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 (Ph.D. student of prof. Władysław Skarbek)

WARSZAWA 2020

Lightweight neural architectures by example - car model classification using GhostNet

Abstract. One of the branches of ongoing research in the field of computer vision and neural networks in general is to develop relatively simple architectures that have limited number of trainable parameters and FLOPS required to process data while still being able to achieve competitive accuracy in common tasks. This need for fast and memory-efficient designs is dictated mostly by the necessity of deploying such models on devices with limited computational power and memory, such as smartphones. This work attempts to explore the process of training and finetuning such networks from scratch, taking as an example one of the newest designs in the field (GhostNet developed by Huawei Noah's Ark Lab) and applying it to the Stanford Cars Dataset for car model recognition task.

Keywords: deep learning, neural networks, image recognition, mobile architectures, GhostNet, Stanford Cars Dataset

Sieci neuronowe o zredukowanej liczbie parametrów na przykładzie architektury GhostNet zastosowanej w zadaniu klasyfikacji modeli samochodów

Streszczenie. Jednym z aktualnych kierunków badań w obszarze sieci neuronowych i wizji komputerowej jest opracowanie możliwie prostych architektur sieci, posiadających jak najmniejszą liczbę parametrów oraz wymagających jak najmniejszej liczby operacji zmiennoprzecinkowych (FLOPS) do przetworzenia danych wejściowych, zachowujących przy tym wysoką jakość predykcji. Główną przyczyną starań dążących do zmniejszenia rozmiarów sieci jest potrzeba wdrażania ich na urządzeniach mobilnych o ograniczonej mocy obliczeniowej i pamięci, takich jak smartfony. Niniejsza praca podejmuje temat uczenia i optymalizacji hiperparametrów tego typu sieci, biorąc za przykład jedną z najnowszysch architektur (sieć GhostNet stworzoną przez zespół badawczo-rozwojowy firmy Huawei) zastosowaną w zadaniu rozpoznawania modeli samochodów z wykorzystaniem zbioru danych Stanford Cars.

Słowa kluczowe: głębokie uczenie maszynowe, sieci neuronowe, rozpoznawanie obrazów, architektury mobilne, GhostNet, zbiór Stanford Cars

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 of 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 in the 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 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 4.1 M parameters and only 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 an 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, then finetune the best model from step one to reflect car models distribution on the streets of Poland.
3. Prepare model for deployment, create a simple Android application that allows 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 attached (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 - images and 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 in the input data. Instead of that, the focus is on obtaining those redundant feature maps in a cost-efficient way.

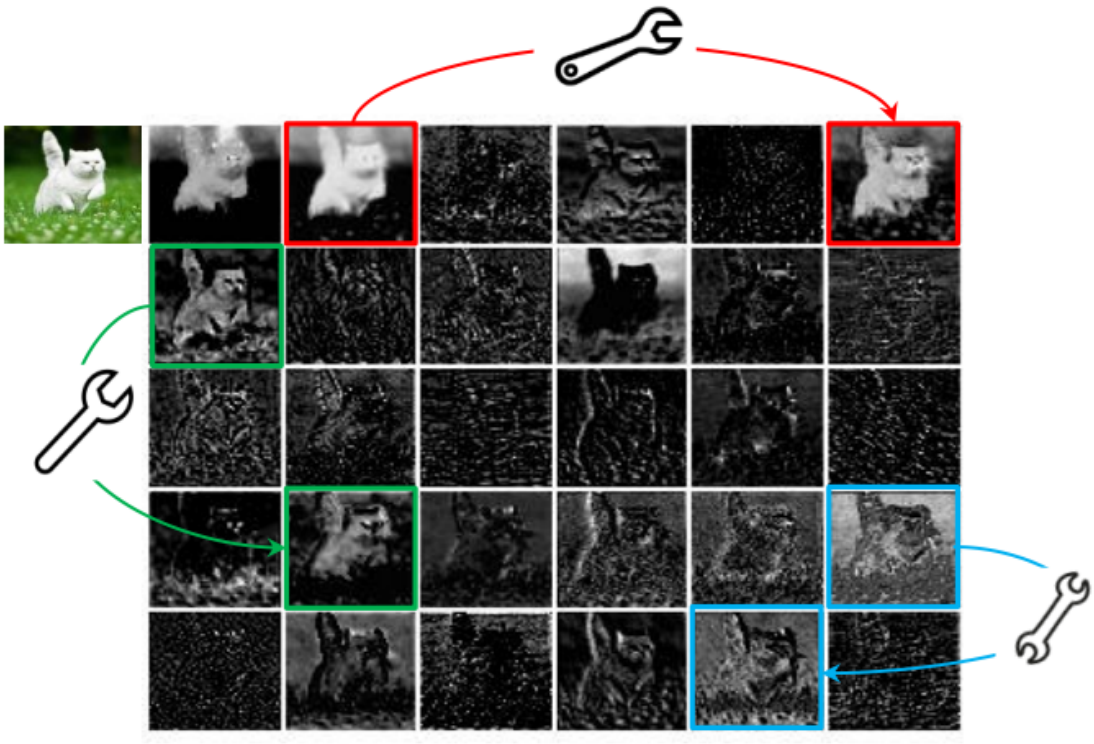
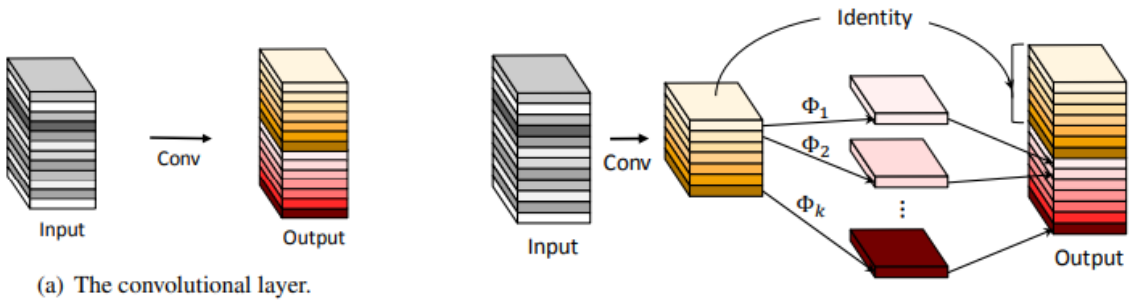


Figure 2: Redundant feature maps from ResNet-50 (picture from paper)

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.



(a) The convolutional layer.

(b) The Ghost module.

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:

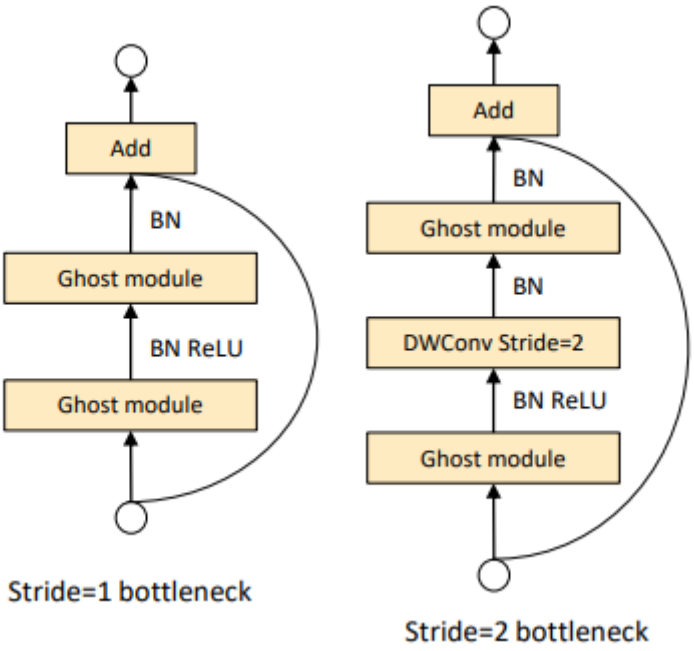


Figure 4: GhostBottleneck (picture from paper)

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
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

Input	Operator	#exp	#out	SE	Stride
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 popular designs like MobileNet or ShuffleNet getting better accuracy with less FLOPS and latency.

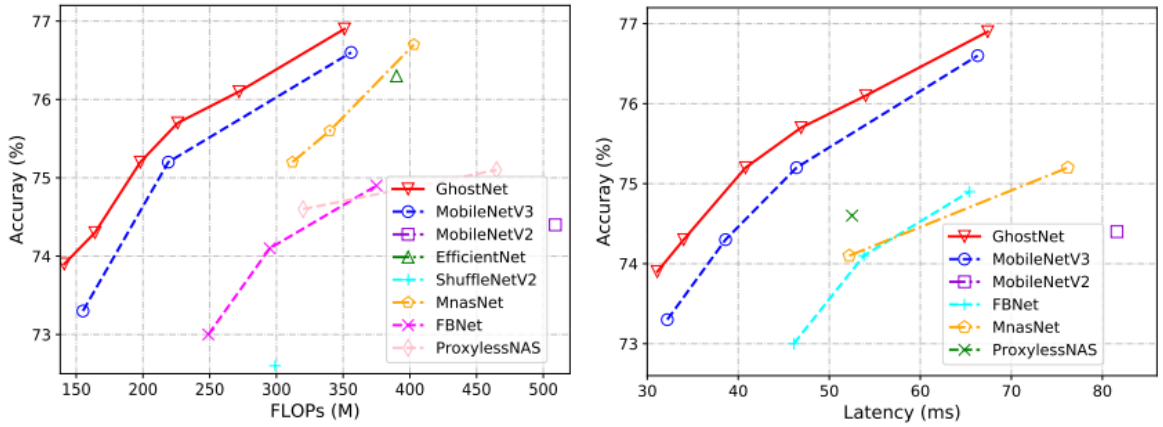


Figure 5: GhostNet comparison with some other mobile architectures (pictures from paper)

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

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. The repository contains 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 the 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
- `notebooks` - folder containing additional Jupyter notebooks (e.g. for normalization parameters calculation)
- `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 to 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:

```
# 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 - if so, recreate local environment by installing packages from `prod_requirements.txt` on Colab (after this step, runtime restart and imports re-execution 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 have 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'
```

```yaml
  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
```

```yaml
# 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
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

Experiment tracking is set up using Neptune experiment management tool. The tool has some useful features like:

- Python API and PyTorch Lightning integration
- Customizable logging of training metrics and model hyperparameters, as well as the storage and versioning of model artifacts
- Customizable plots and experiment comparison dashboards live-updated as the training proceeds
- Easy results sharing via HTTP links
- Option to programmatically download experiments results and parameters for postprocessing using API

Neptune logging can be easily enabled by passing a set of parameters through project `config.yml` file:

```
log_to_neptune: True
neptune_username: '<neptune.ai username>'
neptune_project_name: '<neptune.ai project name>'
neptune_api_token: '<neptune.ai API token>'
```

Results of the performed experiments in the discussed project are available under the following link:

[Neptune `cars` project dashboard]

The main dashboard table is configured to summarize all most important information about each experiment:

- Experiment ID
- Experiment state, running time and runtime utilized (`local` or `colab`)
- Architecture name
- Image and batch size
- Number of parameters of the network
- Image preprocessing settings: grayscale conversion, normalization, usage of image or tensor augmentations, usage of bounding boxes
- Loss function used
- Optimizer type and its most important settings (learning rate and weight decay)
- Learning rate scheduler type
- Network hyperparameters: dropout rate in the classifier, last layer size
- Number of epochs passed before early stopping was triggered
- Best (minimum) training and validation loss and best (maximum) training and validation accuracy
- Tags linking the experiment to specific sections in documentation
- Additional experiment description

After clicking on a particular experiment ID it is possible to check detailed logs and metrics.

In Parameters tab all experiment parameters that are passed through `config.yml` can be checked.

It is also possible to check multiple experiments and make a comparison between their metrics.

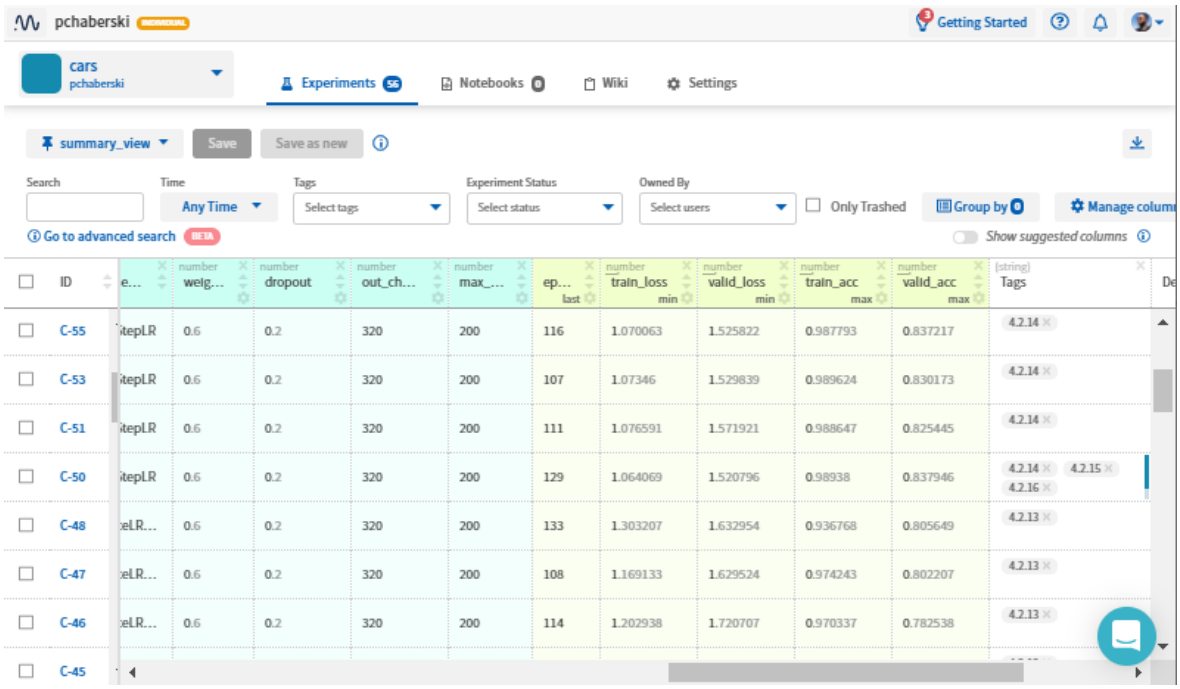


Figure 6: Part of a Neptune main dashboard

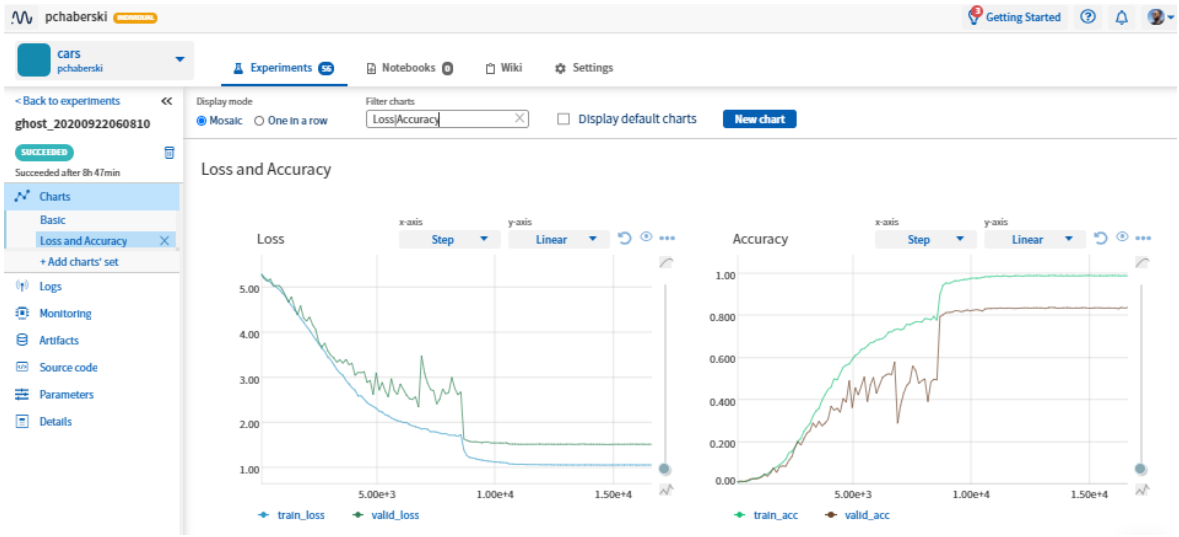


Figure 7: Loss and accuracy plots for particular experiment

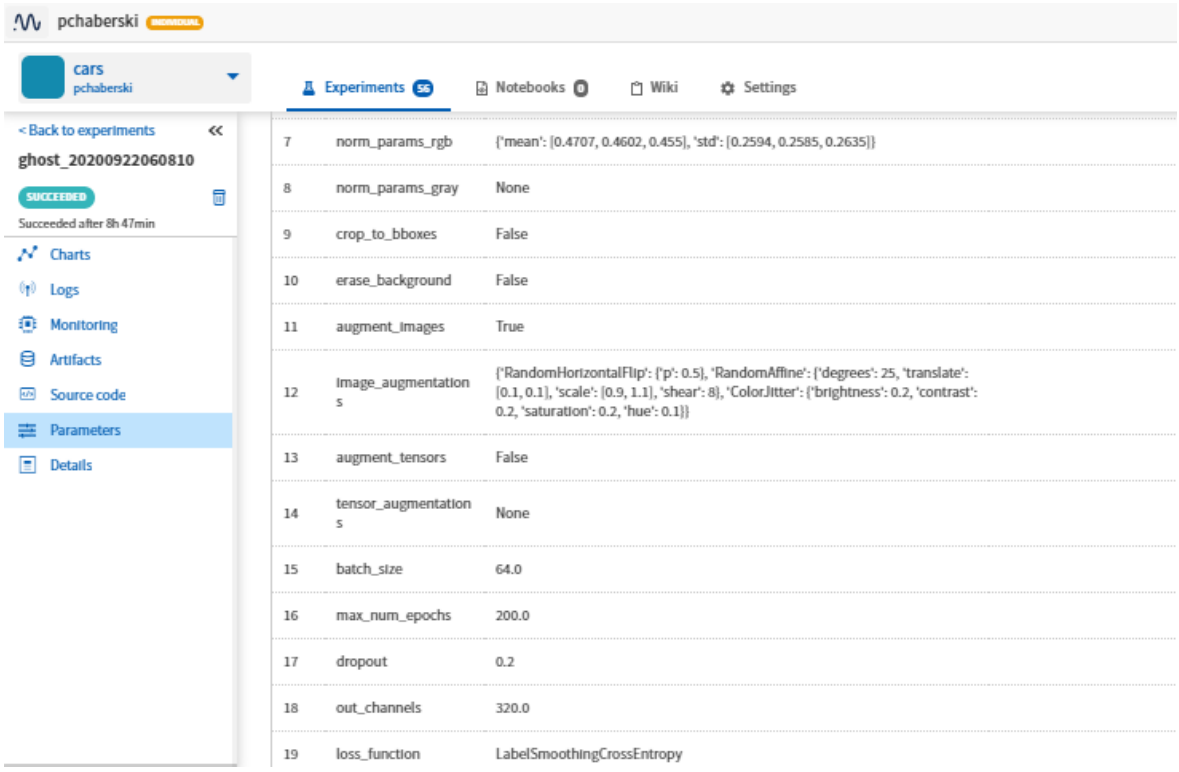


Figure 8: Detailed experiment parameters

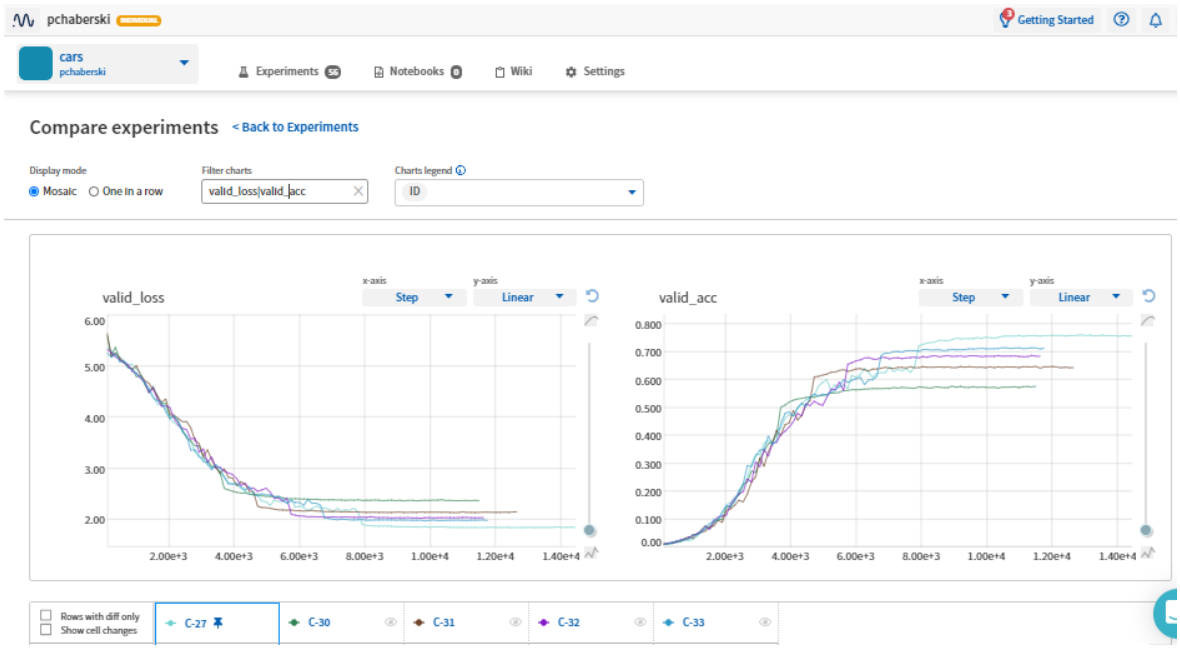


Figure 9: Multiple experiments comparison

4. Results

For the experiments, entire training subset from Stanford Cars Dataset was used for training, and *test* subset was used for validation. There is no additional hold-out testing set so it has to be taken into account that the final accuracy estimate might be somehow biased.

To limit hyperparameter space for the best model search, some assumptions were made at the beginning:

- network is trained from scratch, without using any pretrained weights
- input image size is 227x227 (this assumption results from initial tests on SqueezeNext [6] where this is a minimum image size and all other architectures available in `arch_dict.py` can handle such image size. For GhostNet, minimum image size is 224x224)
- batch size is fixed at 64 mostly because of local GPU memory limitations, however some tests during development phase showed no gain with smaller or larger batch sizes
- Adam with initial learning rate value of 0.001 is chosen as a default optimizer, and may be changed to AdamW [16] when applying weight decay (however SGD was also tested at the development phase, but it was leading to severe overfitting)
- early stopping is triggered when there is no decrease in validation loss for 15 epochs

During experiments, several techniques were used to increase validation accuracy and reduce overfitting, which turned out te be the major issue in training process:

- different loss functions
- pixel value normalization
- various image augmentations
- grayscale conversion

- utilization of bounding boxes
- L2 regularization using weight decay
- dropout rate changing in the classifier module
- last layer size changing
- learning rate scheduling

The search for the best settings was performed in a greedy manner: some arbitrary order of applying different techniques and hyperparameter values was established and after each step the best settings were further augmented using other techniques in order, however a few step-backs and sanity checks were made in the process.

The entire process of obtaining the best model is described step-by-step in section 4.2.

4.1. Best model

[Neptune charts]

The best model that was obtained during the process achieved 83.79 % top-1 accuracy on the validation set after training for 129 epochs with AdamW optimizer and Label Smoothing Cross Entropy function. Best metrics scores for that model are:

Metric	Value
Min. training loss	1.064
Min. validation loss	1.521
Max. training accuracy	98.93%
Max. validation accuracy	83.79%

The best model still shows significant overfitting so there might be some space for further improvement. However, taking into account that the same model achieves 73.98% on ImageNet dataset suggests that the score of 83.79% on the Stanford Cars Dataset is quite decent. While Stanford Cars Dataset contains much less classes (196 in comparison to 1000 in ImageNet), those classes seem harder to distinguish and the dataset itself is much smaller.

It is also important to notice, that due to the lesser number of classes, the size of the last layer was reduced during tests - instead of passing 1280-channel input to the classifier, only 320 channels are passed, which results in the total reduction of parameter count

from 4.2 million to slightly over 3 millions.

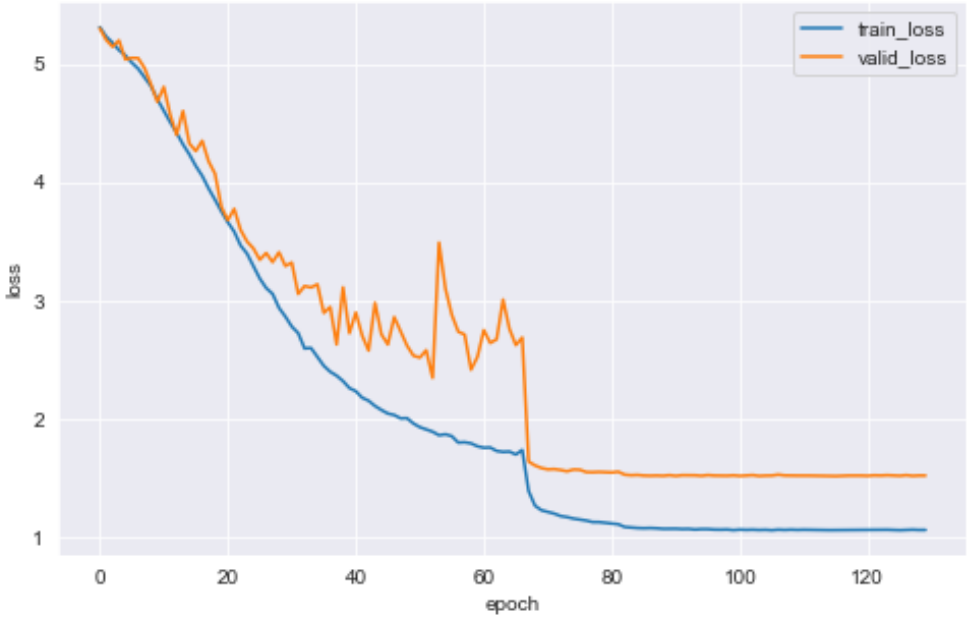


Figure 10: Training and validation loss of the best model (C-50)

The full set of settings and hyperparameters used to train the best performing model is listed below:

- runtime:
  - `colab`
- architecture:
  - `GhostNet`
- num_params:
  - `3041412.0`
- img_size:
  - `[227, 227]`
- grayscale:
  - `False`
- normalize:
  - `True`
- norm_params_rgb:

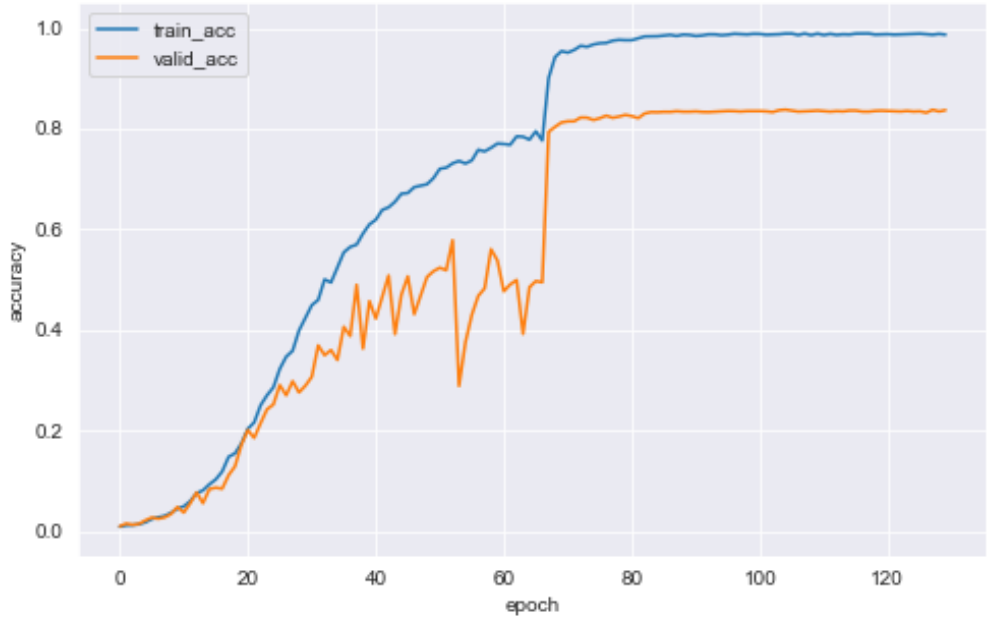


Figure 11: Training and validation accuracy of the best model (C-50)

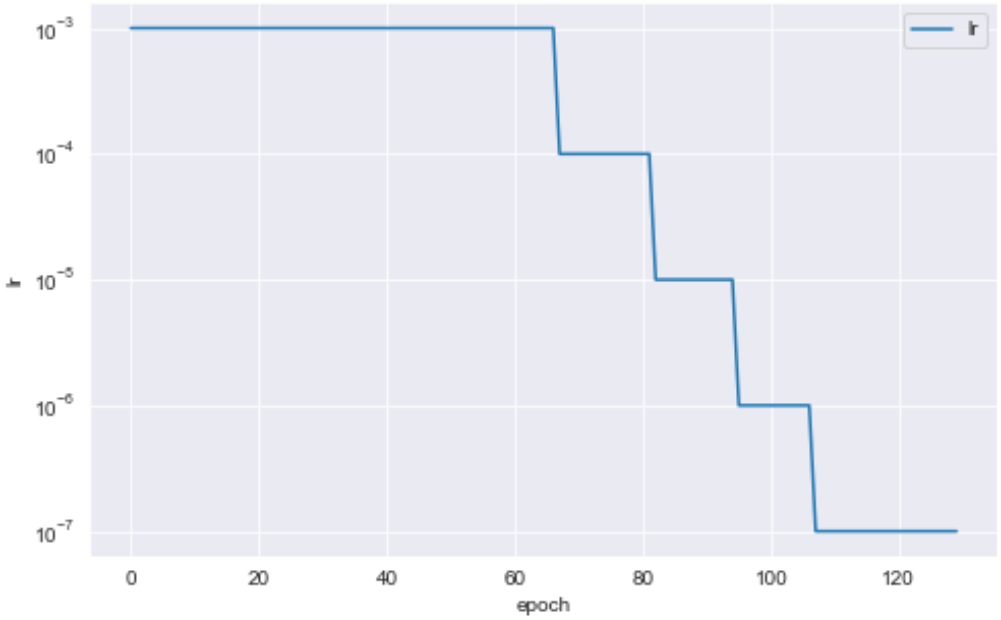


Figure 12: Learning rates for the best model (C-50)

  - `{'mean': [0.4707, 0.4602, 0.455], 'std': [0.2594, 0.2585, 0.2635]}`
- norm_params_gray:
  - `None`
- crop_to_bboxes:
  - `False`
- erase_background:
  - `False`
- augment_images:
  - `True`
- image_augmentations:
  - `{'RandomHorizontalFlip': {'p': 0.5}, '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:
  - `False`
- tensor_augmentations:
  - `None`
- batch_size:
  - `64.0`
- max_num_epochs:
  - `200.0`
- dropout:
  - `0.2`
- out_channels:
  - `320.0`
- loss_function:
  - `LabelSmoothingCrossEntropy`
- loss_params:
  - `None`
- optimizer:
  - `AdamW`
- learning_rate:
  - `0.001`
- weight_decay:

  - `0.6`
- all_optimizer_params:
  - `{'lr': 0.001, 'weight_decay': 0.6}`
- lr_scheduler:
  - `MultiStepLR`
- lr_scheduler_params:
  - `{'gamma': 0.1, 'milestones': [67, 82, 95, 107]}`

4.2. Experiments step-by-step

The table below presents the summary of model accuracy scores for all experiments along with a brief information of techniques used in training. Full and interactive comparison is available through Neptune dashboard. Also, all experiments results (parameters and logged metrics) are archived in a text file on GitHub.

	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	Added RGB normalization	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: horiz. flip, affine, erasing, color jitter	93.68%	38.68%
C-7	Augmentations: horizontal flip, affine, color jitter	99.73%	54.28%
C-8	Grayscale: with normalization, no augmentations	99.49%	6.58%
C-9	Grayscale: with normalization, no augmentations	97.13%	8.68%
C-10	Training set cropping with bounding boxes	7.58%	3.91%
C-11	Training set cropping + background erasing	4.36%	3.07%
C-12	Grayscale: normalization, best RGB augmentations	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%

	experiment description	train_acc	valid_acc
C-20	Dropout rate tests: dropout = 0.4	96.52%	64.88%
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%

	experiment description	train_acc	valid_acc
C-64	LR annealing test: exponentiation base = 0.955	98.49%	60.70%
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

[Neptune comparison]

The first comparison was between standard Cross Entropy loss function and Label Smoothing Cross Entropy. Label smoothing in classification tasks shows some regularization capability [15] resulting from a change in a standard Cross Entropy loss definition.

$$\text{cross-entropy loss} = (1-\epsilon)ce(i) + \epsilon \sum \frac{ce(j)}{N}$$

Figure 13: Label Smoothing Cross Entropy definition

In the Label Smoothing Cross Entropy definition `ce(i)` denotes standard Cross Entropy loss, `epsilon` stands for the label smoothing coefficient being a small positive number, and `N` is the number of classes. This modification results in forcing the model to predict not exactly `1` for correct class and `0` for other classes, but instead somehow *smoothed* values of `1 - epsilon` for correct class and `epsilon` for others.

The comparison shows that indeed with all other hyperparameters fixed, label smoothing allows to achieve a slightly better validation accuracy with the training loss decreasing slower, however the overfitting effect is still very large and in assumed setup results in triggering early stopping after only 26 epochs.

Metric	C-1 (CE Loss)	C-2 (LSCE Loss)
Min. training loss	0.296	1.133
Min. validation loss	4.849	4.873
Max. training accuracy	92.49%	98.89%

Metric	C-1 (CE Loss)	C-2 (LSCE Loss)
Max. validation accuracy	8.15%	9.12%

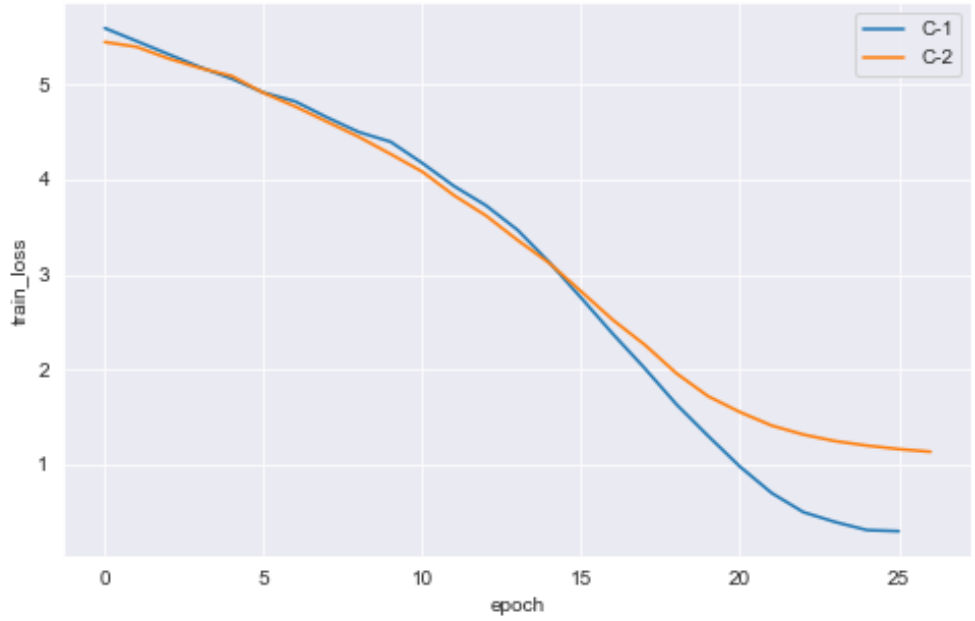


Figure 14: Training loss values for CE Loss (C-1) and LSCE Loss (C-2)

4.2.2. Normalization

[Neptune comparison]

The next step in the process was adding normalization to the data using mean and standard deviation calculated on the training set (see `normalization_coeffs.ipynb` notebook). Centering the data gave a 3 percentage points in validation accuracy, however faster convergence resulted in even faster training loss drop.

Metric	C-2 (No normalization)	C-3 (RGB normalization)
Min. training loss	1.133	1.075
Min. validation loss	4.873	4.792
Max. training accuracy	98.89%	99.45%

Metric	C-2 (No normalization)	C-3 (RGB normalization)
Max. validation accuracy	9.12%	11.95%

4.2.3. Augmentations

[Neptune comparison]

The first milestone experiment series was achieved thanks to adding training data augmentations to the model from C-3 experiment. The transformations that were tested were:

- Random horizontal flip
- Random affine transform
- Random erasing
- Color jittering

With C-3 experiment as the baseline, four combinations of the above-mentioned transformations were tested:

- C-4: `RandomHorizontalFlip` + `RandomAffine`
- C-5: `RandomHorizontalFlip` + `RandomAffine` + `RandomErasing`
- C-6: `RandomHorizontalFlip` + `RandomAffine` + `RandomErasing` + `ColorJitter`
- C-7: `RandomHorizontalFlip` + `RandomAffine` + `ColorJitter`

The results showed that the augmentations in general helped to achieve a very large increase in validation accuracy (from 11.96% to 54.28% in the best case) while reducing overfitting significantly. As for the particular transformations, it turned out that in the tested setup `RandomErasing` did not help, probably introducing too much variance in the training set combined with other augmentations.

Metric	C-3	C-4	C-5	C-6	C-7
Min. training loss	1.075	0.990	1.122	1.315	1.003
Min. validation loss	4.792	2.813	3.339	3.452	2.744
Max. training accuracy	99.45%	99.76%	98.12%	93.68%	99.73%
Max. validation accuracy	11.95%	51.92%	38.08%	38.68%	54.28%

(a) (b)

Figure 15: Original (a) and normalized (b) image

4.2.4. Grayscale conversion

[Neptune comparison]

Testing how the network will behave after converting input images to grayscale before the training came from the idea, that we want the model to distinguish car models only by the details of design, and obviously not to focus on irrelevant differences such as body color. To adapt the original GhostNet architecture to be able to process also 1-channel images, a small customization was made to the first layer of the network by adding `img_channels` parameters, so that the initial convolution could work on any number of channels in general:

```
class GhostNet(nn.Module):
    def __init__(
        self,
        num_classes=1000, img_channels=3, dropout=0.2, out_channels=1280,
        width_mult=1.
    ):
        super().__init__()
        self.num_classes = num_classes
        self.img_channels = img_channels
        self.dropout = dropout
        self.out_channels = out_channels
```

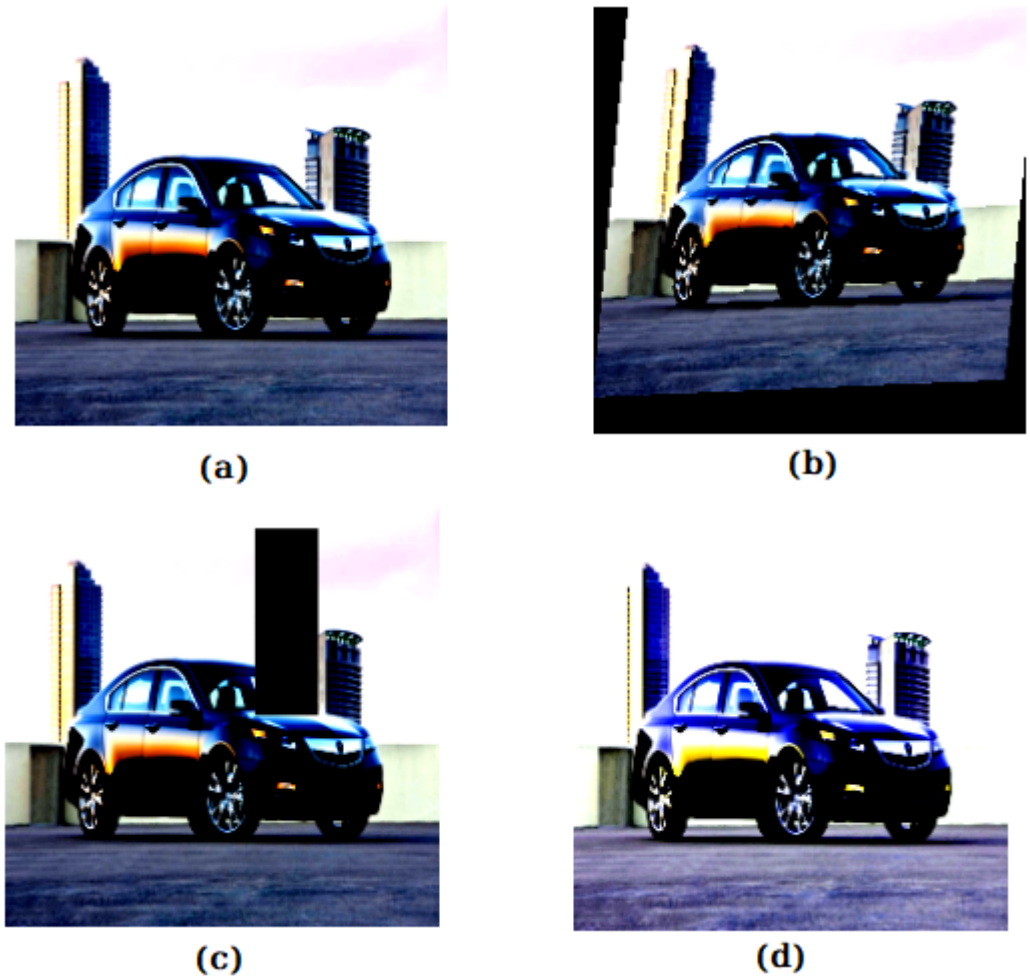


Figure 16: Original normalized image (a); Images with: RandomAffine (b), RandomErasing (c), ColorJitter (d)

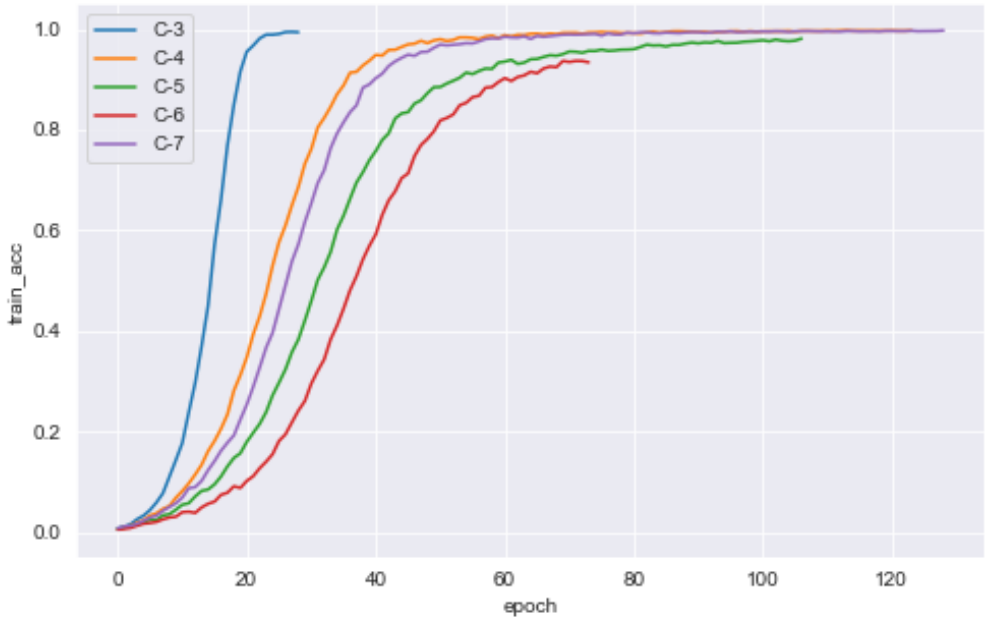


Figure 17: Training accuracy with different augmentations

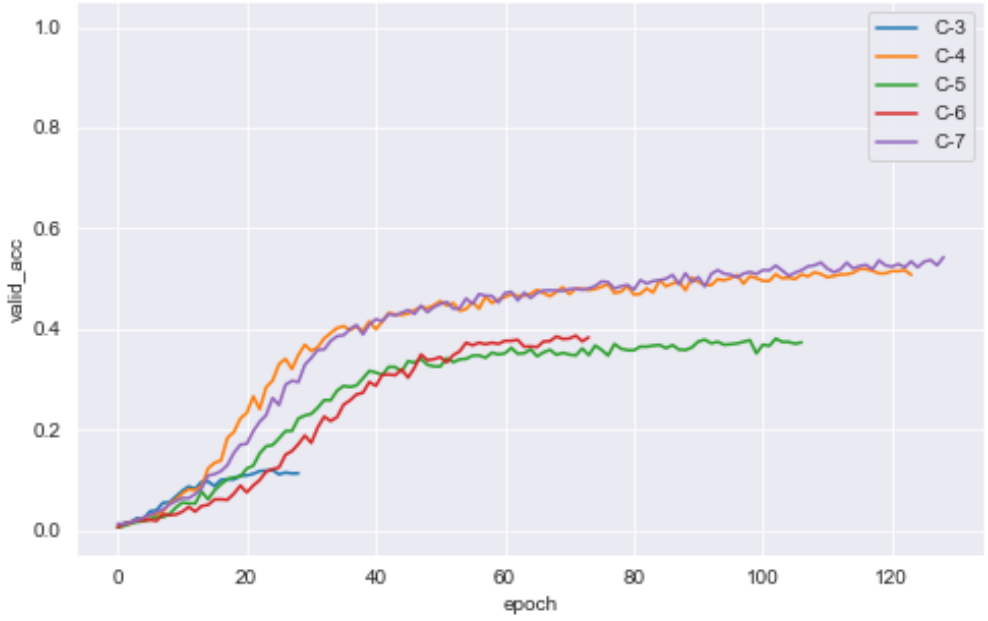


Figure 18: Validation accuracy with different augmentations

```
# ...

        # building first layer
        output_channel = _make_divisible(16 * width_mult, 4)
        layers = [nn.Sequential(
            nn.Conv2d(self.img_channels, output_channel, 3, 2, 1, bias=False),
            nn.BatchNorm2d(output_channel),
            nn.ReLU(inplace=True)
        )]
        input_channel = output_channel

# ...
```

Also some other customizations were made as can be seen above, namely introducing customizable dropout rate in classifier and last layer size by adding `dropout` and `out_channels` parameters. Those will be discussed in sections 4.2.7 and 4.2.8, respectively.

Three comparisons were made taking as baselines identical setups that were previously trained using 3-channel input:

- experiment C-8, being a reflection of C-2 (no normalization and no augmentations)
- experiment C-9, being a reflection of C-3 (normalization added, no augmentations)
- experiment C-12, being a reflection of C-7 (normalization + best augmentations)

The results of these tests clearly show, that probably due to the network design, grayscale conversion brings no gain in model performance (in fact, all comparisons are in favor of RGB variants):

Metric	C-2 (RGB)	C-8 (Gr.)	C-3 (RGB)	C-9 (Gr.)	C-7 (RGB)	C-12 (Gr.)
Min. tr. loss	1.133	1.089	1.075	1.207	1.003	1.017
Min. val. loss	4.873	5.080	4.792	4.746	2.744	2.843
Max. tr. accuracy	98.89%	99.49%	99.45%	97.13%	99.73%	99.67%
Max. val. accuracy	9.12%	6.58%	11.96%	8.68%	54.28%	50.51%

4.2.5. Bounding boxes utilization

[Neptune comparison]

Another idea was to try to somehow utilize car bounding boxes coordinates that are available in Stanford Cars Dataset along with the class labels. The goal of the project was however to get possibly unbiased benchmark on the original test set (despite it was used for model validation), so any operations using bounding box information could be used only on training subset. Another reason for that is that the ultimate objective is to deploy trained model on a mobile device so it cannot use any information that is unavailable during inference to estimate the performance. Using bounding boxes in performance estimation and eventually during real-life prediction would require to stack the discussed classification model with some kind of detector, which would firstly estimate the location of bounding boxes.

Two approaches of utilizing bounding boxes on training set were consecutively tested, taking so far the best C-7 experiment as the baseline:

- C-10: only cropping images to bounding box coordinates before resize
- C-11: cropping mages to bounding boxes and then putting them on the white background of original image size to preserve ratios before resize

Both transformations were intended to get rid of the image background and try to force the network to focus only on relevant image parts and to prevent it from fitting to the background elements.

Figure 19: Original normalized image (a); Image cropped to b-boxes (b); Image with background erased (c)

It turned out that this idea was totally wrong - in the first case (C-10), after crop and resize, all proportions were strongly distorted, which caused a large discrepancy between training and validation data and prevented optimizer from converging. The divergence was even stronger in the second case (C-11), because despite preserving original proportions, the network started to focus only on fitting to the white background instead of car details.

Metric	C-7	C-10	C-11
Min. training loss	1.003	4.570	4.822
Min. validation loss	2.744	5.142	5.196
Max. training accuracy	99.73%	7.58%	4.36%
Max. validation accuracy	54.28%	3.91%	3.07%

4.2.6. Optimizer change and L2 regularization

[Neptune comparison]

For further attempts to reduce overfitting and in consequence increase validation accuracy, one of the most commonly used techniques in parametric machine-learning models training was applied - L2 regularization. Optimizers in PyTorch allow for passing `weight_decay` parameter, which represents the strength of the penalty added for too high model weights. However, as empirical study shows [16], this kind of regularization requires decoupling application of weight decay from optimization steps taken with respect to the loss function when using adaptive algorithms like Adam. So to be able to successfully apply this technique in discussed problem, Adam optimizer was replaced by its variation utilizing decoupled weight decay - AdamW.

With C-7 as a baseline, 5 different `weight_decay` values were tested with AdamW optimizer:

- C-13: `weight_dacay = 0.1`
- C-14: `weight_dacay = 0.2`
- C-15: `weight_dacay = 0.3`
- C-16: `weight_dacay = 0.4`
- C-17: `weight_dacay = 0.5`

The results:

Metric	C-7	C-13	C-14	C-15	C-16	C-17
Min. training loss	1.003	1.023	1.071	1.213	1.193	1.378
Min. validation loss	2.744	2.315	2.089	2.285	2.174	2.301
Max. training accuracy	99.73%	99.44%	98.84%	95.83%	95.95%	90.38%
Max. validation accuracy	54.28%	63.39%	68.50%	61.84%	65.14%	59.95%

It can be observed that adding larger penalty by increasing `weight_decay` in fact reduces overfitting, but at some point the weights become too constrained preventing the model to fit well to training data and therefore limiting validation accuracy increase. However, with all tested values, regularized version managed to improve the score obtained without weight decay.

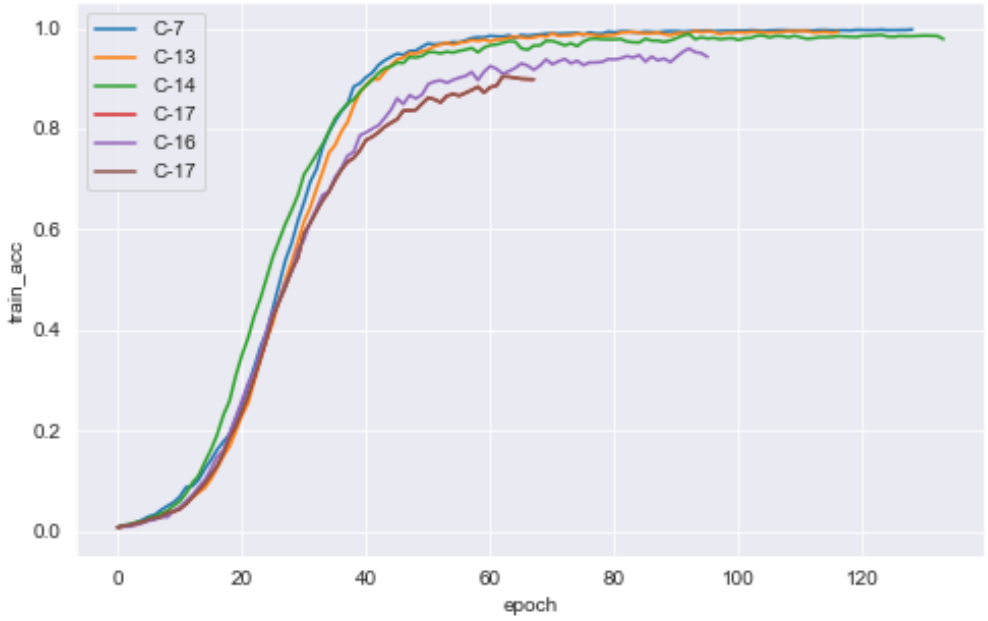


Figure 20: Training accuracy with different values of weight decay

4.2.7. Dropout rate tests

[Neptune comparison]

Basic GhostNet design has dropout rate before last linear layer fixed at 0.2. As men-

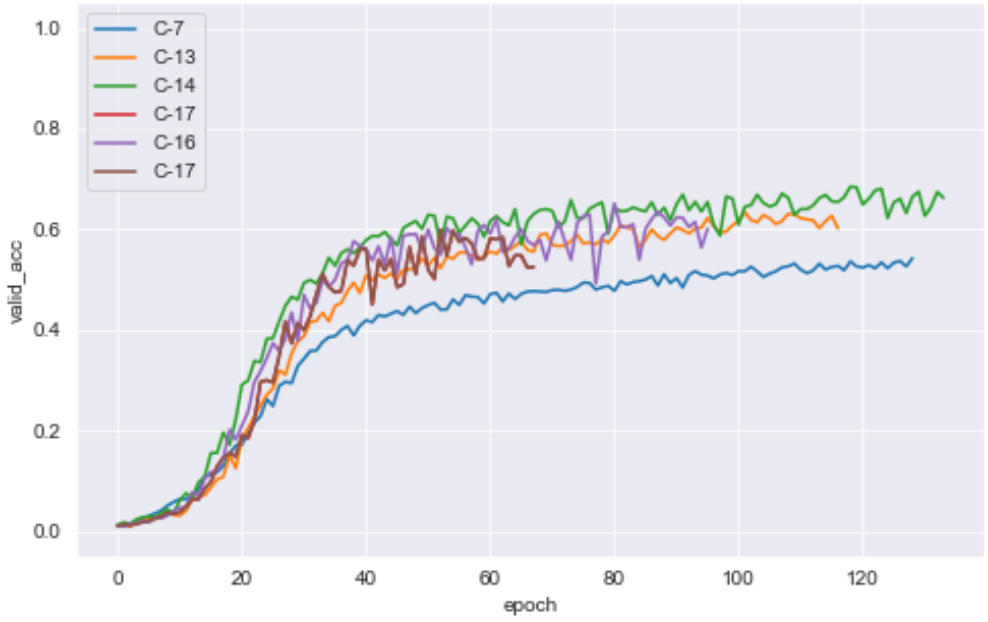


Figure 21: Validation accuracy with different values of weight decay

tioned in 4.2.4 the original design was customized to allow for passing different dropout values. Some other than default values (especially larger, hoping to further reduce overfitting) were checked.

Dropout rate values that were tested with baseline of 0.2 from C-14 experiment setup were:

- C-18: `dropout = 0.1`
- C-19: `dropout = 0.3`
- C-20: `dropout = 0.4`
- C-21: `dropout = 0.5`

The above mentioned experiments show no improvement using values different from default.

Metric	C-14	C-18	C-19	C-20	C-21
Min. training loss	1.071	1.039	1.087	1.173	1.191
Min. validation loss	2.089	2.179	2.108	2.164	2.096
Max. training accuracy	98.84%	99.11%	98.62%	96.52%	96.28%

Metric	C-14	C-18	C-19	C-20	C-21
Max. validation accuracy	68.50%	66.90%	67.81%	64.88%	66.75%

4.2.8. Last layer size tests

[Neptune comparison]

By default, in GhostNet architecture the number of channels in the feature vector passed into the classifier module is fixed at value of 1280 and those channels are finally mapped on the number of classes by the fully connected layer. As mentioned in 4.2.4, this value was parametrized based on the assumption, that it could be strictly associated with the specific output number of classes and the architecture was optimized for 1000-class ImageNet, while Stanford Cars Dataset consists of 196 classes.

The values that were tested with respect to 1280 baseline from experiment C-14:

- C-22: `out_channels = 320`
- C-23: `out_channels = 640`
- C-24: `out_channels = 960`
- C-25: `out_channels = 1600`

It is important to notice, that changing the output channels value strongly affects the total parameter count of the network:

	output channels	number of parameters
C-14	1280	4153090
C-22	320	3041410
C-23	640	3411970
C-24	960	3782530
C-25	1600	4523650

Analysis of the results shows no straightforward relationship between the number of output channels and network's performance on the particular dataset that was used, however the lowest number of channels testes (320) turned out to give slightly better validation accuracy that default with less overfitting, while reducing the number of parameters from 4.15 million to 3.04 million.

Metric	C-14	C-22	C-23	C-24	C-25
Min. training loss	1.071	1.140	1.196	1.108	1.071
Min. validation loss	2.089	2.055	2.250	2.202	2.294
Max. training accuracy	98.84%	97.13%	96.13%	98.23%	98.99%
Max. validation accuracy	68.50%	68.93%	63.13%	64.96%	63.11%

4.2.9. Automatic learning rate scheduling

[Neptune comparison]

The more regularization added, the more difficult it is for optimizer to find the optimal solution with a fixed learning rate what is reflected by more and more jumpy learning curves after certain number of epochs. A natural step in this case is to try to decrease the learning rate after some stagnation starts to emerge in the loss decrease.

The first round of experiments with learning rate schedulers were done using `ReduceLROnPlateau` scheduler from PyTorch, which was set to decrease the learning rate by a factor 0.1 after no decrease in validation loss is observed for 5 consecutive epochs. During the early development phase it was observed that randomness in contents of training batches that results in slightly different learning curves even with the same experiment setup each time, may also result in triggering learning rate decrease at different epochs, and this in turn may affect the final validation accuracy achieved. To test the scale of this phenomenon, 4 attempts of the same experiment were taken and compared with the results obtained without scheduler (C-22):

- C-26
- C-27
- C-28
- C-29

The results are that the learning rate decrease at the right point of training procedure can give a significant benefit, and the choice of that particular moment based on variability in training data feed is also not without significance.

Metric	C-22	C-26	C-27	C-28	C-29
Min. training loss	1.140	0.995	1.014	0.988	1.006
Min. validation loss	2.055	1.903	1.836	1.888	1.874

Metric	C-22	C-26	C-27	C-28	C-29
Max. training accuracy	97.13%	99.82%	99.78%	99.83%	99.78%
Max. validation accuracy	68.93%	74.60%	76.20%	75.14%	74.82%

4.2.10. Controlled learning rate scheduling

[Neptune comparison]

For further investigation of the influence of learning rate drop timing (especially the first drop from 0.001 to 0.0001) on the final validation accuracy, some more experiments with manually set milestones for learning rate decrease were made using `MultiStepLR` scheduler with the same factor of 0.1. Also it was noticed that 4.2.9 the best results were obtained when the scheduler was triggered at the earliest, so the milestones were set to push further in this direction:

With the best take from 4.2.9 (C-27) as the baseline, where automatically triggered milestones were checked to be `[61, 82, 93, 104]`, the experiments were:

- C-30: `milestones = [28, 48, 68, 88]`
- C-31: `milestones = [36, 56, 76, 96]`
- C-32: `milestones = [44, 64, 84, 104]`
- C-33: `milestones = [52, 72, 92, 112]`

Looking at the results and comparing with the baseline it is obvious, that all learning rate drops were triggered too early.

Metric	C-27	C-30	C-31	C-32	C-33
Min. training loss	1.014	1.749	1.321	1.142	1.059
Min. validation loss	1.836	2.364	2.130	2.022	1.971
Max. training accuracy	99.78%	80.66%	95.03%	98.68%	99.60%
Max. validation accuracy	76.20%	57.82%	64.93%	68.79%	71.59%

4.2.11. Weight decay adjustment

[Neptune comparison]

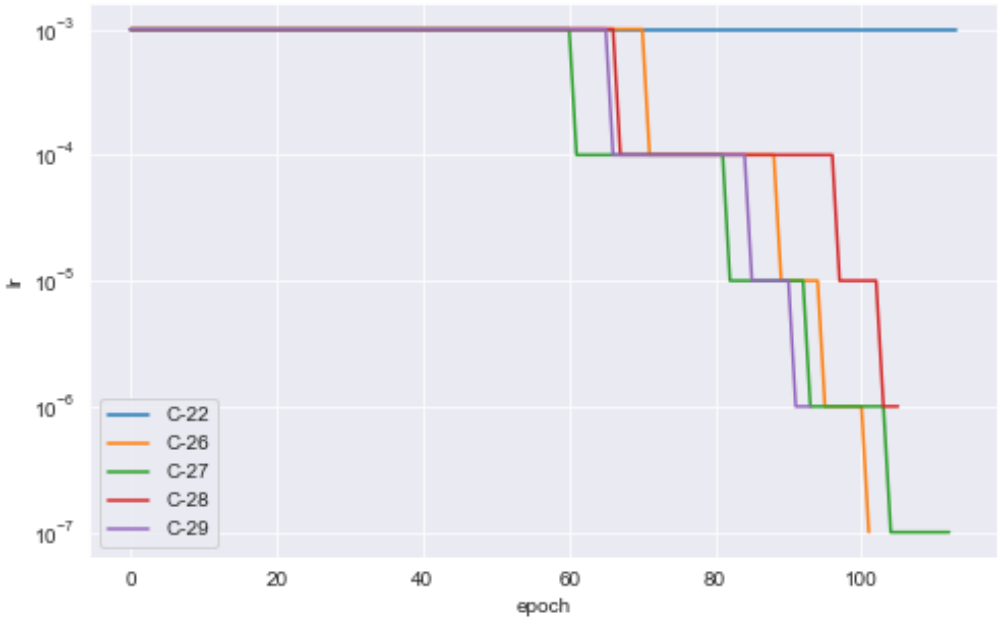


Figure 22: Different LR decrease moments with automatic scheduler

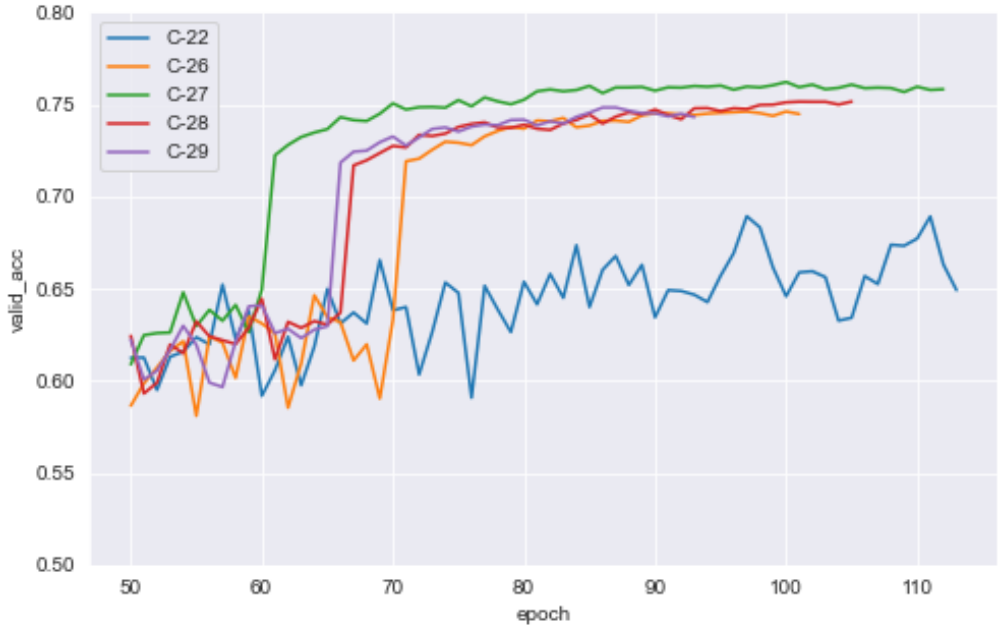


Figure 23: Validation accuracy with different LR drops (from epoch 50)

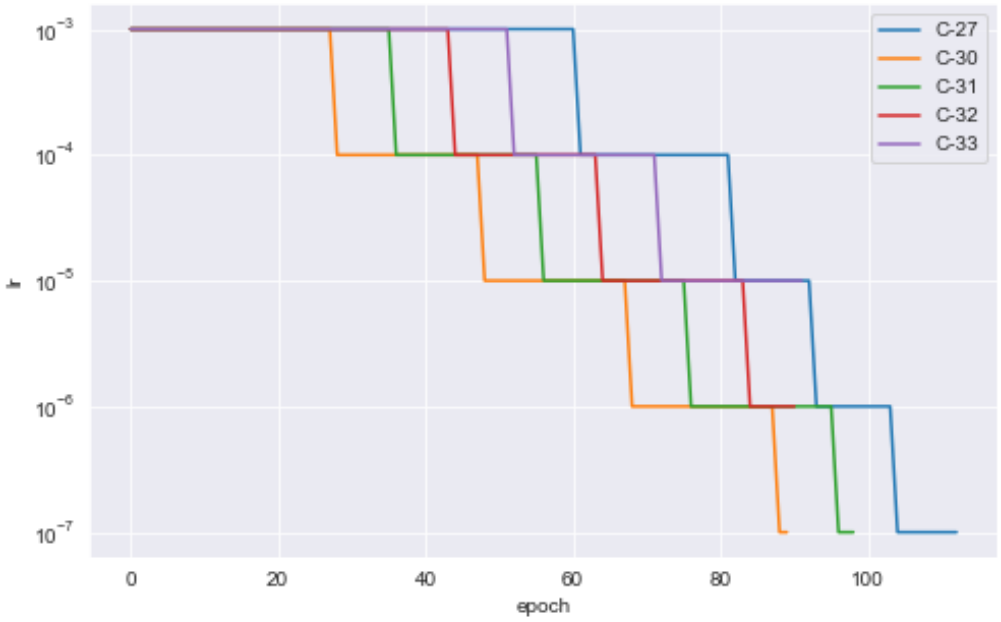


Figure 24: Learning rate drops with manual LR scheduling

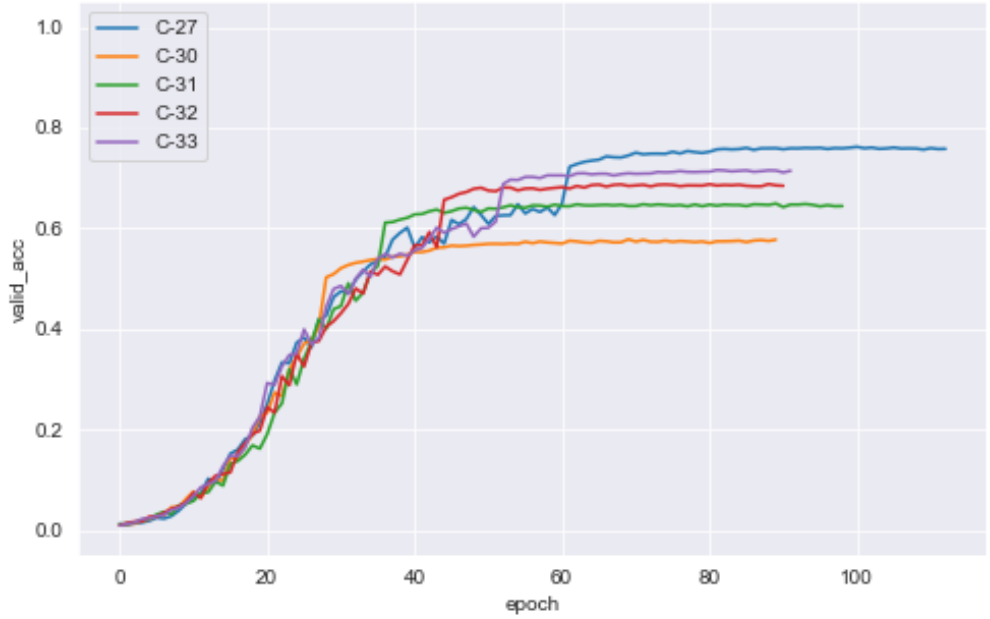


Figure 25: Validation accuracy with manual LR scheduling

After all experiments with learning rates, it seemed reasonable to revisit the most important decisions related to regularization. Especially the optimal weight decay value is strictly dependend on the learning rate used, therefore once again some other values than so-far-best 0.2 were checked at this point. Comparing to C-27, new weight decay values tested were:

- C-36: `weight_decay = 0.5`
- C-37: `weight_decay = 0.3`
- C-38: `weight_decay = 0.4`
- C-39: `weight_decay = 0.6`
- C-40: `weight_decay = 0.7`

Similarly to 4.2.6, some values were to small to prevent overfitting enough while after some point the penalty was to large to let the model fit the data properly (and also triggering learning rate drop too early), but the optimal value from the set tested was much larger than before with fixed learning rate. The best validation accuracy was achieved with weight decay of 0.6.

Metric	C-27	C-36	C-37	C-38	C-39	C-40
Min. training loss	1.014	1.103	1.043	1.048	1.090	1.089
Min. validation loss	1.836	1.669	1.858	1.695	1.563	1.802
Max. training accuracy	99.78%	98.84%	99.57%	99.37%	98.67%	99.24%
Max. validation accuracy	76.20%	79.40%	74.44%	78.82%	82.55%	75.12%

Also it has to be noted, that due to different course of learning process resulting from different regularization, automatic `ReduceLROnPlateau` LR scheduling has been reestablished for the time being.

4.2.12. Dropout rate verification

[Neptune comparison]

Another step-back after introducing learning rate scheduling was to verify the dropout rate that was fixed before. With the C-39 experiment as the baseline, re-checked values were:

- C-41: `dropout = 0.3`
- C-42: `dropout = 0.4`

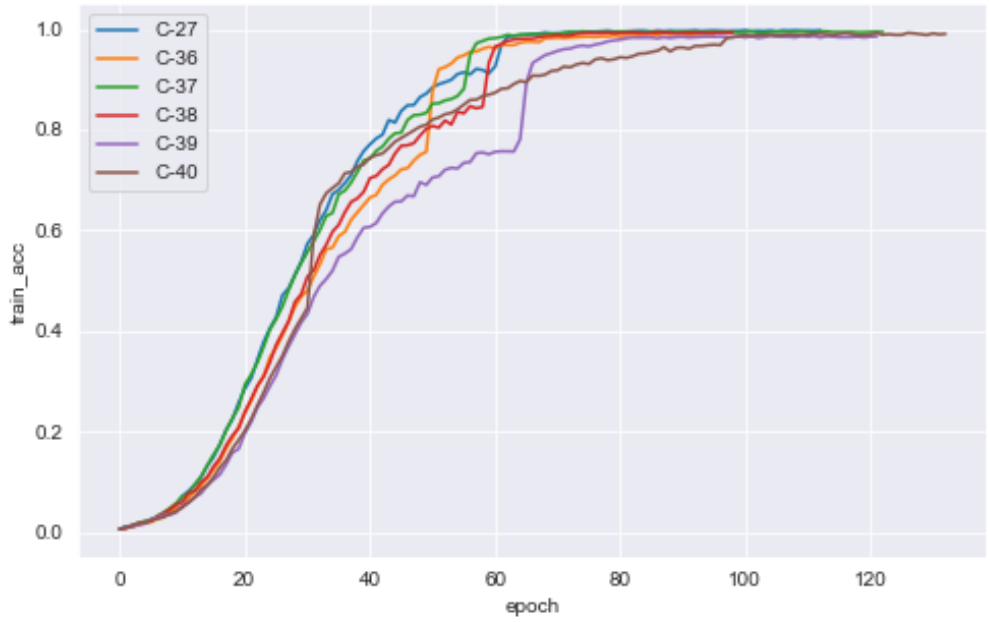


Figure 26: Training accuracy with different weight decay ant auto-scheduling

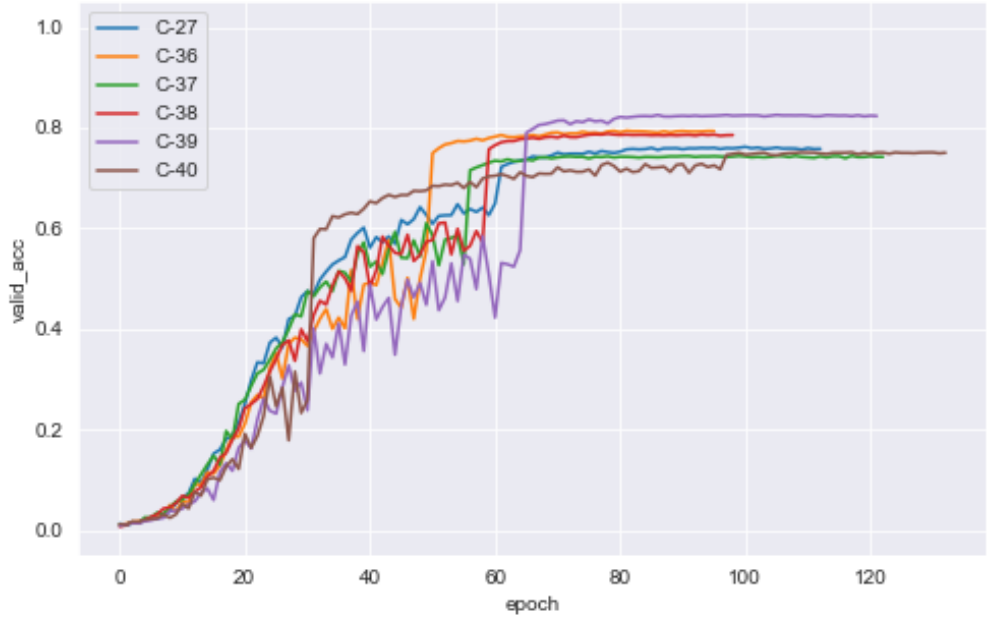


Figure 27: Validation accuracy with different weight decay and auto-scheduling

- C-43: `dropout = 0.5`
- C-44: `dropout = 0.25`

The tests confirmed that the default value seems to be optimal, since all experiments fared worse than the 0.2 baseline, moreover the closer the default value, the higher the validation accuracy.

Metric	C-39	C-41	C-42	C-43	C-44
Min. training loss	1.090	1.111	1.270	1.270	1.087
Min. validation loss	1.563	1.571	1.646	1.692	1.562
Max. training accuracy	98.67%	98.49%	95.34%	96.08%	98.79%
Max. validation accuracy	82.55%	82.08%	79.57%	77.87%	82.45%

4.2.13. Additional augmentations tests

[Neptune comparison]

In attempt to further decrease overfitting some additional augmentations were tested (and also `RandomErasing` was tried again). Every new transformation was added separately to the existing set of augmentations:

- C-45: `RandomResizedCrop`
- C-46: `RandomRotation`
- C-47: `RandomPerspective`
- C-48: `RandomErasing`

Figure 28: Resized crop (a); Rotation (b); Perspective (c)

The results show that adding more augmentations make it too hard for the model to fit the data and therefore limits the validation accuracy increase.

Metric	C-39	C-45	C-46	C-47	C-48
Min. training loss	1.090	1.162	1.203	1.169	1.303
Min. validation loss	1.563	1.699	1.721	1.630	1.633
Max. training accuracy	98.67%	97.56%	97.03%	97.42%	93.68%
Max. validation accuracy	82.55%	78.73%	78.25%	80.22%	80.56%

4.2.14. Learning rate scheduler adjustment

[Neptune comparison]

Remembering the influence of the exact moment of learning rate drop on the model performance, some additional search for the best milestones was performed in the closest neighborhood of so far the best LR drop milestones (from C-39: `[65, 80, 93, 105]`) using `MultiStepLR` scheduler:

- C-50: `milestones = [67, 82, 95, 107]`
- C-51: `milestones = [63, 78, 91, 103]`
- C-53: `milestones = [66, 81, 94, 106]`
- C-55: `milestones = [68, 83, 96, 108]`
- C-56: `milestones = [64, 79, 92, 104]`

The validiation accuracy difference range was narrow, however manual scheduler fine-tuning helped to gain over 1 additional percentage point.

Metric	C-39	C-50	C-51	C-53	C-55	C-56
Min. training loss	1.090	1.064	1.077	1.073	1.070	1.066
Min. validation loss	1.563	1.521	1.572	1.530	1.526	1.560
Max. training accuracy	98.67%	98.94%	98.86%	98.96%	98.78%	98.99%
Max. validation accuracy	82.55%	83.79%	82.54%	83.02%	83.72%	82.79%

4.2.15. Last layer size sanity check

[Neptune comparison]

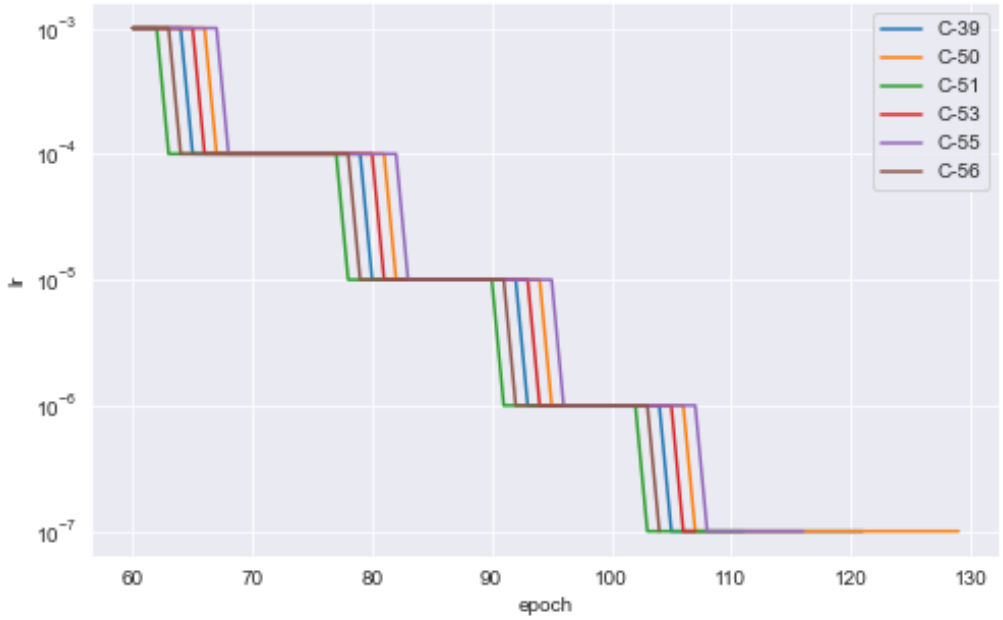


Figure 29: Learning rate drop comparison after scheduler adjustment (from epoch 60)

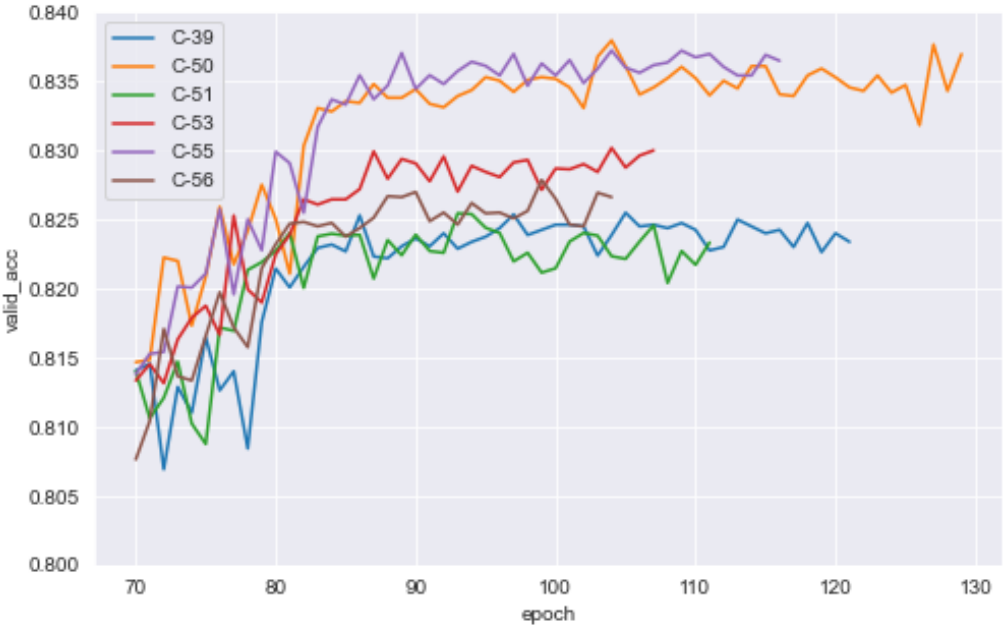


Figure 30: Validation accuracy differences after scheduler adjustment (from epoch 70)

To make sure that the previous choice of the output number of channels made in 4.2.8 is still valid after learning rate scheduling introduction, a comparison between the best model (C-50) and the model with 1280 output channels (C-58) was made, with `ReduceLROnPlateau` scheduler for the latter. The results confirmed that the reduced number of channels work well applied to the discussed problem, since the model with original `out_channels` showed worse validation accuracy and larger overfitting.

Metric	C-50	C-58
Min. training loss	1.064	1.050
Min. validation loss	1.521	1.720
Max. training accuracy	98.94%	99.44%
Max. validation accuracy	83.79%	78.83%

4.2.16. Learning rate annealing tests

[Neptune comparison]

The last series of tests were aimed to check how the learning process will run with learning rate being reduced smoothly using `LambdaLR` scheduler that will decrease LR each epoch by multiplying it by factor `base**epoch`, where `base` is a number less that `1`, but close to that value. The initial exponentiation base was chosen in way, so that the learning rates would be equal at the epoch when the first LR drop occurs during training the best model (C-50) with standard scheduler. Some other values in the neighborhood were also checked:

- C-63: `lr_lambda = lambda epoch: pow(0.0001/0.001, 1/67)**epoch` (exponentiation base calculated as the common ratio of geometric progression, so that learning rates will *meet* at epoch 67)
- C-64: `lr_lambda = lambda epoch: 0.955**epoch`
- C-65: `lr_lambda = lambda epoch: 0.975**epoch`
- C-66: `lr_lambda = lambda epoch: 0.98**epoch`

For all experiments the learning curves were much smoother than with using standard scheduler, but the overfitting started much sooner and the validation accuracy was low.

Metric	C-50	C-63	C-64	C-65	C-66
Min. training loss	1.064	0.994	1.169	0.975	1.057
Min. validation loss	1.521	2.039	2.314	1.907	1.942
Max. training accuracy	98.94%	99.80%	98.49%	99.66%	98.72%
Max. validation accuracy	83.79%	70.51%	60.70%	73.07%	70.46%

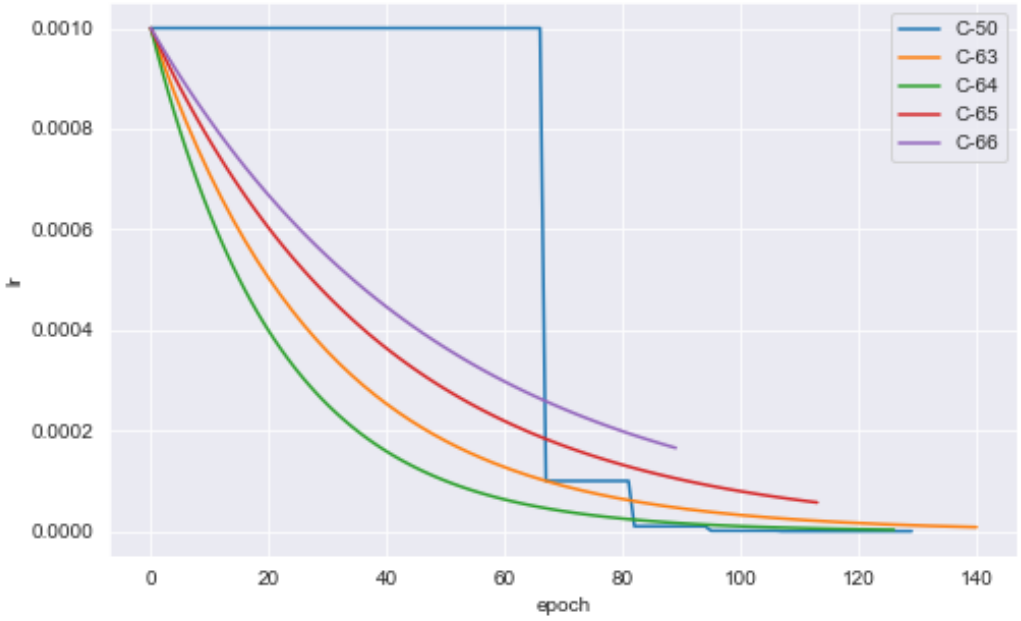


Figure 31: Learning rate annealing (linear scale)

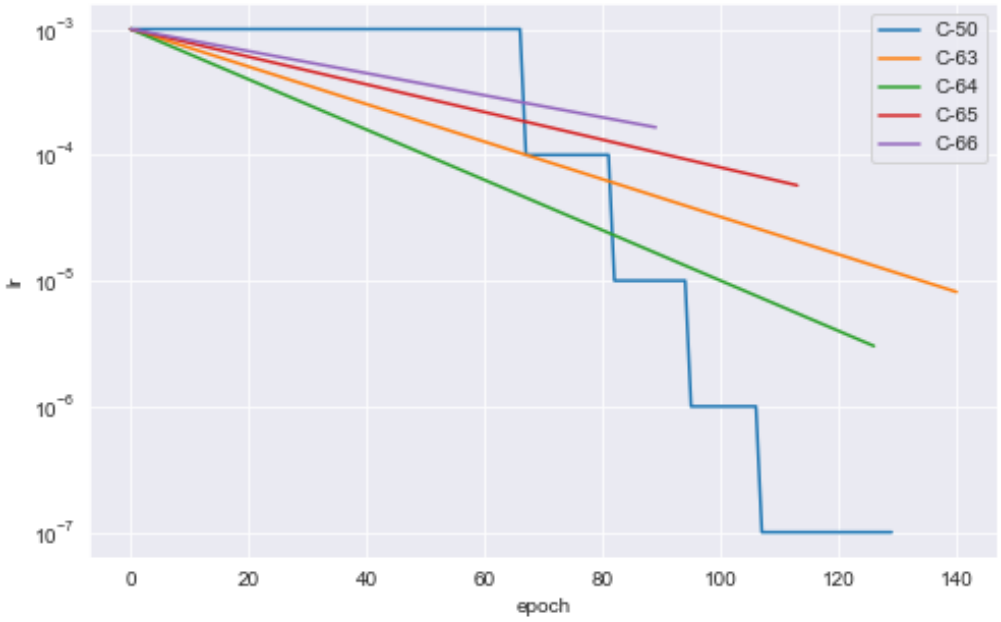


Figure 32: Learning rate annealing (log scale)

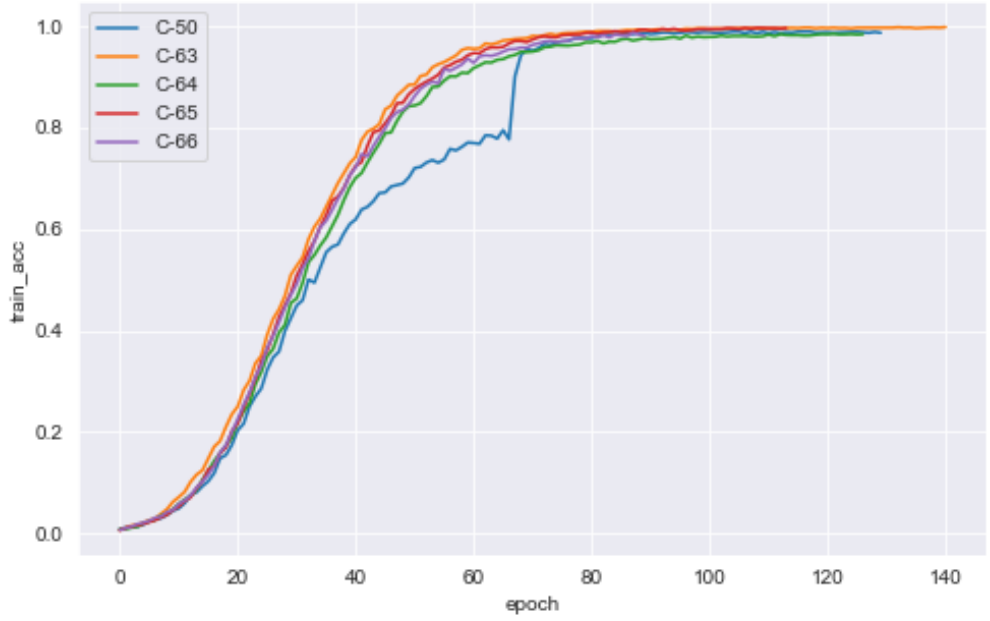


Figure 33: Training accuracy with LR annealing

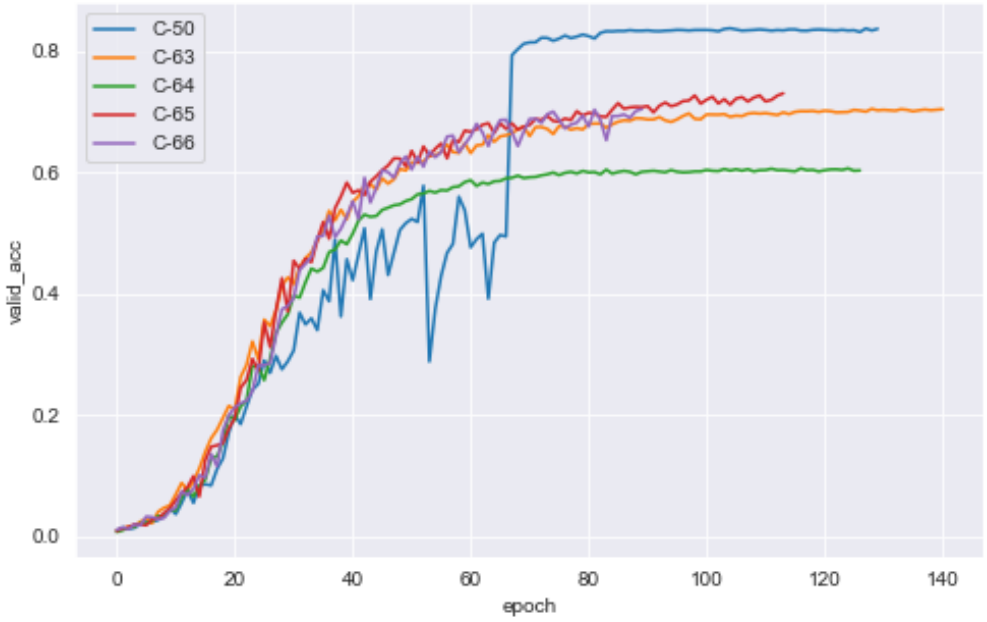


Figure 34: Validation accuracy with LR annealing

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] # 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 of 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 in the 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 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 4.1 M parameters and only 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 an 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, then finetune the best model from step one to reflect car models distribution on the streets of Poland.
3. Prepare model for deployment, create a simple Android application that allows 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 35: 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 attached (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 - images and 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 in the input data. Instead of that, the focus is on obtaining those redundant feature maps in a cost-efficient way.

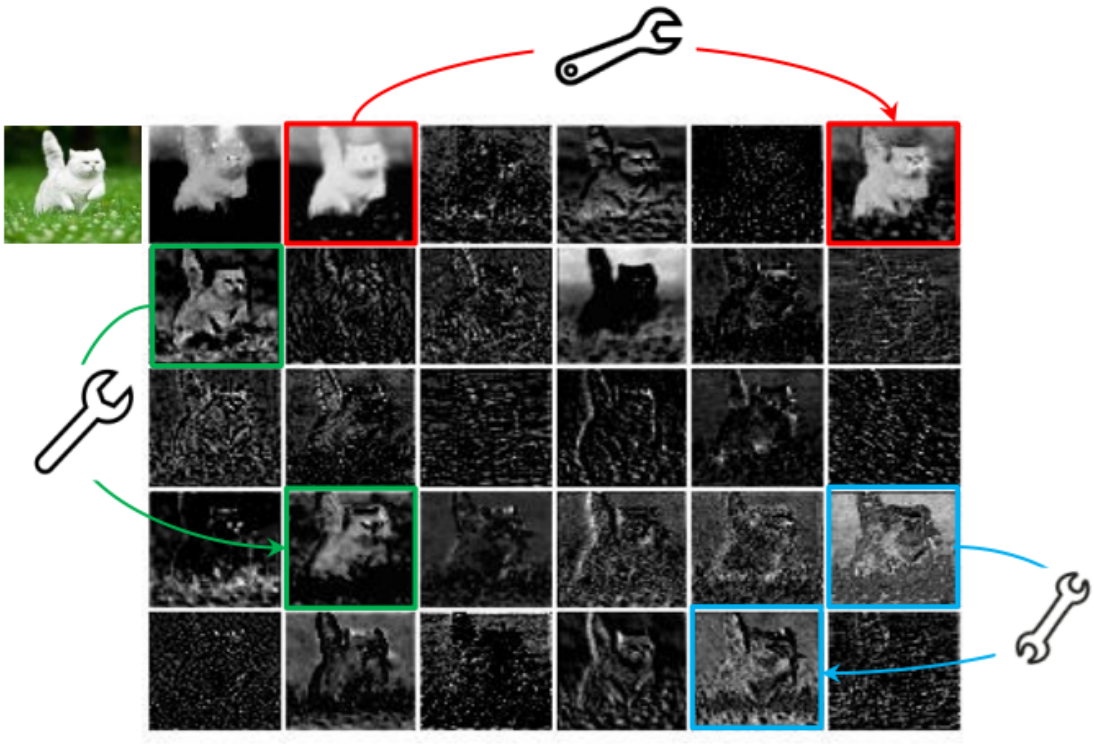
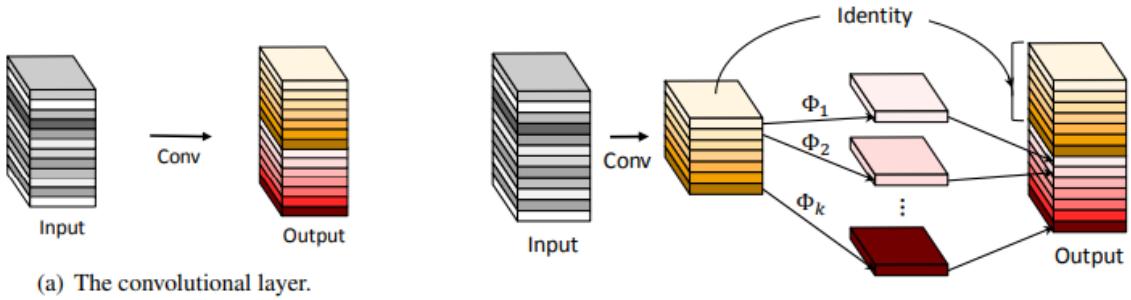


Figure 36: Redundant feature maps from ResNet-50 (picture from paper)

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.



(a) The convolutional layer.

(b) The Ghost module.

Figure 37: 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:

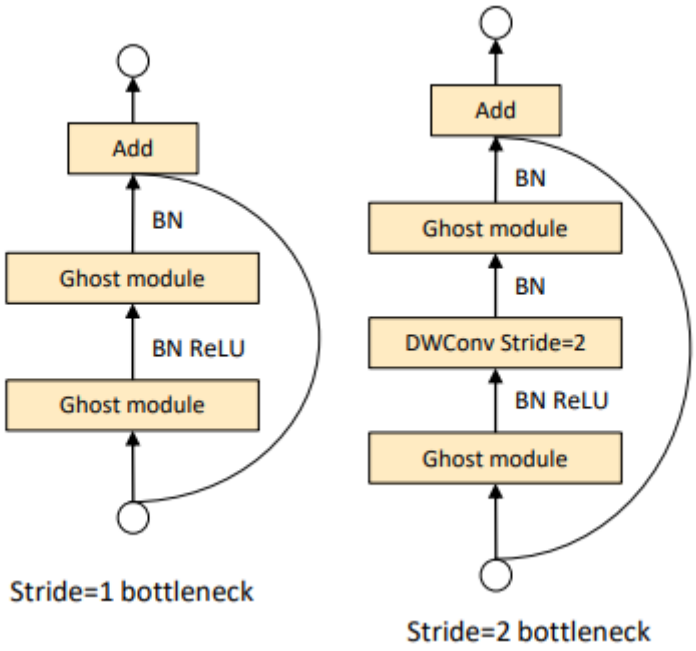


Figure 38: GhostBottleneck (picture from paper)

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
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

Input	Operator	#exp	#out	SE	Stride
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 popular designs like MobileNet or ShuffleNet getting better accuracy with less FLOPS and latency.

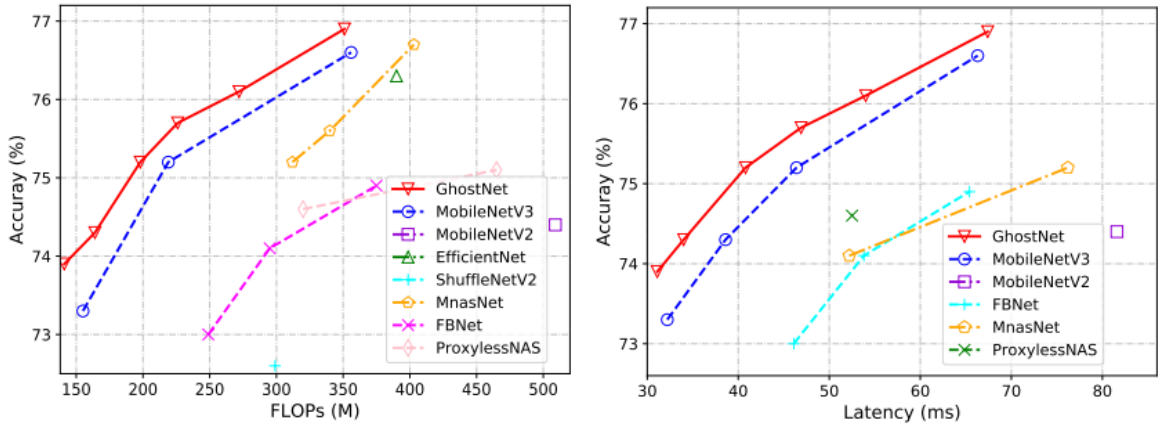


Figure 39: GhostNet comparison with some other mobile architectures (pictures from paper)

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

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. The repository contains 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 the 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
- `notebooks` - folder containing additional Jupyter notebooks (e.g. for normalization parameters calculation)
- `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 to 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:

```
# 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 - if so, recreate local environment by installing packages from prod_requirements.txt on Colab (after this step, runtime restart and imports re-execution 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 have 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'
```

```yaml
  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
```

```yaml
# 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
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

Experiment tracking is set up using Neptune experiment management tool. The tool has some useful features like:

- Python API and PyTorch Lightning integration
- Customizable logging of training metrics and model hyperparameters, as well as the storage and versioning of model artifacts
- Customizable plots and experiment comparison dashboards live-updated as the training proceeds
- Easy results sharing via HTTP links
- Option to programmatically download experiments results and parameters for postprocessing using API

Neptune logging can be easily enabled by passing a set of parameters through project `config.yml` file:

```
log_to_neptune: True
neptune_username: '<neptune.ai username>'
neptune_project_name: '<neptune.ai project name>'
neptune_api_token: '<neptune.ai API token>'
```

Results of the performed experiments in the discussed project are available under the following link:

[Neptune `cars` project dashboard]

The main dashboard table is configured to summarize all most important information about each experiment:

- Experiment ID
- Experiment state, running time and runtime utilized (`local` or `colab`)
- Architecture name
- Image and batch size
- Number of parameters of the network
- Image preprocessing settings: grayscale conversion, normalization, usage of image or tensor augmentations, usage of bounding boxes
- Loss function used
- Optimizer type and its most important settings (learning rate and weight decay)
- Learning rate scheduler type
- Network hyperparameters: dropout rate in the classifier, last layer size
- Number of epochs passed before early stopping was triggered
- Best (minimum) training and validation loss and best (maximum) training and validation accuracy
- Tags linking the experiment to specific sections in documentation
- Additional experiment description

After clicking on a particular experiment ID it is possible to check detailed logs and metrics.

In Parameters tab all experiment parameters that are passed through `config.yml` can be checked.

It is also possible to check multiple experiments and make a comparison between their metrics.

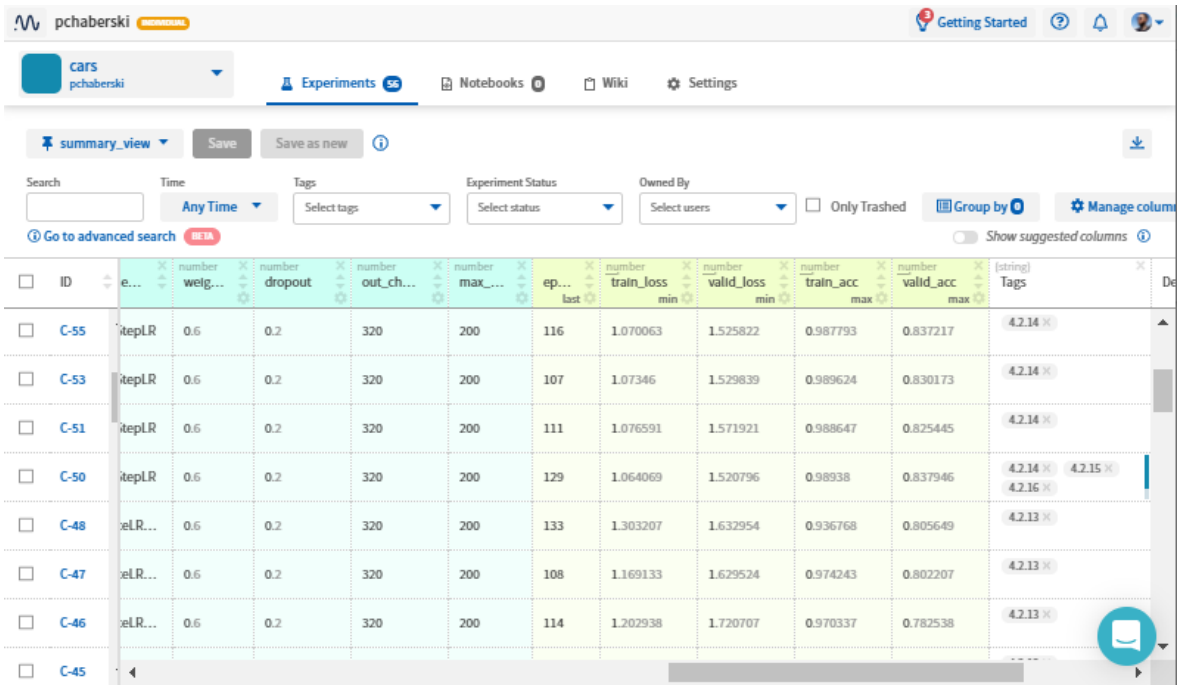


Figure 40: Part of a Neptune main dashboard

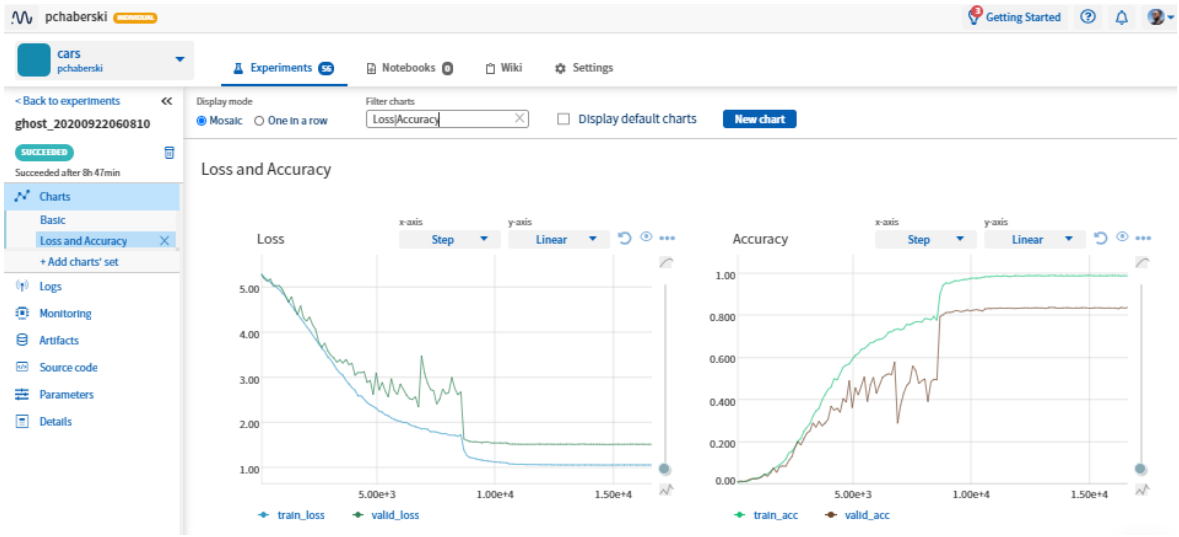


Figure 41: Loss and accuracy plots for particular experiment

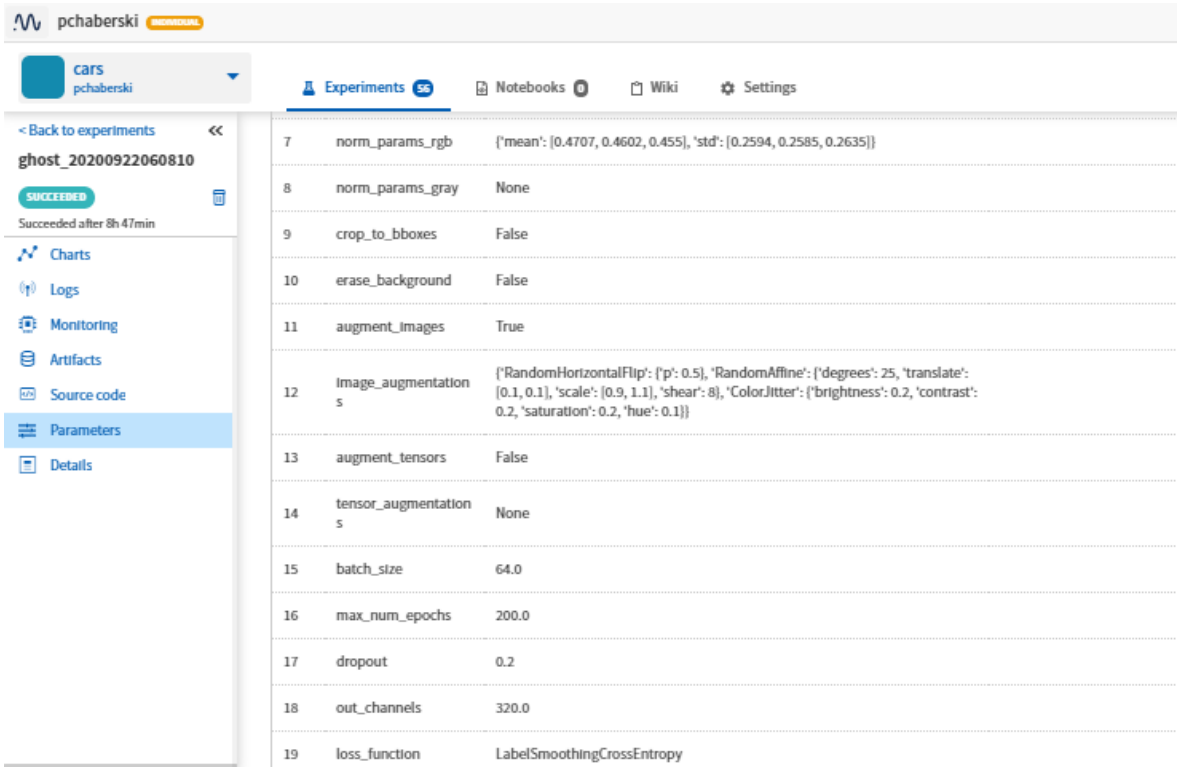


Figure 42: Detailed experiment parameters

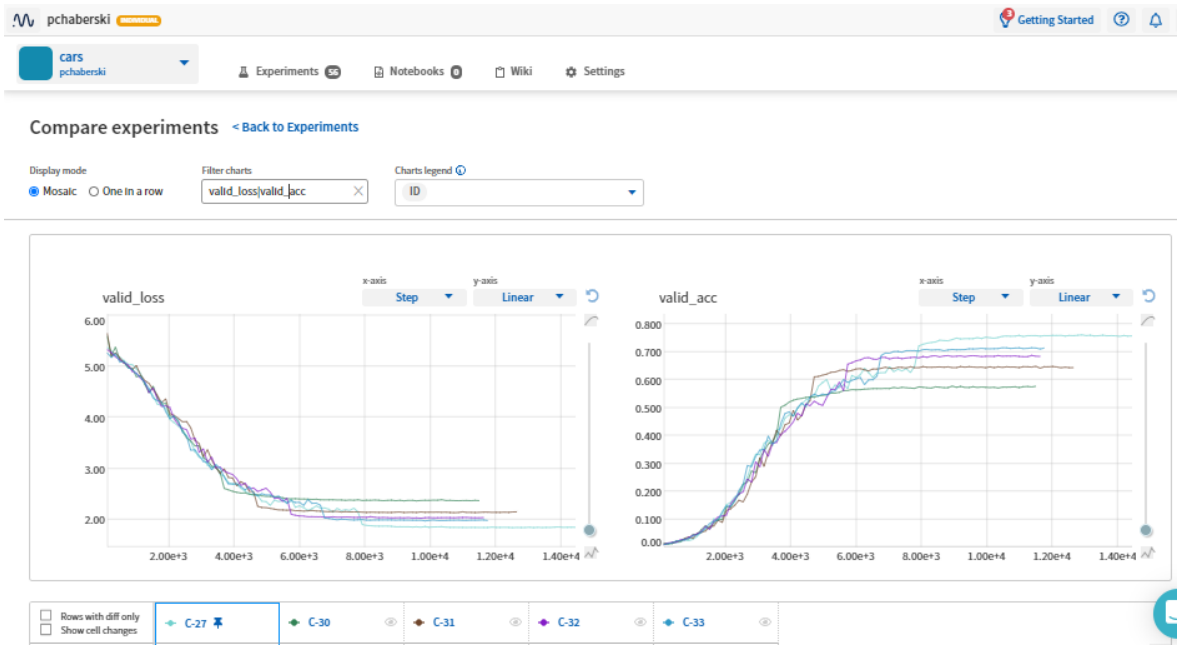


Figure 43: Multiple experiments comparison

4. Results

For the experiments, entire training subset from Stanford Cars Dataset was used for training, and *test* subset was used for validation. There is no additional hold-out testing set so it has to be taken into account that the final accuracy estimate might be somehow biased.

To limit hyperparameter space for the best model search, some assumptions were made at the beginning:

- network is trained from scratch, without using any pretrained weights
- input image size is 227x227 (this assumption results from initial tests on SqueezeNext [6] where this is a minimum image size and all other architectures available in `arch_dict.py` can handle such image size. For GhostNet, minimum image size is 224x224)
- batch size is fixed at 64 mostly because of local GPU memory limitations, however some tests during development phase showed no gain with smaller or larger batch sizes
- Adam with initial learning rate value of 0.001 is chosen as a default optimizer, and may be changed to AdamW [16] when applying weight decay (however SGD was also tested at the development phase, but it was leading to severe overfitting)
- early stopping is triggered when there is no decrease in validation loss for 15 epochs

During experiments, several techniques were used to increase validation accuracy and reduce overfitting, which turned out te be the major issue in training process:

- different loss functions
- pixel value normalization
- various image augmentations
- grayscale conversion

- utilization of bounding boxes
- L2 regularization using weight decay
- dropout rate changing in the classifier module
- last layer size changing
- learning rate scheduling

The search for the best settings was performed in a greedy manner: some arbitrary order of applying different techniques and hyperparameter values was established and after each step the best settings were further augmented using other techniques in order, however a few step-backs and sanity checks were made in the process.

The entire process of obtaining the best model is described step-by-step in section 4.2.

4.1. Best model

[Neptune charts]

The best model that was obtained during the process achieved 83.79 % top-1 accuracy on the validation set after training for 129 epochs with AdamW optimizer and Label Smoothing Cross Entropy function. Best metrics scores for that model are:

Metric	Value
Min. training loss	1.064
Min. validation loss	1.521
Max. training accuracy	98.93%
Max. validation accuracy	83.79%

The best model still shows significant overfitting so there might be some space for further improvement. However, taking into account that the same model achieves 73.98% on ImageNet dataset suggests that the score of 83.79% on the Stanford Cars Dataset is quite decent. While Stanford Cars Dataset contains much less classes (196 in comparison to 1000 in ImageNet), those classes seem harder to distinguish and the dataset itself is much smaller.

It is also important to notice, that due to the lesser number of classes, the size of the last layer was reduced during tests - instead of passing 1280-channel input to the classifier, only 320 channels are passed, which results in the total reduction of parameter count

from 4.2 million to slightly over 3 millions.

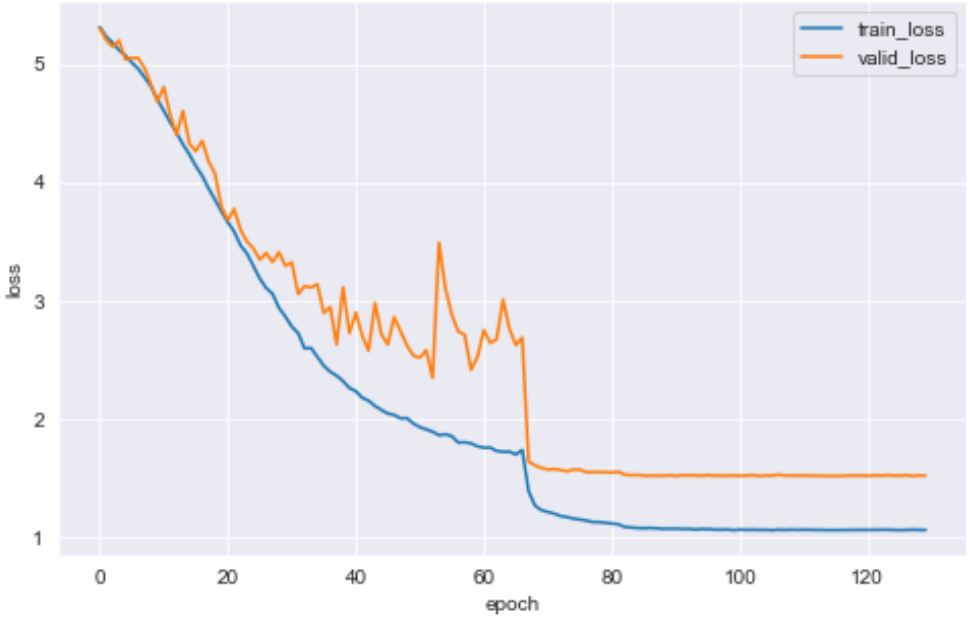


Figure 44: Training and validation loss of the best model (C-50)

The full set of settings and hyperparameters used to train the best performing model is listed below:

- runtime:
  - `colab`
- architecture:
  - `GhostNet`
- num_params:
  - `3041412.0`
- img_size:
  - `[227, 227]`
- grayscale:
  - `False`
- normalize:
  - `True`
- norm_params_rgb:

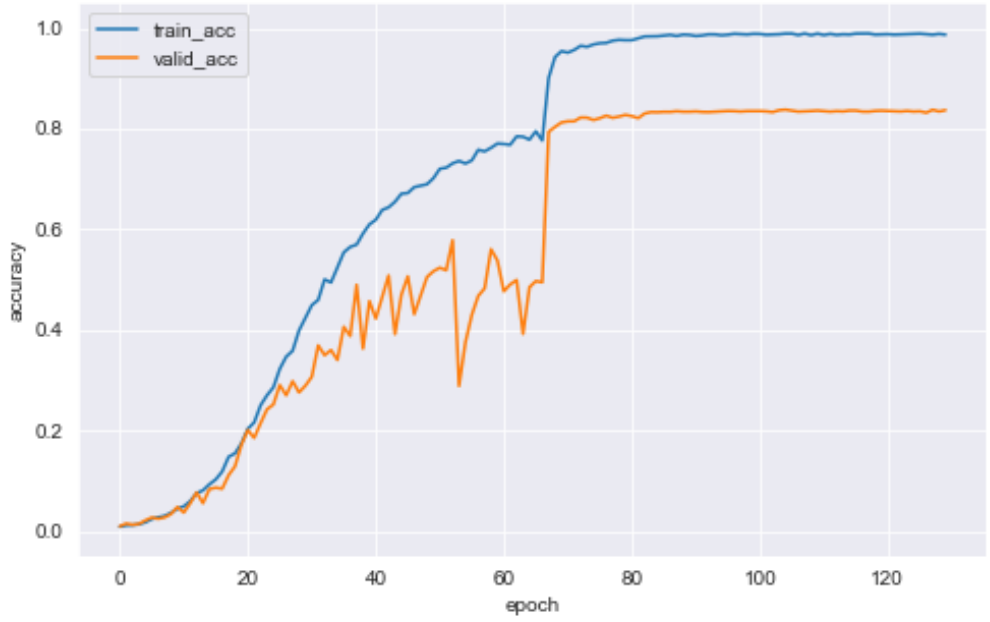


Figure 45: Training and validation accuracy of the best model (C-50)

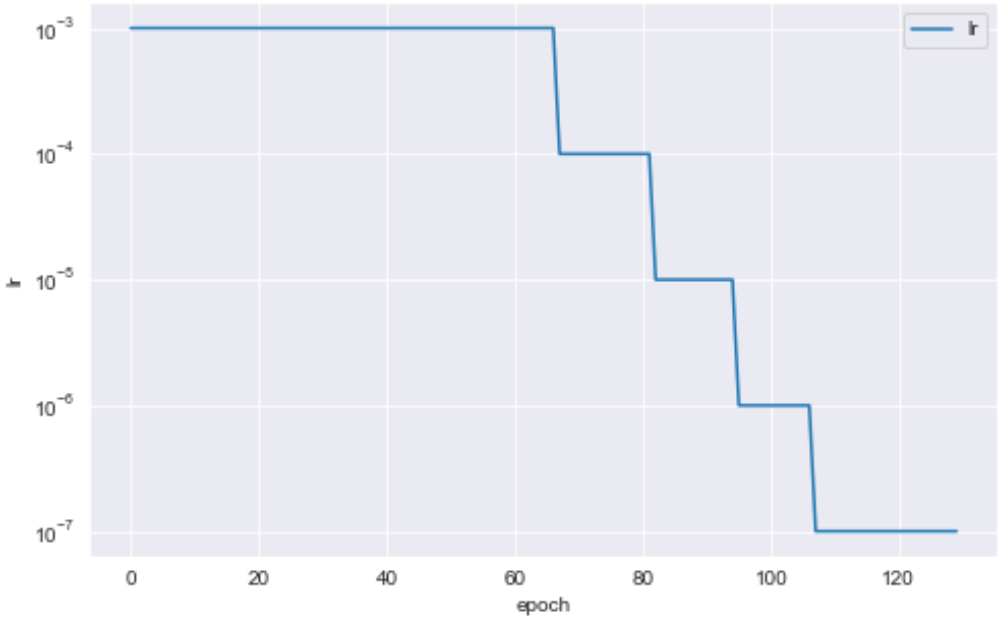


Figure 46: Learning rates for the best model (C-50)

  - `{'mean': [0.4707, 0.4602, 0.455], 'std': [0.2594, 0.2585, 0.2635]}`
- norm_params_gray:
  - `None`
- crop_to_bboxes:
  - `False`
- erase_background:
  - `False`
- augment_images:
  - `True`
- image_augmentations:
  - `{'RandomHorizontalFlip': {'p': 0.5}, '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:
  - `False`
- tensor_augmentations:
  - `None`
- batch_size:
  - `64.0`
- max_num_epochs:
  - `200.0`
- dropout:
  - `0.2`
- out_channels:
  - `320.0`
- loss_function:
  - `LabelSmoothingCrossEntropy`
- loss_params:
  - `None`
- optimizer:
  - `AdamW`
- learning_rate:
  - `0.001`
- weight_decay:

  - `0.6`
- all_optimizer_params:
  - `{'lr': 0.001, 'weight_decay': 0.6}`
- lr_scheduler:
  - `MultiStepLR`
- lr_scheduler_params:
  - `{'gamma': 0.1, 'milestones': [67, 82, 95, 107]}`

4.2. Experiments step-by-step

The table below presents the summary of model accuracy scores for all experiments along with a brief information of techniques used in training. Full and interactive comparison is available through Neptune dashboard. Also, all experiments results (parameters and logged metrics) are archived in a text file on GitHub.

	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	Added RGB normalization	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: horiz. flip, affine, erasing, color jitter	93.68%	38.68%
C-7	Augmentations: horizontal flip, affine, color jitter	99.73%	54.28%
C-8	Grayscale: with normalization, no augmentations	99.49%	6.58%
C-9	Grayscale: with normalization, no augmentations	97.13%	8.68%
C-10	Training set cropping with bounding boxes	7.58%	3.91%
C-11	Training set cropping + background erasing	4.36%	3.07%
C-12	Grayscale: normalization, best RGB augmentations	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%

	experiment description	train_acc	valid_acc
C-20	Dropout rate tests: dropout = 0.4	96.52%	64.88%
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%

	experiment description	train_acc	valid_acc
C-64	LR annealing test: exponentiation base = 0.955	98.49%	60.70%
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

[Neptune comparison]

The first comparison was between standard Cross Entropy loss function and Label Smoothing Cross Entropy. Label smoothing in classification tasks shows some regularization capability [15] resulting from a change in a standard Cross Entropy loss definition.

$$\text{cross-entropy loss} = (1-\epsilon)ce(i) + \epsilon \sum \frac{ce(j)}{N}$$

Figure 47: Label Smoothing Cross Entropy definition

In the Label Smoothing Cross Entropy definition `ce(i)` denotes standard Cross Entropy loss, `epsilon` stands for the label smoothing coefficient being a small positive number, and `N` is the number of classes. This modification results in forcing the model to predict not exactly `1` for correct class and `0` for other classes, but instead somehow *smoothed* values of `1 - epsilon` for correct class and `epsilon` for others.

The comparison shows that indeed with all other hyperparameters fixed, label smoothing allows to achieve a slightly better validation accuracy with the training loss decreasing slower, however the overfitting effect is still very large and in assumed setup results in triggering early stopping after only 26 epochs.

Metric	C-1 (CE Loss)	C-2 (LSCE Loss)
Min. training loss	0.296	1.133
Min. validation loss	4.849	4.873
Max. training accuracy	92.49%	98.89%

Metric	C-1 (CE Loss)	C-2 (LSCE Loss)
Max. validation accuracy	8.15%	9.12%

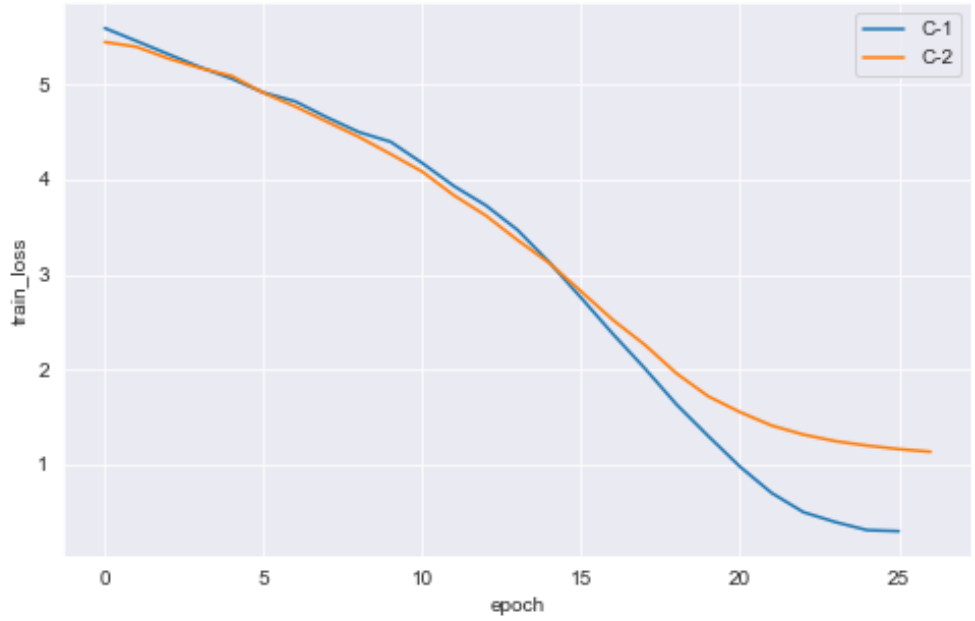


Figure 48: Training loss values for CE Loss (C-1) and LSCE Loss (C-2)

4.2.2. Normalization

[Neptune comparison]

The next step in the process was adding normalization to the data using mean and standard deviation calculated on the training set (see `normalization_coeffs.ipynb` notebook). Centering the data gave a 3 percentage points in validation accuracy, however faster convergence resulted in even faster training loss drop.

Metric	C-2 (No normalization)	C-3 (RGB normalization)
Min. training loss	1.133	1.075
Min. validation loss	4.873	4.792
Max. training accuracy	98.89%	99.45%

Metric	C-2 (No normalization)	C-3 (RGB normalization)
Max. validation accuracy	9.12%	11.95%

4.2.3. Augmentations

[Neptune comparison]

The first milestone experiment series was achieved thanks to adding training data augmentations to the model from C-3 experiment. The transformations that were tested were:

- Random horizontal flip
- Random affine transform
- Random erasing
- Color jittering

With C-3 experiment as the baseline, four combinations of the above-mentioned transformations were tested:

- C-4: `RandomHorizontalFlip` + `RandomAffine`
- C-5: `RandomHorizontalFlip` + `RandomAffine` + `RandomErasing`
- C-6: `RandomHorizontalFlip` + `RandomAffine` + `RandomErasing` + `ColorJitter`
- C-7: `RandomHorizontalFlip` + `RandomAffine` + `ColorJitter`

The results showed that the augmentations in general helped to achieve a very large increase in validation accuracy (from 11.96% to 54.28% in the best case) while reducing overfitting significantly. As for the particular transformations, it turned out that in the tested setup `RandomErasing` did not help, probably introducing too much variance in the training set combined with other augmentations.

Metric	C-3	C-4	C-5	C-6	C-7
Min. training loss	1.075	0.990	1.122	1.315	1.003
Min. validation loss	4.792	2.813	3.339	3.452	2.744
Max. training accuracy	99.45%	99.76%	98.12%	93.68%	99.73%
Max. validation accuracy	11.95%	51.92%	38.08%	38.68%	54.28%

(a) (b)

Figure 49: Original (a) and normalized (b) image

4.2.4. Grayscale conversion

[Neptune comparison]

Testing how the network will behave after converting input images to grayscale before the training came from the idea, that we want the model to distinguish car models only by the details of design, and obviously not to focus on irrelevant differences such as body color. To adapt the original GhostNet architecture to be able to process also 1-channel images, a small customization was made to the first layer of the network by adding `img_channels` parameters, so that the initial convolution could work on any number of channels in general:

```
class GhostNet(nn.Module):
    def __init__(
        self,
        num_classes=1000, img_channels=3, dropout=0.2, out_channels=1280,
        width_mult=1.
    ):
        super().__init__()
        self.num_classes = num_classes
        self.img_channels = img_channels
        self.dropout = dropout
        self.out_channels = out_channels
```

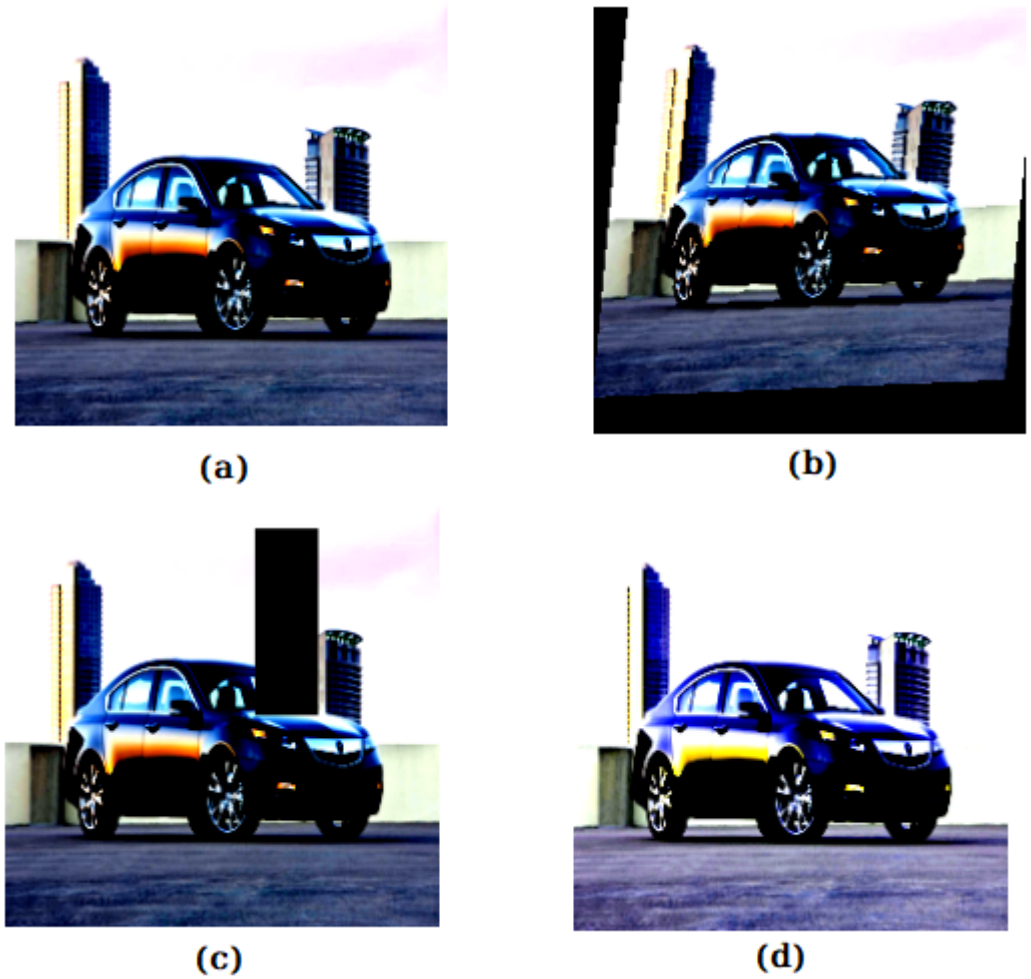


Figure 50: Original normalized image (a); Images with: RandomAffine (b), RandomErasing (c), ColorJitter (d)

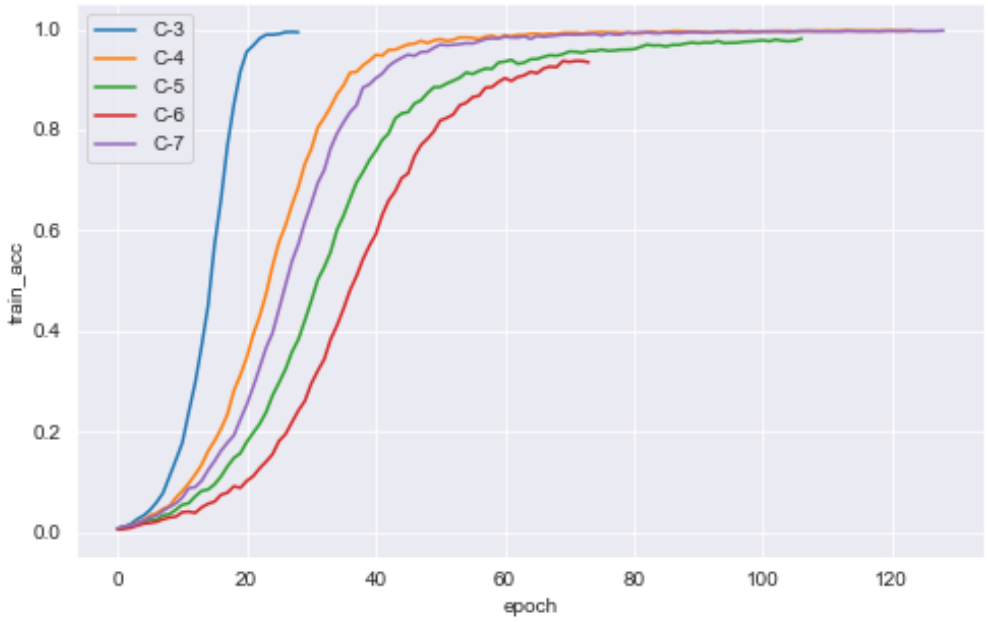


Figure 51: Training accuracy with different augmentations

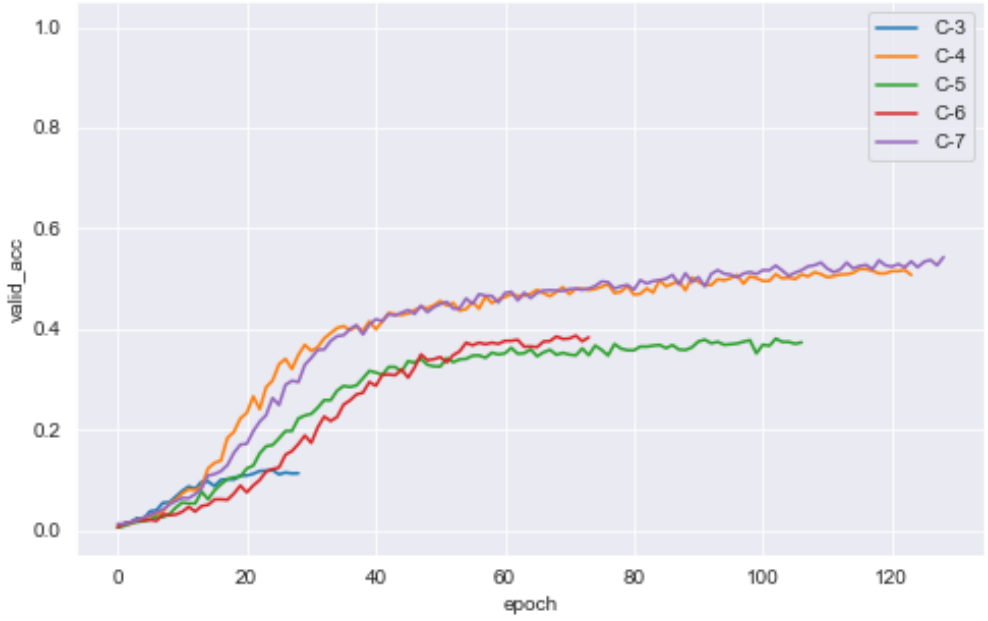


Figure 52: Validation accuracy with different augmentations

```
# ...

        # building first layer
        output_channel = _make_divisible(16 * width_mult, 4)
        layers = [nn.Sequential(
            nn.Conv2d(self.img_channels, output_channel, 3, 2, 1, bias=False),
            nn.BatchNorm2d(output_channel),
            nn.ReLU(inplace=True)
        )]
        input_channel = output_channel

# ...
```

Also some other customizations were made as can be seen above, namely introducing customizable dropout rate in classifier and last layer size by adding `dropout` and `out_channels` parameters. Those will be discussed in sections 4.2.7 and 4.2.8, respectively.

Three comparisons were made taking as baselines identical setups that were previously trained using 3-channel input:

- experiment C-8, being a reflection of C-2 (no normalization and no augmentations)
- experiment C-9, being a reflection of C-3 (normalization added, no augmentations)
- experiment C-12, being a reflection of C-7 (normalization + best augmentations)

The results of these tests clearly show, that probably due to the network design, grayscale conversion brings no gain in model performance (in fact, all comparisons are in favor of RGB variants):

Metric	C-2 (RGB)	C-8 (Gr.)	C-3 (RGB)	C-9 (Gr.)	C-7 (RGB)	C-12 (Gr.)
Min. tr. loss	1.133	1.089	1.075	1.207	1.003	1.017
Min. val. loss	4.873	5.080	4.792	4.746	2.744	2.843
Max. tr. accuracy	98.89%	99.49%	99.45%	97.13%	99.73%	99.67%
Max. val. accuracy	9.12%	6.58%	11.96%	8.68%	54.28%	50.51%

4.2.5. Bounding boxes utilization

[Neptune comparison]

Another idea was to try to somehow utilize car bounding boxes coordinates that are available in Stanford Cars Dataset along with the class labels. The goal of the project was however to get possibly unbiased benchmark on the original test set (despite it was used for model validation), so any operations using bounding box information could be used only on training subset. Another reason for that is that the ultimate objective is to deploy trained model on a mobile device so it cannot use any information that is unavailable during inference to estimate the performance. Using bounding boxes in performance estimation and eventually during real-life prediction would require to stack the discussed classification model with some kind of detector, which would firstly estimate the location of bounding boxes.

Two approaches of utilizing bounding boxes on training set were consecutively tested, taking so far the best C-7 experiment as the baseline:

- C-10: only cropping images to bounding box coordinates before resize
- C-11: cropping mages to bounding boxes and then putting them on the white background of original image size to preserve ratios before resize

Both transformations were intended to get rid of the image background and try to force the network to focus only on relevant image parts and to prevent it from fitting to the background elements.

Figure 53: Original normalized image (a); Image cropped to b-boxes (b); Image with background erased (c)

It turned out that this idea was totally wrong - in the first case (C-10), after crop and resize, all proportions were strongly distorted, which caused a large discrepancy between training and validation data and prevented optimizer from converging. The divergence was even stronger in the second case (C-11), because despite preserving original proportions, the network started to focus only on fitting to the white background instead of car details.

Metric	C-7	C-10	C-11
Min. training loss	1.003	4.570	4.822
Min. validation loss	2.744	5.142	5.196
Max. training accuracy	99.73%	7.58%	4.36%
Max. validation accuracy	54.28%	3.91%	3.07%

4.2.6. Optimizer change and L2 regularization

[Neptune comparison]

For further attempts to reduce overfitting and in consequence increase validation accuracy, one of the most commonly used techniques in parametric machine-learning models training was applied - L2 regularization. Optimizers in PyTorch allow for passing `weight_decay` parameter, which represents the strength of the penalty added for too high model weights. However, as empirical study shows [16], this kind of regularization requires decoupling application of weight decay from optimization steps taken with respect to the loss function when using adaptive algorithms like Adam. So to be able to successfully apply this technique in discussed problem, Adam optimizer was replaced by its variation utilizing decoupled weight decay - AdamW.

With C-7 as a baseline, 5 different `weight_decay` values were tested with AdamW optimizer:

- C-13: `weight_dacay = 0.1`
- C-14: `weight_dacay = 0.2`
- C-15: `weight_dacay = 0.3`
- C-16: `weight_dacay = 0.4`
- C-17: `weight_dacay = 0.5`

The results:

Metric	C-7	C-13	C-14	C-15	C-16	C-17
Min. training loss	1.003	1.023	1.071	1.213	1.193	1.378
Min. validation loss	2.744	2.315	2.089	2.285	2.174	2.301
Max. training accuracy	99.73%	99.44%	98.84%	95.83%	95.95%	90.38%
Max. validation accuracy	54.28%	63.39%	68.50%	61.84%	65.14%	59.95%

It can be observed that adding larger penalty by increasing `weight_decay` in fact reduces overfitting, but at some point the weights become too constrained preventing the model to fit well to training data and therefore limiting validation accuracy increase. However, with all tested values, regularized version managed to improve the score obtained without weight decay.

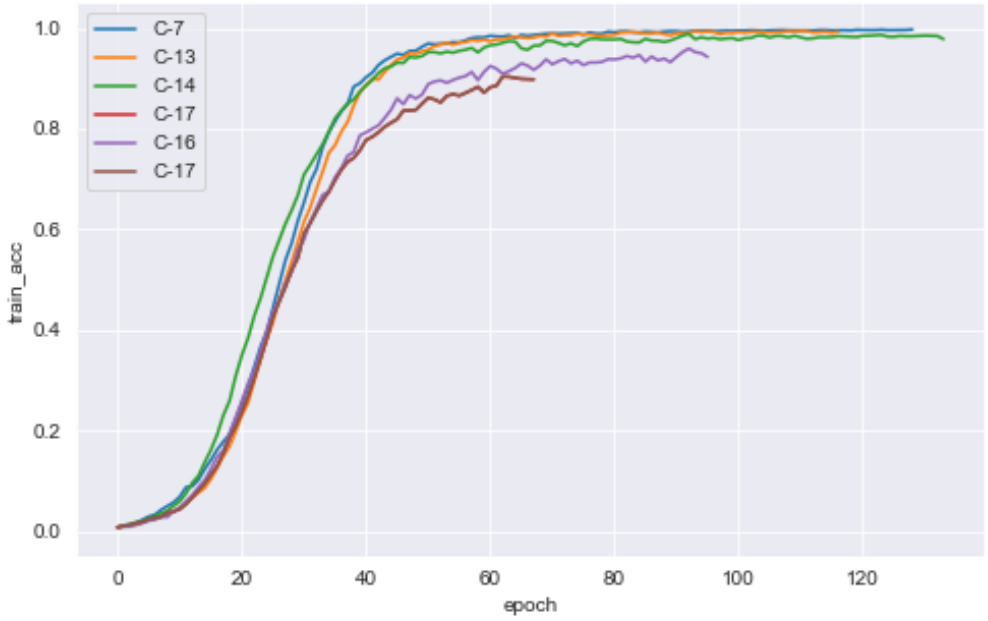


Figure 54: Training accuracy with different values of weight decay

4.2.7. Dropout rate tests

[Neptune comparison]

Basic GhostNet design has dropout rate before last linear layer fixed at 0.2. As men-

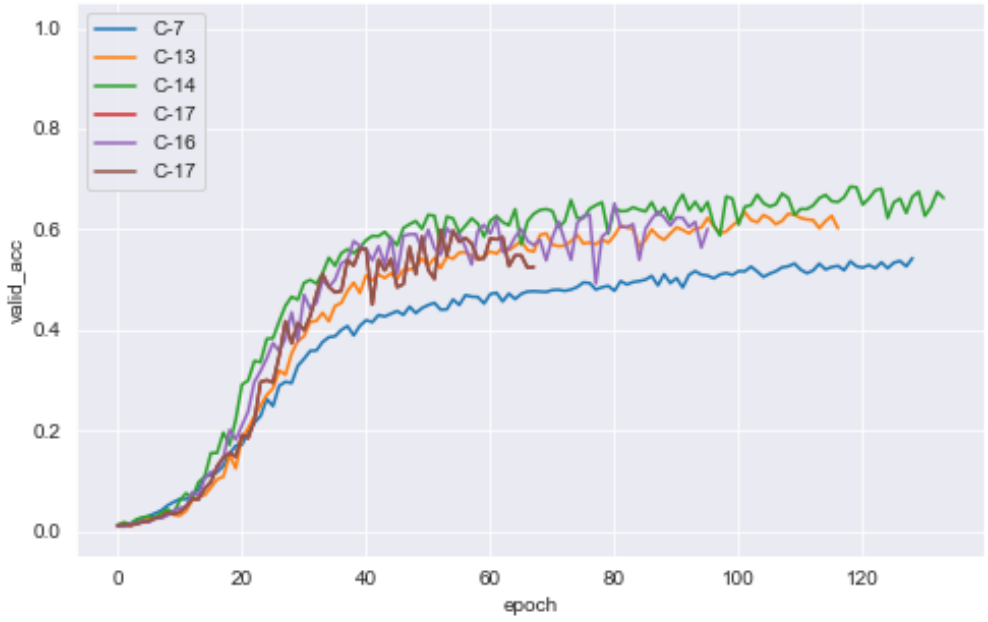


Figure 55: Validation accuracy with different values of weight decay

tioned in 4.2.4 the original design was customized to allow for passing different dropout values. Some other than default values (especially larger, hoping to further reduce overfitting) were checked.

Dropout rate values that were tested with baseline of 0.2 from C-14 experiment setup were:

- C-18: `dropout = 0.1`
- C-19: `dropout = 0.3`
- C-20: `dropout = 0.4`
- C-21: `dropout = 0.5`

The above mentioned experiments show no improvement using values different from default.

Metric	C-14	C-18	C-19	C-20	C-21
Min. training loss	1.071	1.039	1.087	1.173	1.191
Min. validation loss	2.089	2.179	2.108	2.164	2.096
Max. training accuracy	98.84%	99.11%	98.62%	96.52%	96.28%

Metric	C-14	C-18	C-19	C-20	C-21
Max. validation accuracy	68.50%	66.90%	67.81%	64.88%	66.75%

4.2.8. Last layer size tests

[Neptune comparison]

By default, in GhostNet architecture the number of channels in the feature vector passed into the classifier module is fixed at value of 1280 and those channels are finally mapped on the number of classes by the fully connected layer. As mentioned in 4.2.4, this value was parametrized based on the assumption, that it could be strictly associated with the specific output number of classes and the architecture was optimized for 1000-class ImageNet, while Stanford Cars Dataset consists of 196 classes.

The values that were tested with respect to 1280 baseline from experiment C-14:

- C-22: `out_channels = 320`
- C-23: `out_channels = 640`
- C-24: `out_channels = 960`
- C-25: `out_channels = 1600`

It is important to notice, that changing the output channels value strongly affects the total parameter count of the network:

	output channels	number of parameters
C-14	1280	4153090
C-22	320	3041410
C-23	640	3411970
C-24	960	3782530
C-25	1600	4523650

Analysis of the results shows no straightforward relationship between the number of output channels and network's performance on the particular dataset that was used, however the lowest number of channels testes (320) turned out to give slightly better validation accuracy that default with less overfitting, while reducing the number of parameters from 4.15 million to 3.04 million.

Metric	C-14	C-22	C-23	C-24	C-25
Min. training loss	1.071	1.140	1.196	1.108	1.071
Min. validation loss	2.089	2.055	2.250	2.202	2.294
Max. training accuracy	98.84%	97.13%	96.13%	98.23%	98.99%
Max. validation accuracy	68.50%	68.93%	63.13%	64.96%	63.11%

4.2.9. Automatic learning rate scheduling

[Neptune comparison]

The more regularization added, the more difficult it is for optimizer to find the optimal solution with a fixed learning rate what is reflected by more and more jumpy learning curves after certain number of epochs. A natural step in this case is to try to decrease the learning rate after some stagnation starts to emerge in the loss decrease.

The first round of experiments with learning rate schedulers were done using `ReduceLROnPlateau` scheduler from PyTorch, which was set to decrease the learning rate by a factor 0.1 after no decrease in validation loss is observed for 5 consecutive epochs. During the early development phase it was observed that randomness in contents of training batches that results in slightly different learning curves even with the same experiment setup each time, may also result in triggering learning rate decrease at different epochs, and this in turn may affect the final validation accuracy achieved. To test the scale of this phenomenon, 4 attempts of the same experiment were taken and compared with the results obtained without scheduler (C-22):

- C-26
- C-27
- C-28
- C-29

The results are that the learning rate decrease at the right point of training procedure can give a significant benefit, and the choice of that particular moment based on variability in training data feed is also not without significance.

Metric	C-22	C-26	C-27	C-28	C-29
Min. training loss	1.140	0.995	1.014	0.988	1.006
Min. validation loss	2.055	1.903	1.836	1.888	1.874

Metric	C-22	C-26	C-27	C-28	C-29
Max. training accuracy	97.13%	99.82%	99.78%	99.83%	99.78%
Max. validation accuracy	68.93%	74.60%	76.20%	75.14%	74.82%

4.2.10. Controlled learning rate scheduling

[Neptune comparison]

For further investigation of the influence of learning rate drop timing (especially the first drop from 0.001 to 0.0001) on the final validation accuracy, some more experiments with manually set milestones for learning rate decrease were made using `MultiStepLR` scheduler with the same factor of 0.1. Also it was noticed that 4.2.9 the best results were obtained when the scheduler was triggered at the earliest, so the milestones were set to push further in this direction:

With the best take from 4.2.9 (C-27) as the baseline, where automatically triggered milestones were checked to be `[61, 82, 93, 104]`, the experiments were:

- C-30: `milestones = [28, 48, 68, 88]`
- C-31: `milestones = [36, 56, 76, 96]`
- C-32: `milestones = [44, 64, 84, 104]`
- C-33: `milestones = [52, 72, 92, 112]`

Looking at the results and comparing with the baseline it is obvious, that all learning rate drops were triggered too early.

Metric	C-27	C-30	C-31	C-32	C-33
Min. training loss	1.014	1.749	1.321	1.142	1.059
Min. validation loss	1.836	2.364	2.130	2.022	1.971
Max. training accuracy	99.78%	80.66%	95.03%	98.68%	99.60%
Max. validation accuracy	76.20%	57.82%	64.93%	68.79%	71.59%

4.2.11. Weight decay adjustment

[Neptune comparison]

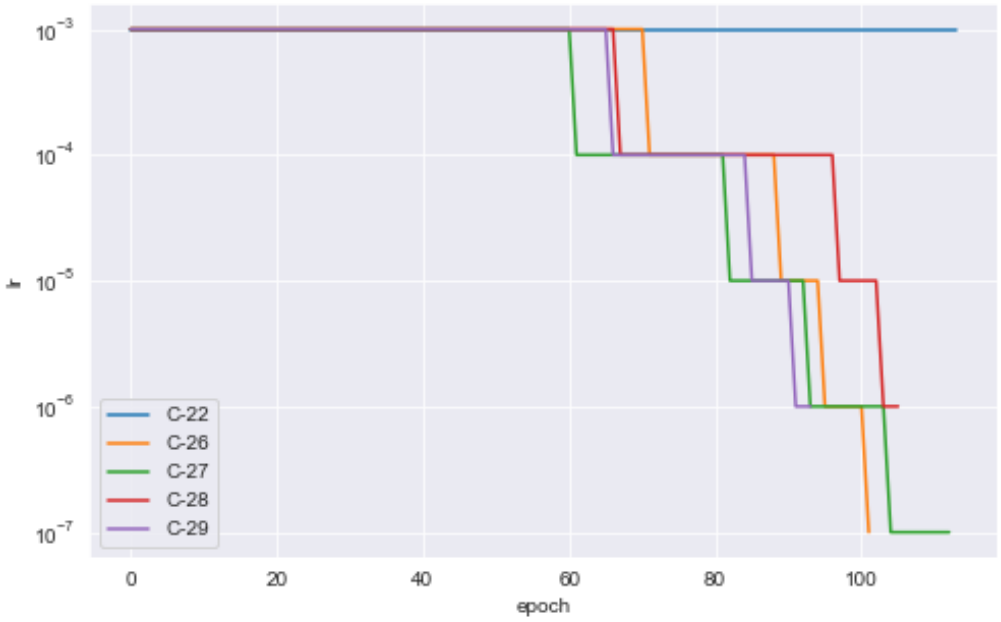


Figure 56: Different LR decrease moments with automatic scheduler

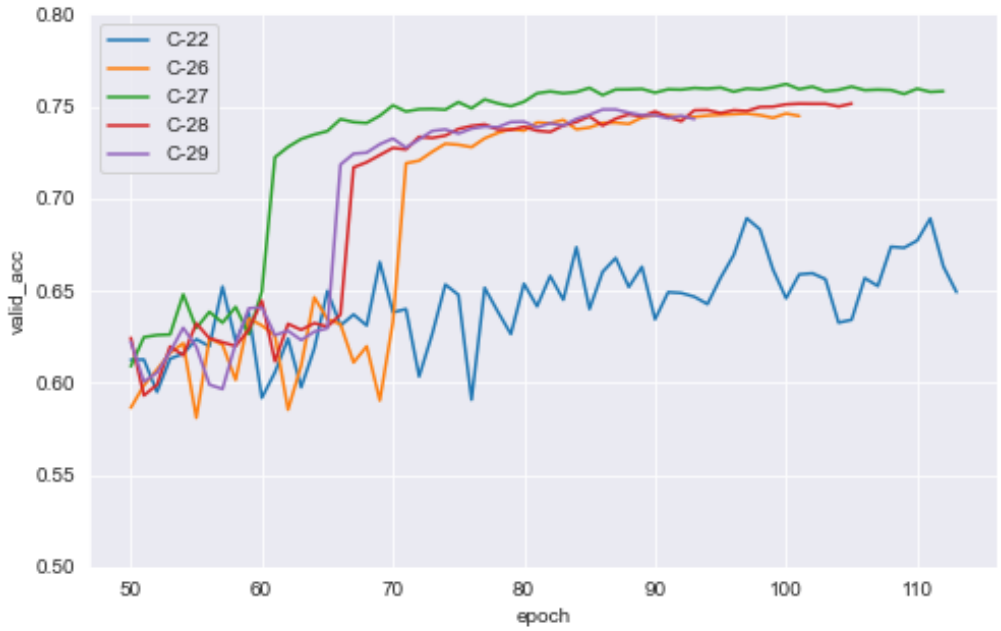


Figure 57: Validation accuracy with different LR drops (from epoch 50)

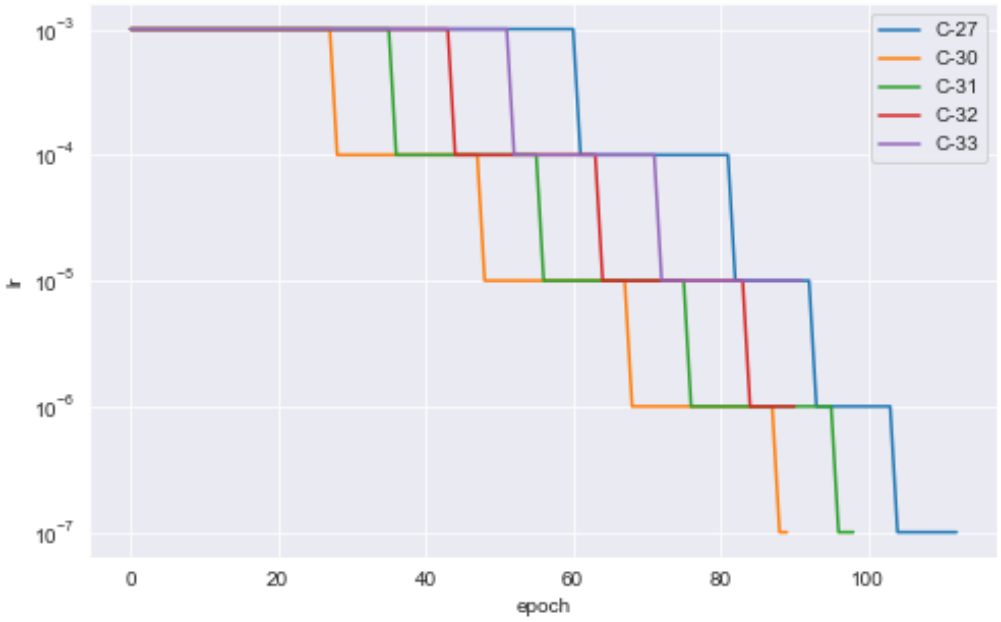


Figure 58: Learning rate drops with manual LR scheduling

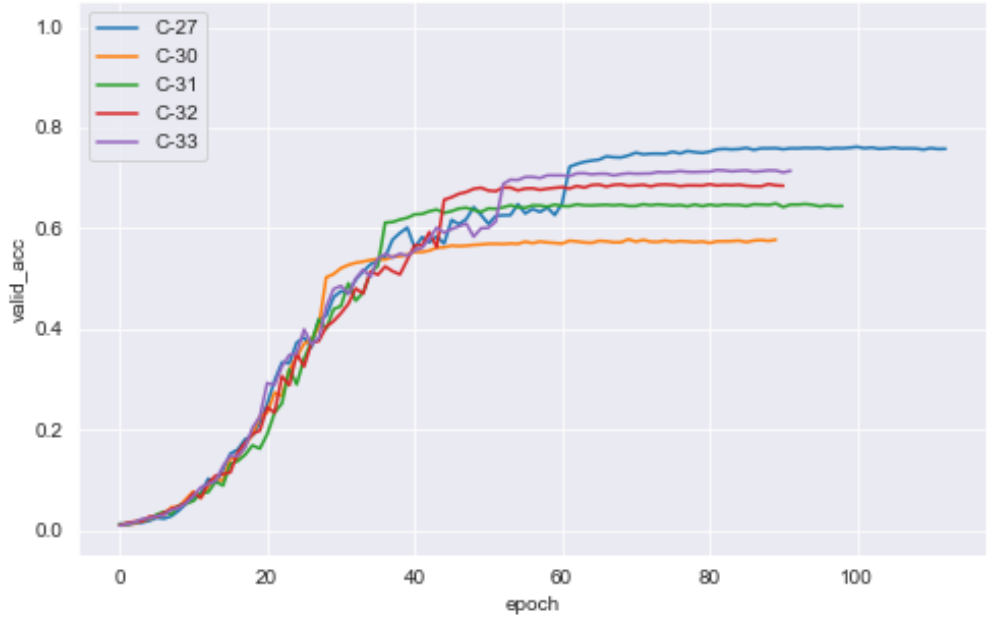


Figure 59: Validation accuracy with manual LR scheduling

After all experiments with learning rates, it seemed reasonable to revisit the most important decisions related to regularization. Especially the optimal weight decay value is strictly dependend on the learning rate used, therefore once again some other values than so-far-best 0.2 were checked at this point. Comparing to C-27, new weight decay values tested were:

- C-36: `weight_decay = 0.5`
- C-37: `weight_decay = 0.3`
- C-38: `weight_decay = 0.4`
- C-39: `weight_decay = 0.6`
- C-40: `weight_decay = 0.7`

Similarly to 4.2.6, some values were to small to prevent overfitting enough while after some point the penalty was to large to let the model fit the data properly (and also triggering learning rate drop too early), but the optimal value from the set tested was much larger than before with fixed learning rate. The best validation accuracy was achieved with weight decay of 0.6.

Metric	C-27	C-36	C-37	C-38	C-39	C-40
Min. training loss	1.014	1.103	1.043	1.048	1.090	1.089
Min. validation loss	1.836	1.669	1.858	1.695	1.563	1.802
Max. training accuracy	99.78%	98.84%	99.57%	99.37%	98.67%	99.24%
Max. validation accuracy	76.20%	79.40%	74.44%	78.82%	82.55%	75.12%

Also it has to be noted, that due to different course of learning process resulting from different regularization, automatic `ReduceLROnPlateau` LR scheduling has been reestablished for the time being.

4.2.12. Dropout rate verification

[Neptune comparison]

Another step-back after introducing learning rate scheduling was to verify the dropout rate that was fixed before. With the C-39 experiment as the baseline, re-checked values were:

- C-41: `dropout = 0.3`
- C-42: `dropout = 0.4`

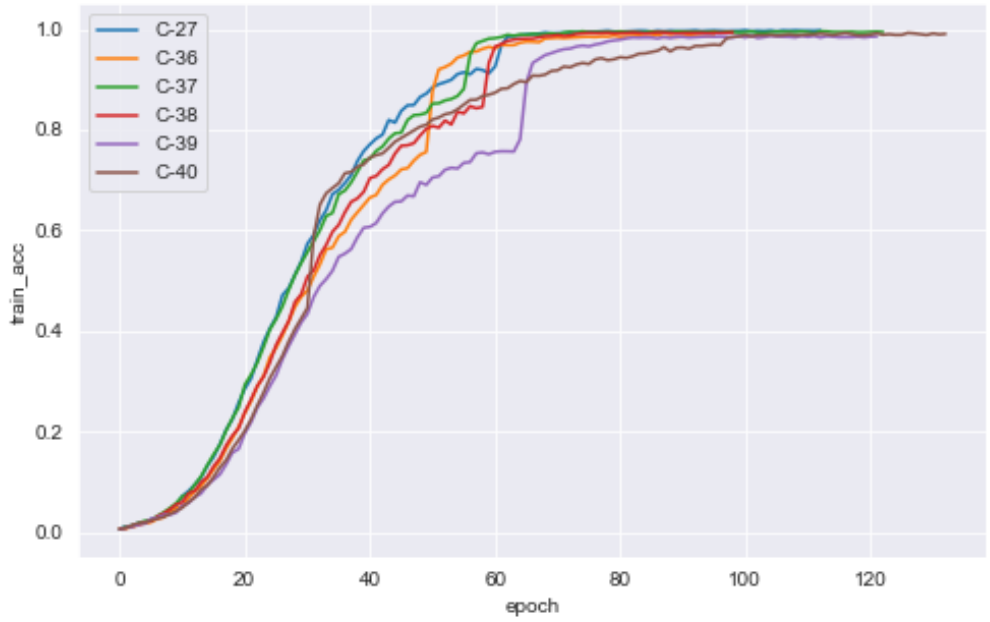


Figure 60: Training accuracy with different weight decay ant auto-scheduling

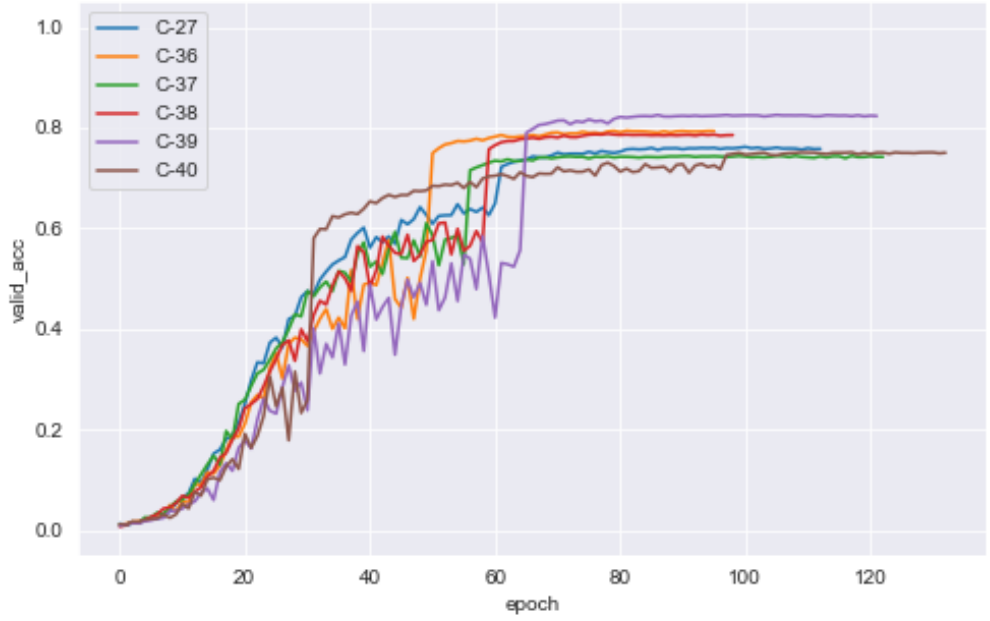


Figure 61: Validation accuracy with different weight decay and auto-scheduling

- C-43: `dropout = 0.5`
- C-44: `dropout = 0.25`

The tests confirmed that the default value seems to be optimal, since all experiments fared worse than the 0.2 baseline, moreover the closer the default value, the higher the validation accuracy.

Metric	C-39	C-41	C-42	C-43	C-44
Min. training loss	1.090	1.111	1.270	1.270	1.087
Min. validation loss	1.563	1.571	1.646	1.692	1.562
Max. training accuracy	98.67%	98.49%	95.34%	96.08%	98.79%
Max. validation accuracy	82.55%	82.08%	79.57%	77.87%	82.45%

4.2.13. Additional augmentations tests

[Neptune comparison]

In attempt to further decrease overfitting some additional augmentations were tested (and also `RandomErasing` was tried again). Every new transformation was added separately to the existing set of augmentations:

- C-45: `RandomResizedCrop`
- C-46: `RandomRotation`
- C-47: `RandomPerspective`
- C-48: `RandomErasing`

Figure 62: Resized crop (a); Rotation (b); Perspective (c)

The results show that adding more augmentations make it too hard for the model to fit the data and therefore limits the validation accuracy increase.

Metric	C-39	C-45	C-46	C-47	C-48
Min. training loss	1.090	1.162	1.203	1.169	1.303
Min. validation loss	1.563	1.699	1.721	1.630	1.633
Max. training accuracy	98.67%	97.56%	97.03%	97.42%	93.68%
Max. validation accuracy	82.55%	78.73%	78.25%	80.22%	80.56%

4.2.14. Learning rate scheduler adjustment

[Neptune comparison]

Remembering the influence of the exact moment of learning rate drop on the model performance, some additional search for the best milestones was performed in the closest neighborhood of so far the best LR drop milestones (from C-39: `[65, 80, 93, 105]`) using `MultiStepLR` scheduler:

- C-50: `milestones = [67, 82, 95, 107]`
- C-51: `milestones = [63, 78, 91, 103]`
- C-53: `milestones = [66, 81, 94, 106]`
- C-55: `milestones = [68, 83, 96, 108]`
- C-56: `milestones = [64, 79, 92, 104]`

The validiation accuracy difference range was narrow, however manual scheduler fine-tuning helped to gain over 1 additional percentage point.

Metric	C-39	C-50	C-51	C-53	C-55	C-56
Min. training loss	1.090	1.064	1.077	1.073	1.070	1.066
Min. validation loss	1.563	1.521	1.572	1.530	1.526	1.560
Max. training accuracy	98.67%	98.94%	98.86%	98.96%	98.78%	98.99%
Max. validation accuracy	82.55%	83.79%	82.54%	83.02%	83.72%	82.79%

4.2.15. Last layer size sanity check

[Neptune comparison]

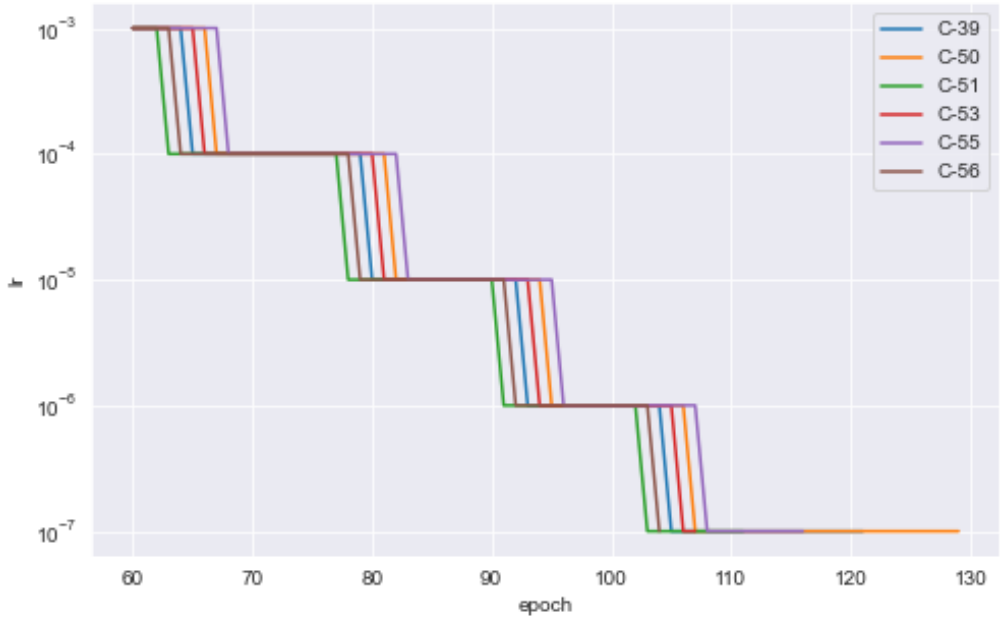


Figure 63: Learning rate drop comparison after scheduler adjustment (from epoch 60)

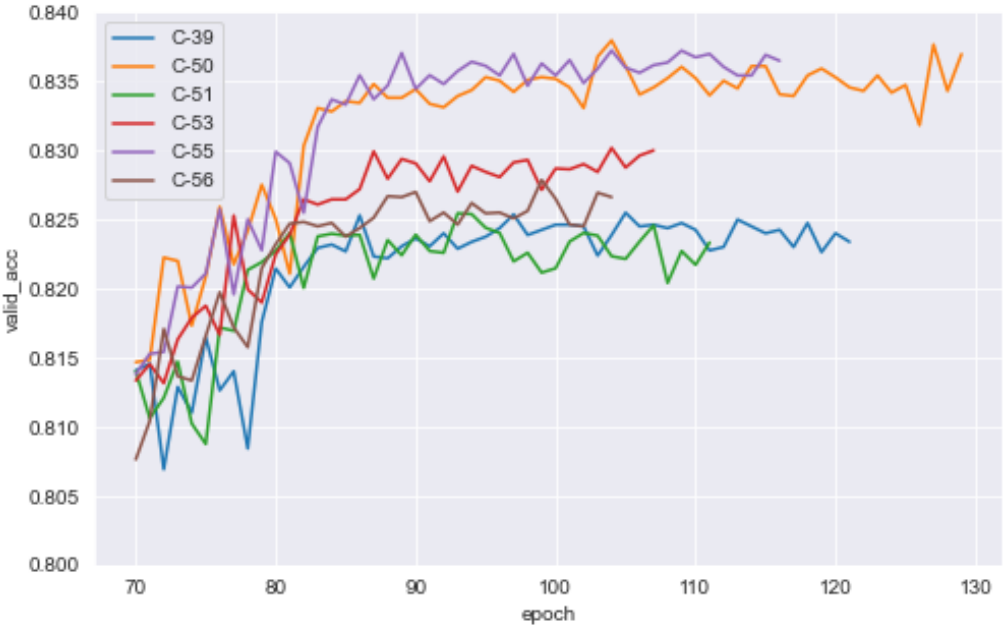


Figure 64: Validation accuracy differences after scheduler adjustment (from epoch 70)

To make sure that the previous choice of the output number of channels made in 4.2.8 is still valid after learning rate scheduling introduction, a comparison between the best model (C-50) and the model with 1280 output channels (C-58) was made, with `ReduceLROnPlateau` scheduler for the latter. The results confirmed that the reduced number of channels work well applied to the discussed problem, since the model with original `out_channels` showed worse validation accuracy and larger overfitting.

Metric	C-50	C-58
Min. training loss	1.064	1.050
Min. validation loss	1.521	1.720
Max. training accuracy	98.94%	99.44%
Max. validation accuracy	83.79%	78.83%

4.2.16. Learning rate annealing tests

[Neptune comparison]

The last series of tests were aimed to check how the learning process will run with learning rate being reduced smoothly using `LambdaLR` scheduler that will decrease LR each epoch by multiplying it by factor `base**epoch`, where `base` is a number less that `1`, but close to that value. The initial exponentiation base was chosen in way, so that the learning rates would be equal at the epoch when the first LR drop occurs during training the best model (C-50) with standard scheduler. Some other values in the neighborhood were also checked:

- C-63: `lr_lambda = lambda epoch: pow(0.0001/0.001, 1/67)**epoch` (exponentiation base calculated as the common ratio of geometric progression, so that learning rates will *meet* at epoch 67)
- C-64: `lr_lambda = lambda epoch: 0.955**epoch`
- C-65: `lr_lambda = lambda epoch: 0.975**epoch`
- C-66: `lr_lambda = lambda epoch: 0.98**epoch`

For all experiments the learning curves were much smoother than with using standard scheduler, but the overfitting started much sooner and the validation accuracy was low.

Metric	C-50	C-63	C-64	C-65	C-66
Min. training loss	1.064	0.994	1.169	0.975	1.057
Min. validation loss	1.521	2.039	2.314	1.907	1.942
Max. training accuracy	98.94%	99.80%	98.49%	99.66%	98.72%
Max. validation accuracy	83.79%	70.51%	60.70%	73.07%	70.46%

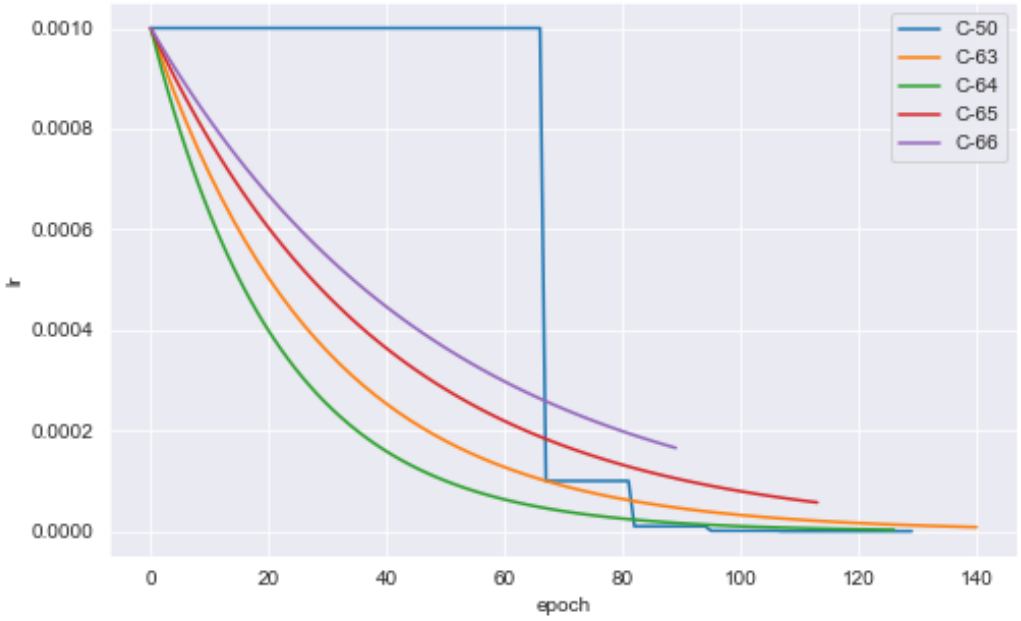


Figure 65: Learning rate annealing (linear scale)

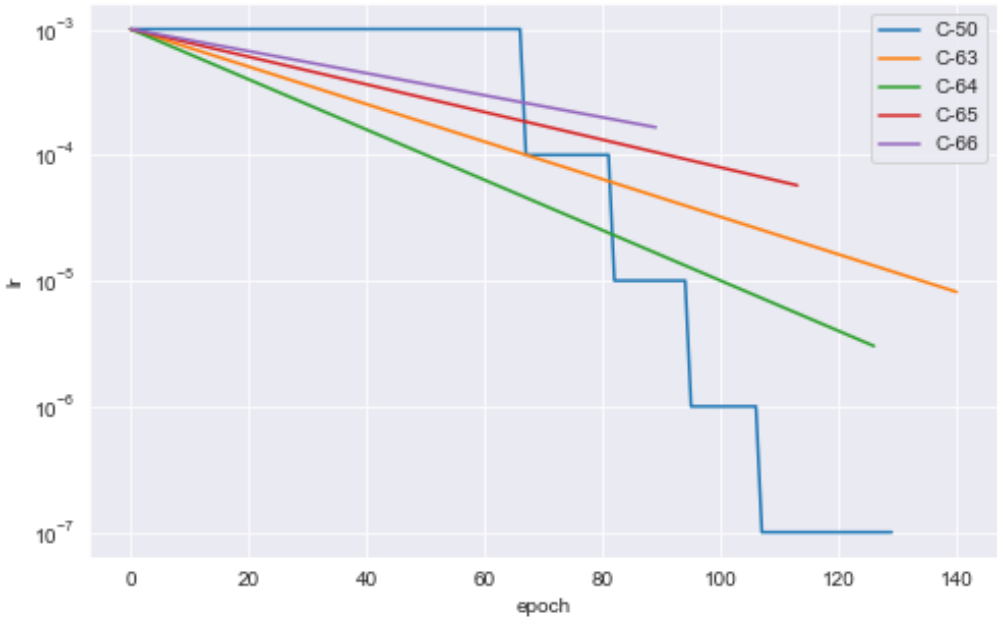


Figure 66: Learning rate annealing (log scale)

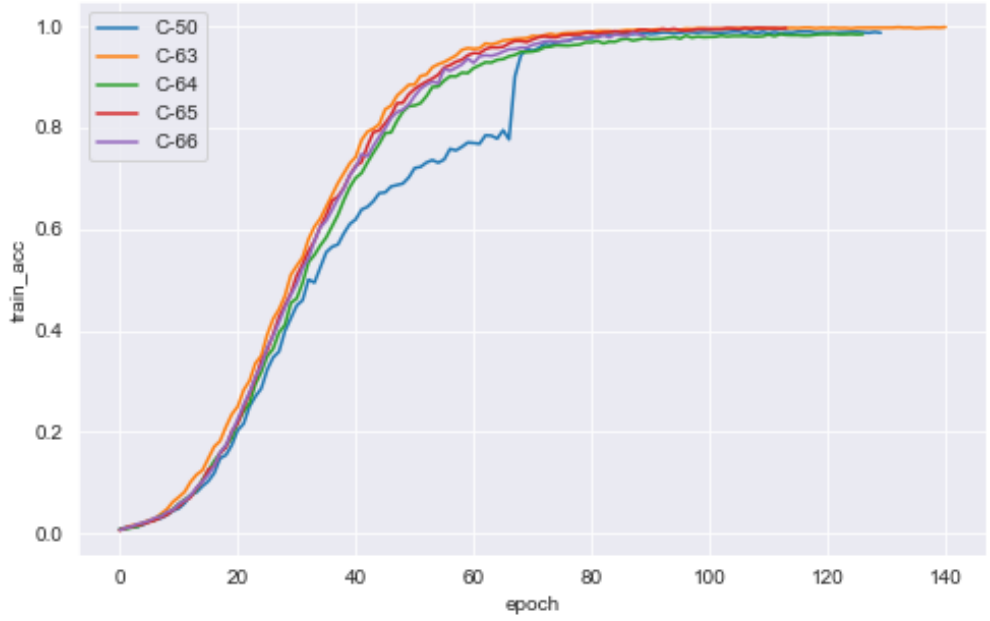


Figure 67: Training accuracy with LR annealing

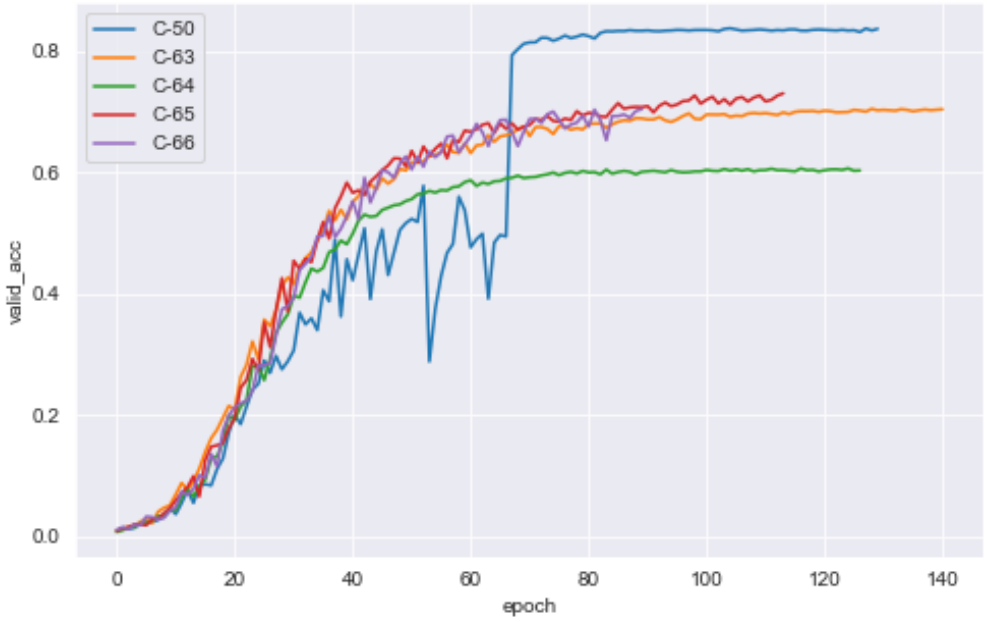


Figure 68: Validation accuracy with LR annealing

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]