BREAKING CAPTCHA

George Rahul

TASKS I DID

GENERATION OF DATASET

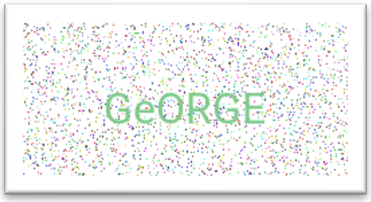
START SMALL, DREAM BIG

This is one thing I realized very late in the project that it is a good idea to start easy and then start making it more complex as you go on. I realized this a little late and this caused a lot of delay in finishing the Classification Model

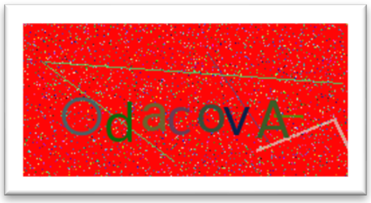
All the tasks in generating the datasets were relatively easily since they were straightforward.

I basically created a simple python script using Python and Pillow Library to generate the Easy Dataset first.

I modified the code for easy dataset to make the Hard Dataset. Here, I realized manually downloading the fonts (around 50) was not a great idea and hence, added a functionality in the script where I can just add a few font families, and it will download it for me. The rest of the functionalities was relatively the same except for adding some noise to the images



Now, for the bonus dataset I tried being a little creative. I tried adding some small jitter to the position of the letters in the image, added straight and curved lines like in real life captcha and tried mixing and matching different font families for each letter in the word and so on. The base code was derived from the Hard Dataset Generation Script. An important point to note is that as the task progressed a variety of changes in bonus dataset has been made for efficient compute and agreeable results. I also preferred darker fonts to be rendered to help with easy identification for me



LET US CLASSIFY!!

STARTING SIMPLE

So, I started by creating a very simple Neural Network that can classify between three words in the easy dataset mainly “Apple”,” Bapple” & “Bpple”. This I felt like would help me evaluate if the model can learn small letter variations as well as different word lengths. It only had two layers, one for input and one for output. I had around 100 images per word, and I first did for 5 epochs which did not work well and then I did for 100 epochs both gave very low results. But accidentally I trained it once for 10 epochs once. This gave me 100% accuracy. Investigating about this further I realized at 5 epochs; it was under fitted and for 100 epochs it was overfitted. Thus, I realized maybe the ideal epochs is 10 for 3 words

Now, thought about training with a larger set of images around 1000 per word and the accuracy reached 100% in the first epoch. This helped me down the lane too where I realized if a model normally takes a lot of epochs to train, it is better to just increase the training size instead of waiting for the model to get trained for a longer time.

One interesting thing when I trained for 100 images per word is that sometimes by 8th or 9th epoch the model got 100% accuracy while sometimes it barely crossed 50%. This led to me to believe that it is maybe because the model might not have learnt some key features to classify the images.

An optimal model I came up with had 250 image per word for train and 100 for test. I tried it for 6 epochs and the model got 100% by second epoch only.

An interesting thing I noticed is that in the first epoch, model only had 67% train accuracy in all the experiments I ran with various epochs. This led me to believe that maybe the model might be making one word wrong due to the similar looking pair of words like “Apple” and “Bpple”.

I also made the model a little complex by adding two hidden to make the model more consistent to avoid problems as discussed above.

LET US CLASSIFY!!

Thinking a little Bigger

Now, I thought I will do it for 10 words on the easy set. Here also, I tried for 250 train, 100 test per word and or 6 epochs & still got 100%. Then, I tried reducing train to 150 words and I still got 100% accuracy. Thus, I would say this is the optimal scenario for it

Let's Go Big

I thought if it works for 10 words, it should work for 100 words also and I added a random dictionary for the same to train the model. Here, I got a 1% accuracy thus me realizing that maybe the model is too simple for the huge amount of data.

I read a medium article: https://medium.com/@ageitgey/how-to-break-a-captcha-system-in-15-minutes-with-machine-learning-dbebb035a710 which helped me realize that going with a CNN architecture would be a great idea than a Simple NN.

So, I created a new CNN model and tried it with 20 words. It was very astonishing to see that it got 100% accuracy in the first epoch only. Then I tried for 100 words and got 100% accuracy in the first epoch. I did not try to optimize the CNN Model or the training parameters like before since for me this was more like a proof of concept for the next level where I thought I will optimize everything well.

Let's Go Hard

So, feeling confident I tried with 20 words from the hard dataset and got a big surprise when the maximum accuracy I got was only 54% at 14th epoch that too with a very standard 3 Layered CNN.

So, for briefly I thought of segmenting letter by letter and then predicting the word and tried some small experimentations to make a letter classification model. But realized it would be a bad idea for a classification since I believed that there was something wrong in my training process than in the architecture itself when I did a small googling.

LET US CLASSIFY!!

After tinkering a lot with the model architecture and the parameters for one or two days with no noticeable difference, I concluded that maybe the problem lies in the dataset. More narrowing down to the fact that I was using 50 random fonts without any logic. I tried directly for 100-word classification task again with a newfound intuition

Input	**Output & Analysis**
Only 4 block fonts and all Lowercase (MC1)	99.9% accuracy in 7th epoch (450 images per word for train)
Added random Uppercase (MC2)	99.73% in 3rd epoch but, had added more datapoints
30 Block Fonts (MC3)	99.85% in 6th epoch
5 Cursive Fonts (MC4)	99% in 4th epoch and 99.82% in 8th epoch but 750 images per word in training. Evaluated the model with 2x,2.5x and 0.5x and gave 97.2%,83.8% & 99.9% respectively
More cursive fonts (MC5)	96.3% accuracy in 7th epoch and 98.25% accuracy in 10th epoch. 600 images per train set
10 Block & 10 Cursive (MC6)	85.8% top accuracy for 240 images per class
All the 60 fonts (MC7)	97% accuracy in 72nd epoch

The biggest takeaway from all these experimentations was that the dataset should be uniform. Equal number of block and cursive is needed and hence, a random selection is not a good idea.

Optimize Everything

One interesting this I realized that in MC6, even though it trained in a total of only 20 fonts, it could generalize the information it learned for all the 60 fonts. So, this was one point of optimization where I decided upon the number of fonts to train

Another method I tried is to convert the image to grayscale and then train the model. I did this with a RGB model and Grayscale model with 300 images per word and grayscale model crossed 97% in epoch 25 while RGB model crossed the same threshold in 18th epoch. Both took almost the same time to train per epoch and hence not seeing any significant improvement, I decided to use the RGB model only.

I tried removing a Convo Block from the Three-Layered model but, got less than 2% accuracy in 50th epoch in test even though it had 99% accuracy in train set thus concluding that the model has lost its ability to generalize. I tried doubling the number of filters in the two layers but, training time increased very significantly to almost 18 min per epoch from 7 or 8 min of a normal model and thus concluded this is also not a great idea to optimize the model.

I tried keeping all the three layers but halving the number of filters in each. This gave me an accuracy of 98.4% in epoch 52 and the training time was very less to almost less than min per epoch thus, seemed to be a very good tradeoff **(MC 9)** and optimization point to be here. I again tried halving the filter size, the accuracy never crossed 94% and hence dropped further optimization on the model.

The “Word Classifier“ is the final convolutional neural network (CNN) designed for image-based word classification, utilizing three convolutional blocks with 3×3 filters (kernels) to extract spatial features. The first block uses 32 filters, detecting basic patterns like edges and textures; the second block increases to 64 filters, capturing more complex structures; and the third block applies 128 filters, extracting high-level features. Each convolutional layer is followed by batch normalization for stable training, ReLU activation for non-linearity, max pooling to reduce spatial dimensions, and dropout for regularization, before an adaptive pooling layer and fully connected classifier predict the word category.

THE GENERATOR

SOME RANDOM EXPERIMENTS

On checking the internet, I realized that the state-of-the-art approach would be combined CNN and bi-LSTM model. I thought to start with this approach since I believed that once I optimize the model well, and have a working prototype, I can try with other models and see why they might be or might not be a good idea.

Stage I

I tried with my non-optimal CNN model and then fitted a bi-Directional LSTM to sequentially predict the features that will be extracted by the CNN model. I used around 20,000 randomly generated images with very limited block letter fonts to do the first trail. It worked well as by 100th epoch I got around 100% accuracy. I just thought of putting cursive fonts in test set and prayed that it will work but, it came up with an accuracy of 0% as expected. This is **GM1**

Stage II

Here, for the same model and parameters as above I tried 20 block fonts and 20 cursive fonts. After 70 epochs, I tried testing it. Here, I noticed that it works great with block letters very well but sometimes gets confused between O and 0 and so on. Also, it was performing a little less than expected with cursive letters

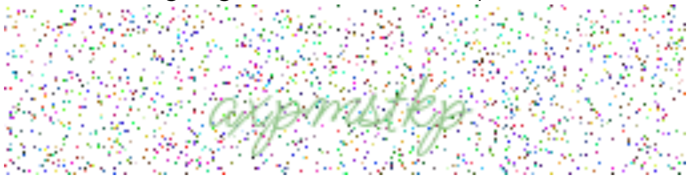
Just to get a very brief idea on how exactly it might look if I brute force the problem, I trained the model with all the 60 fonts using 40,000 images but, as expected the accuracy was less than 50%. **GM2**

Stage III

I realized that my bottleneck was proving to be cursive letters. So, I did a couple of tests where I used 5 cursive fonts to predict 20 other cursive fonts and so on but, none of them gave any meaningful results. So, I thought of just using one cursive font and train it with 20,000 lowercase words to get a very meaningful baseline. I achieved around 99% accuracy in 200 epochs. So, this was my baseline that I had to achieve.

THE GENERATOR

Now, I put around 5000 images of 5 cursive fonts and by 400 epoch, I got around 70% epoch which I was not satisfied with. When I manually tried attempting the test I realized that the images generated were very hard to read even for me. Thus, as before I again



had to tone down the complexity of my dataset. Fonts which were thin, a lot of random noise generated and the color where the main reasons I felt that it was not working perfectly.

Stage IV

Reading this article, I felt that there was a scope to optimize the model: https://medium.com/towards-data-science/a-basic-introduction-to-separable-convolutions-b99ec3102728

By, using a combination of the Separable Convolution Layers described and toning down the dataset a little, I was able to get 97% accuracy in 130th epoch using 10,000 train images. The average epoch time to train the model around 1 min. I also had reduced the filters in each convo layer by half and kept only two bi-LSTM layers in the model to optimize it further. **GM4**

Stage V

Now, was the time to test out the alternative: RNN. So, instead of the LSTM layer, I swapped it with RNN layers and then used the same parameter as GM4. It took almost twice the time per epoch and barely crossed 91% in the 100th epoch. Seeing this dismissal performance, I realized that the GM4 approach was better. Nevertheless, I still named it **GM5**.

MISSION: BONUS ACCOMPLISHED

TAKE 1

On Seeing the task description, the first thing came to my mind was, why not use keep the same GM4 model but, then also create a separate model to see if the image has a red or a green background. On further googling about this idea, I realized that I could combine them to a single model, and I do not have to create and integrate two separate models for it. So, basically, I just must create an extra layer from the input layer that will binary classify if it is red or not, I decided to give cursive some rest since I thought to start simple from my previous experience.

I used the optimal LSTM model that I had made before and then fitted the classification layer on it but, the max accuracy was only 10%. Here, I had two options on why it did not work. Either the model or the dataset. I chose the latter.

So, I trained the model on Green and Red Images separately and got accuracy of 82% and 8% respectively after 100 epochs. So, I realized that the dataset is fine and that somewhere the model was at fault. On a careful deep-dive I realized that I had assumed that the binary classification layer will learn the difference between Green and Red on its own from the loss generated by the final output. I did not realize that for the Binary Classification, I might have to train both the aspects of the model

TAKE 2

So, here I labeled the dataset to include the color of the background as well and then also included another Binary Entropy Loss to train the Binary Classifier along with the LSTM. The binary classifier got an accuracy of 100% by first epoch and got around 82% accuracy by 300th epoch **(BonusGenerationModel5.py)**

TAKE 2.1

I played a little with the Binary Classification part where I used a simple Neural Network as well as a very trimmed down CNN model using only one layer. Both had the same performance of 100% in the first epoch only. I went with the CNN approach since it seemed more scalable in the future. I tried running a few tests and concluded that green images perform better than the Red Images (78% vs 73.4%) this is probably since Red has the disadvantage of learning the reversed word while Green doesn't have that.

MISSION: BONUS ACCOMPLISHED

EXPECT THE UNEXPECTED

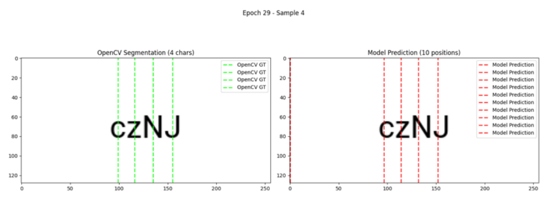
Just out of curiosity I tried with 1 bi-LSTM layer instead of 2. Here, what surprised me is that the model had a 93% accuracy in just 100 epochs. On training it with 20,000 images, it reached a max of 94% while the previous model barely crossed 90% after 175 epochs. Thus, I realized that a single LSTM probably reduces overfitting and thus give better results **(BonusGeneration5.py)**

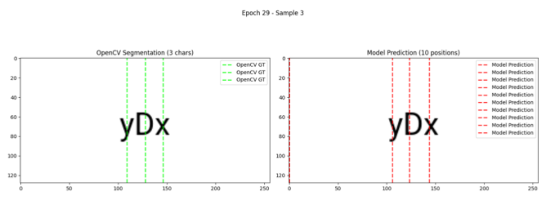
Motivated by this I thought of using a unidirectional LSTM to see for further optimization possibilities but, this one did not work at all and even did not generate anything after 100 epochs and hence I decided not to explore further

LAST BUT, NOT THE LEAST

Why so overkill?

On reading papers like https://medium.com/@ageitgey/how-to-break-a-captcha-system-in-15-minutes-with-machine-learning-dbebb035a710 and https://ceur-ws.org/Vol-1885/93.pdf this, I realized that what if segmenting the letters in a word and then predicting them might also be a very good approach instead of using an LSTM. But the basic idea on how to train the model to segment it was something that eluded me for a while. I knew that OpenCV was a great option for it but, I wanted a NN to do this. So, what I finally came up was to process the images in OpenCV and then try to teach the model where the OpenCV predicted the center of the letter is to be. The model was the optimal CNN that I had discussed earlier, and it predicts 10 values since I had assumed the max length of the captcha to be 10 and tried running the model. I used the easy set since I wanted to do baby steps. I tried with around 1000 images per word-length between 3 & 10 to teach it how to be variable. One idea that I had come up with to teach variable length is to put -1 instead of asking it to predict the model a random value when the word length is less than 10 since, OpenCV does not do it by default and segment based on the number of letters only. I got a top accuracy of 6000 images to get an accuracy of 65% which seemed reasonable since the model need not predict the exact center of the word but, more like an estimate. On manually inspecting it, I realized that the model not only learned the center of the letters but, also the number of letters too since the number of extra segmentation lines drastically reduced since it was trained to output them as -1. **SEG1**





The next part then I thought was to create a letter model to recognize the individual letters. I tried the same approach as in classifying the hard dataset and got a model that was 100% accurate in 70 epoch. **WM1**

Parallelly, I also tried to create another model that can predict the number of letters in a word using the same CNN architecture but, that one did not go well at all. Initially I used a linear regression model at the output layer but, since the number of letters where whole numbers and not continuous, the accuracy of the model was bad. I also tried with a classification layer and got around 91% accuracy but, since the above approach was doing not only this but, also predicting the letter position, I dropped this line of thought.

REAL LIFE EXPECTATIONS

Just to see how well this approach will scale in the real life, I downloaded a random Kaggle dataset for captcha: https://www.kaggle.com/datasets/fournierp/captcha-version-2-images. Initially I thought of using my pretrained model but, it came out to be very bad at predicting this. The main guess I would say is because the dimensions as well as the representation of the image is very different from the toy dataset that I had used. But, to test the architecture, I trained the optimal LSTM model and achieved around 97% accuracy within 160 epochs. Thus, I feel that the model is very fit for real-life scenarios also.