Evan Hollier

CS 4449 Capstone

Midterm project report

7/16/2024

Recognizing Traffic Signs with Keras

Dataset Description

The dataset consists of 51,839 images of 32x32 traffic signs. They were captured under different real-life conditions, including showing obstructions, poor lighting, or even the sign being far away from the camera. There is 43 categories of signs. Some examples are, "Stop,", "Yield," and "Speed limt 50km/h." The labels are not evenly distributed, with the most frequent occurring roughly 12 times more than the least frequent. I would consider this slightly unbalanced, enough to monitor recall and precision in addition to accuracy. Accuracy, recall, and precision were consistently close to each other, so it didn't end up being problematic. The data was already split into train-validation-test with approximately 67-9-24 percents.

Data Preparation And Analysis

Before I could even feed the data into the different pre-trained models from Keras, the images needed to be upscaled. I made them 128x128, so each original pixel was blown up to 4x4. When trying different upscaled resolutions, I made sure to only use factors of 32 so all the original pixels would be maintained. Interestingly, the factor by which I upscaled the images had siginificantly different results on the performance of the pre-model. I think this is because the imagenet weights are different depending on the resolution of the input data. I arbitrarily did 128x128 when running the baselines, and when I started tuning I started with 64x64, in the hopes that the models would run faster. This is when I discovered this quirk, and I decided it wasn't it the spirit of this project to play around with it until I got whatever imagenet weights worked the best. So I stuck with the 128x128 imagenet weights.

I verified the dataset itself was clean by checking for any non 32x32 images. The only other pre-processing I had to do was one-hot encode the label data since it's categorical.

Once I was ready to start training models, I went through a bunch of pre-trained models from Keras. Since I was only looking for a base model to use, I did not add a custom layer to them. I ended up trying six pre-trained models. The best one ended up being EfficientNetB0. There is a series of other EfficientNet models, with the number after B representing how increased the complexity is. I tried a more complex EfficientNet model, EfficientNetB5, to see if it would be better, and found that is was not.

When I had my pre-trained model selected, I set up early stopping and model checkpoints, so that I would only use the best model from each trial. I started with a simple custom layer on top of EfficientNetB0, with just a single Dense layer. I performed a grid search with potentially 32, 64, or 128 neurons, and 0, 0.2, or 0.5 dropout. The results from the grid search, as well as EfficientNetB5 being worse than EfficientNetB0, made it seem that adding complexity to the custom layer was detrimental to performance. I stuck with only 32 neurons for most of my trials with the more straightforward tests, like Data Augmentation, L2 Regularization, and Batch Normalization. I occasionaly tried 64 neurons, but it never outperformed the same model with only 32. Once I was happy with the models not overfitting, I started training for more epochs with longer early stopping patience.

Results

The final model utilized Data Augmentation on 4x upscaled images, EfficientNetB0 pre-trained model as base, Batch Normalization, a custom 32-neuron Dense layer with 0.2 Dropout, Adam optimizer. The best weights were on the 25th epoch out of the 30 I ran. This means I could potentially train for even longer, though I don't expect the improvement to be that much.

Even though complexity seemed to be unhelpful for this dataset, I still would have liked to try adding more just to see if a threshold exists where it's better. This would probably take several more days of training full time though.

Discussion

I believe the main challenge with this dataset was the poor quality images. The combation of very low resolution and poor lighting made most of the images unreadable even to me. If I were to guess, the model probably used sign shape to label signs whose picture/text was unusable. The problem is a lot of signs have the same shape, and even coloring. The results are much better than I was expecting, and definitely better than I could do myself looking at the data from a human perspective.

Additionally, I think there wasn't enough validation data. I'm not sure why the original data only allocated 8.5% for it, but 4,410 images was not enough to get consistent validation results when tuning models. Sometimes simply rerunning the same model would results in a validation score being more than a percentage point higher or lower. I'm only changing a little at a time when tuning the model, so large variation like this makes it difficult to tell whether what I did was helpful or not. Personally, I would have made the validation and test splits more even, maybe something like 67-15-18.

In the context of using this kind of dataset for self-driving cars, something I think would be interesting to develop for this dataset is a custom metric instead of recall or precision. I think false negative or positive for different signs is worse. For example, Stop signs should minimize false negatives, since it would be really bad to label them as something else. On the flip side, false positives on "General Caution" signs wouldn't explicitly add danger, because it's pretty much the same as just missing the actual sign. Operating under the principal that 100% accuracy couldn't be attained, I think a function that specific signs are penalized differently would be important for actual use.