Can you break the CAPTCHA?

Sudheera Y S
sudheera.yelimeli@research.iiit.ac.in

1 Introduction

This project involves training and building a neural network model from scratch for breaking Captchas. We also had to build the model on our custom dataset. I have used the inbuilt functions in the PyTorch library to build the model from scratch. The challenge is to build a model which can handle the variation in fonts and noise in the captcha image to extract the multichar text in the image. I referred to multiple resources to get through this, like Youtube tutorials, online articles and online courses, which I have linked at the end of this report. As part of the task, we are also asked to make a short video explaining the research paper by OpenAI titled "Learning Transferable Visual Models From Natural Language Supervision" popularly called CLIP.

2 Task 0 - Data Generation

Here is how my dataset looks like -

Figure 1: Easy Set - Uniform Capitalization, Fixed Font and No Noise

Figure 2: Hard Set - Random Capitalization, Font 1 and Background Noise

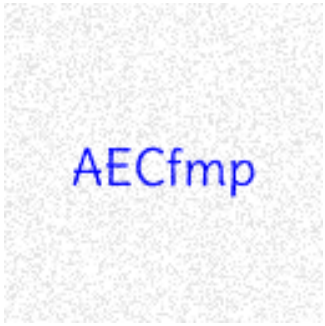

Figure 3: Hard Set - Random Capitalization, Font 2 and Background Noise

and for the bonus task, here are the sample captchas -

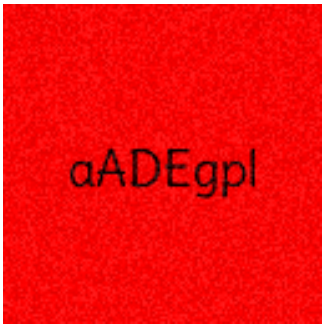

Figure 4: Red Background with Noise

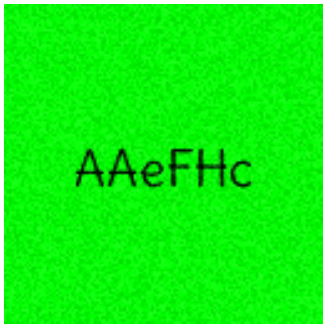

Figure 5: Green Background with Noise

2.1 Dataset for Classification

Generated 100 random words, which will be fixed and will be used as the 100 classes. For each class, generated 1 easy image (1 image since there can be no more variation in it), 20 Hard images with Font 1, and 20 Hard images with Font 2 (Each with it's own noise and random capitalization). Each class had 41 words. To ensure that each captcha class has the same no. of images in the testset, I generated a different test set. For each of the captchas, I generated 5 Hard Images with font 1, and 5 Hard Images with Font 2.

To create the data and labels on the Image dataset, I am using the ImageFolder function in the torchvision library which expects a following structure for the dataset - Each label should be a folder and the data for a particular label should be inside the corresponding folder. Created the dataset following the same folder structure.

Found fonts which look similar to the captcha images given in the task document. For the easy set, used Roboto Font, used Rubik-Iso Font for Font 1 in Hard Set and Delius Font for Font 2 in Hard Set.

2.2 Dataset for Text Generation

Generated a dataset containing 50k images. 25k images with the first font and the rest of the 25k with second font. Unlike the last time, we don't need a separate test dataset since we can partition this dataset into train and test, and maintain uniformity for all characters. I made sure that all 52 (26 upper and 26 lower) appear and their frequency are somewhat balanced.

Since this is not a classification task, we have to follow a different folder structure. Each captcha image has it's own label and it doesn't make sense to have multiple captchas for the same label since the goal of the "Text Generation" task is to teach the model to accurately extract characters from the image.

For storing the (image, captcha) pair in the dataset, I saved each image in the dataset with the filename - captcha.png. For example, if the captcha is AbCdE, I stored the corresponding image in the dataset as AbCdE.png . This helps in extracting the label so as to train and test the model.

Initially, I made an assumption that each captcha has constant no. of characters so as to simplify the model. Later, when I relaxed this constraint, I generated a different dataset with each captcha having a random no. of characters ranging between 5 and 7 characters. Hence, the two datasets, one with fixed no. (5) of characters and the other one with varying no. of characters.

2.3 Dataset for Bonus Task

Generated 50k images in the dataset. Divided the dataset into 4 sets of 12.5k images each, with each set having a one of the combinations out of (Red, Green) background colors and (Font1, Font2) fonts.

Very similar to the previous dataset generation, I just had to change the background color of the image accordingly. But the difference between this dataset and previous dataset was the filename. In this dataset, I also have to mark the background color in the filename in order for me to train the model and test it.

The captchas with red background color were named captcha1.png while the captchas with green background were named captcha2.png . For example, the capthca "AbCdE" with red background was named AbCdE1.png while the green background was named AbCdE2.png . To note here that, even though the captcha in red background would read in the reverse direction, the file was named in the direction the characters are displayed in the captcha.

3 Task 1 - Image Classification

3.1 The Model

The input image to the model is a 128 x 128 RGB image. In the transform function, I first convert the image into a grayscale image, Resize it to 128 x 128 (Even though the size doesn't change, it normalizes the pixel values between 0 and 1) and convert the datatype into a tensor datatype. Now, the dimension of the image is [1,128,128] since there is just one channel now.

My test dataset has 10 images per class, hence 1000 images in total. For the training dataset, even though the dataset has 41 images per class, I only take a subset of them and calculate the optimal training size to reach a reasonable accuracy. The ImageFolder function creates the (features, labels) pairs for a dataset, given that the dataset folder follows the folder structure it expects. I'm using a batch_size of 32 images.

My CNN model has 2 convolution layers and 2 fully connected layers. The input dimension for the first convolution layer is [32,1,128,128] where 32 is the batch size, 1 being the no. of input channels (since it's an grayscale image, just 1 channel) and the dimensions of the image being 128 x 128. In the first layer, there are 32 filters (with padding), and then max pooling is done which shrinks the image dimension by half. Since there are 32 filters (which can be thought of as features), there are 32 output neurons, each representing one feature. The output dimensions from this layer is [32,32,64,64], the channels increasing to 32 and dimensions of the image scaling down by a factor of 2.

In the second convolutional layer, it takes the 32 feature outputs from the previous layer, applies convolution (with padding) with 64 filters. Similar to the previous layer, there is an activation function applied which non-linearises the values and then a batch normalization which normalizes each of the filters which help in stabilizing the model. Since there are 64 filters applied, the output dimensions are [32,64,32,32] since the dimensions were halved again after the maxpooling with stride = 2 (Batch size remains the same and filters become 64). Since there are 64 filters (features), there are 64 neurons in the final convolution layer.

Now, we send these feature predictions to the fully connected layer. Flattening each image results into a 64 (no. of filters) x 32 (height) x 32 (width) sized array. This is the no. of neurons in the input to the first hidden layer. This has 128 outputs which goes to the second layer. The second layer has 128 inputs from the previous layer and gives out 100 outputs which is equal to the no. of classes.

There are 100 predefined classes, hence the same no. of output neurons from the model. The model will output the logits for each class. This is the weight calculated by the model for each class for a given image. These are raw logits and doesn't correspond to a valid probability distribution. To convert these raw logits into a valid probability distribution, we can apply softmax function.

We don't need to explicitly provide the kernel values to extract the features. The model learns the kernel values by itself using the loss values. For example, in the case of a face recognition system, the first layer might extract the edges, and in the next layer, it will combine the edges and try to extract face parts like nose, ear, etc. We won't know what exact features the model is trying to extract, but the idea remains the same for any CNN model.

Since there are 100 classes, we will be using Cross Entropy as our loss function (which also handles the raw logits from the model). I've used learning rate of 0.01 as a higher learning rate was not converging the model. Any no. of epochs work, but 100 felt reasonable and was good enough to see a great accuracy. For updating the weights, I'm using the SGD (Stochastic Gradient Descent) optimizer.

Data	5 per class	10 per class	20 per class
Test	47.8%	86%	96%
Train	99.6%	100%	100%

Table 1: Overfitting Result

Data	5 per class	10 per class	20 per class
Test	21%	79%	95.8%
Train	47%	84%	99.9%

Table 2: Final Result

3.2 Results

Running the model, I found that the model was overfitting. The results without accounting for overfitting are tabulated in Table 1.

Here, we can see that the model is performing extremely well on the training data even with 5 images per class, but performs very poor on the testing data. Similarly, the gap between the accuracy is very high when the no. of datapoints per class were increased. An accuracy of 100% on training data implies that the model is not really learning the data but memorizing the training data, hence the poor performance on a new unseen data like the test data. This is called "Overfitting".

To fix this issue, I incorporated two mechanisms - Weight decay and Dropout. Weight decay is the constant factor in L2 Regularization. It penalizes very large values of weights, ensuring that the model doesn't overfit. The new loss is function is calculated as below -

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \frac{\lambda}{2} \sum_i \|w_i\|^2$$

where the lambda denotes the constant value of weight decay. By adding the penalising term in the new loss function, when we consider the gradient of the total loss function for updating the weights (using SGD Optimizer), we can see that the new term penalizes heavy weights -

$$w_i^{(t+1)} = w_i^{(t)} - \eta \left(\frac{\partial \mathcal{L}_{\text{data}}}{\partial w_i} + \lambda w_i \right)$$

where the "eta" is the learning rate. The new weights are updated, hence adding the L2 regularization term reduces the weights, hence preventing the model from overfitting. The formula for updating the weights in SGD is given by -

$$w^{(t+1)} = w^{(t)} - \eta \nabla \mathcal{L}$$

Taking the gradient of the combined loss function, we get

$$\nabla \mathcal{L}_{\text{total}} = \nabla \mathcal{L}_{\text{data}} + \frac{\lambda}{2} \nabla \sum_i \|w_i\|^2$$

$$= \nabla \mathcal{L}_{\text{data}} + \lambda w$$

Dropout is another technique which disables certain neurons in the model. Given a probability, it disables each neuron with that probability. It is found that the dropout helps in overfitting. This has been proven to improvise I've used dropout in both the hidden layers with a probability of 0.5 and weight decay value of 0.0001.

Now, incorporating these two techniques to solve overfitting, the results are tabulated in Table 2.

We know that weights are adjusted by backward propogation after every epoch so that the loss value decreases over epoch. The variation of average loss per image with increase in epochs is visualized in Figure 6.

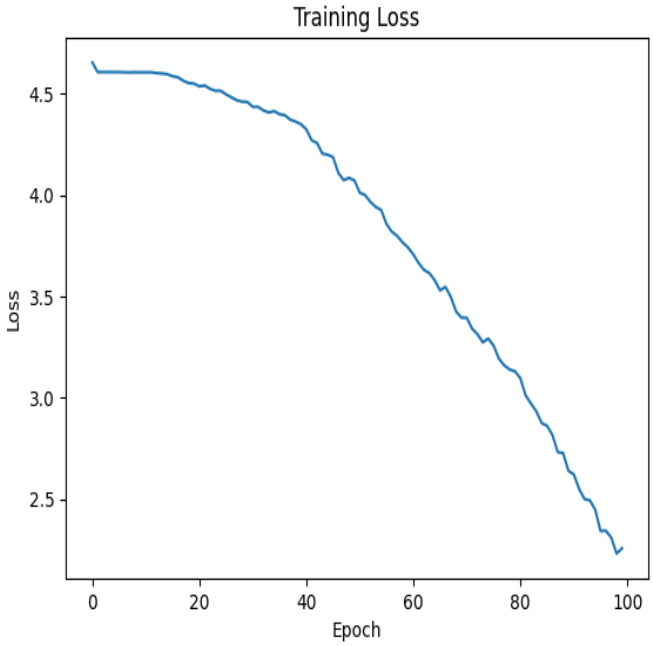

Figure 6: Average loss per image over epochs

From the Table 2, we see that 20 images per class in the training dataset is good enough to obtain a highly accurate model. Even though there are 41 classes per image in the original dataset, using a subset of this dataset containing 20 images per class is more than enough. Using a bigger dataset will slwo down the training and testing phase albeit the increase in accuracy.

3.3 Challenges Faced

- Loss function not decreaasing - Even for 100 epochs, the loss was almost a constant value which means that the model is not exactly able to learn. To fix this, I lowered the learning rate which made the loss converge to 0.
- Once the convergence issue was fixed, I figured out that the model was overfitting. To fix this, I implemented L2 regularization and Dropout which reduced the gap between train accuracy and test accuracy. Also after this, the train accuracy was never 100% which indicated that the model was not memorizing, instead was learning.
- Getting Started - To get started with the classification task, before jumping straight into the captcha classification, first I implemented a CNN on a dataset of playing cards found online on Kaggle. I was able to classify the test images into one of the 53 playing card classes with an accuracy of around 92%.
- Reproducibility - Since I was using a random subset of the dataset to train my model, there is an issue of reproducibility. Every time I run the model, I am not guaranteed to obtain the same results. I tried to fix this issue by manually setting the random seed, but it didn't help.

4 Task 2 - Text Generation

4.1 Attempt 1 - with an assumption

To solve the text generation task, first the major difference was that now, the model is doing a multiclass classification not for 100 classes but with 52 (26 uppercase and 26 lowercase) character classes but for 5 characters. To simplify things, first I assumed that there are fixed no. of characters in the captcha and built a CNN model which can extract the text from any given image. For the labels, I've used one hot encoding of size 5 x 52, since there are 5 characters belonging to one of the 52 classes.

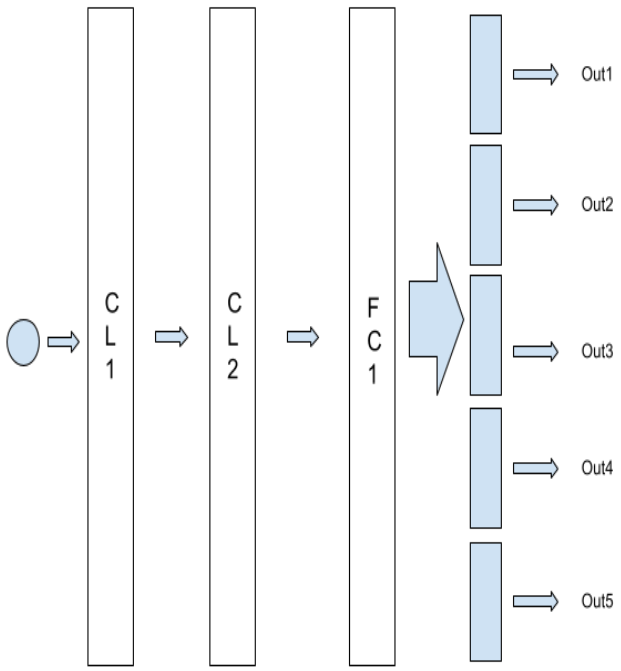

Figure 7: CNN model for 5 character output

After the first layer in the fully connected layer, there are 5 different output layers each trying to classify the corresponding character into one of the 52 classes. Each of the 5 output layers are independent layers and are taking the input from the first fc layer and classifying independently.

This works really good with captcha's having exactly 5 characters. The hyperparameters used are - learning rate = 0.001, weight decay = 0.0001, epochs = 50. With a higher value of learning rate(0.01), the losses were not converging to 0, hence the lower value of learning rate. I've used the same value of weight decay as in the previous model. Since there are around 20k images to train on, the model takes a long time to train (Even running it locally on a powerful GPU), hence just 50 epochs. The results are tabulated in Table 3.

Data	10k set	20k set
Test	Captcha-8% and Character-59%	Captcha-71% and Character-92%
Train	Captcha-23% and Character-74%	Captcha-82% and Character-95%

Table 3: Results for 5-character captcha

I also tried different classes for different fonts. For example, the character "a" of font 1 can be considered a separate class compared to the same character in the second font. I hypothesized that, since the same character is rendered differently in different fonts, the model will have a tough time distinguishing between the same characters of different fonts. But the results weren't that distinctive with very similar accuracies, so decided not to complicate the model with 104 classes (52 characters for both the fonts).

4.2 Attempt 2 - Multichar Text Extraction

Now that there is no constraint on the no. of characters in the captcha, we need to use a CRNN (Convolution Recurrent Neural Network). The dataset I am using has atleast 4 and atmost 7 characters in the captcha. For the labels, similar to the basic generation model, they are stored as one hot encoding vectors, and padded with 0s to make the size of the vector 12 x 52.

Now, coming to the model, it has a similar convolution layer as the previous model. The output dimension from the convolution layer is of the size [batch_size, 64, 32, 32] where 64 are the no. of filters, and the image dimensions are now 32 x 32. There is a average pooling done with a stride so that the final dimension of the image is now 32 x 12. Since the "time step" is in the x axis direction (runs through the width of the image), the width of the image is the time step and hence the width is now reduced to 12.

Since now the width is the "time step" in the sequence, we need to permute the columns (so that the architecture knows that weight is the time step) and also combine the rest values into "feature vectors", which is of size height x filters.

We now have a recurrent layer which has 128 neurons. It takes the first timem stamp with a default activation, and then produces an output which is again fed to the same layer, but now with a different activation vector and 2nd timestamp values.

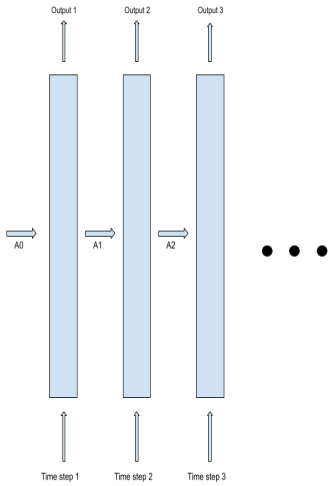

Figure 8: CRNN model for multi-character output

Here, in the Figure 8, we see that the recurrent layer is taking input from the time step, calculating the activation values and sends the 128 values to itself. The no. of input values to the layer is 64 x 32 values (combining filters and height as feature vector), and the layer contains 128 neurons. Now these 128 values are the output from that particular time stamp, and is fed to the same 128 neurons recursively 12 times.

These 128 sized output for each timestamp is now sent to a linear layer with 53 classes (52 characters and 1 blank) which classifies the input to one of the 53 classes. We see that here, each timestep is being classified into one of the 53 classes, helping us with support for multichar extraction.

The Loss function (CTC Loss), takes softmax values, so the model should return softmax values of the output from the linear layer for each timestep. The CTC Loss should also know the blank value, which is the class 0, hence blank = 0 (1 based indexing for the rest of the 52 alphabets). Similar to the previous model, I'm using the same learning rate, weight decay, and epoch values.

The results are tabulated in the Table 4. We also need to clean the predictions, as there might be blanks in between two characters which wont be present in the label. Also, I observed that the same character was being duplicated two times by the model, hence I combined every two consecutive identical values into one. This improved the captcha accuracy and the character accuracy drastically, since the model was predicting all the characters correctly in sequence, but was predicting the same character multiple times.

Data	10k set	20k set
Test	Captcha-42% and Character-65%	Captcha-74% and Character-86%
Train	Captcha-57% and Character-74%	Captcha-87% and Character-93%

Table 4: Results for multi-character captcha

5 Bonus Task

Using the model from the Text Generation model to extract the multi-char text from the images. Additionally, we will have a background color image binary classifier after the convolution layer which will learn to classify between red and green backgrounds given the features.

The model is visualized in the Figure 9. Here, since the input will be an RGB image, there will be 3 channels in the input. Sending them to the common convolution layer, we get the features extracted. We can directly send this to our previous multi-char extraction model which will do it's job.

Now, for the background classifier, we flatten the feature output from the convolution layer, and pass it to the linear layer which is a binary classifier. Based on the features, it will learn to classify between red and green backgrounds.

Since there are two different models with different loss functions, we need a way to combine the two losses. For the Recurrent layer extracting the text, we are using CTC Loss, and for the binary classifier, we are using Cross Entropy Loss. The total loss can be calculated as follows -

$$\mathcal{L}_{\text{total}} = \lambda_1 \cdot \mathcal{L}_{\text{char}} + \lambda_2 \cdot \mathcal{L}_{\text{bg}}$$

Where lamda 1 and 2 are constant values which control the importance of the two losses. By SGD, we will try to minimize both the loss values, but more importance is given to the higher lamda value. Since we have a very big dataset for background colors, and the text extraction is more important than the background color classifier, I set the second lamda value to be 0.1 while the first one to be 1. Even though there weren't any visible changes in the accuracy with different values for these constants, since our training dataset is quite huge.

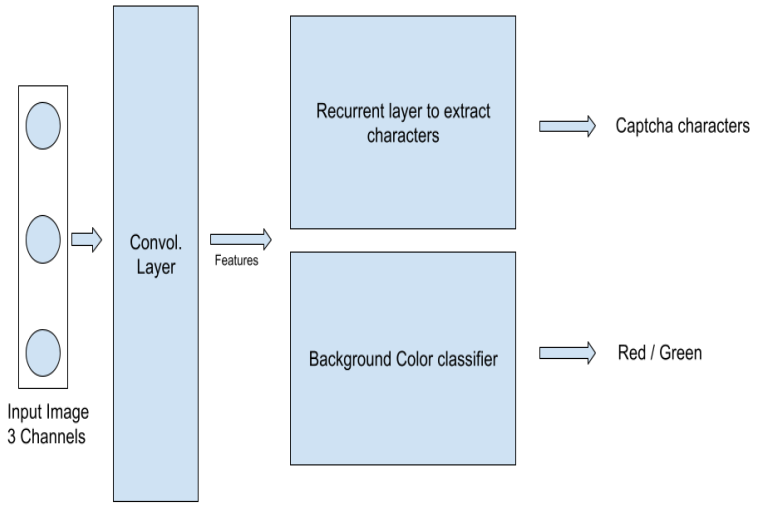

Figure 9: CNN model for 5 character output

Here are the results for the Bonus Task, with each of them having 100% accuracy on Background Color Classifier -

Data	10k set	20k set
Test	Captcha-76% and Character-86%	Captcha-79% and Character-88%
Train	Captcha-86% and Character-92%	Captcha-88% and Character-93%

Table 5: Results for captcha detection with color classifier

We are post processing the extracted characters by looking at the background color classifier. Since the background color classifier is 100% accurate, we can just rely on it to reverse the string. We are not training the model to read the reverse string, but to extract the characters in similar fashion, and reverse based on the color.

6 Conclusion

I built a model from scratch with PyTorch library, for Image Classification, Multichar Text Extraction, Multi-class Model using CNN and CRNN having a high accuracy of 78-93% accross tasks.

7 Resources

- Youtube Playlist 1
- Youtube Playlist 2
- Coursera Course