Effective Sequence Modeling:** The combination of a deep CNN encoder with a bidirectional LSTM effectively captures both spatial and sequential dependencies in the input.

Limitations:

- **Complexity:** The increased depth and the use of residual connections make the model more complex and computationally expensive.
- **Hyperparameter Tuning:** The more complex architecture may require more careful tuning of hyperparameters (e.g., learning rate, dropout rate) to achieve optimal performance.

3. Comparative Analysis and Insights

Why Model 2 (CRNNResidual) is Better Than Model 1 (Simple CRNN):

Enhanced Feature Extraction: Model 1 uses a simple sequential CNN which, although effective in capturing basic features, might not capture more abstract or invariant features needed for robust OCR. Model 2 uses residual blocks, which enable a much deeper network that can learn higher-level representations. The skip connections in residual blocks help in training very deep networks by mitigating the vanishing gradient problem.

Improved Gradient Flow: Residual connections in Model 2 allow the gradient to flow directly through the network. This eases training, particularly in deeper architectures, leading to better convergence and overall performance.

Regularization and Robustness: Model 2 integrates dropout within each residual block, which improves generalization by reducing overfitting. In contrast, the simple CRNN may not have as robust regularization mechanisms, leading to a higher risk of overfitting on noisy or limited data. The combination of batch normalization and dropout in Model 2 makes it more robust against the variability introduced in the hard-set images.

Architecture Depth and Capacity: The deeper architecture of Model 2 allows it to learn more complex, hierarchical features compared to Model 1.

This increased capacity is crucial when dealing with challenging inputs where text might be distorted, noisy, or rendered in various fonts and styles. As a result, Model 2 can better differentiate between similar-looking characters or words under adverse conditions.

Empirical Performance: Your experiments showed that Model 2 outperforms Model 1. This observation is consistent with theoretical expectations: the residual architecture's enhanced feature extraction and regularization capabilities generally yield higher accuracy and better generalization, especially in tasks like OCR where data variability is high.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[2] Youngmin Baek, Kiwon Yun, Donghwan Kim, and Sanghyun Park. Character region awareness for text detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9332–9341, 2019.