Final Project 4390

Heinrich Rausch

May 2024

Contents

1 Introduction

The idea behind this project was to create a model that can solve a CAPTCHA problem for the user. The idea came to me when I was solving an image CAPTCHA and realized that it's quite literally just an image classification problem, something that deep learning is used for quite often. Initially, I had decided to try and solve two different CAPTCHA types, however, once I got to the testing phase, I realized that there were just not enough images out there (that I could find) for me to test on the of types satisfactorily, thus I changed gears and decided to focus on only one type of CAPTCHA but instead look into how differing levels of generalization would affect the model's ability to solve a given CAPTCHA. I ended up training 2 models on the same architecture with one being trained for four categories and one being trained for ten categories. While there are certainly many other models out there that can solve CAPTCHA problems (as counter-intuitive as that is), I still feel that this was a worthwhile project as it focuses less on solving the CAPTCHA accurately and

instead more on how to solve it most efficiently. Furthermore, I also learned some things from this project that have changed how I look at training neural networks and general and that will aid me in further training. As will be repeated later on, but bears mentioning, the largest hindrance in this assignment was time. I feel like I could've made significantly better models were I to have had more time.

2 Topic problem

The original objective for this project was to create a model that can solve 2 of the different CAPTCHA types: identifying what images contain a specific item, and identifying what objects don't belong with the rest in an image. Some example images of the two categories are below:

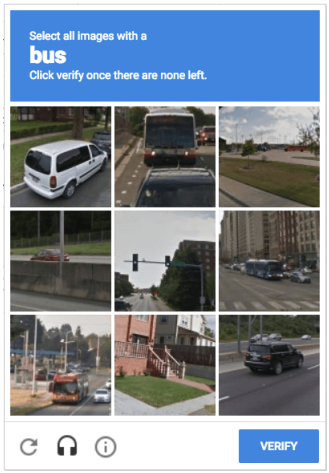

Figure 1: Select all images with X

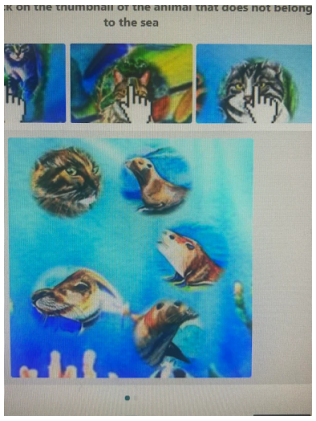

Figure 2: Which image does not belong

However, once I started gathering data for the latter category, I found that there was quite frankly just not enough images out there for me to test it satisfactorily. I spent several hours searching for more examples of the latter category but in all my searching I could only find one source that contained three images; far from enough to do sufficient testing. I thus decided to switch my objective from solving two different categories to solving one category as efficiently as possible.

3 The Project

The plan x-
My plan for the project was to split the work up into four different phases:

- Phase 1: Data Gathering
- Phase 2: Model Creation
- Phase 3: Test Image Interpreter
- Phase 4: Testing

3.1 Phase 1: Data gathering

This phase was by far the simplest; all it was was the gathering of data for the model and it thus was also the phase that took the least amount of time to do. I ended up using three different datasets from Kaggle which I then combined into one, larger, dataset. Those datasets can be found below:

- hCaptcha Dataset
  (https://www.kaggle.com/datasets/aneeshtickoo/hcaptcha-dataset)

- Caltech101 — Stop Sign — Images & Annotations
  (https://www.kaggle.com/datasets/maricinnamon/caltech101-stop-sign-images-annotations)
- Road Sign Detection

The number of categories from those datasets is greater than 10, however I felt 10 categories to be a good number for my larger model as given that the CAPTCHA format I was testing was that of images in a 3x3 grid, 10 categories would allow for all but one category to be present in any given test, something that I felt would be useful in testing. The final categories for the two models are as follows:

4-Category model

- Airplanes
- Stop Signs
- Traffic Lights
- Trucks

10-Category model

- Airplanes
- Bicycles
- Boats
- Motorbus
- Motorcycle
- Seaplane
- Stop Signs
- Traffic Lights
- Trains
- Trucks

3.2 Phase 2: Model Creation

This phase was the most time-consuming of all the phases as it required me to train several models, each of which took some time to train. I finally arrived at the following architecture (with numCat corresponding to the number of categories):

```
input_img = Input(shape=(64,64,3))
output1 = Conv2D(16, (3, 3), activation='relu')(input_img)
for i in range(10):
    output2 = Conv2D(16, (3, 3), padding = 'same', activation='relu')(output1)
    output2 = BatchNormalization()(output2)
    output2 = Dropout(0.3)(output2)
    output2 = add([output1, output2])
    output1 = output2
output3 = Flatten()(output2)
output4 = Dense(units = 16, activation = 'relu')(output3)
output5 = Dense(units = 16, activation = 'relu')(output4)
output6a = Dense(units = 16, activation = 'relu')(output5)
output6 = Dense(units = numCat, activation = 'softmax')(output6a)
model = Model(inputs=input_img, outputs=output6)
```

The end results of the training can be seen in the following table:

Model	Training Time	Validation Accuracy	Epochs
4 Categories	1826 Seconds	91.77%	369
10 Categories	3795 Seconds	72.57%	337

Table 1: Results from model training

The model for four categories had a higher training and validation accuracy, but I expected that as the fact that it had over half as many categories as the ten-category model gave it a clear advantage. The learning curves for the two models are below:

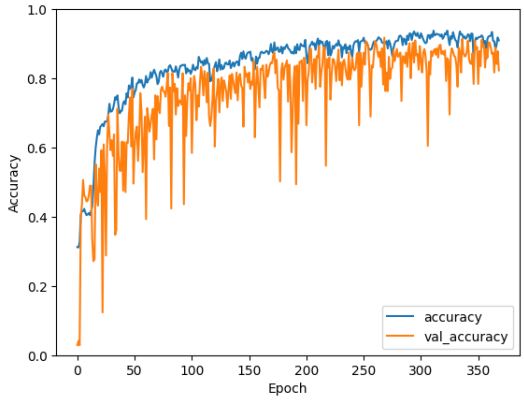

Figure 3: Learning curve for four category model

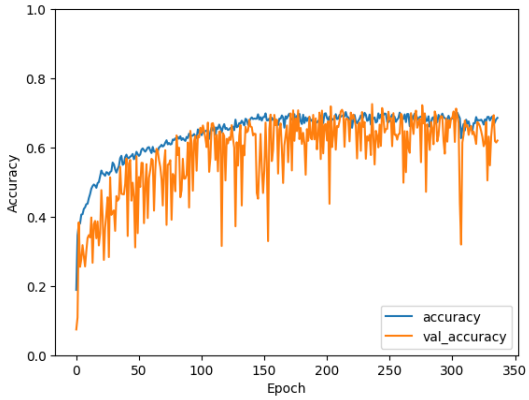

Figure 4: Learning curve for ten category model

Something interesting about the results is that the two models took around the same number of epochs to train, despite the fact that the ten-category model took over twice the time to train as the four-category model. On top of this, the two have very similar learning curves. Despite all that, the four-category model was significantly more accurate than the ten-category model.

3.3 Phase 3: Test Image Interpreter

3.3.1 Making the Interpreter

This phase was the hardest out of all the others, as it involved creating an interpreter for test images. What I mean by that is that I need to automatically take a provided CAPTCHA image a split it up into its component nine images, and while this at first seemed simple, as all that involved was slicing the image

into a grid and then taking the images from there, which wasn't hard at all, the hard part came when I realized that I couldn't expect an input image to be cropped nicely. What I mean by that is that splitting the input image into a grid and then using that grid to get the component images only really works if the image is already nicely cropped so that the only things in the image are the component images themselves, as seen below:

Figure 5: Example pre-cropped image

While the interpreter did work at this point, testing would've required me to go through and hand-crop each CAPTCHA image I found for testing, which would've taken quite some time. I thus decided to make the interpreter (try to) crop out any bordering white space from the provided image so that all that would need to be done is ensure the only thing around the image was white space, so there would be no need to worry about cropping it perfectly. This took quite some time as I had trouble figuring out how to identify the border of the image in order to crop it out. I initially tried to just get rid of all lines of white pixels that were on the edges of the image, but that didn't work due to the varying colors of the pixels requiring me to give quite a lot of leeway for what color counted as "white", which ended up deleting some of the images, which defeated the purpose. I eventually came across the cv2.Canny function, which saved me as it allowed me to use edge detective to get the borders of the main grid which I then used to crop the image down.

3.3.2 Making a Test Image Generator

Once the cropping code was done, I then started trying to find some test CAPTCHA images, however finding and then cropping the images, no mat-

ter how bad the cropping was, still took some time. I thus decided that instead of finding specifically CAPTCHA images to then crop down to a 3x3 grid of images, I could instead create code that would just generate a 3x3 image for me. This was fairly easy, as the "Load and Preprocess images" tutorial on the Tensorflow website (Found Here) contains code to print out a 3x3 grid of images from your dataset, so all I had to do was take that code and use it to print out an image from my test dataset which I then save to put through the image interpreter before using the resulting images as a test. While that does essentially return the images to how they were eventually, there is some small changes in the sizes of the various images due to the low-quality cropping as well as the generated grid of images not being perfectly spaced out. An example test image made by the generator is below:

Figure 6: Example generated test image

3.4 Phase 4

In order to test the two models, I decided that I would instead run each of them while using my test image interpreter and generator and see how many test CAPTCHAs they could solve successfully. While this method isn't as accurate as evaluating the entire test set, this one is more in line with the idea behind the project; it simulates a CAPTCHA question instead of just getting the overall accuracy of the model. I had thought about testing the two on each other's test sets, however, upon further thought, I realized that that wouldn't work very well as the four-category model would never be able to beat the ten-cateogry model on the 10-category dataset, as it just wouldn't be able to classify over half the images.

Model	Test Accuracy	Test Accuracy (CAPTCHA Simulation)
4 Categories	87.77%	32/50
10 Categories	72.69%	38/50

Table 2: Results From Testing

Below is an example of the final output of the ten-category model:

Figure 7: Example Output

Something interesting about the results of my testing was that the ten-category model was actually *more* accurate than the four-category model in terms of solving CAPTCHAs. Going in to testing, I expected the four-category model to be more accurate as it was more accurate on the test set, and given that I was using the test sets to generate my test CAPTCHAs, I figured that that accuracy would carry over. That was not the case, however, as the ten-category model was overall more accurate. Upon some thinking, I realized that the larger size of the ten-category model actually worked to its advantage in this situation. When evaluating the test set, the four-category model got an accuracy of around 88%, meaning that 12% was wrong. The ten-category model was less accurate and got an accuracy of about 73%, meaning that about 27% was wrong. However, when you take the different category sizes into account, 12% wrong in four categories means that the four-category model got 3% wrong per category, while the ten-category model only got 2.7% wrong. Thus, as its inaccuracy was spread out over a larger number of categories, the ten-category

model had a higher chance of getting all the classifications correct as it had a lower chance of actually running into an image that it incorrectly classifies.

4 Conclusion

Overall, this project was very interesting to do and I learned a lot while doing it. I am a bit displeased with the fact that I was unable to do the project I had originally intended to do, however, I still had some fun doing this one. If I had more time to work on it, I feel like I could've made a generator for the other type of CAPTCHA, as it's not conceptually too difficult (just randomly place images into a larger image), but I don't even know where to start on extracting those images, thus I think it is for the best that I ended up doing what I did instead. What I am really happy about is learning about how misleading the accuracy score can be. Just because a model is more accurate in general than another doesn't mean it'll be accurate in more circumstances. While I know that the idea that inaccuracy is 'spread out' between the different categories isn't entirely accurate (a model could have 100% accuracy in one category and 50% in another totaling up to 75% accuracy), I still feel like it has some merit. There is clearly a difference between a model with 90% accuracy over 4 categories and 90% accuracy over 10, a difference which favors the latter. Just that alone makes this project more than worth it as it's resulted in me having a shift in how I think about the accuracy on my models; I have to take the number of categories into account when choosing a target accuracy.

5 References

- hCaptcha Dataset(https://www.kaggle.com/datasets/aneeshtickoo/hcaptcha-dataset
- Caltech101 — Stop Sign — Images & Annotations(https://www.kaggle.com/datasets/maricinnamon/caltech101-stop-sign-images-annotations)
- Road Sign Detection(https://www.kaggle.com/datasets/andrewmvd/road-sign-detection/data)
- Load and preprocess images(https://www.tensorflow.org/tutorials/load_data/images#visualize_the_data)