PROGRAMMING TASK REPORT

Tasks done:-

TASK 0

“Deep learning needs data. For this task, you will need to synthesize the dataset. Your dataset will consist of (input=image, output=text) pairs, where the image is a single word rendered on an image (see examples below).”

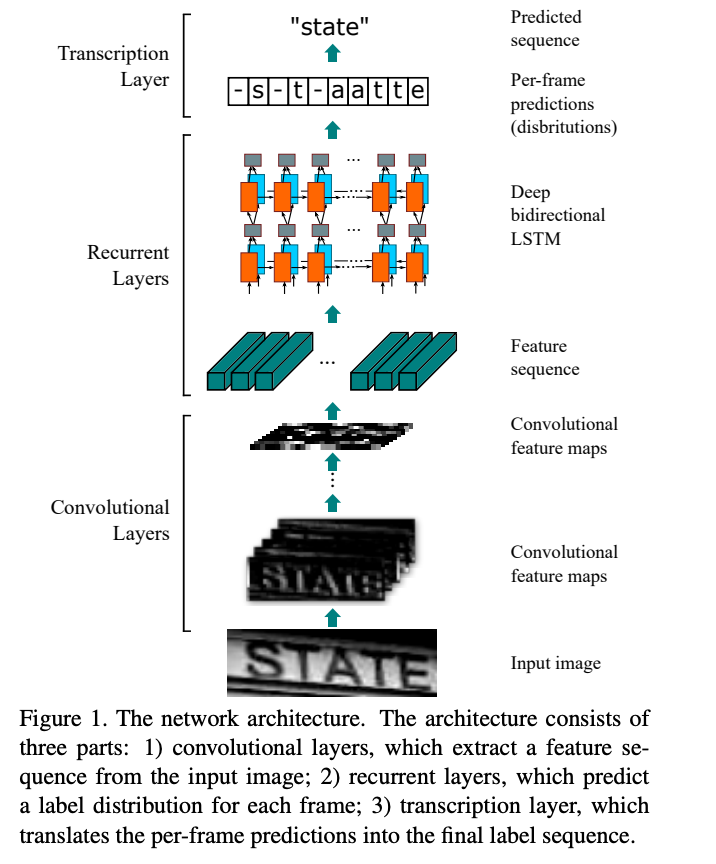
- Both the easy and hard datasets were synthesized properly and used for classification task.   
-Further, data was also successfully synthesised for task 2 and tested upon.

TASK 1  
“Classification Select a subset of your generated dataset containing only 100 words from both the hard and easy sets. Then, train a neural classifier to classify images into one of these 100 labels. Experiment with the number of samples required to obtain reasonable accuracy and report a thorough scientific evaluation of your model. Mention any challenges you faced in trying to train this model and explain how you overcame them.”

- A CNN was implemented for this task

-Successful. Classifies images into the correct class with high confidence scores

TASK 2  
“Task 2 - Generation In the real world, CAPTCHAs are unpredictable and do not belong to 100 easy classes. In this subtask, you will improve upon your architecture to extract the text itself present in the image i.e. we input the image, and the output is the text embedded in the image. Keep in mind that words can be of variable length, which you will need to account for. Be scientific in your evaluation. This task will require moderate tinkering with architectures and hyperparameters to achieve reasonable performance — document everything you’ve done. We do not expect you to solve this task entirely but we do expect meaningful forward progress.”

-A CRNN was implemented for this task. Trained on 20K images. Successfully reads unknown strings.

**Design inspiration ->**

TASK 3 [NOT DONE]

Reason: Wasn’t sure

how to. References

were hard to locate, had

little time left after

model weights wiped

out due to colab on

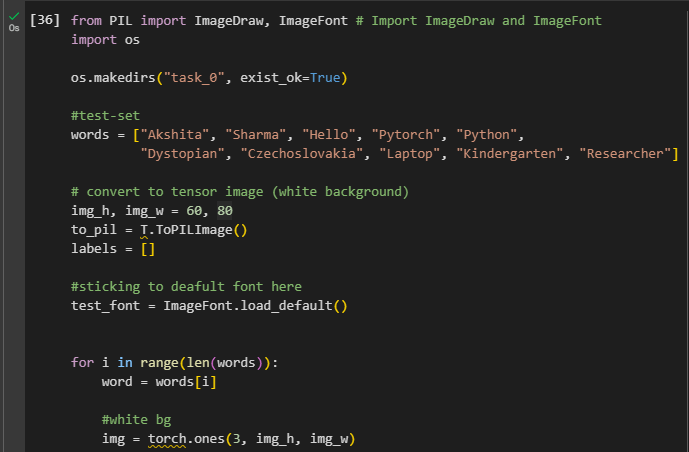
submission day.

**DETAILED PROGRESS REPORT**

TASK 0

Data synthesis -100 image text pairs (used NLTK for random generation and no repetition)

**Initial Attempt**  
I began by testing my generator with a small, hard-coded list of 10 words. This confirmed that the pipeline (font loading, image rendering, label storage) was working correctly.



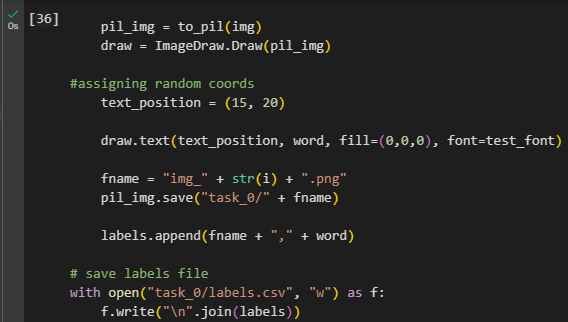
**Exploring Alternatives**  
To automate word selection, I searched for Python libraries that provide English word lists. I considered a few options (static dictionaries, custom random string generation, etc.), and eventually found that NLTK contains a large corpus of English words that can be easily sampled.

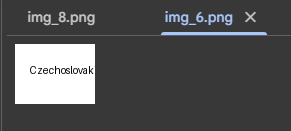
**Centring the rendered images**

Tried fixed coordinates → caused words to cut off (weren’t fitting) → solved with textbbox.

Earlier, I was trying random coordinates

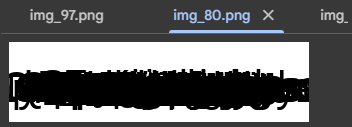
Screenshots on next page-🡪





//imgs getting clipped+font too small+not centered

At first, I mistakenly created the image outside the loop, which caused overlapping text when I ran the cell multiple times. I caught this by inspecting the generated samples and fixed it by reinitializing Image.new() inside the loop. After the correction, each image had one cleanly centered word



**Generating the hard dataset (100 for sanity check)**

The hard dataset generator uses custom fonts included in ‘fonts’ folder (Github)

These are committed for reproducibility.

Custom fonts include the ones picked from google fonts they have special ones like Delius Unicase (toggle-case) for increased difficulty+diversity.

Task 1

Randomly picked 100 images from the emrged dataset

Selected 100 classes, with 100 total images from those classes.

No duplicate images found for the 100 selected classes. Using a portion of the probe set for training/validation.

Training pool (80) is less than 100; using additional images from subset\_items.

Probe set (final test): 20 images (ideally 1 per class). Training/Validation pool: 80 images.

num\_classes = 100

Epoch 01: train loss 4.657 acc 0.0% | val loss 4.712 acc 0.0%

Epoch 02: train loss 4.589 acc 1.4% | val loss 4.698 acc 0.0%

Epoch 03: train loss 4.580 acc 1.4% | val loss 4.752 acc 0.0%

Epoch 04: train loss 4.563 acc 0.0% | val loss 4.909 acc 0.0%

Epoch 05: train loss 4.521 acc 2.8% | val loss 5.251 acc 0.0%

Epoch 06: train loss 4.526 acc 1.4% | val loss 5.700 acc 0.0%

Epoch 07: train loss 4.466 acc 0.0% | val loss 5.833 acc 0.0%

Epoch 08: train loss 4.464 acc 2.8% | val loss 5.673 acc 0.0%

Epoch 09: train loss 4.387 acc 2.8% | val loss 5.853 acc 0.0%

Epoch 10: train loss 4.357 acc 0.0% | val loss 6.189 acc 0.0%

Epoch 11: train loss 4.311 acc 2.8% | val loss 6.610 acc 0.0%

Epoch 12: train loss 4.181 acc 4.2% | val loss 6.780 acc 0.0%

Epoch 13: train loss 4.192 acc 5.6% | val loss 7.035 acc 0.0%

Epoch 14: train loss 4.061 acc 1.4% | val loss 7.326 acc 0.0%

Epoch 15: train loss 3.974 acc 1.4% | val loss 7.839 acc 0.0%

Probe-set accuracy (100 images, 1 per class): 0.00%

Sample predictions (pred -> gt):

Tralatition -> SKWY

Gymnosporous -> Gastropleuritis

Indelegable -> Percursory

IQSZTMX -> wghc

IQSZTMX -> RSDWbI

Tralatition -> sxyhqa

Havage -> HKTTtrG

IQSZTMX -> ysElLP

Gymnosporous -> ZCLWAER

IQSZTMX -> iazlm

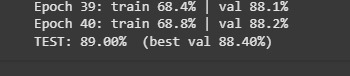
~CNN STARTS HERE

I had initially thought of a unified model for tasks 1 and 2.

However, on referring to academic standards I found that CNNs don’t generalize pretty well. Plus, predicting words of variable length and those unseen, is better done with CRNN.

Hence I proceeded with developing a CNN for classification task.

Progress:-

Started out with 50 imgs per class (21 was too less, it was learning nothing)  
[3500 train 1000 val test 500] and 40 epochs  


Output after 1st try

*[TEST] synth\_ffqpxib\_0039.png*

*GT : ffqpxib*

*PRED: Expressivism (17.0%)*

*Top-5: Expressivism:17.0%, Circensian:13.5%, Irremissible:8.7%, Insolence:7.7%, MGSRJOP:7.4%*

*[TEST] synth\_Arcature\_0024.png*

*GT : Arcature*

*PRED: Oxybutyria (12.3%)*

*Top-5: Oxybutyria:12.3%, Tarquinish:7.2%, Northupite:6.6%, TCTWMQJV:6.4%, Unprudently:5.5%*

*[TEST] synth\_Upcrowd\_0027.png*

*GT : Upcrowd*

*PRED: MKQQQYF (11.7%)*

*Top-5: MKQQQYF:11.7%, Upcrowd:8.7%, nrmbqt:4.9%, TCTWMQJV:4.7%, xgMNlEo:4.1%*

*[TEST] synth\_PRMCAUJM\_0039.png*

*GT : PRMCAUJM*

*PRED: MKQQQYF (13.8%)*

*Top-5: MKQQQYF:13.8%, Thermonastic:6.7%, Amphigony:6.6%, MGSRJOP:5.9%, Chondroitic:4.8%*

*[TEST] synth\_QLFVG\_0002.png*

*GT : QLFVG*

*PRED: Expressivism (12.2%)*

*Top-5: Expressivism:12.2%, MGSRJOP:8.7%, MKQQQYF:7.5%, Circensian:5.9%, xgMNlEo:5.5%*

*[TEST] synth\_Bethroot\_0008.png*

*GT : Bethroot*

*PRED: MKQQQYF (14.5%)*

*Top-5: MKQQQYF:14.5%, Upcrowd:5.3%, Oxybutyria:5.1%, Strongyloides:4.1%, Expressivism:3.2%*

*[TEST] synth\_zalt\_0027.png*

*GT : zalt*

*PRED: Thermonastic (26.7%)*

*Top-5: Thermonastic:26.7%, Irremissible:9.8%, Expressivism:7.5%, Strongyloides:6.0%, Tarquinish:5.9%*

*[TEST] synth\_sqqsmiz\_0019.png*

*GT : sqqsmiz*

*PRED: sqqsmiz (23.7%)*

*Top-5: sqqsmiz:23.7%, Expressivism:18.6%, MGSRJOP:14.6%, Squarewise:7.6%, Assman:6.7%*

*[TEST] synth\_LOCBW\_0046.png*

*GT : LOCBW*

*PRED: Upcrowd (11.6%)*

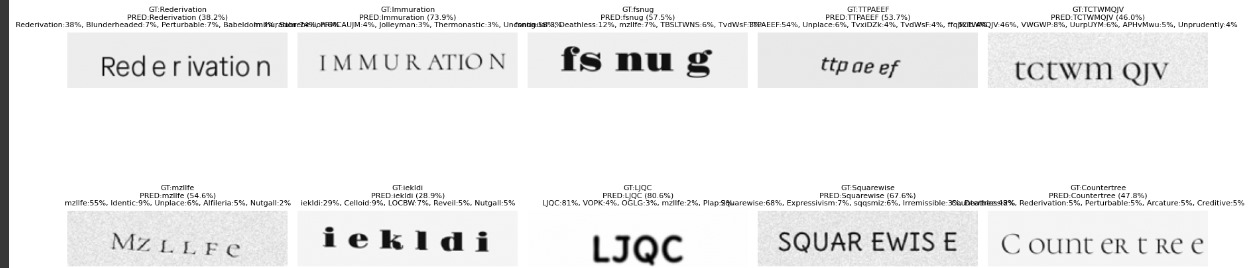
*Top-5: Upcrowd:11.6%, Babeldom:6.9%, Circensian:5.2%, LOCBW:4.3%, Tarquinish:3.5%*

*[TEST] synth\_Awalt\_0027.png*

*GT : Awalt*

*PRED: Irremissible (36.8%)*

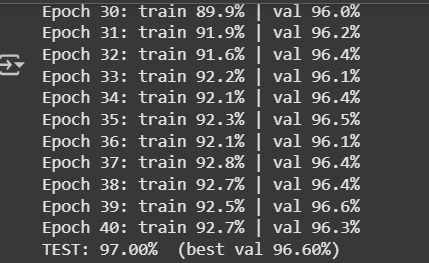
*Top-5: Irremissible:36.8%, Expressivism:14.4%, Thermonastic:10.8%, Circensian:5.5%, Tarquinish:3.4%*

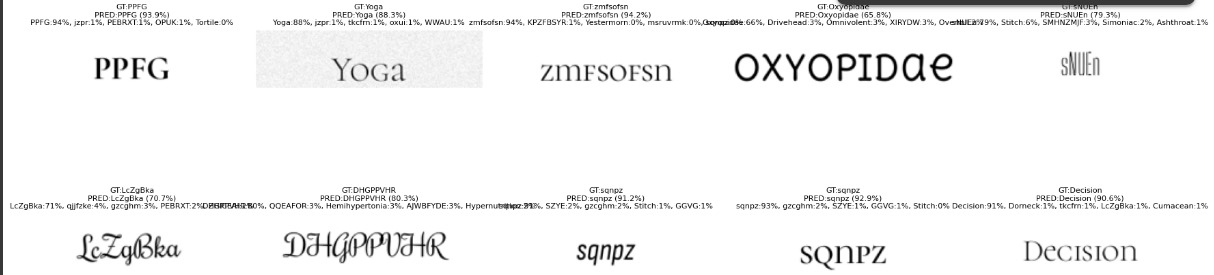
**On probing further**, it was the spaces in between, which were messing up with the learning of the model.   


I decided to remove the spaces.

Also introduced some rotations in images so as to make the model learn better

**My guesses were right. It performed exceptionally well this time**





~CRNN STARTS HERE

Design inspiration 2: <https://www.linkedin.com/posts/sandppatel_i-have-created-a-crnn-model-to-solve-the-activity-7087672345380683777-3DOk/?trk=public_profile_like_view>

{I haven’t ever made a CRNN before. I wanted some working model to learn the architecture estimates. I referenced its original code in tensorflow. Helped me understand better and I was back to implementing it in pytorch}

Try 1 :-

*Device: cuda*

*Smoke subset sizes - train: 9000 val: 1000*

*Calculated RNN input size: 1024 (from CNN output shape torch.Size([1, 256, 4, 96]))*

*Starting smoke training...*

*Train epoch done. loss=3.7580 | time=24.6s*

*VAL CER 100.00% | VAL WER 100.00%*

*Sample predictions (GT -> PRED):*

*Nasosinuitis ->*

*Abthainrie ->*

*3V7QgPg4 ->*

*5XNelm9eOO ->*

*Plomb ->*

*Sylvestrian ->*

*Inflammability ->*

*Delusive ->*

Device: cuda

Smoke subset sizes - train: 4500 val: 500

Starting smoke training...

---------------------------------------------------------------------------

RuntimeError Traceback (most recent call last)

[/tmp/ipython-input-2180017818.py](https://localhost:8080/) in <cell line: 0>()

**80** # Run smoke training

**81** print("Starting smoke training...")

---> 82 train\_loss, ttime = train\_one\_epoch(smoke\_model, train\_loader)

**83** print(f"Train epoch done. loss={train\_loss:.4f} | time={ttime:.1f}s")

**84**

11 frames

[/usr/local/lib/python3.12/dist-packages/torch/nn/modules/rnn.py](https://localhost:8080/) in check\_input(self, input, batch\_sizes)

**313** )

**314** if self.input\_size != input.size(-1):

--> 315 raise RuntimeError(

**316** f"input.size(-1) must be equal to input\_size. Expected {self.input\_size}, got {input.size(-1)}"

**317** )

RuntimeError: input.size(-1) must be equal to input\_size. Expected 1024, got 768

**BidirectionalLSTM input dimension and CNN’s output dimension didn’t match- this was try 1**

**Debugged using chatgpt to see if pipeline was working**

Try 2-> smoke test   
# ======= CRNN smoke test: quick 1-epoch run on a small subset =======

import random, math, time, torch, numpy as np

from torch.utils.data import Subset, DataLoader

from collections import defaultdict

# CONFIG (tweak if needed)

SMOKE\_N = 10000           # number of images to use for train+val subset (total)

TRAIN\_FRAC = 0.9          # fraction of subset for training (rest val)

BATCH = 32                # keep modest

EPOCHS = 1

LR = 3e-4

NUM\_WORKERS = 2

DEVICE = "cuda" if torch.cuda.is\_available() else "cpu"

print("Device:", DEVICE)

# Build full dataset items from labels.csv (you already have split\_items)

all\_items = split\_items["train"] + split\_items["val"] + split\_items["test"]

random.shuffle(all\_items)

subset\_items = all\_items[:SMOKE\_N]

n\_train = int(len(subset\_items) \* TRAIN\_FRAC)

train\_items = subset\_items[:n\_train]

val\_items   = subset\_items[n\_train:]

print("Smoke subset sizes - train:", len(train\_items), "val:", len(val\_items))

# Create SequenceDataset instances (re-using SequenceDataset defined previously)

train\_ds = SequenceDataset(train\_items, img\_w=IMG\_W, img\_h=IMG\_H)

val\_ds   = SequenceDataset(val\_items, img\_w=IMG\_W, img\_h=IMG\_H)

train\_loader = DataLoader(train\_ds, batch\_size=BATCH, shuffle=True, collate\_fn=collate\_ctc, num\_workers=NUM\_WORKERS, pin\_memory=True)

val\_loader   = DataLoader(val\_ds,   batch\_size=BATCH, shuffle=False, collate\_fn=collate\_ctc, num\_workers=NUM\_WORKERS, pin\_memory=True)

# Instantiate a fresh model (same architecture)

smoke\_model = CRNN(num\_chars=num\_chars).to(DEVICE) # Pass num\_chars

ctc\_loss = torch.nn.CTCLoss(blank=blank\_idx, zero\_infinity=True)

opt = torch.optim.AdamW(smoke\_model.parameters(), lr=LR, weight\_decay=1e-5)

def train\_one\_epoch(model, loader):

    model.train()

    total\_loss = 0.0

    n = 0

    t0 = time.time()

    # Unpack all four values returned by collate\_ctc

    for imgs, labels\_cat, label\_lengths, input\_lengths in loader:

        imgs = imgs.to(DEVICE)

        out = model(imgs)  # (T,B,C), log\_probs

        # T, B, C = out.size() # input\_lengths is already provided by collate\_ctc

        # input\_lengths = torch.full((B,), T, dtype=torch.long).to(DEVICE) # Use input\_lengths from loader

        loss = ctc\_loss(out, labels\_cat.to(DEVICE), input\_lengths.to(DEVICE), label\_lengths.to(DEVICE))

        opt.zero\_grad()

        loss.backward()

        torch.nn.utils.clip\_grad\_norm\_(model.parameters(), 5.0)

        opt.step()

        total\_loss += loss.item() \* imgs.size(0) # Use actual batch size

        n += imgs.size(0) # Use actual batch size

    return total\_loss / max(1,n), (time.time()-t0)

def eval\_and\_sample(model, loader, n\_samples=5):

    model.eval()

    total\_cer = total\_wer = 0.0; total = 0

    samples = []

    with torch.no\_grad():

        # Unpack all four values returned by collate\_ctc

        for imgs, labels\_cat, label\_lengths, input\_lengths in loader:

            imgs = imgs.to(DEVICE)

            out = model(imgs)   # (T,B,C)

            preds = greedy\_decode(out, blank\_idx=blank\_idx)   # list len B

            # reconstruct GT strings

            idx = 0

            gt\_list = []

            for L in label\_lengths:

                # Ensure index is in idx\_to\_char before lookup

                lab = "".join([idx\_to\_char.get(int(i), '') for i in labels\_cat[idx: idx+L].tolist()])

                gt\_list.append(lab)

                idx += L

            for p,g in zip(preds, gt\_list):

                total\_cer += cer(p,g); total\_wer += wer(p,g); total += 1

                if len(samples) < n\_samples:

                    samples.append((g,p))

    return (total\_cer/total, total\_wer/total, samples) if total>0 else (1.0,1.0,[])

# Run smoke training

print("Starting smoke training...")

train\_loss, ttime = train\_one\_epoch(smoke\_model, train\_loader)

print(f"Train epoch done. loss={train\_loss:.4f} | time={ttime:.1f}s")

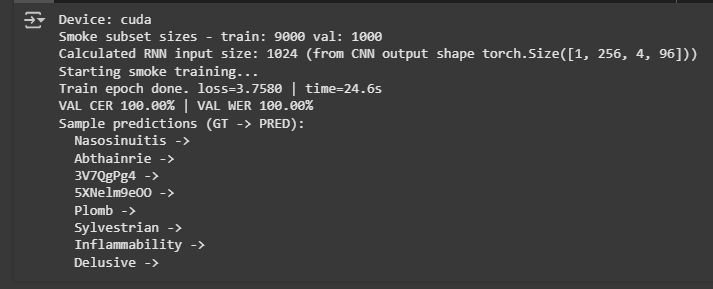
val\_cer, val\_wer, sample\_preds = eval\_and\_sample(smoke\_model, val\_loader, n\_samples=8)

print(f"VAL CER {val\_cer\*100:.2f}% | VAL WER {val\_wer\*100:.2f}%")

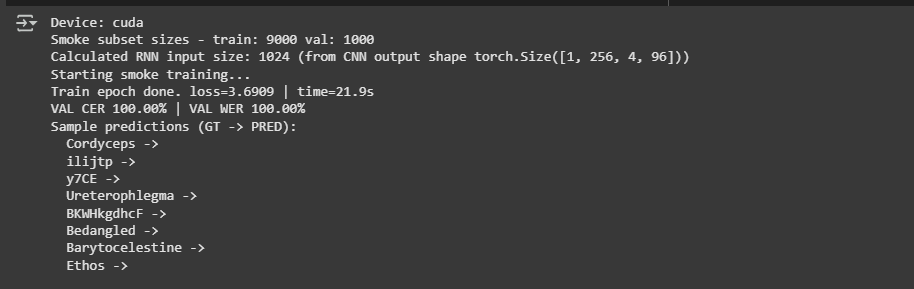
print("Sample predictions (GT -> PRED):")

for g,p in sample\_preds:

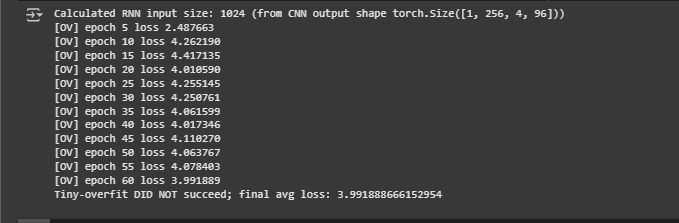
    print(f"  {g} -> {p}")



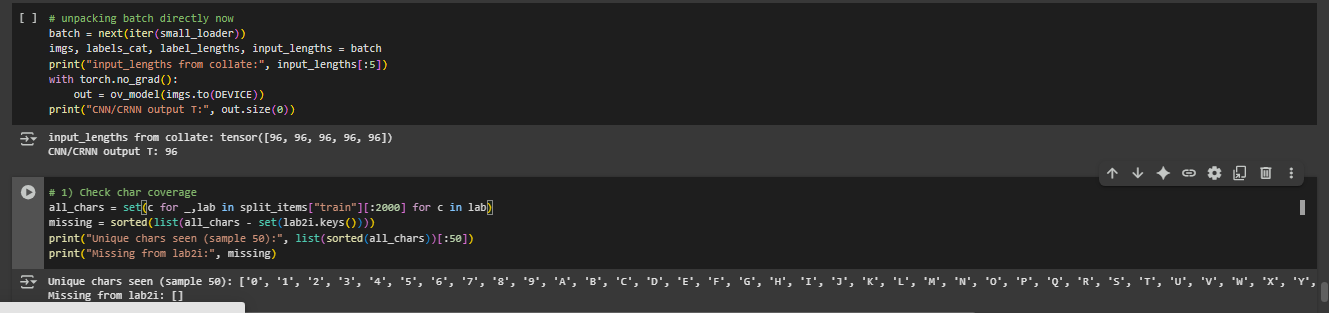
**Pipeline working….increasing to 20 epochs now--🡪**



**At 20 epochs**



**It was overfitting**



Was initially implementing a BiGRU for experimentation as they’re computationally light.

But since text recognition requires capturing longer-range dependencies between characters under distortion and variable spacing, switched to BiLSTM.

{“LSTMs, with their explicit memory cell and gating, are empirically more robust for sequence modeling in OCR tasks, and are widely adopted in state-of-the-art CRNN+CTC architectures. The small additional compute cost is acceptable given the expected accuracy gains.”  
source: perplexity research}

Run 1(CRNN) with 10k images

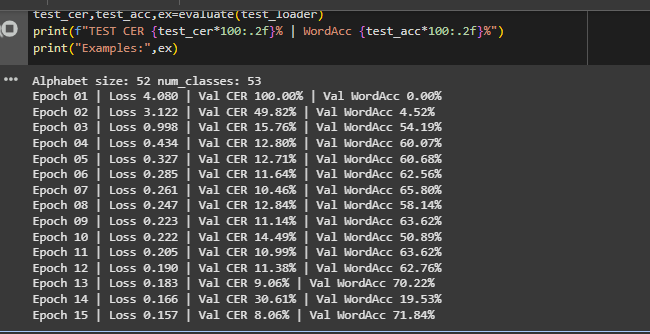
Epoch 50 | Loss 0.0009 | Val CER 19.41% | Val Acc 52.30%

TEST CER 21.42% | TEST WordAcc 50.78%

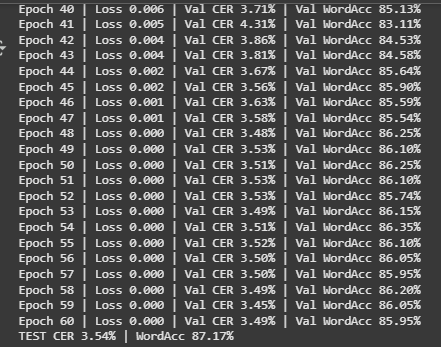
Sample preds: [('eYplOSIVE', 'EXPLOSIVE'), ('Miasmatic', 'miasmatic'), ('unofending', 'UNOFFENDING'), ('jYu', 'jYu'), ('aRjun', 'arjun'), ('Xxt', 'Xxt'), ('unHigh', 'unhigh'), ('SEDUIeTfRED', 'Sequestered'), ('PPhW', 'PPhW'), ('Rogero', 'Rogero')]

SWITCHING TO 20K NOW

Started out like this:-

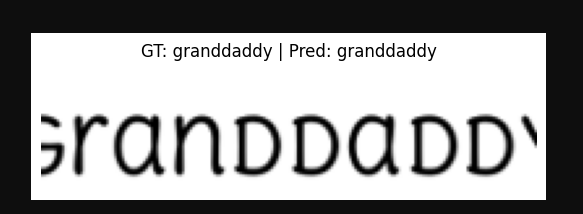


**Ended up learning well**



**CRNN can even read clipped words now  
refer: last image ‘Granddaddy’**





**METHODOLOGY**

1. Dataset Synthesis

Two primary datasets were synthesized to train and evaluate the models: a classification dataset and an Optical Character Recognition (OCR) dataset.

1.1. Classification Dataset Synthesis (cls100\_fixed\_nospacing)

This dataset was created for a 100-class classification task, where each class corresponds to a unique word.

Word Selection: 100 unique words were randomly sampled from a combination of words from the nltk.corpus.words list and randomly generated alphanumeric strings.

Image Generation: For each selected word, 50 image samples were generated, totaling 5000 images. The image size was fixed at 198x50 pixels.

Rendering Variations: To introduce variability and simulate real-world CAPTCHAs, two rendering styles were used:

Clean: Images with clear text, minimal noise, and centered alignment.

Hard: Images with added noise, random background colors, variations in font size and style (selected from a provided font directory), random case transformations (uppercase, lowercase, capitalize, toggle), text jitter, rotation, and Gaussian blur. 25% of the images were generated with the "hard" style.

Dataset Split: The generated images for each word were randomly split into training, validation, and testing sets with an approximate ratio of 70% train, 20% validation, and 10% test.

Labeling: A labels.csv file was generated, mapping each image file path to its corresponding word label.

1.2. OCR Dataset Synthesis (ocr\_20k\_fixed)

This dataset was created for an end-to-end OCR task, allowing for variable-length word recognition.

Word Selection: A pool of approximately 2500 unique words was created by mixing words from the nltk.corpus.words list (60%) and randomly generated alphanumeric strings (40%). Word lengths were constrained between 3 and 12 characters.

Image Generation: A total of 20,000 images were generated using the words from the pool. The images were rendered on a 256x64 canvas and then resized to a target size of 198x50 pixels.

Rendering Variations: Similar to the classification dataset, images were generated with "clean" and "hard" styles (25% hard ie 0.25), incorporating variations in fonts, colors, case, jitter, rotation, blur, and noise to enhance robustness.

Dataset Split: The dataset was split into training, validation, and testing sets with a ratio of approximately 80% train, 10% validation, and 10% test.

Labeling: A labels.csv file was generated, mapping each image file path to its corresponding word label.

2. Model Architecture

Two distinct neural network architectures were implemented and evaluated: a Convolutional Neural Network (CNN) for the classification task and a Convolutional Recurrent Neural Network (CRNN) for the OCR task.

2.1. CNN Architecture (for Classification)

The CNN model is designed for fixed-length input and outputs a probability distribution over the 100 predefined classes.

Architecture: The model consists of a convolutional layer followed by four ConvBlock modules and a head section.

Stem: A 2D convolutional layer with 32 output channels, followed by Batch Normalization and ReLU activation.

ConvBlock: Each block comprises two 2D convolutional layers (with 3x3 kernels, padding 1), Batch Normalization, ReLU activation, Dropout2d, and a MaxPool2d layer with a kernel size of 2x2. The number of output channels increases in successive blocks (32->64->128->192->256).

Head: An Adaptive Average Pooling layer to reduce spatial dimensions to 1x1, followed by a Flatten layer, a linear layer with 256 output features and ReLU activation, a Dropout layer, and a final linear layer with num\_classes (100) output features.

Activation: ReLU activation is used throughout

Regularization: Dropout and Dropout2d are used to prevent overfitting. Batch Normalization is applied after convolutional layers.

Initialization: Convolutional layers are initialized using Kaiming Normalization, while linear layers use Xavier Uniform Initialization for weights and zeros for biases.

2.2. CRNN Architecture (for OCR)

The CRNN model is designed to handle variable-length input images and predict a sequence of characters using Connectionist Temporal Classification (CTC).

Architecture: The model combines a CNN for feature extraction and a Recurrent Neural Network (RNN) for sequence prediction.

CNN: Consists of six convolutional layers with varying kernel sizes, strides, and padding, interspersed with ReLU activations, Batch Normalization, and Max Pooling layers. The pooling layers are designed to reduce the height significantly while preserving the width dimension to create a sequence of features.

RNN: Two layers of bidirectional LSTM follow the CNN. The input to the LSTMs is a sequence of feature vectors extracted by the CNN.

Output Layer: A linear layer maps the LSTM output to the size of the character set (including a blank character).

Loss Function: Connectionist Temporal Classification (CTC) loss is used to train the model, which allows for alignment-free training on sequences.

Decoding: Greedy decoding is used during inference to obtain the predicted character sequence from the model's output probabilities.

3. Training and Evaluation

3.1. Classification Model Training and Evaluation

Training Data: The training set of the classification dataset (cls100\_fixed\_nospacing) was used.

Loss Function: Cross-Entropy Loss with label smoothing (smoothing factor 0.1) was used.

Optimizer: AdamW optimizer with a learning rate of 3e-4 and weight decay of 1e-4 was employed.

Learning Rate Scheduler: Cosine Annealing Learning Rate Scheduler was used to adjust the learning rate during training over 40 epochs.

Gradient Clipping: Gradient clipping with a maximum norm of 5.0 was applied to prevent exploding gradients.

Evaluation Metrics: Training and validation accuracy were monitored per epoch. The model with the best validation accuracy was saved. Finally, the saved model was evaluated on the test set to report the final accuracy.

Prediction: A helper function predict\_topk\_tensor was implemented to get the top-k predicted classes and their probabilities. Visualizations of test samples with ground truth and top-5 predictions were generated using show\_grid.

3.2. CRNN Model Training and Evaluation

Training Data: The training set of the OCR dataset (ocr\_20k\_fixed) was used.

Loss Function: CTC loss with zero\_infinity=True was used. The blank index was set to 0.

Optimizer: AdamW optimizer with a learning rate of 1e-3 and weight decay of 1e-5 was employed.

Learning Rate Scheduler: OneCycleLR scheduler was used to manage the learning rate over 60 epochs.

Gradient Clipping: Gradient clipping with a maximum norm of 5.0 was applied.

Evaluation Metrics: Character Error Rate (CER) and Word Accuracy were used as evaluation metrics.

CER: Calculated using the edit distance between the predicted and ground truth sequences, normalized by the length of the ground truth.

Word Accuracy: The percentage of correctly predicted words (exact match with ground truth).

Decoding: Greedy decoding was used to obtain the predicted sequence from the model's output.

Best Model Saving: The model with the lowest validation CER was saved.

Testing: The best performing model was evaluated on the test set to report the final CER and Word Accuracy. Examples of predictions on test samples were also displayed.

Generalizability Test: A custom image of a word not present in the training data was generated and passed through the trained CRNN model to qualitatively assess its generalization capability.

4. Implementation Details

Libraries: The implementation utilizes Python libraries such as torch for model building and training, numpy for numerical operations, PIL for image processing, matplotlib for visualization, and nltk for word sourcing.

Hardware: Training was performed on a GPU (CUDA) for accelerated computation.

Reproducibility: Random seeds were set for Python's random module, numpy, and torch to ensure reproducibility of results.