Intro to Machine Learning Through Traffic Sign Recognition Using CNN

Chiadika Vincent
University of California, San Diego
cbvincen@ucsd.edu

Sachintha Brandigampala
University of California, San Diego
sbrandig@ucsd.edu

Rey Simon
University of California, San Diego
rsimon@ucsd.edu

Abstract

This paper will delve into the application of image recognition and classification on German traffic signs. We took on this task using a deep learning approach and was aided with the help of Tensorflow. The architecture that was used for our Convolutional Neural Network was LeNet 5, however, we deviated from the original by adding an extra convolutional block and changed the kernel size from (5,5) to (3,3). This algorithm turned out to be very simple and effective and even provided higher accuracy than the original project tutorial in some cases. However, this was due to overfitting and caused us to lose some accuracy when working with the test dataset. In the end, we were able to achieve an accuracy of 95% on average.

1. Introduction/Motivation

Advancements in technology have now reached all aspects of our lives. All activities we now partake in have been transformed by new technological innovations, such as our daily commute. Vehicles have grown to be exponentially smarter within the past few years, aiming to be able to one day go fully autonomous. Tesla is the current pioneer in this segment with their Autopilot technology that is available in all their cars. This led us to wonder how the car is capable of understanding what exactly is going on in its environment. Through some research, we learned that in order to simultaneously coordinate between all of the sensors and cameras on the car, as well as all of the environmental variables that they encounter, Tesla uses an advanced architecture in machine learning called HydraNets. They contain a 'backbone' which is trained on all objects and 'heads' which train on specific tasks. For this project, we decided to focus on one specific task, which in this case would be traffic sign recognition. We went in intending to attempt to use the knowledge we have obtained from ECE 196, coupled with newfound knowledge we gained through personal exploration to create and implement a machine learning algorithm that is both accurate, and has real-world applications.

2. Problem Solving Approach

The first hurdle that we encountered was finding a workable dataset. Through analysis of multiple datasets, we found one

the German Traffic Sign Recognition Benchmark dataset which was simple, organized, and split into training and testing data for our convenience. The training dataset was further organized by class with each image class being sorted into different folders. These folders were numbered 0-42, as we had 43 different classes of traffic signs. After doing some further research, we decided that implementing a Convolutional Neural Network (CNN) algorithm was the best approach for an image recognition and classification project. Despite our lack of experience and knowledge of deep learning, we decided to take on the challenge. We would be using the functionality of Tensorflow, an open-source software library for Machine Learning, to aid us with the heavy lifting. The final challenge was to actually implement what we set out to do. Though we made great strides during this process, we also took some steps back in our progress due to trial and error. Data management and algorithm implementation gave rise to these errors. In the next section, we will introduce our solutions to the obstacles we encountered and how we implemented our algorithm.

3. Solution

The first thing we wanted to do was to visualize our data to see the types of images we are working with, which we did by creating a function that outputs the first 9 images of a specific class of traffic signs and resized them to 30x30 pixels.

The next thing we wanted to do was visualize our overall dataset. We noticed that our dataset was heavily skewed, in that some classes had substantially more images than others. This raised questions regarding whether or not this affected our accuracy, which we will discuss later in the report.

Before beginning the actual algorithm, we needed to store the images in our training data set into a NumPy array so that we could use them in our algorithm. This array contained about 40,000 images, each with an RGB value. We then created another NumPy array containing the labels of each of the images. These labels corresponded to the folder that each image belonged to in the training dataset. Next, we used the train_test_split() method from sklearn to split the training data into an 80/20 split.

We then used the to_categorical utility in Keras, a library in Tensorflow to one-hot encode our dataset to the different classes. In y_train, we have 31,367 different photos of traffic signs. Each of these belongs to 1 of the 43 different classes we have. However, the algorithm has no way of understanding the connection between the two. One-hot encoding creates a matrix that is 31367x43. Each row is a different image

```
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)

print(y_train.shape)
print(y_test.shape)
```

```
(31367, 43)
(7842, 43)
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 1. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 1. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

Figure 1. One hot encoding visualization

and each column represents the different possibilities for the classes it may belong to, as seen in Figure 1. In each row, all of the columns would be filled with 0, except for the column that matches with the class number that photo belongs to because that index will have 1. This is how the algorithm can encode categorical variables into numeric ones.

Finally, we arrived at implementing the CNN algorithm. A CNN is able to detect patterns and make sense of images. What makes a CNN unique is that it contains hidden layers which are called convolutional layers, which take in images as inputs.

These convolutional layers contain filters, which are convolved with the images. This convolution is what detects the patterns in our images. The output of these layers is then passed to other subsequent layers. Another type of layer that we decided to use is called an average pooling layer. These layers section an image into 2x2 patches and then convert those patches into the average value of those patches. The output is an image with reduced dimensionality. The reason for utilizing this pooling layer is to account for any slight changes that may occur to the input, as these changes will result in different outputs from the convolutional layers. The pooling layer then corrects this by downsampling and instead gives a more generalized image of the important features.

Now that we gained an understanding of what CNN is and how it works, it was time for us to choose a specific architecture. This refers to the way we wanted to position our layers and how many layers we wanted to use. After careful consideration, we narrowed it down to the LeNet-5 architecture (Figure 2) and the VGG-16 architecture (Figure 3). LeNet-5 starts with a black and white image, then passes the image through two iterations of convolution and average pooling. A VGG-16 architecture consists of 13 convolutional layers and 5 max-pooling layers. As CNN's begin to get more sophisticated, they start to include more layers.

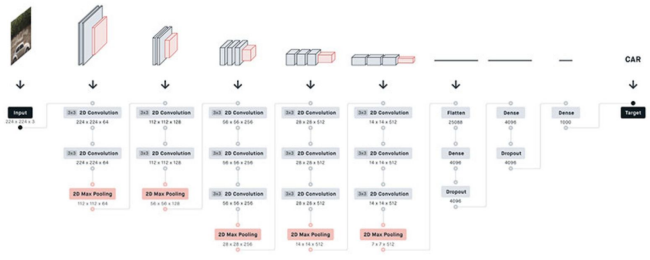

Figure 2: VGG-16 architecture

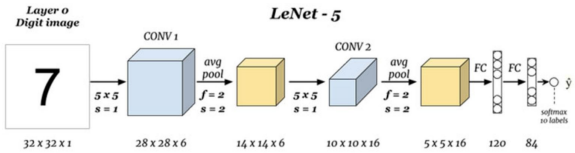

Figure 3: LeNet-5 architecture

Taking into account elements from both architectures, we came up with a version of our own. Some parameters we varied were the number of convolutional layers in our architecture, the use of average pooling and max pooling, the number of filters in each convolutional layer, and kernel size. The typical kernel size used in convolutional layers is (5,5), but we reduced it down to (3,3) to get more convolutions in each layer. This way, the output of these layers would be more precise. Furthermore, although we read, through extensive research, that max-pooling would work better than average pooling, our accuracy ended up increasing when we switched to the latter. Max pooling is generally preferred over average pooling because it highlights the most important features of the patch and not just an average.

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 28, 28, 32)        896
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 26, 26, 32)        9248
_________________________________________________________________
average_pooling2d_3 (Average (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 9, 9, 64)          36928
_________________________________________________________________
average_pooling2d_4 (Average (None, 4, 4, 64)          0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 2, 2, 128)         73856
_________________________________________________________________
average_pooling2d_5 (Average (None, 1, 1, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 256)               33024
_________________________________________________________________
dense_3 (Dense)              (None, 43)                11051
=================================================================
Total params: 183,499
Trainable params: 183,499
Non-trainable params: 0
_________________________________________________________________
```

Figure 4: Input/output of convolutional and pooling layers

Another aspect of our architecture is that we considered adding a dropout layer at the end of each convolutional block. Dropout layers act as a buffer and cause some layer outputs to randomly drop out. This is supposed to correct for instances when the CNN network overcorrects for mistakes from previous layers, making the network stronger overall. However, we made the mistake of not including it because we produced better accuracy scores (about 95%) on our training data without it. This also minimized our loss. However, we realized that this led to overfitting and as a result, we produced subpar results (about 96%) when using our algorithm on the testing data.

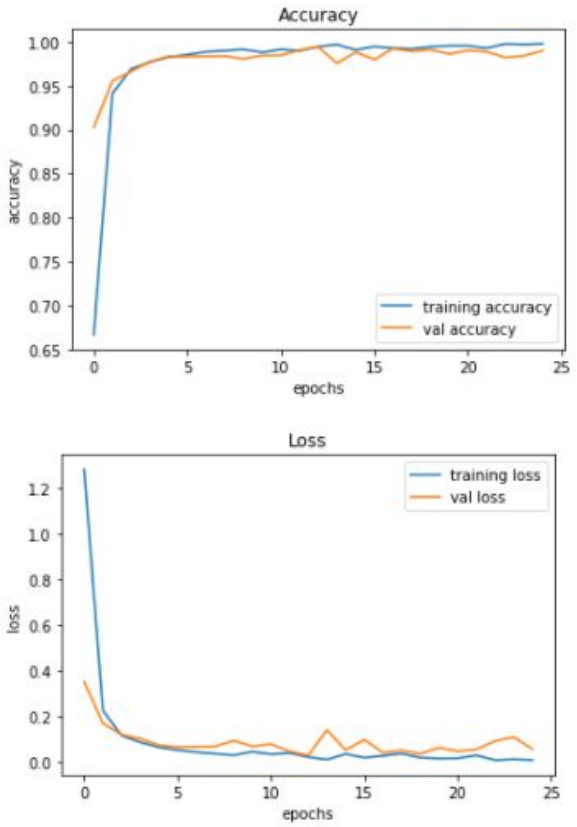

Figure 5: Training accuracy/loss summary

Further questions we explored examined our accuracy score and different factors that could affect it. One aspect we changed was utilizing balanced accuracy scores from sklearn. This worked better for us than the accuracy score because of how skewed our dataset was, with some classes having thousands of more images than others. We also calculated the precision of our overall model and each class, and the F1-score(weighted average) as we were dealing with an unbalanced sample dataset.

	precision	recall	f1-score	support
0	0.95	1.00	0.98	60
1	0.95	0.99	0.97	720
2	0.97	0.99	0.98	750
3	0.95	0.98	0.96	450
4	1.00	0.96	0.98	660
5	0.89	0.97	0.93	630
6	0.99	0.77	0.86	150
7	1.00	0.93	0.96	450
8	0.99	0.92	0.95	450
9	0.91	1.00	0.95	480
10	1.00	0.99	1.00	660
11	0.97	0.97	0.97	420
12	0.96	0.96	0.96	690
13	0.98	0.99	0.98	720
14	1.00	0.99	0.99	270
15	0.85	1.00	0.92	210
16	0.99	0.95	0.97	150
17	1.00	0.99	1.00	360
18	0.95	0.95	0.95	390
19	0.81	1.00	0.90	60
20	0.85	0.99	0.91	90
21	0.95	0.61	0.74	90
22	0.93	0.96	0.95	120
23	0.95	0.98	0.97	150
24	0.97	0.84	0.90	90
25	0.98	0.93	0.95	480
26	0.97	0.84	0.90	180
27	0.88	0.50	0.64	60
28	0.94	1.00	0.97	150
29	0.92	0.98	0.95	90
30	0.82	0.85	0.83	150
31	0.97	0.99	0.98	270
32	0.80	1.00	0.89	60
33	1.00	1.00	1.00	210
34	0.92	1.00	0.96	120
35	0.99	0.99	0.99	390
36	0.98	0.98	0.98	120
37	0.65	1.00	0.78	60
38	1.00	0.99	0.99	690
39	1.00	0.93	0.97	90
40	0.92	0.54	0.69	90
41	1.00	0.37	0.54	60
42	0.85	0.98	0.91	90
accuracy			0.96	12630
macro avg	0.94	0.92	0.92	12630
weighted avg	0.96	0.96	0.96	12630

Figure 6: Accuracy Table

The model that we implemented consists of 32 filters per layer and each filter is used to localize various patterns in the image. These filters go through 40,000 images and produce 2d activation maps that have the size of (filters x images). The visualization of these filters was crucial to understanding how our model works and we used a reference code[1][7] to visualize the filter sets and activation outputs of each layer.

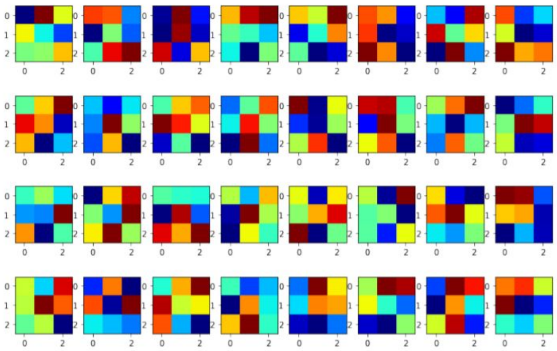

Figure 7: Map of 32 filters in first layer

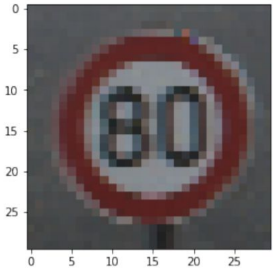

Figure 8: Speed limit (80km/h)

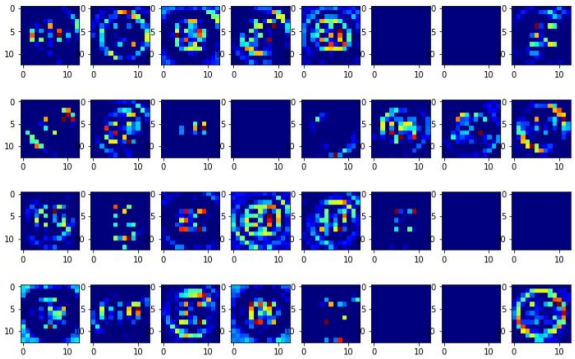

Figure 9: 2nd activation layer of image

4. Conclusion and Outlook

Throughout the project, we experimented with the learning process, by implementing CNN with elements from VGG-16 and LeNet-5 architectures to establish a proper algorithm structure that could perform well with the German traffic light dataset. The performance of the CNN algorithm was slightly inefficient during the training process and required a large data size to store its coefficients, while the VGG-16 was too complicated and excessive for the task. We settled with the short structure of the LeNet-5 architecture while including the order of layers from the VGG-16 architecture, as it provided the most reliable accuracy, and the debugging process was effortless. The use of Keras, simplified the layer implementation, analysis of the project, and facilitated the tuning process of the parameters to optimize the neural network to achieve better results. The conventional LeNet-5 architecture had 92% average accuracy on the dataset, but the addition of a convolutional layer and the removal of drop-out functions, plus data preprocessing increased the average accuracy up to 94%. The proposed and developed model for the final traffic sign detection project performed slightly above the conventional LeNet-5 architecture and the use of higher epochs (i = 5,15,25) increased its average training accuracy up to 98%-99%. The selected dataset had an imbalanced data distribution, and to address the issue, we used undersampling of the dataset to equalize the distribution.

However, the performance of the model reduced down to 92%, when the sampling size was reduced by 25%. The finalized model uses the training data of the entire dataset and was able to identify more complicated localized patterns as it reaches deeper layers of the architecture. Final architecture is sufficient to classify 43 classes of about 40,000 image data within

300us, with a 99% accuracy on training data and 95% on testing data. This accuracy difference is a cause of overfitting data, and a resampling technique, such as cross-validation can be used to estimate model accuracy and evaluate the CNN algorithm to limit overfitting.

The results were able to exceed our expectations for the project, however, further improvements can be done to increase the functionality of this algorithm. As of now, the program only identifies a class of an image upon feeding one image into the system. We could improve this by expanding the detection system to identify multiple objects in a real-time video, which could facilitate navigation applications such as autonomous vehicles, robots, and medical technology. However, due to the limited time, we were unable to expand our model to solve complicated tasks, but the experience and the knowledge we gained as a team was enormous.

5. References

[1]. "How to get the output of Intermediate Layers in Keras?," knowledge Transfer, 01-Feb-2020. [Online]. Available: https://androidkt.com/get-output-of-intermediate-layers-keras/. [Accessed: 09-Feb-2021].

[2]. "Kenny Miyasato," Medium. [Online]. Available: https://medium.com/@kennymiyasato. [Accessed: 09-Feb-2021].

[3]. "4 CNN Networks Every Machine Learning Engineer Should Know," knowledge Transfer, 08-Feb-2020. [Online]. Available: https://www.topbots.com/important-cnn-architectures/ [Accessed 28-Jan-2021]

[4]. Mykola, "GTSRB - German Traffic Sign Recognition Benchmark," Kaggle, 25-Nov-2018. [Online]. Available: https://www.kaggle.com/meowmeowmeowmeow/gtsrb-german-traffic-sign. [Accessed: 09-Feb-2021].

[5]. J. Cohen, "Computer vision at tesla," 15-Oct-2020. [Online]. Available: https://heartbeat.fritz.ai/computer-vision-at-tesla-cd5e88074376. [Accessed: 09-Feb-2021]

[6]. "Python Project on Traffic Signs Recognition with 95% Accuracy using CNN & Keras," *DataFlair*, 06-Aug-2020. [Online]. Available: https://data-flair.training/blogs/python-project-traffic-signs-recognition/. [Accessed: 09-Feb-2021]

6. Code Appendix

PDF of Code