PPMROB

Jetracer Road Sign Detection

Github: https://github.com/KonstantinRoehrl/PPMROB_RoadSignDetection

Konstantin Röhrl Yang Yikai

March 4, 2023

1 Introduction

Within the scope of the course 'Programming Principles of Mobile Robotics' at TU Vienna, we have gathered a small-scale dataset in order to train a neural network to correctly identify four different classes of roadsigns. Since the main purpose is the onboard use on the Waveshare Jetracer [1], all images have been created with the mounted camera.
By deploying our trained model to the embedded device, we could evaluate the performance in the lab-environment with 'real-world' data.

2 Gathering the Data

To keep the effort reasonable for our purpose, we have agreed on creating our dataset with four different classes of roadsigns as shown in figure 1.
Images are directly gathered from the mounted camera on the Jetracer in the lab environment. Each class is represented by 420 images covering different viewpoints, scales and backgrounds. Only the first class offers an additional 60 images. Therefore, the entire dataset consists of 1740 images.

Figure 1: Classes from left to right: stop sign, 50 speed limit, turn right, turn left

3 Implementation

In order to establish the pipeline for training our classifier, we adapted most of the structure provided by the corresponding PyTorch tutorial [6].

3.1 Overview

In the following, we quickly list the components of our implementation as well as explaining its purpose.

3.1.1 `config.py`

Within this file, we are defining multiple parameters such as the path to the the dataset, which model-architecture we want to use and where to store the trained model. Last but not least, we set parameters for the training process as well as the device on where to execute the calculations.

3.1.2 `dataloader.py`

For our custom dataset, we have created a simple dataloader in order to iterate the data.

- *Defining the datasplit:* In order to keep things simple, we just take each fifth image for the evaluation-split. The rest remains in the training-split.
- *Loading data:* Each image is converted to YUV with consecutive data augmentation in order to be more robust with different lightning conditions.
- *Augmenting data:* For both splits, images are resized to `(w,h)=(256, 256)` and normalized based on previously calculated mean and standard deviation (Hardcoded into dataloader). Furthermore, we rely on PyTorch's `ColorJitter` augmentation [4] in order to bring a little bit more randomness in our training data.

3.1.3 `main.py`

This file implements the core functionality, i.e. the train- and evaluation-function, which is described in section 3.2. Besides that, the model is quantised after the training process as described in section 3.4.
We also provide a method to calculate mean and standard deviation for a new dataset. Since this is already hardcoded into our custom dataloader, it currently remains unused in the `main.py`.

3.1.4 `predictImages.py`

Run this file to step through the evaluation-split with a visualization of the current batch and corresponding predictions.

3.2 Training

As mentioned before, we strictly followed the suggested training- and evaluating-structure for a classifier by PyTorch. The only exception to this is that we added a softmax function to the model's output in order to let the sum of the confidences be equal to one.

3.2.1 Parameters

For all of our experiments, we have used the parameters as listed in figure 2.

Parameter	Value
`Learning rate`	0.0001
`Batch size`	16
`Epochs`	20
`Loss criterion`	`torch.nn.CrossEntropyLoss`
`Optimizer`	`torch.optim.Adam`

Figure 2: Caption

Model	Accuracy [%]
AlexNet	91.09
GoogLeNet	99.14
MobileNetV2	52.59
ResNet18	100.0
ResNet50	75.86

Figure 3: Results on the defined evaluation-split

3.2.2 Choosing a model

Since image classification is a well solved task in machine learning, we relied on the large collection of the prebuilt models in pytorch and tested multiple architectures. The percentage of correct predictions serves as evaluation metric for all models.
As shown in figure 3, `ResNet18` as well as `GoogLeNet` achieve extraordinary accuracy on our evaluation-split. Since the training-time for `ResNet18` is roughly half of the training-time for `GoogLeNet`, we consider the architecture of `ResNet18` to be less complex and therefore more suitable for deployment on the Waveshare Jetracer.

3.3 Training: Usage

In order to train a model by yourself, just follow these steps:

1. In `dataloader.py`, set `dataset_path` to the path of the RoadSignSet. Per default, you can just place it into the top-level directory.
2. [*Optional*] In `dataloader.py`, change `model_path` to your desired location.
3. [*Optional*] In `dataloader.py`, select your desired `model_class` and comment the others
4. [*Optional*] In `dataloader.py`, change and experiment with different training parameters.
5. Run `main.py`

3.4 Quantisation

"Quantization is the process of constraining an input from a continuous or otherwise large set of values (such as the real numbers) to a discrete set (such as the integers)." [Wikipedia][9] In the context of deep learning quantisation is used to improve model memory usage and run-time performance by utilizing integer scale data-types (e.g. INT8) instead of floating point data-types (e.g. FP32). Especially in mobile robotics quantization could play a role to enable embedded devices with limited memory and computing performance to use machine learning models with reasonable performance. For the JetRacer device in particular, quantization could for example be utilized to lower the latency between capturing and classifying images from the camera in addition to running more powerful, but more memory-intensive models on the device.

3.4.1 Quantisation Methods

Three different quantization methods exist and are provided by the framework PyTorch and differences between those as described as follows in the corresponding PyTorch tutorial [5]:

1. Dynamic quantization: Weights quantized with activations read/stored in floating point and quantized for compute.
2. Post-static quantization: Weights quantized, activations quantized, calibration required post training
3. Quantization-aware training: Weights quantized, activations quantized, quantization numerics modeled during training

It should be noted that not all operators in neural networks available in PyTorch are supported throughout all quantization methods, as seen in Figure 4:

	Static Quantization	Dynamic Quantization
nn.Linear	Y	Y
nn.Conv1d/2d/3d	Y	N
nn.LSTM	Y (through custom modules)	Y
nn.GRU	N	Y
nn.RNNCell	N	Y
nn.GRUCell	N	Y
nn.LSTMCell	N	Y
nn.EmbeddingBag	**Y (activations are in fp32)**	**Y**
nn.Embedding	**Y**	**N**
nn.MultiheadAttention	**Y (through custom modules)**	**Not supported**
Activations	**Broadly supported**	**Un-changed, computations stay in fp32**

Figure 4: Supported operators using quantization methods in PyTorch framework. Source: [5]

In particular, the missing support for convolution using dynamic quantization means that convolutional neural networks, like ResNet-18 will not be supported by using this method.

Quantization of models in PyTorch can be carried out using two different backends: 1) QNNPack (Quantized Neural Networks PACKage) [8] 2) FBGemm (Facebook GEneral Matrix Multiplication) [2]. The backends are platform-dependent and can only be used on ARM-devices (QNN-Pack) and x86-devices (FBGemm) respectively.

3.4.2 Application of Quantization

While the tutorials and code samples found on the PyTorch website suggest a seamless mostly automatic quantization procedure, we found the process not as straightforward as it seemed like and encountered multiple issues:

1. Using the QNN-Pack (ARM) backend we found that no modifications were applied to the model. The size of the model (44.5 MB) even remained the same after quantization.

2. While the quantization process could be executed without any issues, running the quantized models themselves proved to be difficult. Out-of-the box none of the models could be executed, the reason for that is because the applied modifications to the model architecture were either not

sufficient (Missing quantisation/dequantization on input/output layer) or incompatible (certain operators e.g. add have to be patched into the model manually).

3. Loading the quantized models, once saved, posed a challenge due to lacking documentation. To load a quantized model one must instantiate an empty uninitialized regular model and run through the quantization process again to regain the changes made to the model architecture during quantization, before subsequently loading the weights '`state_dict`' saved from a prior quantization. While conceptually clear, the lack of documentation, and un-intuitive error messages caused many trial and error attempts.

With regard to issue 2) we discovered a tutorial which explains how to modify the Res-Net50 model to be compatible with quantization [3]. In particular, the part where a QuantizationStub, and DequantizationStub are inserted in the input-/output-layer was missing from the models created by the aforementioned automatic quantization procedures from PyTorch. Those parts had to be manually created in a wrapper class built around the regular Res-Net18 class. Furthermore, as described in the blog-post [3], layer fusion was applied to fuse together layers where there was no direct quantizable equivalent. In particular this concerned the layer `BatchNormalization` which was fused together with `convolution` and the `relu` layer.

The following attempts were also made to resolve the issue, but ultimately failed in the end:

1. PyTorch itself provides an implementation of a quantized ResNet18 model under `https://pytorch.org/vision/main/models/resnet_quant.html` which is similar to the model created in the tutorial and available in the PyTorch framework. However, while running quantization using that class an exception is thrown suggesting that the method `get_parameters()` was not implemented by the class.

2. In addition, NVIDIA provides a library called `PyTorch-Quantization` [7] which aims to automatically swap out operators for their quantized equivalents. However, similar to PyTorch's internal implementation, models created by the library could not be executed.

Method	Model size	Creates model	Runnable model
Dynamic quantization (FBGemm)	44.8MB	✓	- (Erroneous, No quantization applied)
Post-Static quantization (FBGemm)	11..4MB	✓	✗
Quantization-Aware-Training (FBGemm)	11.4MB	✓	✗
Post-Static Quantization (FBGemm + Manual model modification)	11.3MB	✓	✓
Dynamic quantization (QNNPack)	44.8MB	✓	- (Erroneous, No quantization applied)
Post-Static quantization (QNNPack)	-	✗	✗
Quantization-Aware-Training (QNNPack)	-	✗	✗
Various Methods (FBGemm + Nvidia PyTorch-Quantization library)	Same as above	✓	✗

3.4.3 Usage: Quantization

1. Please make sure that the folder `weights` exists and the non-quantized model is placed there. Otherwise, please follow the steps in Training: Usage to train and create the model prior.

2. Please make sure that your device is an x86-device and running the CPU version of PyTorch, since we have found other platforms unfortunately non-working, in particular ARM-devices.

3. Run `quantization.py`

4. The script will create quantized models using the three different quantization methods above and save them as: 1) 'dynamic.zip' for Dynamic quantization 2) 'static.zip' for Post-static quantization. 3) 'qat.zip' for Quantizaton-aware training 4) 'static_fused.zip' for Post-static quantization where the model architecture was manually customized.

3.4.4 Optional Task: Profiling

Out of curiosity, we profiled both the quantized, and non-quantized models. We generated a randomized tensor (n x c x h x w) of size (100x3x128x128) with the parameters denoting: n=number of images in the image series, c = channels, h = height of image, w=width of image. The results for the non-quantized model can be seen in Figure: 5 and for the quantized model in Figure: 6.

Contrary to the expected result of quantized models achieving better run-time performances, we observed that the non-quantized version of the model performed worse than it's regular counterpart.

However, when considering the profiling setup it must be acknowledged that this profiling result may not be representative, since a randomized image is loaded into the model instead of real images. As quantization aims to approximate a wide value range using lower precision data types, often the value range is capped. Hence, it is not surprising that the quantized model leads to worse results since it may be optimized to work with images and value ranges resembling closest to the training set. Unfortunately, due to time-constraints we have not repeated the profiling experiment for real images which may have turned the results around.

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU avg	# Calls
model_inference	5.72%	242.661ms	100.00%	4.239s	4.239s	1
aten::conv2d	0.00%	113.0µs	69.54%	2.948s	147.387ms	20
aten::convolution	0.01%	423.0µs	69.54%	2.948s	147.381ms	20
aten::_convolution	0.01%	226.0µs	69.53%	2.947s	147.360ms	20
aten::mkldnn_convolution	69.51%	2.946s	69.52%	2.947s	147.349ms	20
aten::batch_norm	0.00%	75.0µs	12.74%	540.234ms	27.012ms	20
aten::_batch_norm_impl_index	0.01%	229.0µs	12.74%	539.960ms	26.998ms	20
aten::native_batch_norm	12.72%	539.332ms	12.74%	539.960ms	26.998ms	20
aten::max_pool2d	0.00%	9.000µs	7.16%	303.634ms	303.624ms	20
aten::max_pool2d_with_indices	0.00%	303.625ms	7.16%	305.908ms	303.624ms	20

Figure 5: Profiling: Non-quantized ResNet18 model: (Total time: 4.239s)

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU avg	# Calls
model_inference	9.59%	492.251ms	100.00%	5.136s	5.136s	1
aten::conv2d	0.00%	140.0µs	62.60%	3.215s	160.755ms	20
aten::convolution	0.01%	512.0µs	62.60%	3.215s	160.748ms	20
aten::_convolution	0.01%	266.0µs	62.59%	3.214s	160.722ms	20
aten::mkldnn_convolution	62.57%	3.213s	62.58%	3.214s	160.709ms	20
aten::batch_norm_impl_index	0.00%	91.0µs	11.37%	584.114ms	29.206ms	20
aten::_batch_norm_impl_index	0.01%	281.0µs	11.37%	584.023ms	29.201ms	20
aten::native_batch_norm	11.35%	582.953ms	11.36%	583.686ms	29.184ms	20
aten::relu	0.01%	561.000µs	6.08%	312.454ms	18.380ms	17
aten::clamp_min	7.09%	322.893ms	7.09%	311.893ms	18.347ms	17

Figure 6: Profiling: Quantized ResNet18 model: (Total time: 5.136s)

3.5 Profiling: Usage

To profile the models yourself, follow these steps:

1. The script will create profiling reports for the regular model and the quantized model. Please make sure that those models exist in the folder 'weights' before running this script. If not, please following the steps to train, and quantize your models prior. (Segments: Training: Usage, and Quantization: Usage)
2. Make sure that the folder `weights` exists. If not, the script will create an empty folder.
3. Make sure that the folder `profiling` exists. If not, the script will create an empty folder.
4. Run `profiling.py`.
5. The script will generate the reports in the folder `profiling`, and create a .txt files for each model named after the class name.

4 Real-World Experiment

After training the models on our devices at home, the real-world experiment was conducted in the lab environment on Fri, 18^{th} February 2023. We deployed the best-performing model (ResNet-18) on the JetRacer device, and set up the evaluation environment in both novel and familiar scenarios.

1) First, a novel setup was chosen where the JetRacer was placed on a desk with sun exposure from the windows situated nearby the desk. Neither the location, and the novel environment condition was encompassed by the data collected in the training set. Therefore, the camera covered a never-before-seen background environment. We positioned the signs 'Turn left' and the 'Speed limit 50' in front of the JetRacer camera to evaluate the model. We observed that the model miss-classified the signs in the vast majority of the cases independent of the positional distance, or placement of the traffic sign. This points to our model lacking robustness when encountering novel situations. Furthermore, following characteristics could be observed:

- *A* behaviour of classifying the 'STOP' sign as a default, when either no sign was placed, or the confidence of the other classes was low.
- *T*he pictures feeding from the camera of the JetRacer device were transmitted with a significant delay.

Issues regarding the first behaviour can be explained, due to our training data not containing any training data of situations without any signs, and thus does not contain the class 'no signs'. This issue can be resolved by introducing the aforementioned class and the accompanying training data, or using an approach with multiple models, where an upstream model ensures that our model is only feed with situations where road signs are detected. With regard to the issue of the delay encountered feeding the model, we resolved it by encoding the timestamp in the filename of the saved images. While this solution still incurs a time-delay between the observation and classification of the situation, it is sufficient for our evaluation process.

2) In our second scenario we wanted to create an environment with more similar environmental characteristics to the scenarios when capturing the training data. We pulled down the sun-shades of the nearby windows and turned on indoor lighting. However, the improvement was only slight. We were able to classify the 'Turn left' signs in many cases. However, we could observe an 'trailing effect' where empty frames were classified as 'Turn left' sign, even after removing the sign for a time-span of about 10s-30s. Upon more detailed inspection of the confidence scores, we were only observe a confidence score of around 0.5 which points to a very low confidence, but still sufficient to entail a classification. With regard to the 'Speed limit 50' sign, we were only able to classify it in occasional situations. We obtained a high amount of miss-classifications of the other classes when using this object.

3) In our third scenario we positioned the JetRacer to a position which was used during the collection of the training data. Thus, we wanted to recreate the original environment conditions as possible. For that we even tried to recreate the pole which was used to attach the signs by using a slice of white-colored paper. However this factor was insignificant and did not improve the results. Even in this scenario our model was only able to distinguish the different classes with a low-level confidence. We hence can assume that our model is likely over-fitted and has learned unnecessary features (such as repetitive patterns from the background, noise patterns...) from the training data unrelated to the traffic signs.

With regard to the signs 'Turn right', and the 'Stop' sign they performed equally 'well' with a low level of confidence, and are likewise prone to miss-classifications in both scenarios 2 and 3, with the experiment never carried out in the first scenario due to the bad performance.

5 Conclusion

While our model (ResNet-18) performed excellently on our evaluation-set, the deployment on the device and testing in real-world scenarios has revealed that our model was under-performing under those testing conditions. A more robust training process using randomized data-set shuffling, in addition to a more stringent data gathering process which encompasses a more systematic positional image

capturing to create a balanced training set, may have alleviated those issues. In addition image-augmentation could be applied to insert a higher degree of randomness with regard to the background of the images captured in the training-set.

References

[1] Waveshare Jetracer AI Kit. https://www.waveshare.com/jetracer-ai-kit.htm. [Online; accessed 21. Feb. 2023].

[2] Github: FBGemm. Github: Fbgemm, howpublished = "https://github.com/pytorch/FBGEMM", note = "[online; accessed 2. march. 2023]".

[3] Lei Mao. Pytorch static quantization, howpublished = "https://github.com/pytorch/FBGEMM", note = "[online; accessed 2. march. 2023]".

[4] PyTorch. ColorJitter. https://pytorch.org/vision/main/generated/torchvision.transforms.ColorJitter.html. [Online; accessed 21. Feb. 2023].

[5] PyTorch. Introduction to quantization, howpublished = "https://pytorch.org/docs/stable/quantization.html", note = "[online; accessed 2. march. 2023]".

[6] PyTorch. Training a classifier. https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html. [Online; accessed 21. Feb. 2023].

[7] PyTorch-Quantization. Pytorch: Quantization, howpublished = "https://docs.nvidia.com/deeplearning/tensorrt/pytorch-quantization-toolkit/docs/userguide.html", note = "[online; accessed 2. march. 2023]".

[8] Github: QNN-Pack. Github: Qnn-pack, howpublished = "https://github.com/pytorch/QNNPACK", note = "[online; accessed 2. march. 2023]".

[9] Wikipedia. Quantization definition, howpublished = "https://en.wikipedia.org/wiki/Quantization", note = "[online; accessed 2. march. 2023]".