Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Instytut Radioelektroniki i Technik Multimedialnych

Praca dyplomowa

na studiach: Studium Podyplomowe
Głębokie sieci neuronowe - zastosowania w mediach cyfrowych

Lightweight neural architectures by example - car model classification using GhostNet

Piotr Chaberski

opiekun pracy:
Xin Chang

WARSZAWA 2020

Lightweight neural architectures by example - car model classification using GhostNet

Abstract. This is the abstract!

Keywords: XXX, XXX, XXX

Contents

1. Introduction

1.1. Problem background

One of the ongoing directions of deep learning research in computer vision and image recognition (but not only) is related to the reduction of neural network size and the number of operations needed for inference while preserving good level of performance in terms of classification accuracy or other metrics. The one important reason to do so is to reduce training times and costs and speed up the iterative process od hyperparameter optimization. Another drawback of large networks in some applications can be extensive overfitting. But the most important reason justifying the search for more efficient neural architectures is that in many practical applications models are needed to be deployed not on a multi-GPU servers or on cloud, but rather as a part of embedded systems on devices with very limited computational power and memory like smartphones, car systems or other devices with so-called intelligent modules.

At the time of completing this work there is already a significant number of different propositions of architectures aiming to reduce the number of parameters and FLOPS needed to efficiently perform image classification tasks [1]. Those architectures are most commonly trained on ImageNet (http://www.image-net.org/) and, among others, two metrics are reported on this dataset: accuracy and FLOPS, along with the total number of parameters. Those values give an impression about architecture efficiency in terms of trade-off between prediction quality, inference speed and required memory. Some, but definitely not all, of the successful implementations are:

- SqueezeNet (2016) [2]
- MobileNet (V1: 2017, V2: 2018, V3: 2019) [3][4][5]
- SqueezeNext (2018) [6]
- ShuffleNet (2018) [7][8]

- EfficientNet (2019) [9]
- HarDNet (2019) [10]
- GhostNet (2020) [11]

Most of these architectures come with different customizable variants. For example, EfficientNet has 8 different basic configurations (named `b0` to `b7`) that differ in terms of complexity. Others, like MobileNet, were reworked and upgraded resulting in different versions (there are currently three versions of MobileNets, named simply `V1`, `V2` and `V3`).

The above-mentioned architectures are capable of achieving good accuracy scores with very limited number of parameters and floating point operations required. For example, EfficientNet-b0 has 77.1% accuracy on ImageNet with 5.3 M parameters and 0.39 GFLOPS. GhostNet gets 73.98% accuracy with only 4.1 M parameters and 0.142 GFLOPS. On the contrasts, ResNet-50 to achieve 75.3% accuracy requires 25.6 M parameters and 4.1 GFLOPS to process the image of the same size (224x224 RGB).

1.2. Problem statement

This project is a part of a broader conception to create a mobile application to recognize car models from pictures taken by the users. The initial idea was to:

1. Pick some of the efficient mobile architectures (the project was intended to be carried out in a group), train them on a open dataset of car images and compare in terms of accuracy, model size and FLOPS.
2. Prepare custom dataset of images taken and labelled personally, finetune the best model from step 1 to reflect car models distribution on the streets of Poland.
3. Prepare model for deployment, create a simple Android application that allow to take a picture and recognize a car model.

This work focuses only on step one with selected architecture. Specifically, it describes the process of training and optimizing hyperparameters of GhostNet [11] model using Stanford Cars Dataset [12] to check the performance of this particular novel mobile architecture in a car model recognition task.

2. Project description

2.1. Stanford Cars Dataset

Stanford Cars Dataset [12] is a dataset published by Jonathan Krause of Stanford University and is publicly available at https://ai.stanford.edu/~jkrause/cars/car_dataset.html.

Figure 1: Example images from Stanford Cars Dataset

The dataset contains 16,185 images of 196 classes of car models (precisely, class label contains information about make, model and production year of a car). Dataset has been splitted with stratification into two parts:

- 8,144 images as a training set
- 8,041 images as a test set

In addition to class labels, both subsets have also bounding boxes (as 4 coordinates in metadata files).

Images are originally of different sizes, mostly in RGB, but there are some grayscale images which has to be taken into account during preprocessing. Another thing to be aware of is that the dataset has been updated at some point - the images and the split did not change, but the file names were reordered and metadata was reorganized for the ease of use.

2.2. GhostNet architecture

GhostNet [11] is the architecture designed and first implemented by the research team at Huawei Noah's Ark Lab (http://www.noahlab.com.hk/). It is based on the observation, that standard convolutional layers with many filters are large in terms of number of parameters and computationally expensive, while often producing redundant feature maps that are very much alike each other (they might be considered as "ghosts" of the original feature map). The goal of the GhostNet design is not to get rid of those redundant feature maps, because they often help the network to comprehensively understand all the features of the input data. Instead of that, the focus is on obtaining those redundant feature maps in a cost-efficient way.

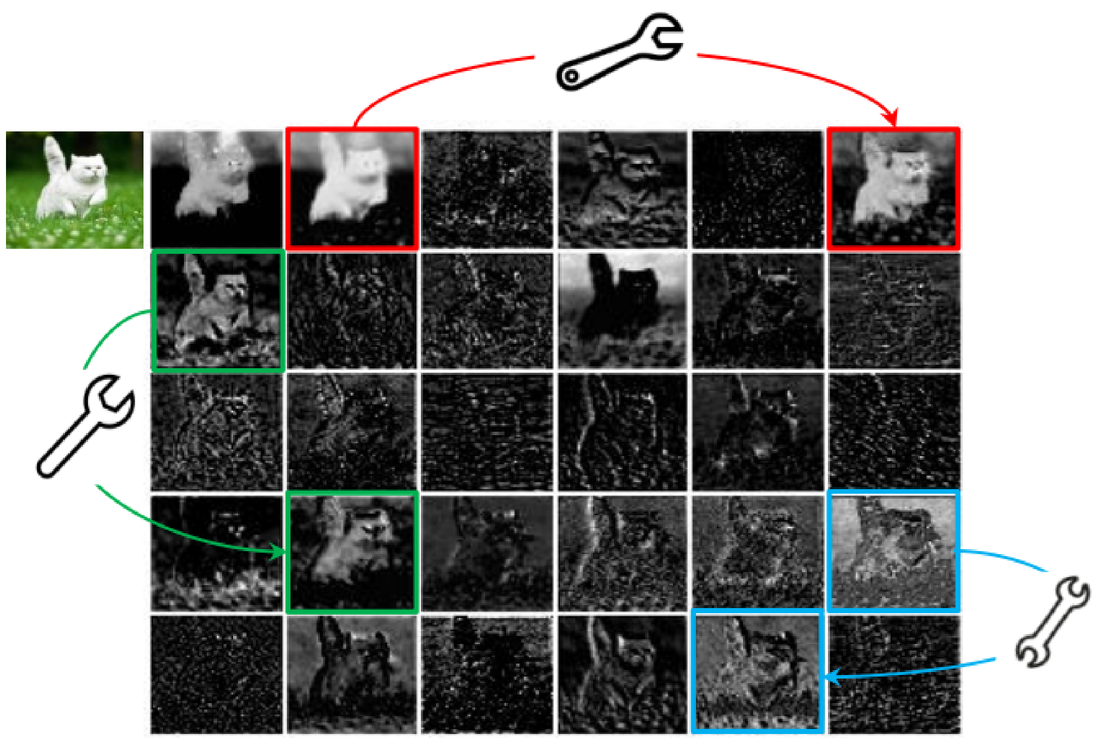

Figure 2: Redundant feature maps from ResNet-50 (picture from paper)

This cost-efficiency in creating feature maps is achieved by introducing GhostModule, namely splitting standard convolutional layer with many filters into two parts. The first part, still being a standard convolutional layer but with less filters, produces a set of base feature maps. Then the second part, by applying cheap linear operations, produces redundant feature maps from the original set (so-called "ghosts"). In the end, the outputs of the first and the second part are concatenated.

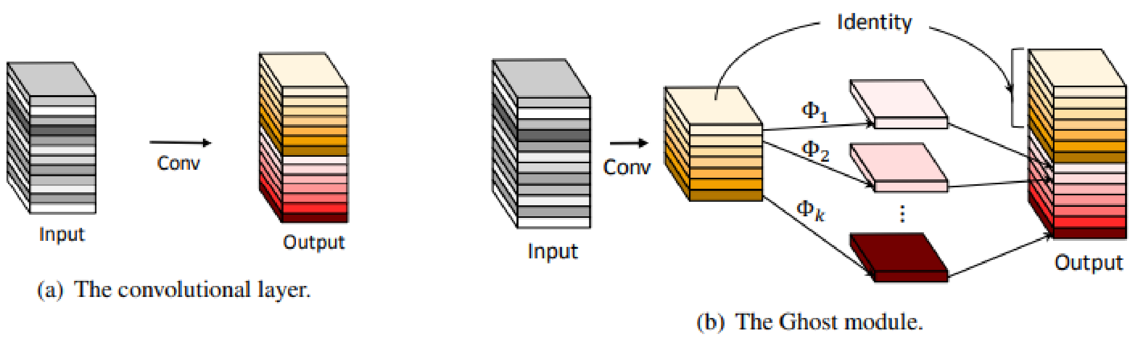

Figure 3: Comparison of standard convolution (a) and GhostModule (b) (picture from paper)

The above mentioned cheap linear operations are implemented using depthwise convolutions [13] (although other options like affine or wavelet transforms were also tested by the authors). With this assumption, GhostModule can be implemented in PyTorch as follows:

```
class GhostModule(nn.Module):
    def __init__(
        self, inp, oup,
        kernel_size=1, ratio=2, dw_size=3, stride=1,
        relu=True
    ):
        super().__init__()
        self.oup = oup
        init_channels = math.ceil(oup / ratio)
        new_channels = init_channels*(ratio-1)

        self.primary_conv = nn.Sequential(
            nn.Conv2d(
                inp, init_channels, kernel_size, stride,
                kernel_size//2, bias=False
            ),
            nn.BatchNorm2d(init_channels),
```

```
            nn.ReLU(inplace=True) if relu else nn.Sequential(),
        )

        self.cheap_operation = nn.Sequential(
            nn.Conv2d(init_channels, new_channels, dw_size, 1,
            dw_size//2, groups=init_channels, bias=False
        ),
            nn.BatchNorm2d(new_channels),
            nn.ReLU(inplace=True) if relu else nn.Sequential(),
        )

    def forward(self, input):
        output_1 = self.primary_conv(input)
        output_2 = self.cheap_operation(output_1)
        output = torch.cat([output_1, output_2], dim=1)
        return output[:, :self.oup, :, :]
```

Two GhostModules combine for a basic building block of GhostNet - the GhostBottleneck, which is based on the concept taken from MobileNet-V3 design [5] (additionally, in some GhostBottlenecks, similarly to MobileNet-V3, Squeeze-and-Excitation modules are used [14]). The first GhostModule in a GhostBottleneck expands the number of channels, while the second one, after ReLU, reduces them again. There is also a residual connection over the two GhostModules. GhostBottleneck has also strided version (with `stride=2` depthwise convolution between GhostModules) which is applied at the end of each stage of GhostNet.

To form up the entire GhostNet architecture several GhostBottlenecks are combined in a sequence which is followed by global average pooling and a convolution which transforms feature maps to the feature vector of length 1280. This feature vector, after dropout layer, is then transformed with a fully connected layer to the size of output number of classes.

GhostNet architecture based on paper:

Input	Operator	#exp	#out	SE	Stride
224 x 224 x 3	Conv2d 3x3	-	16	-	2
112 x 112 x 16	G-bneck	16	16	-	1
112 x 112 x 16	G-bneck	48	24	-	2

Input	Operator	#exp	#out	SE	Stride
56 x 56 x 24	G-bneck	72	24	-	1
56 x 56 x 24	G-bneck	72	40	1	2
28 x 28 x 40	G-bneck	120	40	1	1
28 x 28 x 40	G-bneck	240	80	-	2
14 x 14 x 80	G-bneck	200	80	-	1
14 x 14 x 80	G-bneck	184	80	-	1
14 x 14 x 80	G-bneck	184	80	-	1
14 x 14 x 80	G-bneck	480	112	1	1
14 x 14 x 112	G-bneck	672	112	1	1
14 x 14 x 112	G-bneck	672	160	1	2
7 x 7 x 160	G-bneck	960	160	-	1
7 x 7 x 160	G-bneck	960	160	1	1
7 x 7 x 160	G-bneck	960	160	-	1
7 x 7 x 160	G-bneck	960	160	1	1
7 x 7 x 160	Conv2d 1x1	-	960	-	1
7 x 7 x 960	AvgPool 7x7	-	-	-	-
1 x 1 x 960	Conv2d 1x1	-	1280	-	1
1 x 1 x 1280	FC	-	1000	-	-

GhostNet architecture described above (and in original paper as well) is the basic setup which can be modified by structuring GhostBottlenecks in different sequences. This basic setup, as mentioned before, gets 73.98% accuracy on ImageNet with 4.1 M parameters and requires only 0.142 GFLOPS to process 224x224 RGB image. Other more complex variations, as presented in paper, show superiority over previous designs like MobileNet or ShuffleNet getting better accuracy with less FLOPS and latency.

Full PyTorch implementation of GhostNet that was used in this work is available at GitHub repository of the project.

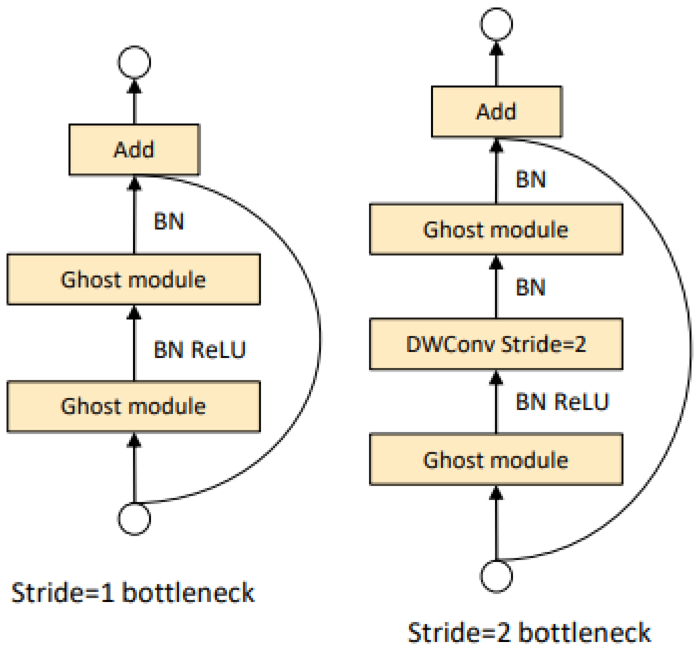

Figure 4: GhostBottleneck (picture from paper)

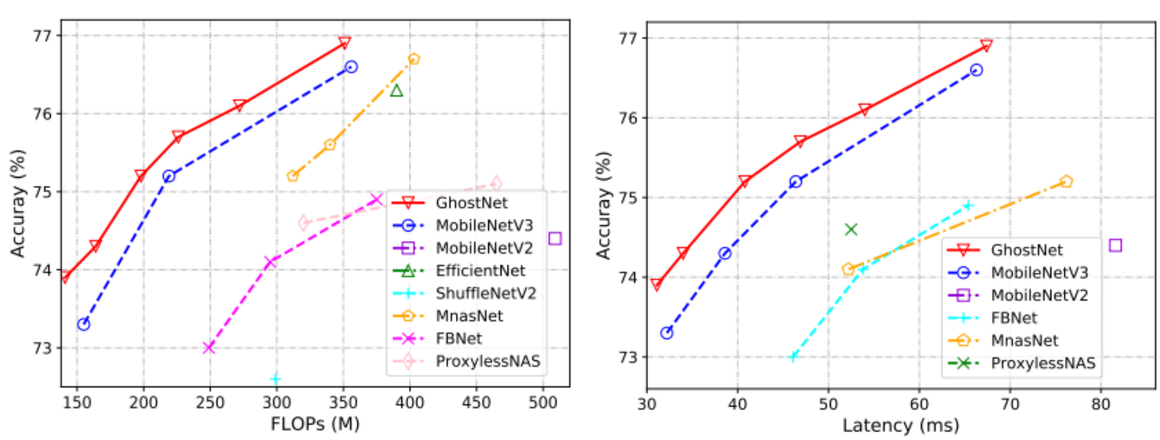

Figure 5: GhostNet comparison with some other mobile architectures (pictures from paper)

3. Experimentation setup

The experimentation setup is entirely based on Python. GhostNet (and some other networks, which also can be used) implementation is written in PyTorch. Training process is orchestrated using `pytorch-lightning` package and controlled by parameters passed through YAML config file. Neptune experiment management tool (https://neptune.ai/) was used for experiment tracking. To build an environment for data preparation and model training Python `virtual env` utility was used. In addition to local training setup there is also a possibility to recreate the project environment and run training on Google Colab platform using a prepared Jupyter Notebook.

3.1. Project structure

Source code for the project is available in GitHub repository: https://github.com/pchaberski/cars. Repository contains the following elements:

- `documentation` - folder containing markdown files with project documentation, images, bibliography as a `.bib` file and some tools for document conversion
- `datasets` - Python package containing:
  - `stanford_data.py` module implementing class for Stanford Cars data loading and preprocessing
  - `stanford_data_module.py` - module implementing `LightningDataModule` defining data loaders for main training `LightnigModule`
  - `stanford_utils.py` - utility to process raw files downloaded from dataset webpage to be suitable for training and validation
- `models` - Python package containing:
  - `architectures` - folder with modules implementing `GhostNet` and several other architectures that were briefly tested during initial stage of the project (`SqueezeNet`, `SqueezeNext`, `EfficientNet`, `MobileNet-V2`, `ShuffleNet`,

  HarDNet)
  - `arch_dict.py` - module with a dictionary of architectures that can be used in experiments
  - `net_module.py` - module containing main `LightningModule` used for network training and evaluation
  - `label_smoothing_ce.py` - implementation of Label Smoothing Cross Entropy loss function [15]
- `utils` - Python packages with utilities for configuration parsing, logging and execution time measurement
- `config_template.yml` - YAML configuration file template; it is supposed to be filled and saved as `config.yml` to allow controlling training settings (mostly data preprocessing settings and model hyperparameters) without interference with source code
- `prod_requirements.txt` - list of external PyPI Python packages to be included in `virtual env` to run the training
- `dev_requirements.txt` - list of additional PyPI Python packages that were used during development and results postprocessing
- `prepare_stanford_dataset.py` - executable Python script that prepares raw files from dataset website to the form suitable for training and validation
- `train.py` - main executable Python script for running experiments
- `train_colab.ipynb` - Jupyter Notebook that can be used to recreate local working environment on Google Colab and run `train.py` remotely

3.2. Working environment

Project structure allows to run experiments in two modes, also simultaneously:

- locally on a machine with GPU and CUDA drivers
- remotely on Google Colab

Local setup was tested on Windows laptop (although experimentation environment should be also reproducible on Linux with no changes in the project) with mobile GeForce RTX 2060 and Python 3.7.6. Google Colab setup recreates the environment to mirror all local package versions and runs on Python 3.6.9, however no compatibility issues were observed.

The first step to prepare for running experiments is to **clone the project GitHub repository**. If the training is to be performed on Colab, project folder should be cloned into `Google Drive` folder that is synchronized with remote Google Drive directory. This will allow to to sync all local changes on the fly and run Colab training without the need of pushing all changes made locally to git remote origin each time and then pulling them on Colab drive.

Before running data preprocessing or local training, the Python environment has to be prepared. It is advised to recreate the environment using Python `virtual env` utility and `prod_requirements.txt` file attached to project repository (using Anaconda is also an option). To do so, the following steps has to be performed using `cmd` or emulated `bash` on Windows or native `bash` on Linux:

Using `cmd` on Windows:

```
:: Go to the project directory that was cloned from GitHub
> cd C:\Users\username\Google Drive\cars

:: Create Python virtual env in some other directory
:: (different than Google Drive to prevent constant syncing of new packages)
> python -m venv C:\projects\venvs\cars

:: Activate newly created virtual env
> C:\projects\venvs\cars\Scripts\activate.bat

:: Install dependencies from prod_requirements.txt file
:: (explicitely pointing to PyTorch repository)
(cars) > pip install -r prod_requirements.txt -f ^
https://download.pytorch.org/whl/torch_stable.html
```

Using `bash` on Linux:

```bash
# Go to the project directory that was cloned from GitHub
$ cd ~/Google Drive/cars

# Create Python virtual env in some other directory
# (different than Google Drive to prevent constant syncing of new packages)
$ python -m venv ~/venvs/cars

# Activate newly created virtual env
$ ~/venvs/cars/bin/activate
```

```
# Install dependencies from prod_requirements.txt file
(cars) $ pip install -r prod_requirements.txt
```

To allow data loaders to process data during training, **raw files have to be preprocessed** using `prepare_stanford_dataset.py` script. It takes three files downloaded from Stanford Cars website, assuming they are stored in a directory passed through `stanford_raw_data_path` parameter of the configuration file (please see section 3.3 for details):

- `car_ims.tgz` - updated collection of train and test images
- `cars_annos.mat` - updated train and test labels and bounding boxes
- `car_devkit.tgz` - original devkit containing class names

The script processes the above-mentioned raw files to obtain:

- `train` and `test` folders with images used for training and validation, separated for the ease of data loaders implementation
- `train_labels.csv` and `test_labels.csv` files with image names and class numbers associated with them, as well as bounding box coordinates and class names. It is important to notice, that in raw data class are numbered within range of 1 to 196, while PyTorch Lightning requires classes to be represented by numbers starting from 0. This issue is handled internally within `StanfordCarsDataset` class and has to be taken into account during interpretation of model predictions.

Preprocessed images and metadata are saved within the directory pointed by `stanford_data_path` configuration parameter (by default, `input/stanford` folder is created within project folder). **If the training is supposed to be run on Colab** it is strongly advisable to prepare also a `.tar.gz` archive (e.g. `stanford.tar.gz`) from `train`, `test`, `train_labels.csv` and `test_labels.csv` and put it on Google Drive. This will allow to quickly copy and unpack the the data from Google Drive to Colab drive before training which will speed up data loading, and therefore training multiple times, as reading image by image from Google Drive takes incomparably more time than reading directly from Colab drive.

After cloning the repository and preparing the data (also creating and filling up `config.yml` from `config_template.yml` as described in 3.3) it is possible to run experiments.

To run experiment locally, after setting all parameters in `config.yml`, `virtual_env` has to be activated and `train.py` has to be run from command line using `python`.

To run experiment on Colab, after making sure that project files and data is put on Google Drive, `train_colab.ipynb` notebook has to be opened. In the first cell there are some additional Colab-specific parameters to be set:

- `colab_google_drive_mount_point` - where the Google Drive is to be mounted on Colab drive
- `colab_remote_project_wdir` - working directory for remote project - should point to `cars` project folder
- `local_project_wdir` - can be ommited if running on Colab, however notebook will also work locally if correct local path to `cars` project folder is provided
- `DATA_ON_COLAB` - if `True`, images and labels are copied and unpacked before training from Google Drive to Colab drive, assuming that they are originally stored at `$colab_remote_project_wdir/input/stanford.tar.gz`
- `colab_data_dir` - where to unpack data copied from Google Drive

After setting all above paths, the notebook is designed to: - check if session is running on Colab runtime - recreate local environment by installing packages from `prod_requirements.txt` on Colab (after this step, runtime restart might be needed to reload new versions of packages) - copy and unpack data from to Google Drive to Colab drive if `DATA_ON_COLAB=True` - run `training.py` script on Colab

If project folder is stored on Google Drive, regardless the runtime used (Colab or local), all outputs and logs are stored in the same place, which allows to run up to three simultaneous experiments (two Colab sessions plus one local session).

3.3. Configuration

All experiments are controlled using `config.yml` file stored in `cars` project folder. This allows to change all experiment-related parameters without any interference in the source code. Initially, after cloning, the repository default settings are stored in `config_template.yml` file. This file has to be copied and renamed as `config.yml`. Configuration file contains parameters related to:

- logging - locally and using Neptune experiment tracking tool (see section 3.4)
- directories where data and outputs (PyTorch lightning model checkpoints) are stored
- image preprocessing and augmentation settings
- network hyperparameters
- optimizer and loss function settings

Before running the training all directory-related settings has to be provided. As for the data preprocessing and modelling settings, `config_template.yml` already contains all parameter values that were used during training the best model achieved in experiment series.

Full contents of `config_template.yml` are listed below:

```yaml
# Logging settings:
loglevel: 'INFO'
logging_dir: 'logs'
log_to_neptune: False
neptune_username: '<neptune.ai username>'
neptune_project_name: '<neptune.ai project name>'
neptune_api_token: '<neptune.ai API token>'

# Train/test dataset and devkit location
stanford_raw_data_path: '<path to the folder containing: \
car_ims.tgz, cars_annos.mat, car_devkit.tgz>'
stanford_data_path: 'input/stanford'

# Output settings
output_path: 'output'

# General data preprocessinng settings
image_size: &img_size [227, 227]  # Anchor to use in augmentations if needed
convert_to_grayscale: False
normalize: True
normalization_params_rgb:  # Applied when 'convert_to_grayscale==False'
  mean: [0.4707, 0.4602, 0.4550]
  std: [0.2594, 0.2585, 0.2635]
normalization_params_grayscale:  # Applied when 'convert_to_grayscale==True'
  mean: [0.4627]
  std: [0.2545]

# Training data augmentation settings
crop_to_bboxes: True  # crop training images using bounding boxes
erase_background: True  # erase background outside bboxes to preserve ratios
```

```
                        # (only if 'crop_to_bboxes==True')
augment_images: True
image_augmentations:  # to be applied consecutively
  RandomHorizontalFlip:  # has to be a valid transformation
                         # from 'torchvision.transforms'
    p: 0.5  # transformation parameters to be passed as '**dict'
  RandomAffine:
    degrees: 25
    translate: [0.1, 0.1]
    scale: [0.9, 1.1]
    shear: 8
  ColorJitter:
    brightness: 0.2
    contrast: 0.2
    saturation: 0.2
    hue: 0.1
augment_tensors: True
tensor_augmentations:  # to be applied consecutively
  RandomErasing:
    p: 0.5
    scale: [0.02, 0.25]

# Network and training settings
architecture: 'ghost'  # Possible options in 'models.arch_dict'
batch_size: 64
num_epochs: 200

# Architecture modifications (right now GhostNet only!)
dropout: 0.2  # dropout rate before the last Linear layer
output_channels: 320  # output channels to be mapped to the number of classes

# Optimizer settings
optimizer: AdamW  # valid optimizer from 'torch.optim'
optimizer_params:
  lr: 0.001
  weight_decay: 0.6
```

```yaml
lr_scheduler: ReduceLROnPlateau  # valid lr_scheduler from 'torch.optim' or None
lr_scheduler_params:  # scheduler parameters to be passed as '**dict'
  factor: 0.1
  patience: 5
  threshold: 0.001
  min_lr: 0.0000001

# Loss function settings
loss_function: LabelSmoothingCrossEntropy  # valid loss function from 'torch.nn'
                                           # or custom LabelSmoothingCrossEntropy
loss_params:  # loss parameters to be passed as '**dict'
```

3.4. Experiment tracking

4. Results

4.1. Best model

4.2. Experiments step-by-step

	experiment description	train_acc	valid_acc
C-1	Baseline (Cross Entropy Loss)	92.49%	8.15%
C-2	Loss function change (Label Smoothing Cross Entropy)	98.89%	9.12%
C-3	Augmentations: horizontal flip, affine	99.45%	11.96%
C-4	Augmentations: horizontal flip, affine, erasing	99.76%	51.92%
C-5	Augmentations: horizontal flip, erasing, color jitter	98.12%	38.08%
C-6	Augmentations: horizontal flip, affine, erasing, color jitter	93.68%	38.68%
C-7	Augmentations: horizontal flip, affine, color jitter	99.73%	54.28%
C-8	Grayscale: no normalization, no augmentations	99.49%	6.58%
C-9	Grayscale: with normalization, no augmentations	97.13%	8.68%
C-10	Grayscale: normalization, best RGB augmentations	7.58%	3.91%
C-11	Training set cropping with bounding boxes	4.36%	3.07%
C-12	Training set cropping + background erasing	99.67%	50.51%
C-13	L2 regularization with AdamW: weight decay = 0.1	99.44%	63.39%
C-14	L2 regularization with AdamW: weight decay = 0.2	98.84%	68.50%
C-15	L2 regularization with AdamW: weight decay = 0.3	95.83%	61.84%
C-16	L2 regularization with AdamW: weight decay = 0.4	95.95%	65.14%
C-17	L2 regularization with AdamW: weight decay = 0.5	90.38%	59.95%
C-18	Dropout rate tests: dropout = 0.1	99.11%	66.90%
C-19	Dropout rate tests: dropout = 0.3	98.62%	67.81%
C-20	Dropout rate tests: dropout = 0.4	96.52%	64.88%

	experiment description	train_acc	valid_acc
C-21	Dropout rate tests: dropout = 0.5	96.28%	66.75%
C-22	Last layer size tests: out channels = 320	97.13%	68.93%
C-23	Last layer size tests: out channels = 640	96.13%	63.13%
C-24	Last layer size tests: out channels = 960	98.23%	64.96%
C-25	Last layer size tests: out channels = 1600	98.99%	63.11%
C-26	Automatic LR scheduling: take #1	99.82%	74.60%
C-27	Automatic LR scheduling: take #2	99.78%	76.20%
C-28	Automatic LR scheduling: take #3	99.83%	75.14%
C-29	Automatic LR scheduling: take #4	99.78%	74.82%
C-30	Controlled LR scheduling: milestones = [28, 48, 68, 88]	80.66%	57.82%
C-31	Controlled LR scheduling: milestones = [36, 56, 76, 96]	95.03%	64.93%
C-32	Controlled LR scheduling: milestones = [44, 64, 84, 104]	98.68%	68.79%
C-33	Controlled LR scheduling: milestones = [52, 72, 92, 112]	99.60%	71.59%
C-36	Weight decay adjustment: weight decay = 0.5	98.84%	79.40%
C-37	Weight decay adjustment: weight decay = 0.3	99.57%	74.44%
C-38	Weight decay adjustment: weight decay = 0.4	99.37%	78.82%
C-39	Weight decay adjustment: weight decay = 0.6	98.67%	82.55%
C-40	Weight decay adjustment: weight decay = 0.7	99.24%	75.12%
C-41	Dropout rate verification: dropout = 0.3	98.49%	82.08%
C-42	Dropout rate verification: dropout = 0.4	95.34%	79.57%
C-43	Dropout rate verification: dropout = 0.5	96.08%	77.87%
C-44	Dropout rate verification: dropout = 0.25	98.79%	82.45%
C-45	Additional augmentations test: resized crop	97.56%	78.73%
C-46	Additional augmentations test: rotation	97.03%	78.25%
C-47	Additional augmentations test: perspective	97.42%	80.22%
C-48	Additional augmentations test: erasing	93.68%	80.56%
C-50	**LR scheduler adjustment: milestones = [67, 82, 95, 107]**	**98.94%**	**83.79%**
C-51	LR scheduler adjustment: milestones = [63, 78, 91, 103]	98.86%	82.54%
C-53	LR scheduler adjustment: milestones = [66, 81, 94, 106]	98.96%	83.02%
C-55	LR scheduler adjustment: milestones = [68, 83, 96, 108]	98.78%	83.72%
C-56	LR scheduler adjustment: milestones = [64, 79, 92, 104]	98.99%	82.79%
C-58	Last layer size sanity check: out channels = 1280	99.44%	78.83%
C-63	LR annealing test: LR geometric sequence	99.80%	70.51%
C-64	LR annealing test: exponentiation base = 0.955	98.49%	60.70%

	experiment description	train_acc	valid_acc
C-65	LR annealing test: exponentiation base = 0.975	99.66%	73.07%
C-66	LR annealing test: exponentiation base = 0.98	98.72%	70.46%

4.2.1. Loss function

4.2.2. Normalization

4.2.3. Augmentations

4.2.4. Grayscale conversion

4.2.5. Bounding boxes utilization

4.2.6. Optimizer change and L2 regularization

4.2.7. Dropout rate tests

4.2.8. Last layer size tests

4.2.9. Automatic learning rate scheduling

4.2.10. Controlled learning rate scheduling

4.2.11. Weight decay adjustment

4.2.12. Dropout rate verification

4.2.13. Additional augmentations tests

4.2.14. Learning rate scheduler adjustment

4.2.15. Last layer size sanity check

4.2.16. Learning rate annealing tests

References

[1]

M. Hollemans. New mobile neural network architectures. Accessed 2020-09-26: https://machinethink.net/blog/mobile-architectures, 2020.

[2]

F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size, 2016. [arXiv]

[3]

A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017. [arXiv]

[4]

M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Jun 2018. [DOI | http]

[5]

A. Howard, M. Sandler, B. Chen, W. Wang, L.-C. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, R. Pang, and et al. Searching for mobilenetv3. 2019 IEEE/CVF International Conference on Computer Vision (ICCV), Oct 2019. [DOI | http]

[6]

A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. Jin, S. Zhao, and K. Keutzer. Squeezenext: Hardware-aware neural network design. 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), Jun 2018. [DOI | http]

[7]

X. Zhang, X. Zhou, M. Lin, and J. Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Jun 2018. [DOI | http]

[8]

N. Ma, X. Zhang, H.-T. Zheng, and J. Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. Lecture Notes in Computer Science, page 122–138, 2018. [DOI | http]

[9]

M. Tan and Q. V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2019. [arXiv]

[10]

P. Chao, C.-Y. Kao, Y. Ruan, C.-H. Huang, and Y.-L. Lin. Hardnet: A low memory traffic network. 2019 IEEE/CVF International Conference on Computer Vision (ICCV), Oct 2019. [DOI | http]

[11]

K. Han, Y. Wang, Q. Tian, J. Guo, C. Xu, and C. Xu. Ghostnet: More features from cheap operations. 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Jun 2020. [DOI | http]

[12]

J. Krause, M. Stark, J. Deng, and L. Fei-Fei. 3d object representations for fine-grained categorization. In 4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13), Sydney, Australia, 2013.

[13]

A. Pandey. Depth-wise convolution and depth-wise separable convolution. Accessed 2020-09-26: https://medium.com/@zurister/depth-wise-convolution-and-depth-wise-separable-convolution-37346565d4ec, 2018.

[14]

J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Jun 2018. [DOI | http]

[15]

D. Poulopoulos. How to use label smoothing for regularization. Accessed 2020-09-27: https://medium.com/towards-artificial-intelligence/how-to-use-label-smoothing-for-regularization-aa349f7f1dbb, 2020.

[16]

I. Loshchilov and F. Hutter. Decoupled weight decay regularization, 2017. [arXiv]