Traffic Sign Classifier Project Writeup

Charlie Wartnaby

Applus IDIADA

charlie.wartnaby@idiada.com

Udacity Self-Driving Car Engineer Nanodegree

Term 1 Computer Vision and Deep Learning

4 Aug 2017

Abstract

A modified version of the LeNet convolutional neural network was implemented in TensorFlow and trained on the provided set of German traffic sign images. The images were normalised in brightness but left in colour.

Initial experiments were performed to discover good hyperparameters to use with a LeNet model to which only essential changes had been made to accommodate the different number of output classes and image colour attributes. Validation accuracy was found to depend strongly on training batch size.

Further experiments were then performed in which the first convolutional layer depth of the model was varied. A validation accuracy of ~96% was achieved with a layer 1 depth of 24, exceeding the project requirement of 93%. An accuracy of 93.5% was obtained on the test set, suggesting that the model was not overfitted or biased towards the training and validation data.

The model was then applied to 6 new German traffic sign images obtained from the internet. Initially 4 of 6 signs were classified correctly. An experimental re-run with the first-layer depth correctly classified 6 of 6 signs, but of course choosing one run over the other would be introducing bias into the result.

Although ultimately successful, using the TensorFlow library introduced some difficulties in terms of model saving and restoring, and debugging, necessitating costly additional training runs. Naming every tensor in the model, just in case its retrieval from the version saved on disk is required, would be recommended in future. Issues were also encountered in TensorFlow values being lost on a save/restore cycle in the Jupyter notebook environment but not when run on the command line directly by the python interpreter.

Write-Up and Code

This document is the project write-up, meeting the rubric points specified.

The project code is in the accompanying Jupyter notebook, Traffic_Sign_Classifier.ipynb.

Data Set Summary & Exploration

See Jupyter notebook code and output for details. The code used Numpy methods to size the data, then ordinary Python to count the number of unique classes.

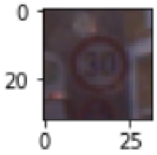
```
Number of training examples = 34799
Number of validation examples =  4410
Number of testing examples = 12630
Image data shape = (32, 32, 3)
Number of classes = 43
```

Include an exploratory visualization of the dataset.

The code used matplotlib to show one example of each class.

```
Class ID=0 first example:
```

```
Class ID=1 first example:
```



(... etc, see notebook for complete output set.)

I noted that the images provided varied considerably in brightness, so as a result of this visualisation I decided to normalise brightness (see below).

Design and Test a Model Architecture

Image Preprocessing

For traffic signs, I intuitively felt that the colour information should be retained, because (for example) the red ring around a speed sign is a distinctive element that the model may well use to identify that sign. However, during debug experiments I did sometimes convert the image set to greyscale to compare the performance. In fact it made little difference, though retaining colour was slightly better in some trials and so I did retain colour finally.

Whether to use greyscale or colour was an input hyperparameter:

```
##############################################
#  Hyperparameters
num_colours = 3  # 1 greyscale, 3 colour
EPOCHS = 700
BATCH_SIZE = 256
...
```

```
if num_colours < 3:
    x_train_norm = debug_to_gray(x_train_norm)
    x_valid_norm = debug_to_gray(x_valid_norm)
    x_test_norm = debug_to_gray(x_test_norm)
```

I normalised images for intensity however, noting that some of the images provided were rather murky while others were quite bright.

In the notebook, **`normalise_one_image()`** analyses an image to find the maximum and minimum intensity value (in any of the RGB channels). The values are then rescaled to an overall range of [-1, 1] for input to the model, where -1 corresponded to the minimum value found in that specific image and +1 the maximum value. Hence all images then had an equal range of RGB values. **`normalise_image_set()`** applies this operation to a complete list of images.

Model Architecture

As suggested I started with the LeNet architecture from the previous course exercise, as this was already known to be successful in classifying images of the required size. I modified it however to work with colour images as an option, and in later experiments (see below) I varied the first layer depth too, so these became input hyperparameters rather than hard-coded numbers:

```
def LeNet(x, num_classes, num_colours, layer1_depth):
```

My final model consisted of the following layers:

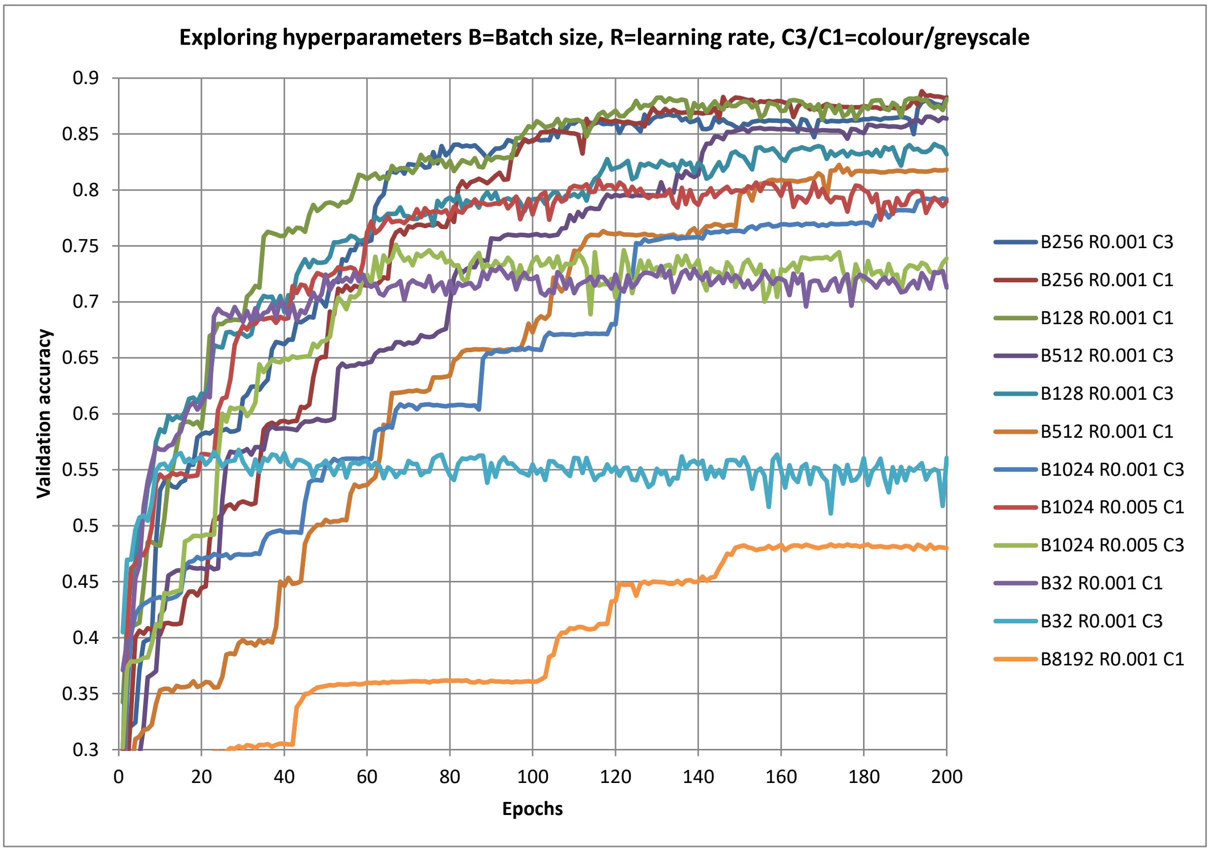
Layer	Description
Input	32x32x3 RGB image (with num_colours=3) or 32x32x1 (num_colours=1); 3 colours worked slightly better and were used in the final tests.
Convolution	Applying 5x5 patch with valid padding and 1x1 stride hence output shape 28x28 horizontally. Vertical depth = layer1_depth hyperparameter; LeNet original was 6, but performance improved markedly improving this to larger values such as 10, 12, 18 or 24. Final value was 24, hence this layer output 28x28x24. (An additional experiment at the end used 18, which happened to match more of the road sign pictures found on the internet, but it would be biased to prefer this version.)
RELU	
Max pooling	As per LeNet, max pooling in horizontal direction only with 2x2 stride hence output shape 14x14x(layer 1 depth), or finally 14x14x24.
Convolution	As per LeNet, again 5x5 patch with valid padding and 1x1 stride, with output depth of 16. Hence output 10x10x16.
RELU	
Max pooling	As per LeNet, max pooling in horizontal direction only with 2x2 stride hence output shape 5x5x16.
Fully connected	As per LeNet, a fully connected layer applied to a flattened version of the previous layer, output 84.
RELU	
Fully connected	As per LeNet, a fully connected layer mapped to the number of output classes (here num_classes=43 rather than the original 10 for the LeNet decimal numerals)
Softmax	To obtain final logit probability values for each class

Model Training

I performed a number of experiments with different hyperparameters to explore how to get the best performance out of the model firstly as an essentially unmodified LeNet, and then with various layer 1 depth values.

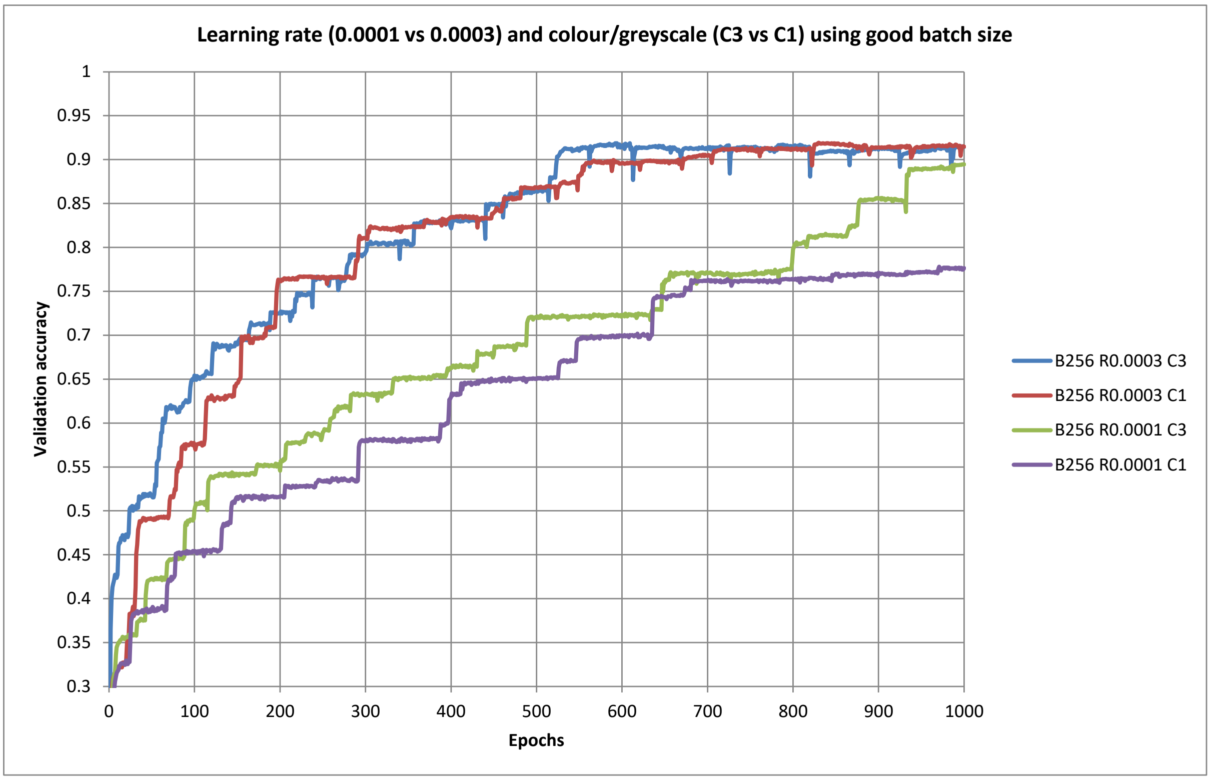
The model code was resaved as a standalone Python script to allow execution on the command line, such that the output could be directed to a tab-delimited file for convenient import into Microsoft Excel for plotting. Working as a standalone script also made debugging in Visual Studio possible. Also this avoided problems restoring TensorFlow values which occurred only in the Jupyter environment (see below).

Firstly, running over 200 epochs I experimented with different batch sizes (B), learning rates (R) and colour depths (C), with the original LeNet layer 1 depth of 6:



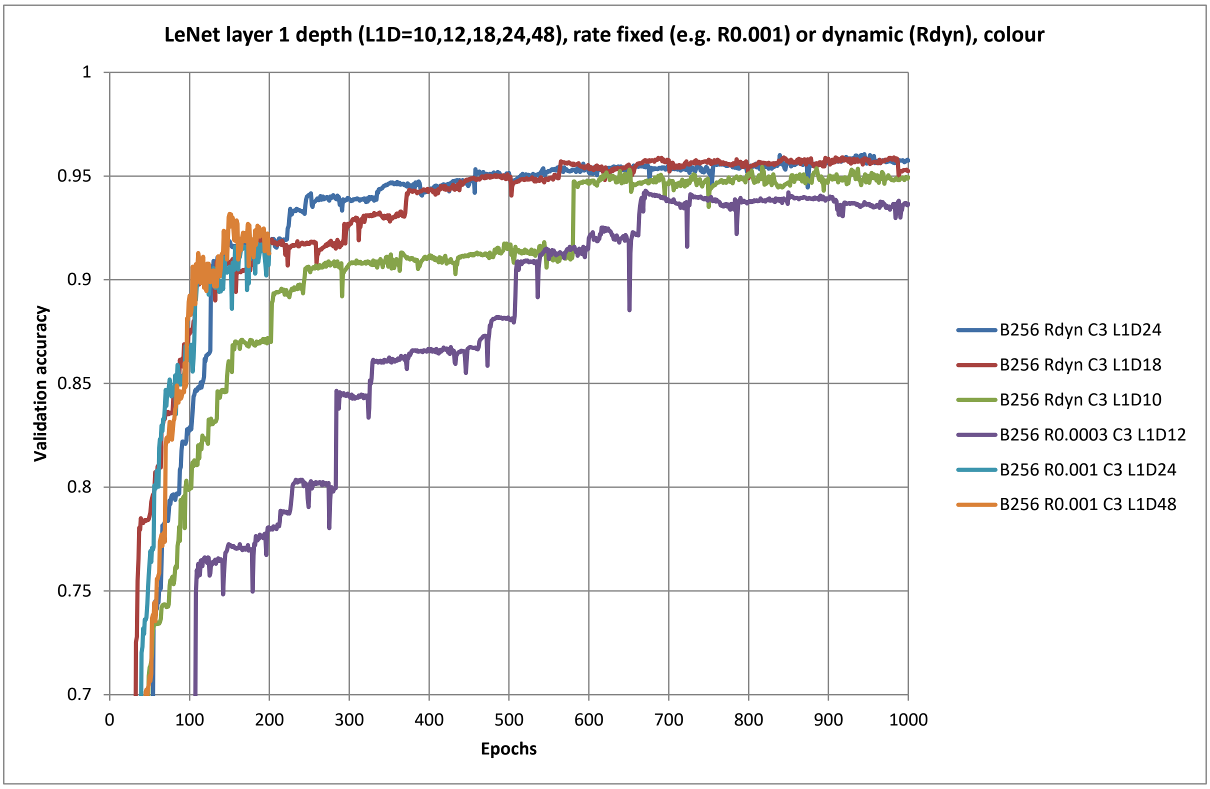
The quality of the fit was a strong function of the batch size. Good batch sizes (e.g. 128, 256, 512) were of the order of 10x the number of classes to be identified (here 43). This fits with the previous LeNet exercise using batches of 128 where there were only 10 classes. Using significantly smaller (e.g. 32) or larger (e.g. 8192) images per batch resulted in much poorer performance. I selected 256 as the best batch size on the basis of these experiments.

Next, I explored the effect of changing learning rate and colour depth (grey or colour). Using low learning rates and large numbers of epochs meant long execution times, so only a limited number of experiments were possible:



Clearly the model trained faster with a higher learning rate. Yet also, the trend was still upwards after 1000 epochs with a smaller learning rate. I therefore decided to modify the code for future experiments so that the learning rate was adjusted dynamically, starting with a larger value (0.001) for faster initial training, progressively reducing (to around 0.0001 after 1000 epochs) for better final convergence.

Finally I experimented with the LeNet model itself, starting at the beginning with the first layer depth, to see what improvements could be obtained by restructuring or resizing the model; I felt that given the greater information content of traffic signs compared to the decimal numerals LeNet was intended for, a more complex network might be required. Simply using higher values of the layer 1 depth was sufficient to exceed the project required accuracy of 0.93 however:



A validation accuracy of 0.96 was obtained using colour (not greyscale) processing with a layer 1 depth of 24 and batch size of 256, so this was used as the final implementation.

Final Model Accuracy

Then running on the test data (hitherto left alone as advised), a final test accuracy of 93.5% was obtained almost as good as the validation accuracy (95.9%). This, and the fact that the validation accuracy curves had not started declining, suggest that overfitting was not a problem:

```
Retrieving normalised data from file...
... data loaded.
INFO:tensorflow:Restoring parameters from traffic_sign_classifier_model-E1000-B256-
R0.000091-C3-L1D24
Test Accuracy = 0.935
```

TensorFlow Difficulties

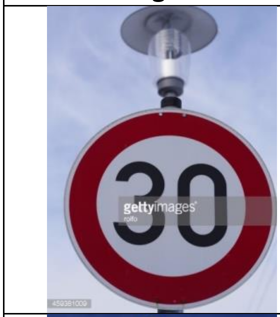
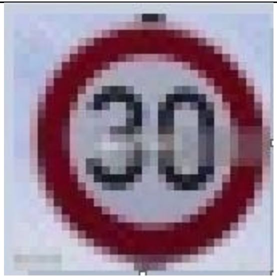
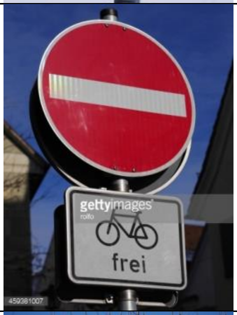
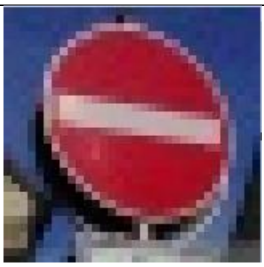
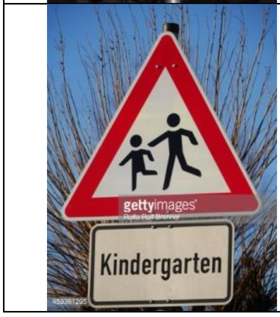
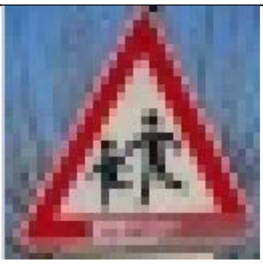
This was not actually the final model run however, as used for the additional tests on images found on the internet. In order to extract named tensors for those reporting stages, I had to go back and name the output tensors I found I needed, and then retrain the model all over again, taking of the order of 12 hours, and (as it happened) not obtaining only 94% accuracy. Not anticipating every output I would need, I had to repeat this cycle a number of times, with validation accuracies in the range 94-96%. In general the difficulty of saving and restoring TensorFlow models such that they could be re-run and useful data extracted became an obvious limitation of the library, though with experience one would know to name all tensors to make them accessible after restoring from disk.

Debugging was also difficult with TensorFlow. I set up the program to run in a Visual Studio PTVS (Python Tools for VS) environment to allow line-by-line debugging and examination of Python variables. But the internal values of tensors are not directly available during execution; the best I could do was to add **`tf.Print()`** "fake" tensor assignments to output some values when I had to debug what was really happening inside TensorFlow.

I encountered a specific problem whereby reloading a trained model would give accuracy no better than chance, even applied to the original training set. Dumping the layer 1 biases just before saving and after restoring, it was clear that the trained values had been lost, reverting to the pseudorandom initial values. This problem was sometimes avoided by restarting the notebook kernel before execution. The only sure way to avoid it was to copy the code into a script file outside the Jupyter notebook and run it directly on the command line using the Python interpreter directly; that always worked. **`train_model.py`** and **`process_internet_images.py`** were used for that purpose (note that plotting had to be commented out however).

Testing the model on new images

6 images were taken from the Getty image library (http://media.gettyimages.com). These were cropped and resized to 32x32 but otherwise not processed in any way. This resize tended to blur out the 'getty images' copyright message but leaving a bit of a smudge across the middle as a result, presenting a slight additional challenge to the model (we might imagine that the model would have to fit faded or dirty signs in any case):

Original	32x32 as input to model	Filename
		1-resized_30_limit.jpg
		17-resized_no_entry.jpg
		28-resized_children.jpg

		18-resized_general_caution.jpg
		27-resized_pedestrians.jpg
		23-resized_slippery.jpg

See the notebook for full output. The output automatically listed the top 5 softmax classifications and their descriptive names for each, highlighting the correct choice (even if that did not obtain the top probability).

Note that the correct class ID (e.g. 23 for slippery road) was encoded in the filename, so that the code could systematically highlight whether or not the correct choice had been found, and summarise overall accuracy.

Results with model using convolutional layer 1 depth of 24 (original)

Output from **`process_internet_images.py`**:

```
Loading internet test images from disk...
... 6 images loaded.
43 class descriptions loaded
Normalising images...
... 6 images normalised.
Running model to classify images...
... model run complete.
Final test accuracy on 6 internet images = 0.667
4 out of 6 correctly classified
Listing top 5 ranked softmax probabilities per image (correct choice flagged with
'*')

Top 5 softmax probabilities for image filename 1-resized_30_limit.jpg:
* 1 1.000000 Speed limit (30km/h)
```

```
  2 0.000000 End of speed limit (80km/h)
  3 0.000000 Speed limit (20km/h)
  4 0.000000 Speed limit (70km/h)
  5 0.000000 End of no passing by vehicles over 3.5 metric tons

Top 5 softmax probabilities for image filename 17-resized_no_entry.jpg:
  1 1.000000 Yield
* 2 0.000000 No entry
  3 0.000000 No passing
  4 0.000000 Stop
  5 0.000000 Speed limit (60km/h)

Top 5 softmax probabilities for image filename 18-resized_general_caution.jpg:
* 1 1.000000 General caution
  2 0.000000 Right-of-way at the next intersection
  3 0.000000 Children crossing
  4 0.000000 Traffic signals
  5 0.000000 No passing

Top 5 softmax probabilities for image filename 23-resized_slippery.jpg:
* 1 0.997872 Slippery road
  2 0.002005 No passing for vehicles over 3.5 metric tons
  3 0.000058 Dangerous curve to the left
  4 0.000055 Wild animals crossing
  5 0.000009 End of no passing by vehicles over 3.5 metric tons

Top 5 softmax probabilities for image filename 27-resized_pedestrians.jpg:
  1 0.998650 Traffic signals
  2 0.001304 General caution
  3 0.000038 Dangerous curve to the left
  4 0.000003 Slippery road
  5 0.000002 No passing

Top 5 softmax probabilities for image filename 28-resized_children.jpg:
* 1 0.999972 Children crossing
  2 0.000028 Dangerous curve to the right
  3 0.000000 Speed limit (80km/h)
  4 0.000000 End of no passing
  5 0.000000 Road work
```

Results with model using convlutional layer 1 depth of 18 (additional experiment)

The model was retrained and re-run after reducing the layer 1 depth (for speed) to check if the whole experiment was reproducible. This happened to give a better result of 100% correct classification. However, it would be biased to choose this model over the previous one given the benefit of hindsight.

Output from **`process_internet_images.py`** (also run successfully in notebook):

```
Loading internet test images from disk...
... 6 images loaded.
43 class descriptions loaded
Normalising images...
... 6 images normalised.
Running model to classify images...
... model run complete.
Final test accuracy on 6 internet images = 1.000
6 out of 6 correctly classified
```

```
Listing top 5 ranked softmax probabilities per image (correct choice flagged with
'*')

Top 5 softmax probabilities for image filename 1-resized_30_limit.jpg:
* 1 1.000000 Speed limit (30km/h)
  2 0.000000 End of all speed and passing limits
  3 0.000000 End of speed limit (80km/h)
  4 0.000000 Dangerous curve to the right
  5 0.000000 Keep right

Top 5 softmax probabilities for image filename 17-resized_no_entry.jpg:
* 1 0.714969 No entry
  2 0.285031 Stop
  3 0.000000 No passing
  4 0.000000 Beware of ice/snow
  5 0.000000 No passing for vehicles over 3.5 metric tons

Top 5 softmax probabilities for image filename 18-resized_general_caution.jpg:
* 1 1.000000 General caution
  2 0.000000 End of no passing by vehicles over 3.5 metric tons
  3 0.000000 End of no passing
  4 0.000000 Roundabout mandatory
  5 0.000000 Keep left

Top 5 softmax probabilities for image filename 23-resized_slippery.jpg:
* 1 1.000000 Slippery road
  2 0.000000 No passing for vehicles over 3.5 metric tons
  3 0.000000 Dangerous curve to the left
  4 0.000000 Double curve
  5 0.000000 Go straight or right

Top 5 softmax probabilities for image filename 27-resized_pedestrians.jpg:
* 1 0.999982 Pedestrians
  2 0.000018 Bicycles crossing
  3 0.000000 General caution
  4 0.000000 Double curve
  5 0.000000 Speed limit (20km/h)

Top 5 softmax probabilities for image filename 28-resized_children.jpg:
* 1 1.000000 Children crossing
  2 0.000000 Go straight or right
  3 0.000000 Bicycles crossing
  4 0.000000 Dangerous curve to the right
  5 0.000000 Bumpy road
```

Conclusions

A modified version of the LeNet architecture was successfully applied to the analysis of German road sign images. The main modifications were increasing the number of colour channels processed to be a variable (so the three RGB channels could be accepted, or 1 for greyscale), and making the convolutional layer 1 depth a variable, increased from the original LeNet depth of 6 to a higher value such as 24.

Initially (with a layer 1 depth of 24) only 4 of 6 'new' images were successfully categorised, compared to a test set accuracy of ~94%. This relatively poor result may be due to the 'new' images being differently processed in terms of cropping; also some were taken from slightly oblique angles, resulting in somewhat distorted shapes.

Rerunning the entire model training and internet image classification again to assess reproducibility of the whole experiment, but with the layer 1 depth reduced from 24 to 18 (for speed), all 6 signs were correctly classified. However, preferring this second run over the first would be introducing the bias of hindsight. A better test would be to find still more internet images for further experiments.

Considerable time was involved not in the core project work, but on the difficulties of debugging the somewhat inaccessible TensorFlow model, and further problems with model save/restore cycles which worked on the command line but failed when identical code was run in the Jupyter notebook environment.