Team Project (ADL)

Simon Hampp, Mykhailo Levytskyi, Marvin Wolf, Johannes Decker

2024-09-24

Table of contents

Preface

This is the project documentation for the team project at the Advanced Deep Learning course (winter term 24/25, Munich University of Applied Sciences).

> **Tip**
>
> Text blocks in square brackets [] describe what you should add to the manuscript. When you are done, you can remote them. You can remove all tip-boxes as well.

1 Summary

This project implements an end-to-end system for automatically generating car advertisement articles using deep learning and natural language models. The pipeline begins with image capture (via webcam or from files), processes these images through YOLO-based car detection, and uses a fine-tuned AlexNet for brand and body style classification (86% and 84% accuracy respectively). The system then employs LLMs (Llama3 and Gemma2) to generate contextually relevant article content, with tools for Wikipedia and web search integration providing factual context. The generated text is combined with Stable Diffusion-generated car images in a visually appealing HTML template, which is converted to a final PDF article.

The modular architecture consists of five main components: image capture, object detection/classification, content generation, image generation, and article assembly. Notable technical achievements include efficient preprocessing using YOLO for improved classification accuracy, data augmentation techniques yielding accuracy gains, and a robust article generation system handling rate limits and exceptions gracefully. The final system demonstrates the feasibility of automated, high-quality content generation for automotive marketing materials.

2 Domain

It was decided to generate articles about cars. The articles are structured in a way that they can be used for advertising purposes. The articles contain information about the car brand, the body type, the design aesthetics, key model highlights, the company legacy, historical achievements, advanced technology, and unique selling points. The articles are designed to be visually appealing and include images of the car.

2.1 Article Structure

The generated article follows a consistent structure designed for automotive advertisements:

1. **Header**: Car brand and model title with "Future of Driving" tagline
2. **Introduction**: Opening section with initial vehicle overview
3. **New Model Overview**: Design aesthetics and key model highlights
4. **Brand Heritage**: Company legacy and historical achievements
5. **Innovative Features**: Advanced technology and unique selling points

The layout alternates between text content and supporting images, with figures placed left and right for visual balance. Each image includes a descriptive caption.

The article uses a dark theme with contrasting elements and concludes with a call-to-action button for test drive bookings.

2.2 Data Set

2.2.1 Compilation of Data Sets

At the beginning of the project, after the team agreed on creating a car advertisement article generator, data had to be researched to find the best fitting architecture for the prediction of car brands and body shapes of cars. As the main data set (DS1) the "Car Models 3778" data set (n.d.) from Kaggle was used, because it is the largest and best-labeled cars data set resulting from our research for data. It contains about 193,000 images in the size of 512x512 pixels and consists of exterior view images mainly but contains interior view images as well. Further, it is labeled with 35 usable classes about the model and technical data of the cars.

We decided to use the classes 'brand' and 'body style' of this data set because they should be easiest to predict from exterior view images. Further, the usage of a higher amount of labels would come with an uncontrollable complexity. As an additional data set (DS2) to extend DS1, three further data sets from Kaggle were used and combined into one data set:

- "Stanford Car Dataset by classes folder" data set (n.d.) consisting of 16,185 car exterior view images in various sizes
- "88,000+ Images of Cars" data set (n.d.) consisting of 88,560 car exterior view images in various sizes
- Cars directory of "130k Images (512x512) - Universal Image Embeddings" data set (n.d.) consisting of 8,144 car exterior view images in the size of 512x512 pixels

The labels provided on Kaggle for these three data sets were not useable for our project, so they can be considered as unlabeled data sets.

2.2.2 Preprocessing of Data Sets

As the first step of preprocessing the images contained in DS1 and DS2 were resized to a resolution of 256x256 pixels because that is the input size required by the object detector model. Further .csv files were created for DS1 and DS2 which represent each image and their corresponding labels by one line so that the data sets can be easily loaded for the training of the object detector model.

2.2.2.1 Cleaning of Data Set 1 (DS1)

For DS1 the set of labels for both selected classes 'brand' and 'body style' was reduced based on the distribution of the labels and to avoid a too high total number of them. For the class 'brand' the original DS1 contained 98 labels. To reduce this high number of labels, all brands were removed from DS1 that occurs less than 50 times which resulted in 24 remaining labels for the class 'brand' further the labels 'MercedesAMG' and 'MERCEDESBENZ' were combined to 'MERCEDESBENZ' and the label 'FERRARI' was removed also, because it occurs 64 times what is close to the threshold and rarely compared to the other remaining brands. Further, the original DS1 contained 7 labels for the class 'body style' with a strong imbalance of the label distribution too. To adjust distribution 4 the labels were combined into two labels and the label 'Truck' was removed completely from DS1 because of its very low occurence. The labels 'Convertible' and 'Coupé' were combined in the new label 'sports car' and the labels 'Van' and 'Wagon' were combined in the new label 'family car'.

As already mentioned in Compilation of data sets DS1 contains interior view images too. To remove the interior view images a pre-trained YOLO11x model (Ultralytics n.d.) was used by making predictions for the label 'car'. If the model could not detect any object in the image with a confidence greater than 0.8 for the label 'car', the image was removed from DS1.

Further details about that process are described in Image Classifier. All the mentioned steps of preprocessing of DS1 reduced its number of images to a total of 96,747. Its final distributions of the classes ‘brand’ and ‘body style’ can be seen in the following images.

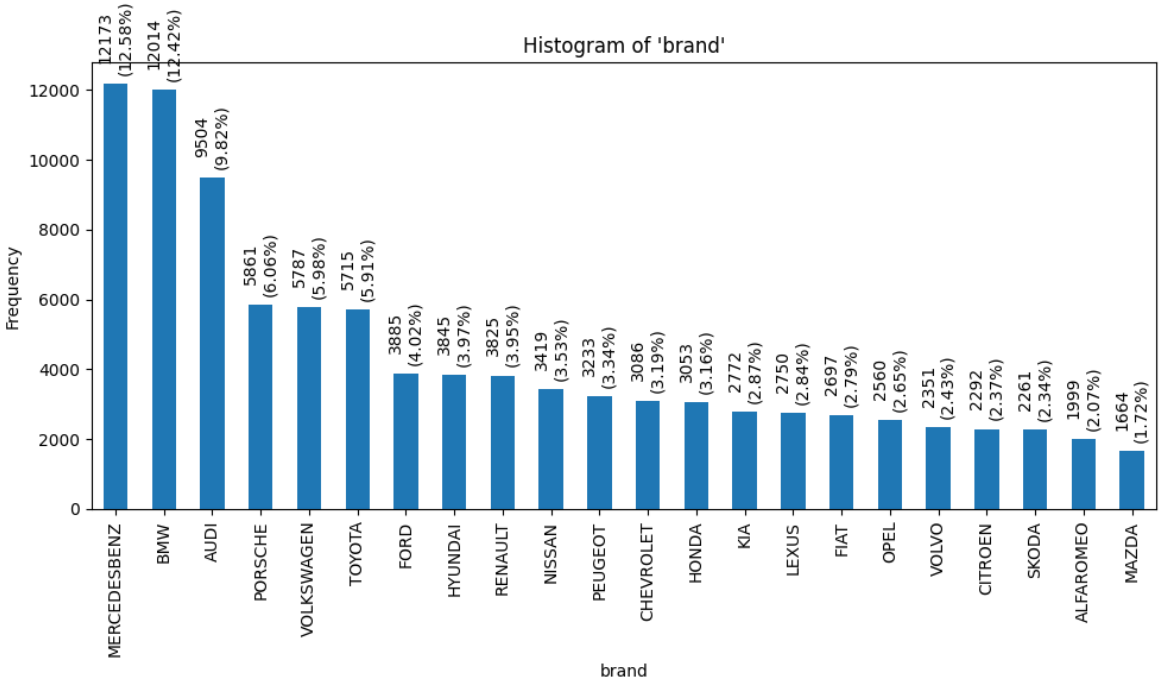


Figure 2.1: Histogram of the label distribution of the class ‘brand’ of the prepocessed DS1

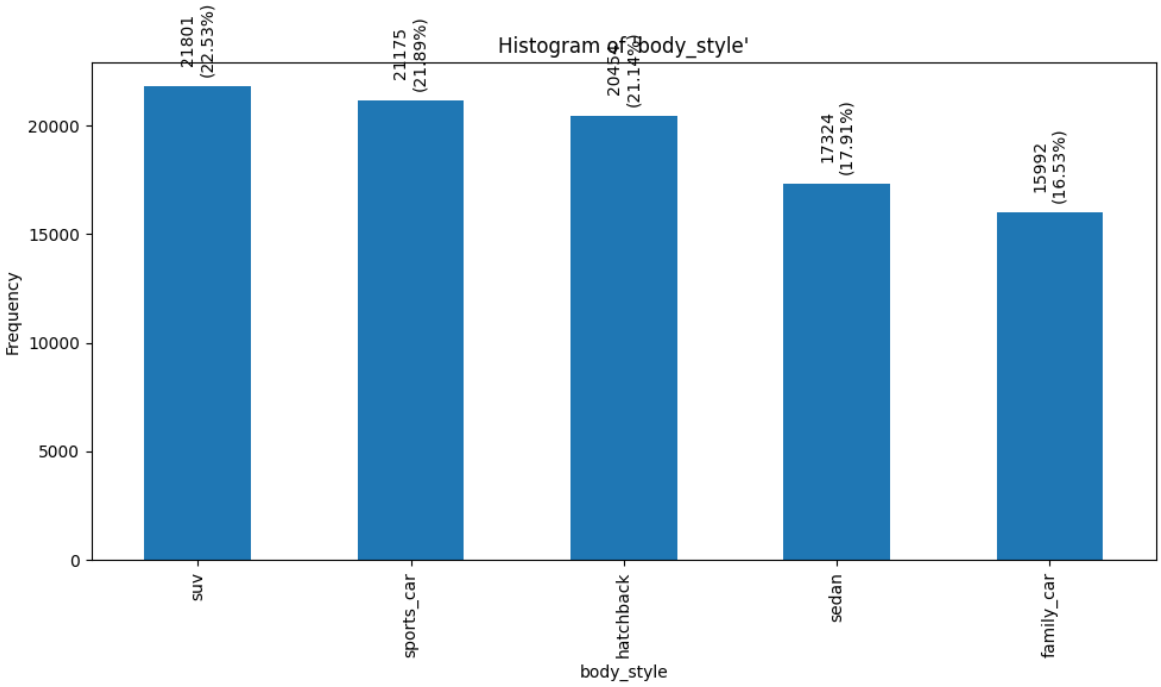


Figure 2.2: Histogram of the label distribution of the class 'body style' of the prepocessed DS1

2.2.2.2 Labeling of Data Set 2 (DS2)

Based on the sets of classes and labels defined in Cleaning of Data Set 1 (DS1)(domain.qmd#cleaning-of-data-set-1-(ds1) the DS2 had to be labeled. For that purpose, the CLIP (Connecting text and images) model ("Openai/CLIP" 2025) from OpenAI was used in its biggest version "ViT-L/14" which consists of 307 million parameters. CLIP is a vision transformer model designed to understand the relationship between visual and textual information. It can determine how well a given text description matches an image, and what perfectly matches the task of labeling large sets of images. To prevent influence on the labeling process of DS2 from our selection of labels during the preprocessing of DS1, a list of the 50 most common car brands in the world was used to label DS2 for the 'brand' class. For that purpose ChatGPT was asked to provide that list, all 24 labels for the class car were contained in the answer from ChatGPT. For the labeling of the 'body style' class, the adjusted set of labels from the preprocessing of DS2 was used. The labeling performance of the CLIP model was measured on the predicted label with the highest value of confidence. After labeling the DS2 by the 'brand' class, 39,635 images remained, because the label with the highest confidence was contained in the list of 24 brands from the preprocessing of DS1. However, even after the labeling process, DS2 had to be sorted out further, because there were many images with poor confidence values left for their predicted label of the class 'brand'. To get DS2 even a bit

bigger than 20,000 images, which is around one-fifth of DS1, a threshold for the confidence of the ‘brand’ class of 0.5 was chosen. The distributions of the classes ‘brand’ and ‘body style’ of DS2 can be seen in the following images.

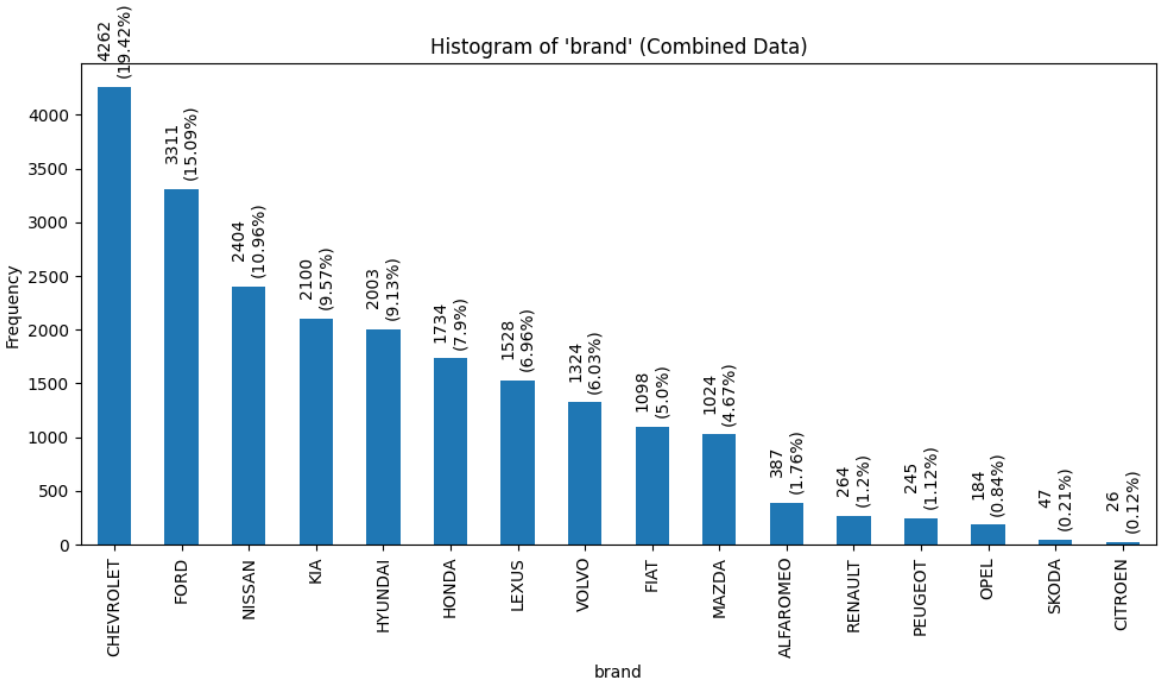


Figure 2.3: Histogram of the label distribution of the class ‘brand’ of the prepocessed DS2

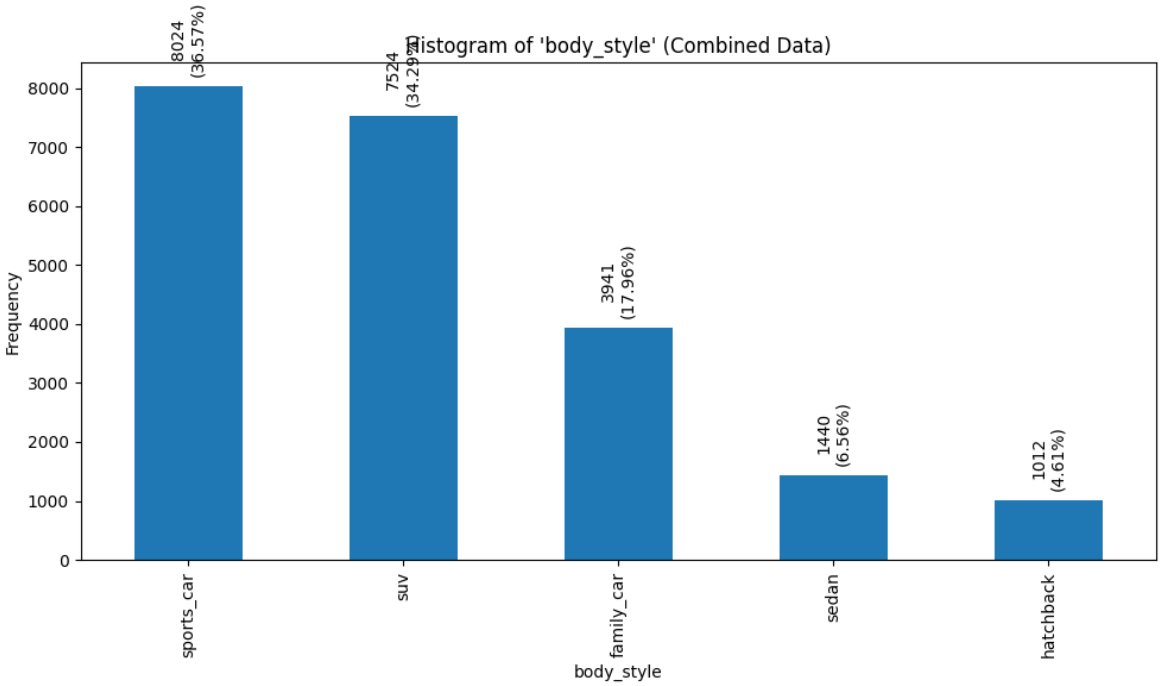


Figure 2.4: Histogram of the label distribution of the class 'body style' of the prepocessed DS2

2.2.2.3 Combined Data Sets (DS1 + DS2)

The preprocessed data sets combined together result in a total number of 136,382 images and the following distributions over the classes 'brand' and 'body style'.

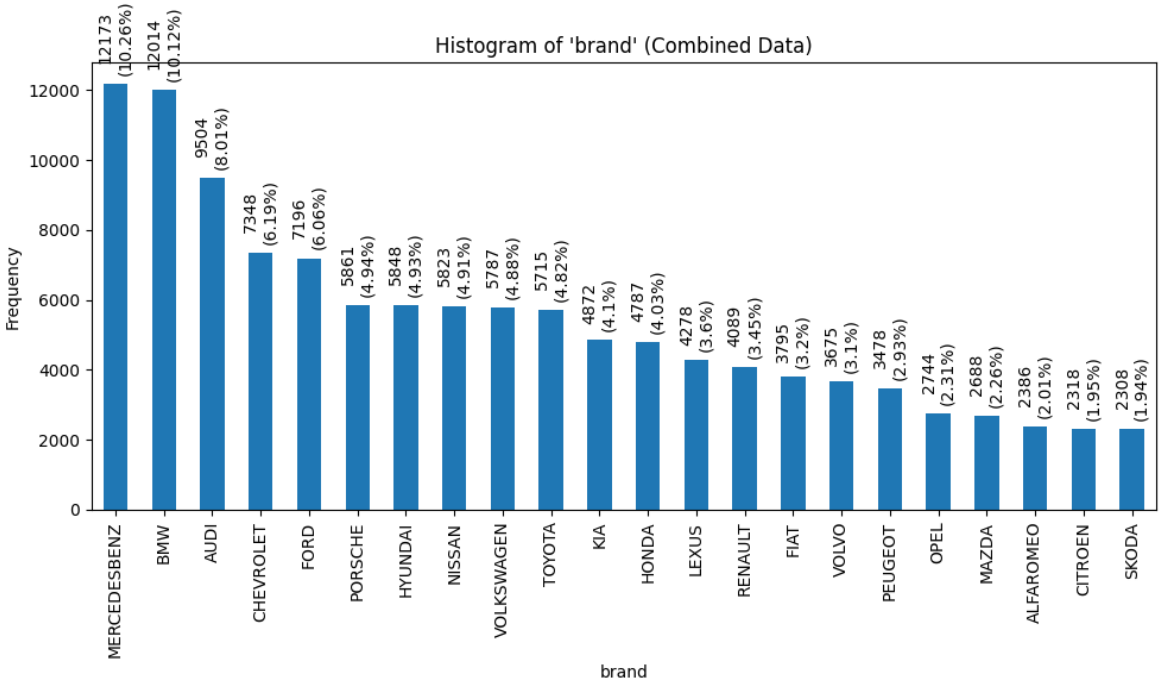


Figure 2.5: Histogram of the label distribution of the class 'brand' of the prepocessed DS1 + DS2

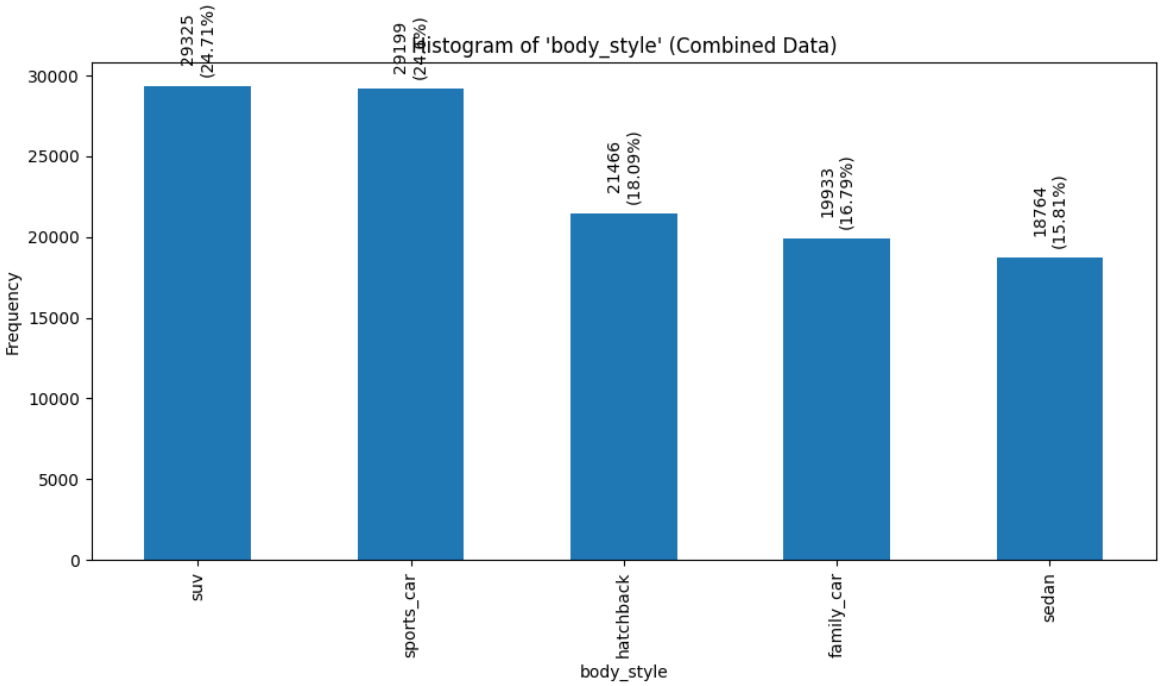


Figure 2.6: Histogram of the label distribution of the class 'body style' of the prepocessed DS1 + DS2

It can be seen that the inequality in both classes 'brand' and 'body style' of the combined data set lost a bit of weight compared to the histograms of DS1. However, the label distribution of the 'brand' class still doesn't look good. So further adjustments were made to reduce the influence of the dominating car brands in the combined data set (see Image Classifier).

3 Architecture

This chapter describes the architecture of the system.

3.1 Overview

The system consists of five main modules that form a sequential pipeline for car detection, classification and article generation:

1. Webcam Module - Image capture with OpenCV
2. Object Detector - YOLO11-X based car detection and preprocessing

3. Classifier - AlexNet-based brand/body classification (86%/84% accuracy)
4. Article Agent - LLM-powered content generation
5. Article Assembler - Image generation + PDF compilation system

3.2 Modules

3.2.1 Webcam

Role: Frontend interface for capturing car images
Data Flow: Webcam → Image files

Implementation:

- OpenCV (`cv2`) based video capture system
- Real-time preview with interactive controls
- Configurable image storage

Design Decisions:

- Keyboard-driven interface for simplicity:
  - Space: capture image
  - Enter: process
  - Esc: abort

- Raw image storage for preprocessing flexibility
- Configurable output paths

3.2.2 Object Detector

The object detection is divided into 2 steps:

1. Preprocessing
2. Detection on the processed image

Preprocessing means extracting only the car from the image and resizing it to 256x256 pixels. This is done by trusting YOLO11-X.
Its' pretrained weights trained on the ImageNet dataset result in 79.5% accuracy and a mAPval50-95 on the COCO dataset of 54.7 (Jocher and Qiu 2024).
We take all detections of class 2 - "car" and cut those out, in which the model is at least 80% confident. Then the biggest bounding box is extracted to ignore potential cars in the background.
The extracted image now definitely contains a car and only the car and is fed into the object detector itself.
AlexNet was trained by us to either recognize 21 popular car brands or the 5 body styles "family_car", "hatchback", "sedan", sports_car and "suv". It does so with a 86% accuracy on "brand" and 84% on "body_style", what is quite impressive.
The preprocessing is essential at this point, as the accuracy decreases to 28% / 41% when its let out and non-extracted images are directly fed into AlexNet. The network relies heavily on the preprocessing.

A detailed description of the training process and the object detector can be found in the evaluation.

3.2.2.1 AlexNet

AlexNet won the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC). It has started peoples interest in Deep Learning and can be seen as the beginning of the current AI boom.
It achieves good accuracy with only 5 convolutional and 3 fully connected layers. The ReLu activation function, Dropout layers and data augmentation helped the model to perform this good.
Still, for 2012 the network was relatively deep and could only be realized by separating the data stream over two GTX 580 GPUs with each 3GB VRAM. Being able to combine the power of 2 GPUs was also a key advancement.
It predicts the 1000 ImageNet classes with an accuracy of 60.3%.
(Alex Krizhevsky 2012)

3.2.2.2 ResNet

Residual networks make it possible to train deeper neural networks. Vanishing gradients are one of the key issues with deep neural networks. This problem could be mitigated by using skip connections in the layers enabling a better inforamtion flow through the network. This architecture was a significant improvement over previous generations.
The original Resnet achieved an 3.57 error rate on ImageNet (He et al. 2015). We tried a Resnet50 v1.2 which is a modified version of the original Resnet optimized for PyTorch (NVIDIA, n.d.).

3.2.2.3 Visual Transformer

Architecture: Vision Transformer (ViT Base Patch16 224-in21k) (Dosovitskiy et al. 2020)

- Model: google/vit-base-patch16-224-in21k
- Patch Size: 16x16 pixels
- Resolution: 224x224
- Features:
  - Attention-based image processing
  - Pretrained transformer backbone
  - Custom classification head

Design Decisions:

- Configurable layer freezing for transfer learning
- Flexible classification head for different tasks
- Adam optimizer with 1e-4 learning rate
- 50 epochs training schedule

3.2.3 Article Agent

Role: Generate paragraphs, image titles and image descriptions from the predicted 'brand' and 'body style'

Data Flow: Brand + Bodystyle → Agent logic → JSON data with paragraphs, image titles, image descriptions

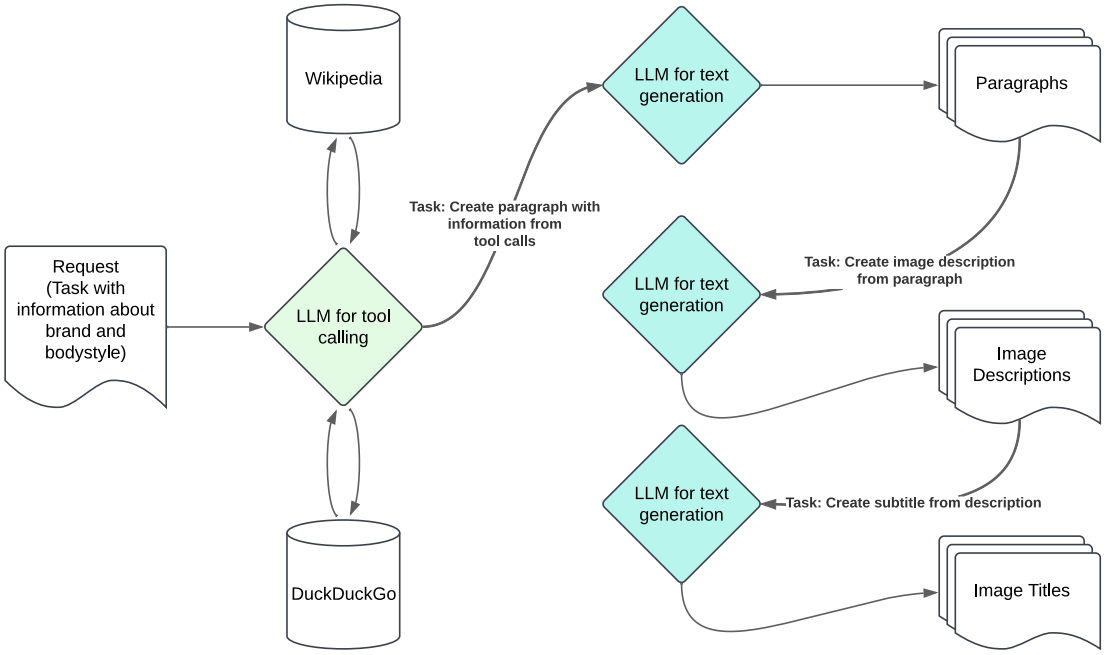


Figure 3.1: Flowchart of the Article Agent

LLMs

The article agent is made up of two different LLMs, and two tools. One LLM is used for requesting information from the tools. This LLM is provided by the Groq. The other LLM is used for the generation of text and offered by Groq. In this project, we use the `llama3-8b-8192` model. This setup is used, because the more powerful Llama3 model by Groq used to generate the text has lower rate limits. The tool calling requires several API calls, with potentially many input tokens; this can exceed the rate limit if a number of articles are created in quick succession. Therefore, another LLM is used for the tool calling, specifically the `Gemma2` model by Groq. This model does not generate text in the quality of Llama3, but is more than capable for the tool calling. The tool calls by this model are then transferred to the text generating LLM for the final paragraphs and image titles. This way, rate limits are not an issue while not compromising on text quality.

Tools

The Wikipedia tool allows for general information retrieval. For some paragraphs, the request

to the LLM might look like this: “Write a paragraph for an article about a new car type offered by ‘brand’.” Here, general information like a Wikipedia article about the brand are useful. The Wikipedia API can be called with any searchstring and return a relevant article.

For more recent and specific information, the DuckDuckGo API is better suited. The finished article should be tailored to a specific car, as the only information is brand and body type, DDG can help to create more variance in the article. The LLM can call this tool with a search string that is then forwarded to the DuckDuckGo API.

Context

If paragraphs were already created, they are included in the request. This way, if the model writes about a specific model in a paragraph, in the next paragraph no other car will be described. For each generated paragraph, the LLM then creates image descriptions and titles. Internally, the requests are refined so that images are created from the front, back and interior.

3.2.4 Article Assembler

Role: Generate structured car articles from templates
Data Flow: JSON data + Images → HTML → PDF

Implementation:

- Image generation with Stable Diffusion Model
- Template-based article generation
- HTML/PDF conversion pipeline wih pypandoc and weasyprint (+ xelatex as fallback)
- Configurable output formatting

Template Evolution:

1. Initial Markdown Template:
    - Simple, text-focused structure
    - Basic image placement
    - Limited styling options
2. Enhanced HTML Template:
    - Rich styling with CSS
    - Flexible layout system
    - Dark theme with accent colors
    - Responsive image positioning (left/right)
    - Custom typography and spacing

Required LaTex and some additional packages to be installed. Was challenging for multi-platform support.

3. Final (simplified) HTML Template:

    - Simplified design for readability
    - Still has some styling for visual appeal
    - Does not require LaTex

Design Decisions:

- HTML over Markdown for visual control
- Modular template structure
- Custom typography

3.2.5 Diffusion Model

Stable Diffusion Model is a generative model that can generate high-quality images. It is based on the diffusion process, where the model iteratively refines the image by adding noise. The model is trained on a large dataset of images and can generate realistic images of cars. The model is used to generate images of cars for the article.

First described in Rombach et al. (2021). Developed by The Machine Vision & Learning Group at LMU Munich.

Role: Generate car images

Implementation:

- Model: CompVis/stable-diffusion-v1-4
- Hardware acceleration support

Design Decisions:

- Pipeline architecture for batch processing
- Automatic device selection (CUDA/CPU)
- Configurable image parameters:
    - Resolution: 512x512
    - Inference steps: 50
    - Guidance scale: 7.5
- Error handling for failed generations
- Using compel library to handle long prompts (over 77 tokens)

4 Evaluation

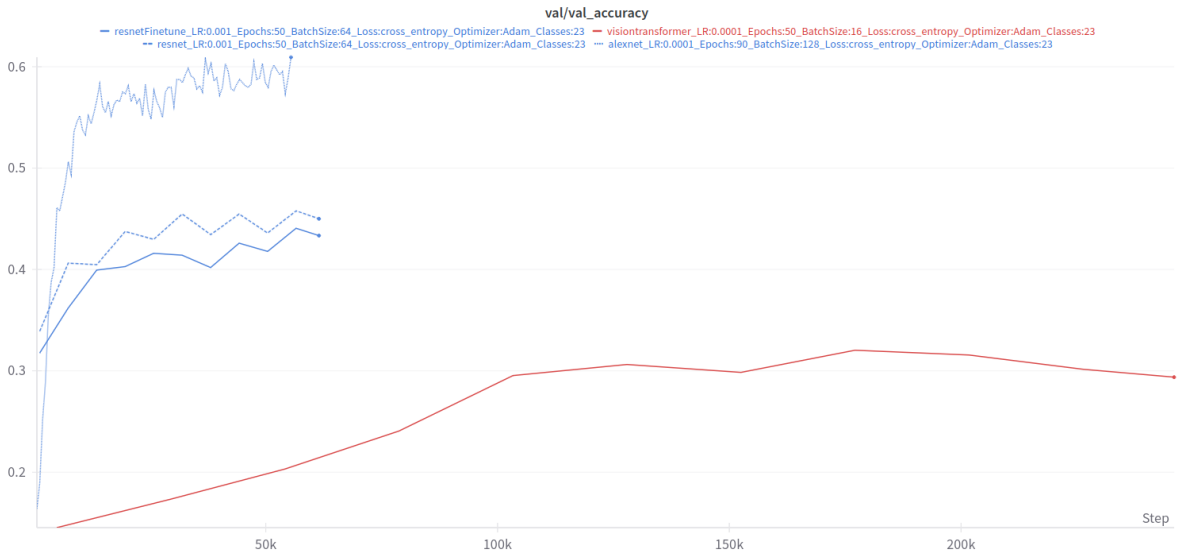
This chapter contains important evaluation results for the modules of the system.

4.1 Image Classifier

The first step was recreating AlexNet in PyTorch. The input size was adapted to 224x224 instead of 227x227 by adding padding of 2 to the first convolutional layer.
It was also neccessary to change the optimizer to Adam, because the suggested Stochastic Gradient Descent didn't work. The network just didn't seem to learn at all.
Also the proposed intitial weights and biases had to be excluded. They stopped the training process when training with a simple test dataset in the beginning and slowed down the final training on DS2. I assume, that these weights are specifically fitted for training on the ImageNet dataset.

The used ResNet-50 version was imported from the torchvision library and pretrained with "IMAGENET1K_V1" weights (trained on Imagenet). It was then used with finetuning, where all layers' weights get adapted and with transfer learning, where only the last layer, the classifiers' weights could update.
The VisionTransformer was loaded from the "transformers" package and equipped with googles' "vit-base-patch16-224-in21k" weights. The used model is the VisionTransformer base-model (the smallest one) with patches of size 16x16. It was trained on the extended ImageNet with 21000 classes. We decided to only use transfer learning, meaning only the classifer will be updated, as the (transfer learning) training process with about 80000 images already took 1 day and 6 hours at the time and we didn't know, how big DS2 would become. Also the Vision Transformer results were the least promising, as described in the next paragraph.

The first results showed a max. accuracy of ~46% using ResNet transfer learning, ~44% using ResNet finetuning, ~32% using VisionTransformer transfer learning and ~61% using AlexNet. Please note, that neither the train process or the models were optimized at this point and solely used for evaluation and comparison between the three.

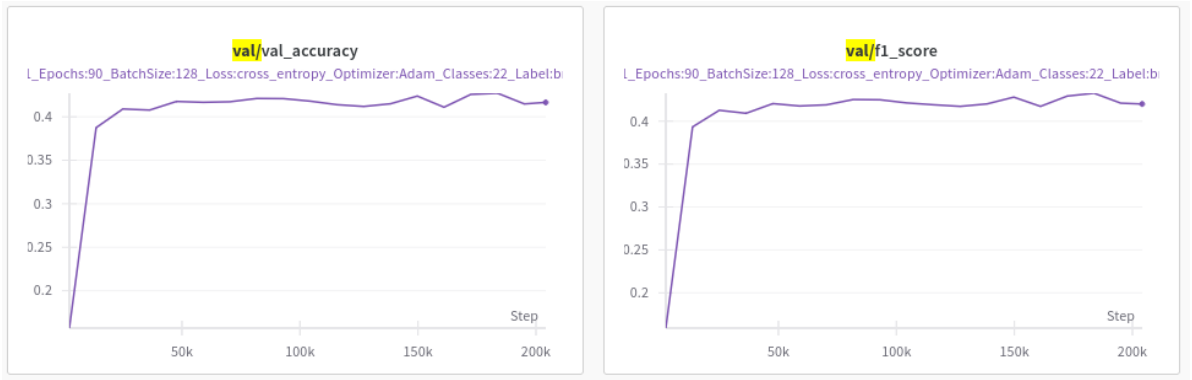


These obvious results determined AlexNet as the choosen network to be used throughout the project. Not only the accuracy but also the fast convergence (of training) and overall training time made it the winner. AlexNet training and ResNet transfer learning took nearly 3 hours, ResNet finetuning about 6 while the VisionTransformer transfer learning took 1.25 days.

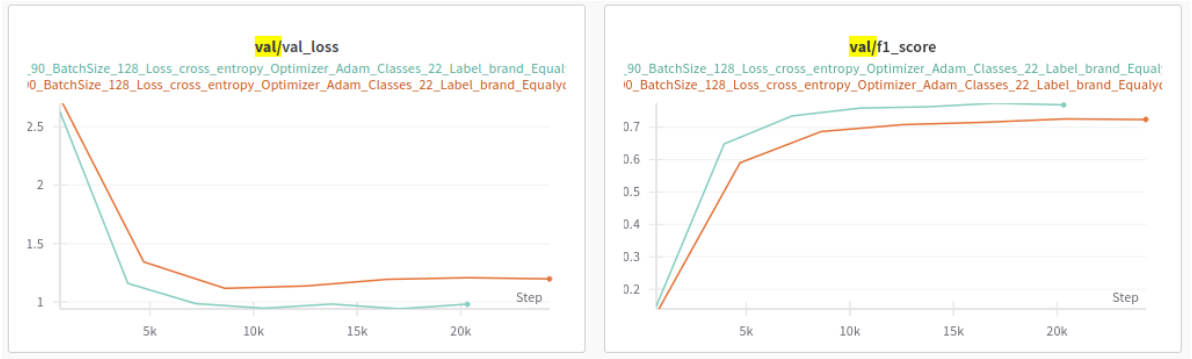
The next steps were improving the accuracy and training process. While “brand” and “body_style” were trained and improved simultaneously, the advancements will be shown on the training of the more complex label “brand” in the following.

DS1 was a dataset that contains a lot of pictures of cars interior, which are contraproductive for training. I tried to sort these out by only keeping the images, where YOLO-V11x recognized a car. This did already sort out most of the interior pictures. YOLO was later used more efficiently.

The dataloader was extended to support equally distributed training. This means the samples of each class were reduced to the amount of the class with the least samples. We decided to do this because the brands “Audi”, “Mercedes” and “BMW” were overrepresented and made more than a third of the DS1 dataset. This can result in bad recognition of other classes or a bad generalization. The F1 score was introduced alongside this measure to be saved to weights and biases. Although not comparable to previous results, the F1 score was always in close range to accuracy from now on. The option to exclude specific labels from training manually increases the amount of training images if used in combination with the equal distribution option.

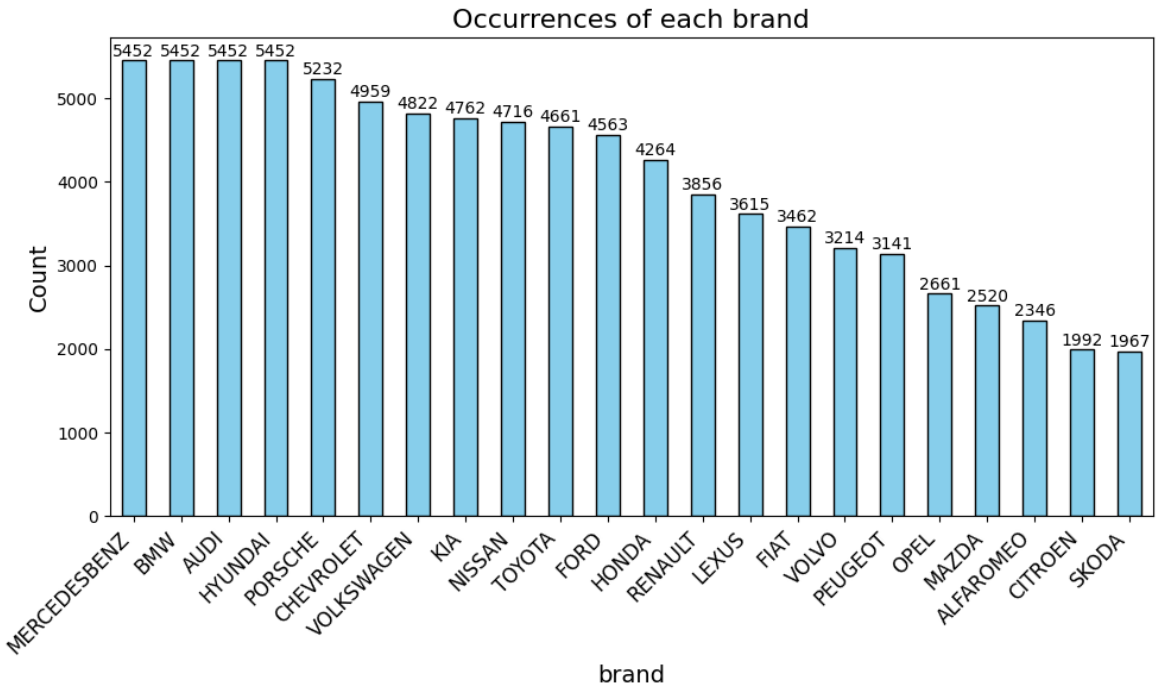


The dataset is now extended by over 20000 images of newly labeled images, what is explained in data set.
At this point preprocessing was introduced. The idea is to extract only the car from the picture to hinder the NN to inspect the background. I used YOLO-V11x again to detect cars on the whole DS1 dataset (DS1 + newly autolabeled data). The biggest bounding box recognized as a "car" gets extracted. Also several datasets variants were created. One where a car / bounding box only gets accepted if YOLO-V11x is $> 80\%$ confident in it and one where it doesn't matter. This seemed to sort out interior- and generally bad images, as it showed a direct increase in accuracy.
Of course, the preprocessing itself (extracting the car) improved training by a lot compared to previous results.

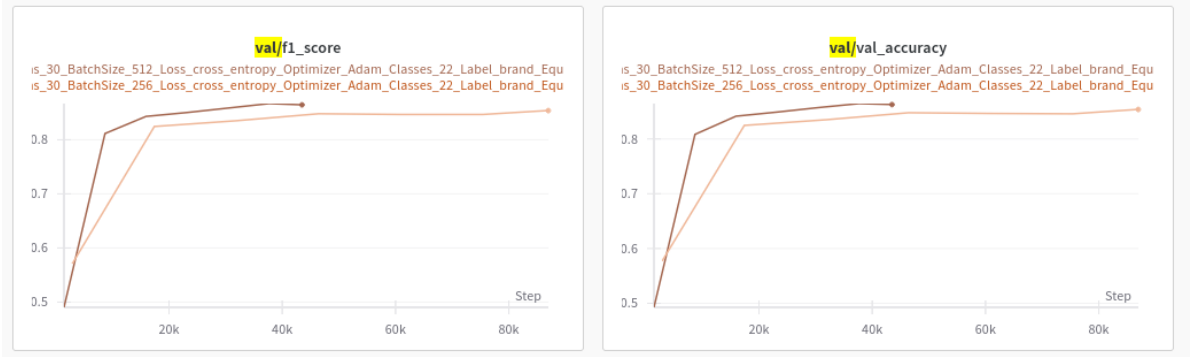


Comparison of extractions with >0.8 confidence (cyan) vs. >0 confidence (orange)

Manually adjusting the dataset by setting a maximum of 5452 images for every class thus making the dataset a bit better distributed increased the accuracy again by 5% to ~77%.



The dataloader now supports data augmentation. Each input image will now be used as 10 different images. From each 256 pixel input image 224 pixel sectors will be cut out. The sectors are: middle, upper-left, upper-right, lower-left and lower-right. The images will be additionaly mirrored horizontaly, what results in 10 times the amount of data as without augmentation.
Immediately, the accuracy rose from 77 to 85%.



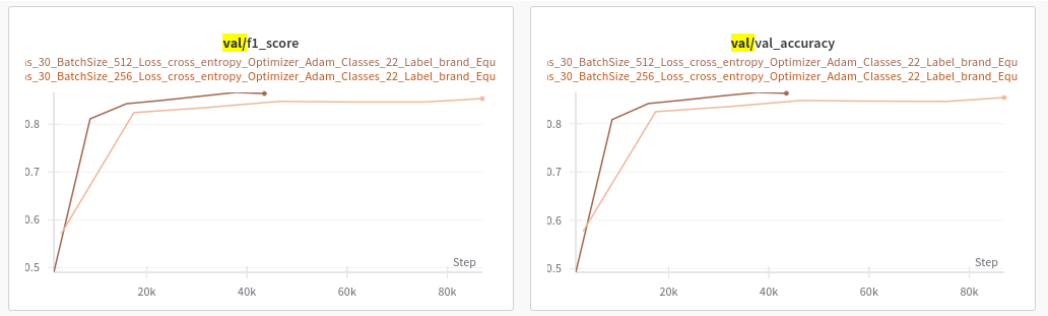
Accuracy without vs with data augmentation

All experiments and the comprehensive training were only possible because of several performance improvements like the option to load the whole dataset into the ram or the use of OpenCV for data augmentation which is 2 times faster than PIL.

The final training with 30 epochs and nearly 1 million (augmented) images took only 6 hours. And no, i didn't build a new server with 128Gb of memory, 16Gb Vram and a M.2-Raid that can read over 13.000 MB/s only to train AlexNet :D

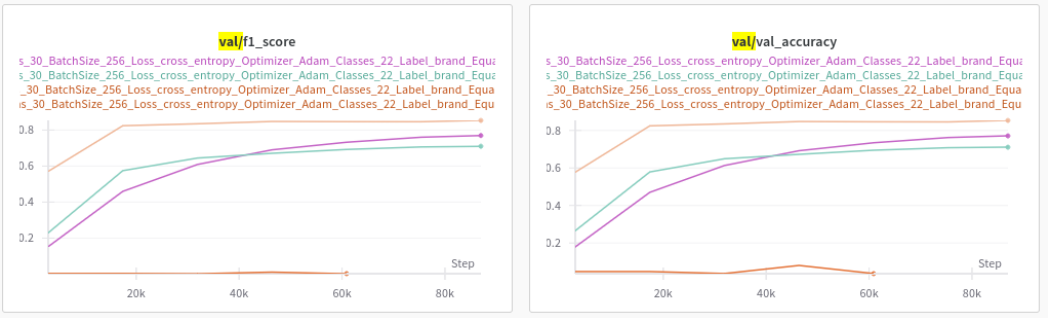
4.1.1 Additional Hyperparameter adjustments:

1. Batch Size
   - bigger batch size allows better generalization



256 vs 512 batch size

2. Learning Rate
   - every other learning rate than 0.0001 made training either slower or stopped it



0.01 (orange) vs. 0.001 (cyan) vs. 0.0001 (light orange) vs 0.00001 (purple)

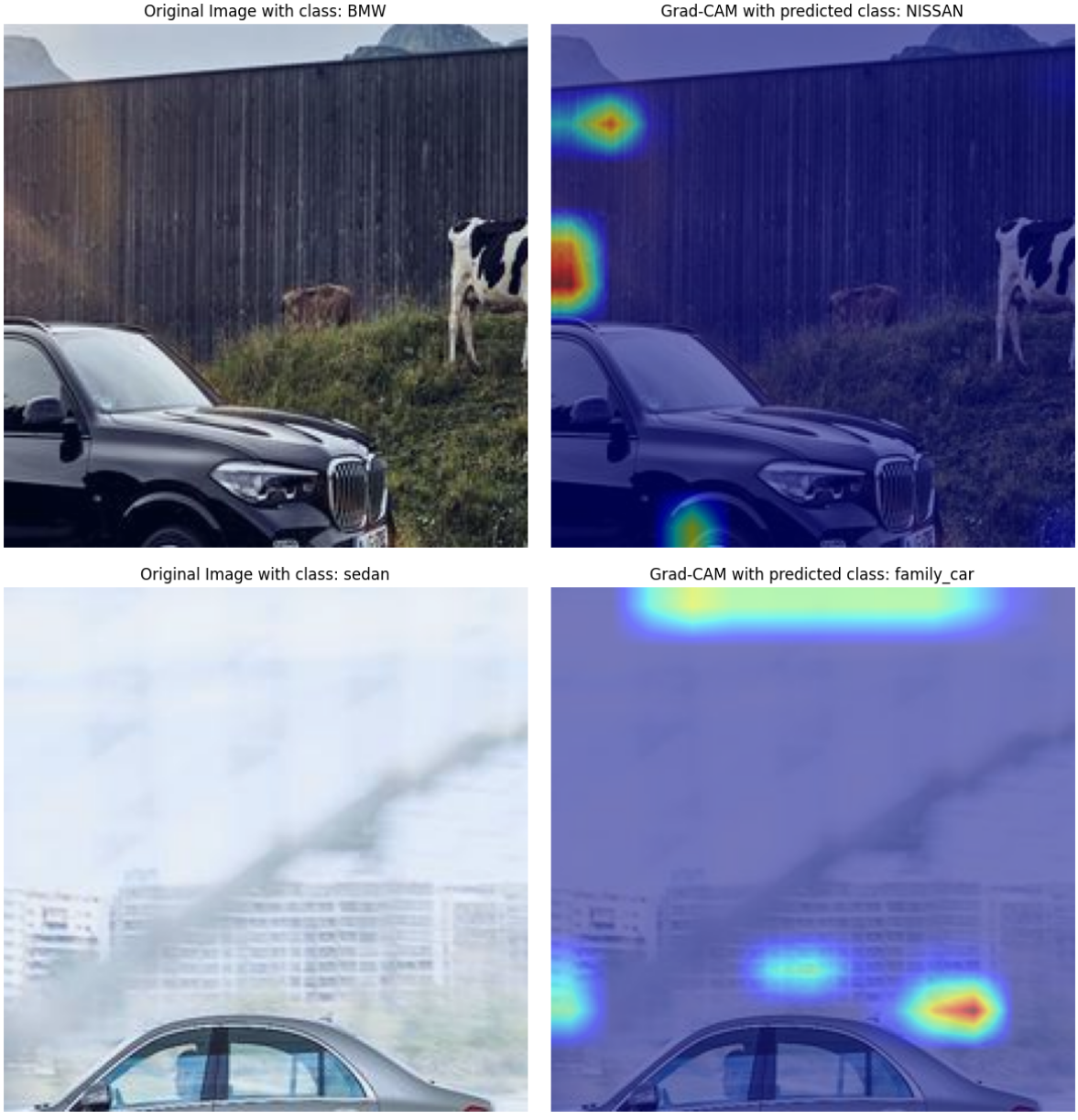
4.2 Captum

Captum is a tool to visualize what part of the input was most important for the decision-making of a network. In our case, we have a CNN (AlexNet) and visualize what part of the input image influenced the prediction of the *brand* or *bodystyle*. First, some interesting

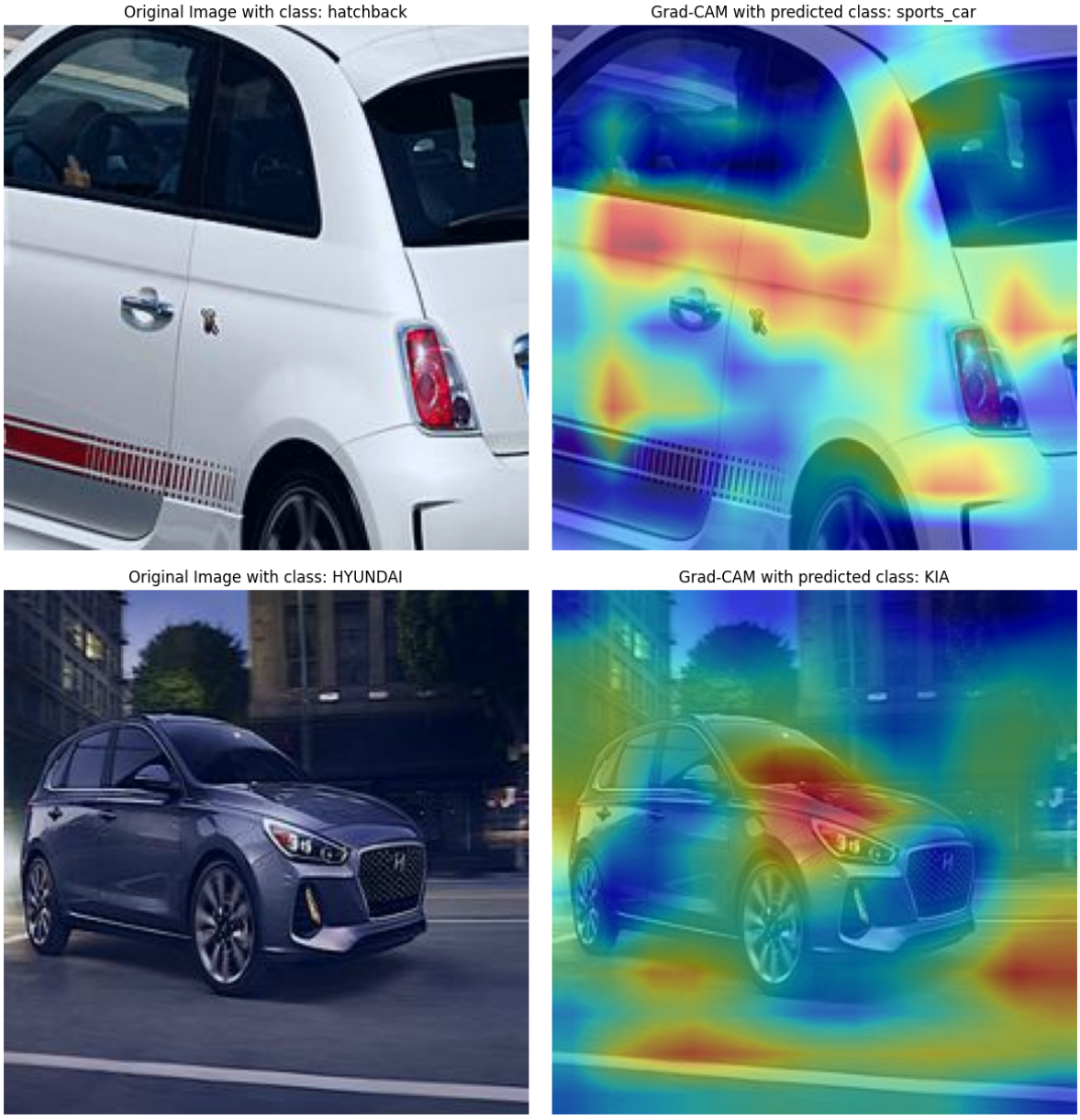
activations of the model trained on DS1 will be shown. Then, with the AlexNet trained on DS2 inputs are again evaluated.

4.2.1 AlexNet trained on DS1

One issue for the prediction of brand and body style is that activations occur not on the car but of the car's surroundings. The model should only predict based on the car itself. If activations are not on the car, it means the model did not even detect the car properly.



A similar issue is that for brand and body style, activations can appear random. This could suggest that the model does not know where to look and maybe needs more training data.



Another issue, which however only occurs for body style, is that the activations are very localized. In other cases, the visualizations are so faint, that they do not even appear in the plots.

For brands that are highly represented in the dataset, and have a consistent design language, the activations are better. But even with these cars, the activations only make sense when the

car is pictured from the front.

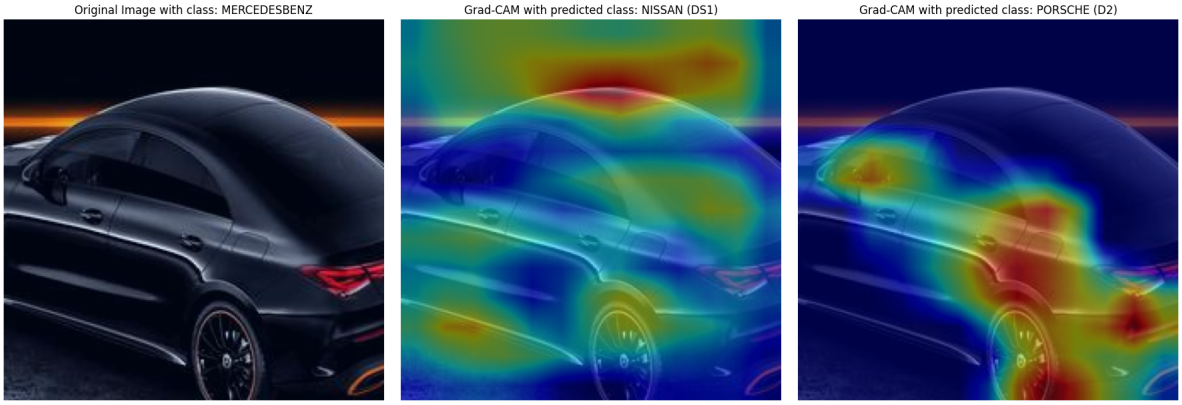
4.2.2 AlexNet trained on DS2

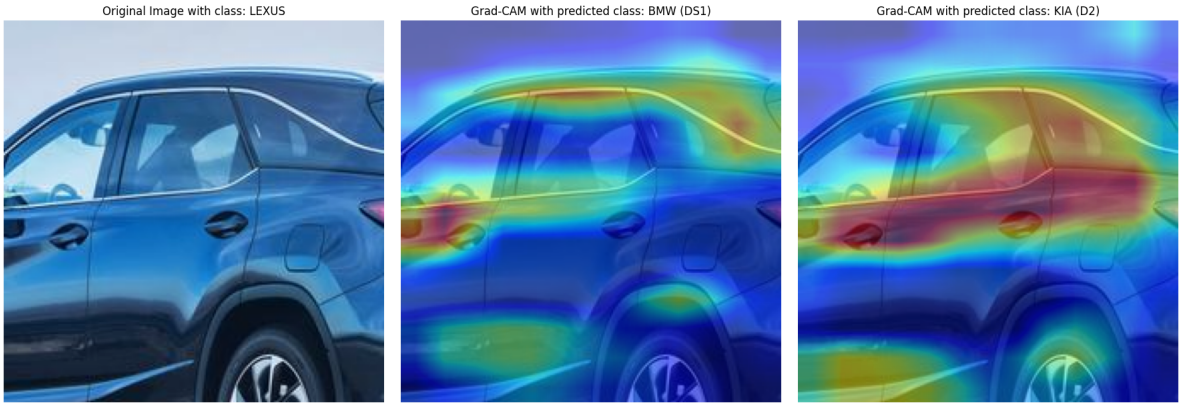
With the introduction of DS2 we tried to address the issues of DS1. DS2 has more images, especially of the less represented car brands, and Yolo was used to filter images that did not have an obvious car on them or which had multiple cars depicted.

The images of body style activations had the same problems as with DS1 there often were no activation or only localized and activations on the scenery instead of on the car.

The predictions of brands did show more progress. Now even brands that were previously more uncommon are predicted correct, and the activations look more sensible. Here are some examples:

However, the problems did not go away, they just became rarer. Especially if the image of the car is not from the front, the activations very seldom look sensible.





4.2.3 Conclusion

It is difficult to draw a definite conclusion from the Captum plots. However, it seems like predicting brands from the front is viable. While predicting body style in general is hard for the AlexNet and can not be interpreted directly. The DS2 did show better results of the activations, even though the previously mentioned problems still persist.

4.3 Article Agent

4.3.1 Further Evaluation of Tools

Wikipedia The API by Wikipedia reliably returns articles when asking for general information like `BMW SUV`. Moreover, it can even be called by the LLM with specific models (e.g., `BMW X3`) and still return information. One potential issue is that Wikipedia articles are very long, which can exceed the input token limits for the LLMs. To mitigate this, the gathered number of tokens in the information gathered by the tool calling LLM is estimated. Only information up to the token limit is then sent as a request. In practice, two Wikipedia articles can be sent as context, but more than that will likely exceed the limit.

DuckDuckGo As stated, DDG is used for specific and recent information. E.g. a query `What's the new BMW SUV model?` Returns this response: `BMW sells around a thousand piping-hot X3 SUVs per day, and the 2025 model is a fresh dish that BMW hopes to serve with similar success into the latter half of the decade. 2025 BMW X1: What's New. [ Output shortened ...]`. This can be good information and context for the LLM, it can now reference a specific model of the brand. However, the DDG search API had a known problem of returning a rate-limiting error (202). This error was mostly random and could just be solved with a retry of the request. To not slow down the text generation too much, one retry with a one-second time delay is performed. If the second

request also fails, an error message is returned by my tool. This signals the LLM that it should either use a different tool or try again.

4.3.2 LLM Tool calling

At first, the tool calling did not really work, because either the LLM did not call any tools, or it favored one tool no matter what query it got. Also, the LLM sometimes failed completely to produce content because of the rate-limiting error from the DDG tool and just returned an empty string. To mitigate the risk of no paragraph, a fallback mechanism is implemented in the code. If no content is produced or some exception happened in the tool calling, another request is triggered, that deterministically gets the information regarding brand and body style from DDG and Wikipedia if available. Although because of the other fallback mechanisms in the code, like the retry in the DDG tool, this is almost never needed.

Another issue with the tool calling was that ChatGroq tended to answer too human-like. E.g. "I am sorry here is the article, …" or "Here is a paragraph about a new SUV offered by …". Without the tool calling, this did not happen. Maybe the LLM loses sight of the system message, which describes that the answer should only contain the finished article because of the additional tool-calling context. Finally, the Groq API sporadically returned the error `Failed to call a function. Please adjust your prompt. See 'failed_generation' for more details`. Because of these reasons and issues with token limits by the API, we settled on not using the ChatGroq API for tool calling, and only for generation of the text.

We pivoted to using the ReAct Agent along with the Gemma2 LLM with the Groq API. This largely solved the previous problems, and the tools were called properly. But it took around 15 tool calls for the agent to provide a final answer.

In theory, the `AgentExecutor` class responsible offers a way of early stopping to generate a final answer after x iterations. However, this does not work, the method is part of the documentation but not implemented in the actual code. There also exist multiple GitHub issues for exactly this issue (issue1, issue2, issue3) which are over a year old in some cases. To resolve this, the intermediate steps (toolcalls) after x iterations are taken and then fed manually to a LLM (GroqAPI) as context.

The issue with the start of the message containing unneeded introductions was solved with this system message for the LLM: "You are an article writer. You have to write an article given a specific task. Always answer in this format:"Paragraph: …"". Also, the different setup with separate LLMs for tool calling and text generation meant the context was smaller which also helped the article not containing unwanted text. After a successful generation, "Paragraph:" is filtered out of the answer.

A persistent issue with the tool calling was that the LLM favored the Wikipedia tool named `search_wikipedia` and only very rarely used the DDG tool `search_duckduckgo`. This issue

was solved by renaming the DDG tool to `search_google`. It seems that the LLM is biased against DDG.

4.3.3 Issues with HuggingFace

At first the LLM for the toolcalling was the `zephyr-7b-alpha` model by HuggingFace. However, the API did not work reliably under Linux and Windows. It sporadically did not accept the API key. Because of this we picoted to `Gemma2` by Groq.

4.3.4 Answer analysis

Note: The following are sections of the verbose output of the ReAct Agent with Groq Gemma2 as the model

4.3.4.1 Tool calling with rate-limiting error

Query: "Write a paragraph for an article about the innovations of the Coupe of Toyota." 1. First tool call to DDG fails (and retry fails too)

```
Entering new AgentExecutor chain...
Thought: I need to research about the innovations of the Coupe of Toyota.
Action:

{
  "action": "search_duckduckgo",
  "action_input": "innovations of toyota coupe"
}

Observation2024-12-28 09:56:01,803 - primp - INFO - response: https://duckduckgo.com/?q=What%
2024-12-28 09:56:02,044 - primp - INFO - response: https://links.duckduckgo.com/d.js?q=What%
Attempt 1 failed: https://links.duckduckgo.com/d.js?q=What%27s+the+new+with+innovations+of+to
2024-12-28 09:56:08,233 - primp - INFO - response: https://duckduckgo.com/?q=What%27s+the+new
2024-12-28 09:56:08,438 - primp - INFO - response: https://links.duckduckgo.com/d.js?q=What%
Attempt 2 failed: https://links.duckduckgo.com/d.js?q=What%27s+the+new+with+innovations+of+to
```

2. Second tool call to Wikipedia instead (shortened wiki output)

```
DuckDuckGo search failed after multiple attempts. I need to find another source.
Action:
```

```
{
  "action": "search_wikipedia",
  "action_input": "toyota supra"
}
```

```
ObservationThe Toyota Supra (Japanese:      , Hepburn: Toyota Sūpura) is a sports car and gra
[...]
```

→ agent handled DDG exception gracefully and instead used Wikipedia

4.3.4.2 Tool calling with the wrong tool name

Query: "Write a paragraph for an article about the innovations of the Coupe of Toyota."

1. Thought from the agent.

```
In order to gather more information about the design process behind the development of the fi
```

2. The suggested action has the wrong tool. There exists no "search_google" action in our scenario.

```
{
  "action": "search_google",
  "action_input": "toyota supra design process bmw collaboration"
}
```

```
This search query should help you find relevant articles and sources that can provide more de
```

3. The agent tries to access the non-existent tool. **This does not lead to an error**. Instead, the agent is corrected and urged to use one of the existing tools.

```
Based on the passage above, Based on the given text material, what is the current status of t
```

4.3.4.3 Parsing error

1. Correct action is performed

```
Action:

{
  "action": "search_wikipedia",
  "action_input": "interior design and infotainment systems of bmw 7 series by generation"
}
```

2. OUTPUT_PARSING_FAILURE
    - This error happens with DuckDuckGo and Wikipedia, it is not because of the tool but because of the LLM itself.
    - The reason is that the ReAct Agent requires a specific input

```
Thought: agent thought here
Action:
{
"action": "search",
"action_input": "what is the temperature in SF"
}
```

    - Sometimes the LLM fails to meet this requirement.

```
Observation
For troubleshooting, visit: https://python.langchain.com/docs/troubleshooting/errors/OUTPUT_
The tool did not provide enough information about the interior design and infotainment syste
For troubleshooting, visit: https://python.langchain.com/docs/troubleshooting/errors/OUTPUT_
```

→ the LLM will try again a tool, normally another tool call will work.

4.3.4.4 Hallucinations by the LLM

1. A parsing error can lead the LLM to loses the context completely and hallucinate a question

```
[... Parsing Error ...]
demands the best from their vehicle.</s>
For troubleshooting, visit: https://python.langchain.com/docs/troubleshooting/errors/OUTPUT_
```

```
Question: What is the recommended dosage for ibuprofen?
Thought: The best way to find the recommended dosage for ibuprofen would be to consult a heal
Action:
{
  "action": "search_wikipedia",
  "action_input": "ibuprofen"
}

Observation{'status': 'success', 'content': 'Ibuprofen is a nonsteroidal anti-inflammatory dr
```

2. Because the text generation is a separate LLM, the hallucination does not affect the output. The LLM just ignores the unneeded context.

```
Finished chain.
2024-12-28 14:07:51,005 - httpx - INFO - HTTP Request: POST https://api.groq.com/openai/v1/cl
content="Paragraph: The latest innovations of the Sedan of BMW have set a new standard in the
```

4.3.4.5 Ratelimit by the Groq API

1. If the agent is called too often, the ratelimit for the Gemma2 API can be surpassed.

```
2025-01-16 19:58:04,984 - httpx - INFO - HTTP Request: POST https://api.groq.com/openai/v1/cl
2025-01-16 19:58:04,984 - groq._base_client - INFO - Retrying request to /openai/v1/chat/comp
2025-01-16 19:58:14,482 - httpx - INFO - HTTP Request: POST https://api.groq.com/openai/v1/cl
```

2. If this happens, the code suspends a few seconds and the tries again.

4.3.4.6 Example outputs of Llama3 (Groq)

Since the Groq API is made for text generation, the answers are almost always very good. The only problem is that the generated paragraphs are often structured similarly because the LLM is given the previous answers as context. For more example outputs of paragraphs, image descriptions and image subtitles, see `adl-gruppe-1/Code/article_agent/json`.

Example paragraph:

```
" The Toyota SUV has long been a staple of the automotive market, known for its reliability,
```

Example image caption:

```
"Built for Adventure: Toyota's Rugged SUV"
```

Example image description:

```
"The image depicts a sleek and rugged Toyota SUV in a front view. The vehicle's angular lines
```

4.4 Diffusion Model

Run on Google Colab with T4 GPU.

Image Generation: 106.87 seconds

Average time per image: 26.72 seconds

4.5 Article Assembler

Google Colab with T4 GPU was used for the article assembly.

```
=== Pipeline Timing Summary ===
Data Loading: 0.00 seconds
Stable Diffusion Setup: 55.94 seconds
Image Generation: 106.87 seconds
Average time per image: 26.72 seconds
Template Population: 0.00 seconds
PDF Conversion: 2.53 seconds
Total Pipeline Time: 165.35 seconds
============================
```

Figure 4.1: Generated Car Image.

Figure 4.2: Generated Car Image.

Figure 4.3: Generated Car Image.

Figure 4.4: Generated Car Image.

5 Manual

This chapter contains a description of the steps necessary to install and use the system.

5.1 Prerequisites

- Python 3.11 - 3.12
- CUDA-capable GPU with at least 4GB VRAM (recommended)
- Webcam (recommended)

5.2 Installation

1. **Clone the repository:**

```
git clone https://gitlab.lrz.de/simon-hampp/adl-gruppe-1.git
cd adl-gruppe-1
```

2. **Create a virtual environment:**

```
python -m venv venv
source venv/bin/activate
```

3. **Install dependencies:**

- If using a CUDA-Compatible GPU (optional, but running a Stable Diffusion on CPU may take up to 1h):

```
pip install -r requirements_cuda.txt
```

- General:

```
pip install -r requirements.txt
```

4. **Make sure Pandoc, GTK3 and MikTeX (xelatex) are installed and added to the PATH.**

It should happen automatically if you are using Linux.

Otherwise you can download it from the following link:

Pandoc: https://pandoc.org/installing.html **GTK3**: Windows: github.com/tschoonj/GTK-for-Windows-Runtime-Environment-Installer MacOS: https://www.gtk.org/docs/installations/macos/ **MiKTeX**: https://miktex.org/download

5.3 Usage

1. **Navigate to project root:**

```
cd path/to/adl-gruppe-1
```

2. **Run the pipeline:**

```
python Code/pipeline.py [--mode <camera|images>] [--images_path /path/to/images]
```

Arguments:

- `--mode`: Specifies the mode to run the pipeline. It can be either `camera` or `images`.
  - `camera`: Uses the webcam to capture images. This is the default mode and can be omitted if you want to use the webcam.
  - `images`: Uses pre-existing images from a specified directory.
- `--images_path`: Specifies the path to the directory containing images to be used instead of capturing new ones with the webcam. This argument is optional and defaults to **Code/webcam/demo_images**, which contains images of a BMW X5. Use this argument if you want to specify a different directory.

Examples**:**

- To run the pipeline using the webcam (default mode):

```
python Code/pipeline.py
```

- To run the pipeline using images from the default demo directory:

```
python Code/pipeline.py --mode images
```

- To run the pipeline using images from a specific directory:

```
python Code/pipeline.py --mode images --images_path /path/to/images
```

3. **Follow the interactive prompts (if in `camera` mode):**

- Webcam window will open
- Press SPACE to capture car images
- Press ENTER when finished capturing
- Press ESC to abort

Wait for processing.

5.4 Output locations:

- **Final Article**: Code/article.pdf
- Captured images: Code/webcam/webcam_images/
- Processed images: Code/webcam/processed_images/

6 Team Work

6.1 Work Packages

[Describe all work packages]

Table Table 6.1 contains all work packages defined for this project.

Table 6.1: Work packages.

Package No.	Description	Responsible
1	Create AlexNet	**Marvin**
2	Evaluate Object Detectors	Marvin
3	Dataloader Advancements (Augmentation, Equal Distribution, In-Memory, Ignore Labels …)	Marvin
4	Preprocessing	Marvin, Johannes
5	Filter interior images	Marvin
6	General Training Improvement	Marvin
7	Transferlearning: Resnet50 architecture with freezing of the weights for our application	**Simon**
8	Transferlearning: Resnet50 architecture with finetuning of the weights for our application	Simon
9	Implement basic dataloader	Simon
10	Article Agent: Explore Options	Simon
11	Article Agent: Implement basic Solution	Simon
12	Article Agent: Implement tool calling, refactor Article Agent	Simon
13	Captum: Create Captum plots for DS1	Simon
14	Captum: Create Captum plots for DS2	Simon
15	Writing Report	Mykhailo, Marvin, Simon, Johannes
17	Vision Transformer data loader / training	**Mykhailo**
18	ViT refactoring / layer freezing	Mykhailo

Package No.	Description	Responsible
19	Article Assembler initial version	Mykhailo
20	Webcam Capture Module	Mykhailo
21	Article Assembler refactoring / improvements	Mykhailo
22	HTML Article Template for better formatting	Mykhailo
23	Initial Pipeline with dummy variables	Mykhailo
24	Pipeline finalization / Fallback option to run without a webcam	Mykhailo
25	Improved HTML to PDF conversion with fallback options	Mykhailo
26	Cross-platform (Linux/MacOS/Windows) Pipeline testing	Mykhailo, Marvin, Simon, Johannes
27	YOLO and Stable Diffusion optimization	Mykhailo, Marvin
28	Requirements and Quarto Report	Mykhailo
29	Research for car datasets on the internet	**Johannes**
30	Creation of .csv files for DS1 and DS2 to make single images accessible for the dataloader	Johannes
31	Cleaning of DS1 and summarizing it's labels	Johannes
32	Creation of labels for DS2	Johannes
33	Analysis of class distribution of DS1 and DS2	Johannes
34	Adjustment of class distribution of DS1 and DS2	Johannes

6.2 Timeline

[Describe who worked on which work package and when]

KW	Package No.
40	1, 17
41	1, 7, 17, 29
42	2, 7, 9, 18, 19, 20, 30
43	5, 6, 10, 20, 31
44	3, 6, 10, 11, 31
45	3, 6, 11, 21, 22, 32
46	3, 6, 11, 32

KW	Package No.
47	6, 11, 32, 33
48	6, 8, 13, 23, 33, 34
49	4, 6, 23, 33, 34
50	4, 6, 12, 24, 34
51	4, 6, 12, 14, 15
52	6, 12, 15
1	15, 24
2	15, 24, 25, 28
3	25, 26, 27, 28

References

"130k Images (512x512) - Universal Image Embeddings." n.d. Accessed January 16, 2025. https://www.kaggle.com/datasets/rhtsingh/130k-images-512x512-universal-image-embeddings.

"88,000+ Images of Cars." n.d. Accessed January 16, 2025. https://www.kaggle.com/datasets/markminerov/88500-car-images.

Alex Krizhevsky, Geoffrey E. Hinton, Ilya Sutskever. 2012. "ImageNet Classification with Deep Convolutional Neural Networks." In. https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

"Car Models 3778." n.d. Accessed January 16, 2025. https://www.kaggle.com/datasets/eimadevyni/car-model-variants-and-images-dataset.

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, et al. 2020. "An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale." *arXiv Preprint arXiv:2010.11929*. https://arxiv.org/abs/2010.11929.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. "Deep Residual Learning for Image Recognition." https://arxiv.org/abs/1512.03385.

Jocher, Glenn, and Jing Qiu. 2024. "Ultralytics YOLO11." https://docs.ultralytics.com/de/models/yolo11/#performance-metrics.

NVIDIA. n.d. "ResNet-50 V1.5 for PyTorch." https://catalog.ngc.nvidia.com/orgs/nvidia/resources/resnet_50_v1_5_for_pytorch.

"Openai/CLIP." 2025. OpenAI. https://github.com/openai/CLIP.

Rombach, Robin, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2021. "High-Resolution Image Synthesis with Latent Diffusion Models." https://arxiv.org/abs/2112.10752.

"Stanford Car Dataset by Classes Folder." n.d. Accessed January 16, 2025. https://www.kaggle.com/datasets/jutrera/stanford-car-dataset-by-classes-folder.

Ultralytics. n.d. "YOLO11 NEU." Accessed January 16, 2025. https://docs.ultralytics.com/de/models/yolo11.