Distributed Malaria Detection

Project report
by
Fabian Schimpf, B.Sc.

Institute of Aerodynamics and Gas Dynamics
University of Stuttgart
Stuttgart, May 2019

1 Overview

Machine Learning has gained tremendous amounts of attention over last couple of years due to the success of Artificial Neural Networks (NNs). In this project goes through the main steps of developing a Machine Learning Pipeline and present improvements to the original code that result from knowledge gained or interest sparked in the lecture Effizientes Programmieren. The repository for this project can be found at `https://github.com/2fasc/Distributed_Malaria_Detection`.

Outline

The core function is contained in `blob_detection.py` and consists of a selected object detection algorithm and and a trained Neural Network for classification. The resulting labels are shown in figure1.1.
Another core component was solving the problem of privacy that would otherwise prohibit the real-

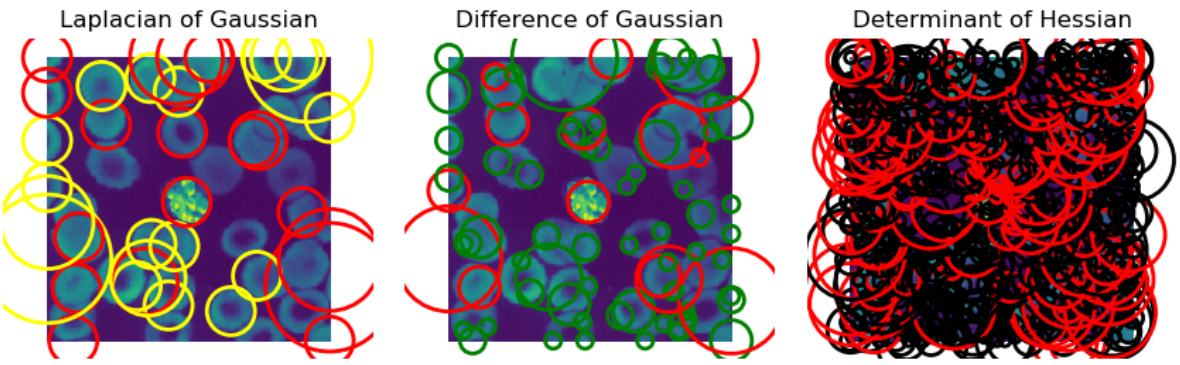

Abb. 1.1: Classification of cells - red circles indicate infected cells

world usage of these algorithms. Therefore, research was done into how training and inference can be done in a way that is allowed under the General Data Protection Regulation (GDPR).
Eventually, the system of choice was PySyft[10] by openmined[1] but it became clear that the code base is not production ready yet and still very restrictive in its usage. Therefore, it one of the key difficulties was to figure out which functions from PyTorch are compatible and which ones are still under development. This issue partly resulted from the fact that the development branch had to be used for PySyft to provide all functions necessary for this project, so it is not to be expected that everything works smoothly. Overall, PySyft looks like a very promising tool though, incorporating Federated Learning, Secure Evaluation and Differential Privacy and has a vibrant community of developers behind it.

[1] `https://www.openmined.org/`

Inhaltsverzeichnis

Nomenclature

CI	Continuous Integration
CNN	Convolutional Neural Network
DoG	Difference of Gaussian
DoH	Determinant of Hessian
GDPR	General Data Protection Regulation
GPU	Graphics Processing Unit
LoG	Laplacian of Gaussian
MLP	Multi Layer Perceptron
NN	Neural Network
RCNN	Regional Convolutional Neural Network
RNN	Recurrent Neural Network
RTD	Readthedocs
SGD	Stochastic Gradient Descent
SVM	Support Vector Machine
YOLO	You Only Look Once

2 Introduction

Malaria is a disease that can be caused by different obligate parasites and infects about 300-600 million people per year. UNICEF claims that it is also responsible for the death of a child about every 30 seconds. [1] The severe effects of the disease can be resolved by using medication when it is diagnosed and treated promptly [2]
Unfortunately, there are no distinct features to malaria since the main symptoms according to the Centers for Disease Control and Prevention (CDC) are fever, sweats, chills, headaches, malaise, muscles aches, nausea, and vomiting. Since time is of such critical importance and the main way of diagnosing malaria is examining blood samples under a microscope [3] The goal of this project is to create the key functions for an augmentation system for microscopes to help doctors detect potentially infected cells.
The procedures used for this can be applied to any other medical condition that requires microscopic examinations. The dataset used for this project is provided by the U.S. National Library of Medicine.

[1] https://www.unicef.org/health/files/health_africamalaria.pdf
[2] http://www.malaria.com/questions/why-malaria-dangerous
[3] https://www.cdc.gov/malaria/about/faqs.html

3 Object Detection

Images usually contain a variety of objects of which only a fraction is of interest. In Computer Vision finding these is called Object Detection and gained increasing interest staring with the Viola Jones algorithm for face detection. The result can either be a bounding box or a parameterized representation like for Hough transforms.

3.1 Choosing an algorithm

There are many algorithms for general purpose object detection. Some examples are the already mentioned Hough transforms and neural network based approaches like You Only Look Once (YOLO) and iterations of Regional Convolutional Networks[8] (RCNNs). Training a neural network to detect cells sounds promising since for example YOLOv3[7] has shown impressive results for autonomous driving applications but labeling images and properly training a network would not be feasible and take away from the core problem that was taken on in this project. Hough transforms on the other hand don't scale well for complex problems and need prior knowledge of the object in question. A common practice to enhance object detection accuracies if the object has sufficient contrast to the background is to firstly run an edge detection algorithm and then dilate them. This approach would certainly work for this project but is potentially to computationally expensive since real-time augmentation requires rapid image processing.

3.2 Blob detection

The previous discussion of available algorithms shows that a general approach is not feasible. Therefore, it was decided to use blob detection algorithms that respond to circular shaped objects with enough contrast to the background. Pythons Scikit-learn library[6] offers three popular algorithms readily available that show promising results. The mathematics underlying these selected algorithms, namely Laplacian of Gaussian (LoG), Difference of Gaussian (DoG) and Determinant of Hessian (DoH) shall not be further explained here since their development was not part of this project[1].
An example of the results of these algorithms is shown in figure 3.1. To quantify their performance a test setup was developed creating random blobs of various sizes in an otherwise uniformly black image. The results of multiple runs are presented in figure 3.2.

[1] To speed up the blob detection efforts were made to re-engineer LoG with PyTorch and GPU support but it turned out that the highly optimized code behind sklearn does not allow for significant improvements and heuristics are needed to weed out false positives. Developing these seemed not feasible in the context of this project

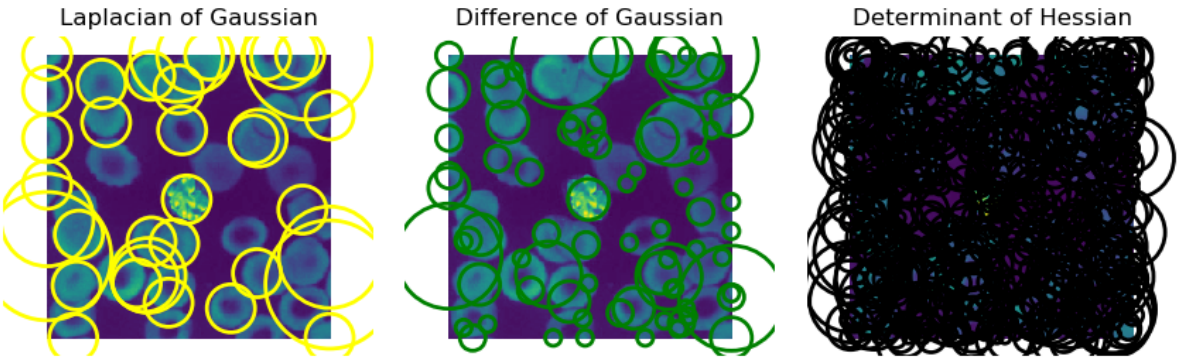

Abb. 3.1: Different blob detection algorithms

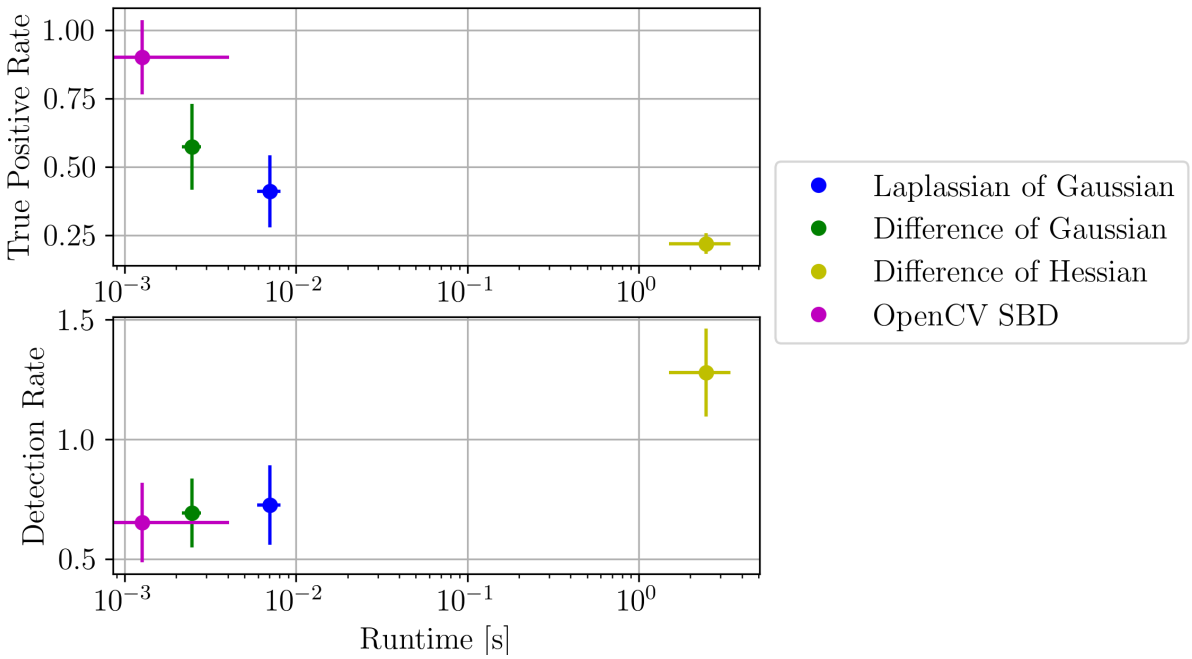

Abb. 3.2: Analysis of different blob detection algorithms

4 Classification

Convolutional Neural Networks are the state of the art since Alex Krizhevsky and Ilya Sutskever won the ImageNet competition in 2012 with over 10% higher accuracy than the runner up[5]. There are alternative approaches like Support Vector Machines (SVMs), exotic approaches like Capsule Nets but they are not competitive anymore. Theoretically it is possible to use all Classification algorithms like logistic Regression, Decision Trees and others but these tend to not scale well with the high dimensional problem images provide.

4.1 Convolutional Neural Networks

The big breakthrough was that Krizhevsky developed a way to train the Parameters of the Network with GPUs which allowed them to train networks that were previously unfeasible to train. The inner workings of Neural Networks would take a separate report to explain. The basic ideas are that Multi Layer Perceptrons (MLPs) are Universal Function Approximators (UFA) that are capable to map high dimensional input to almost arbitrary outputs. The deeper the net and the more neurons in each layer the higher the potential range of functions that can be approximated. Since the number of inputs scales exponentially with the training time of a network it is helpful to find more compressed representations. This is what the convolutional part is for: It combines convolutions also used for edge detection and other tasks in image processing with pooling operations which decreases the number of pixels in the resulting images. The so called pixel maps can then be reformatted to a 1d vector after the number has been sufficiently reduced. The training process works by applying the so called backpropagation algorithm to get the derivatives of the network parameters with respect to a chosen loss function that should be minimized. This means in practice applying the chain rule to the matrix representations that are underlying the operations in the CNN and using variants of Stochastic Gradient Descent (SGD) like Adam to adjust the parameters in the net.
It is important to keep in mind that PyTorch is built around CUDA .Therefore, only NVIDIA GPUs with the correct CUDA version and matching drivers are supported. A common problem with GPU

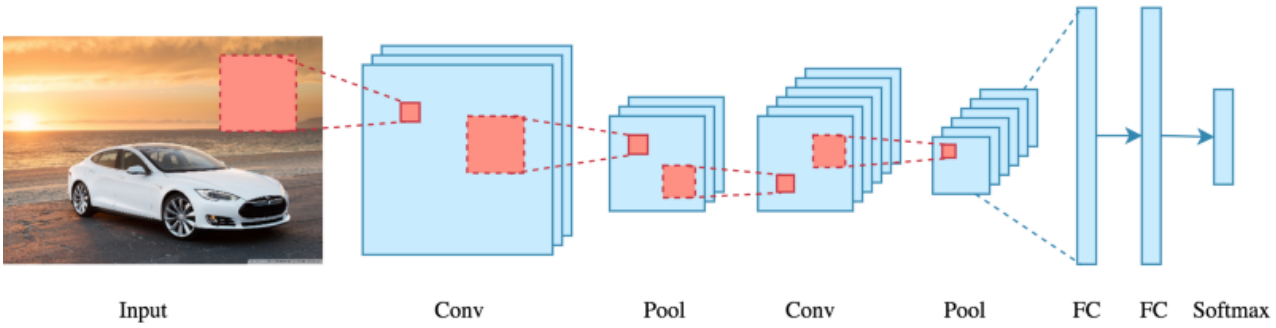

Abb. 4.1: Main components of a CNN, source:towardsdatascience.com

based training is that weights trained on a GPU are not loadable on a CPU since their type depends on the memory location. This can be mitigated by detaching the tensors from the GPU after training and saving them with the CPU.

4.2 Training Speed

Training a Neural Network is structured around the idea of making predictions based on input data that is served in batches, performing SGD for every batch and starting over once the last batch has been parsed. One iteration through the training data set is called an episode or epoch. The time necessary for one epoch on a CPU depends mostly on the size of the network and not on the number of trainable parameters. This becomes clear by comparing the model summaries in section 8.2.1 and 8.2.2 with figure 5.2: A training step for the small net is more than 10 times faster even though it has over $2 \cdot 10^6$ parameters compared to roughly 1000 for Squeezenet[3]. Another observation is that the speedup provided by a GPU also depends on the size of the network: Training times for Squeezenet and the small CNN are approximately the same when they are being trained on a GPU.

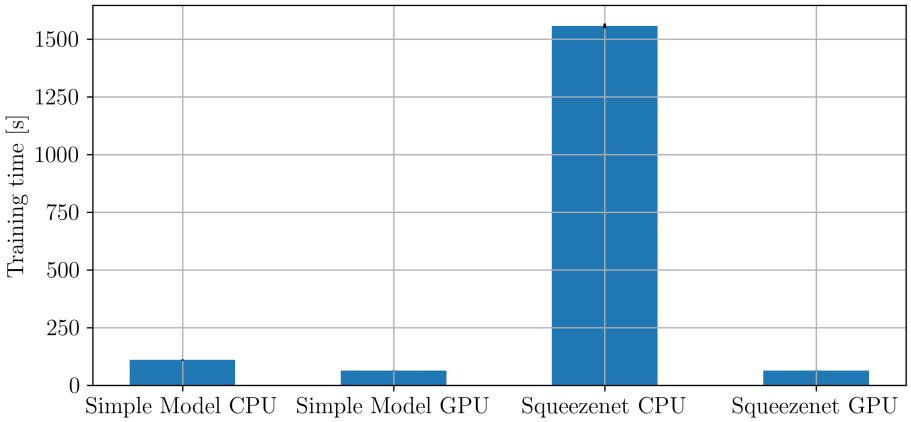

Abb. 4.2: Training times for 3 episodes with different networks

4.3 Inference Speed

Making a prediction with a network is called inference. The time it takes the the smaller CNN and Squeezenet to make predictions for roughly 5400 samples of the test set is plotted in figure 4.3 and 4.4 respectively. The observations about GPU speedups from the previous section hold true: The speedups for the small model are negligible whereas execution times drop significantly for Squeezenet. Nonetheless, executing the small CNN on a GPU is still about an order of magnitude faster than running the bigger model.
Tracing is a feature added with PyTorch 1.0[1] and allows to JIT compile models. The speedups by tracing models are negligible but the execution times seem to be more consistent according to the standard deviation plottet as error bars in figure 4.3 and 4.4.

[1] `https://pytorch.org/blog/the-road-to-1_0/`

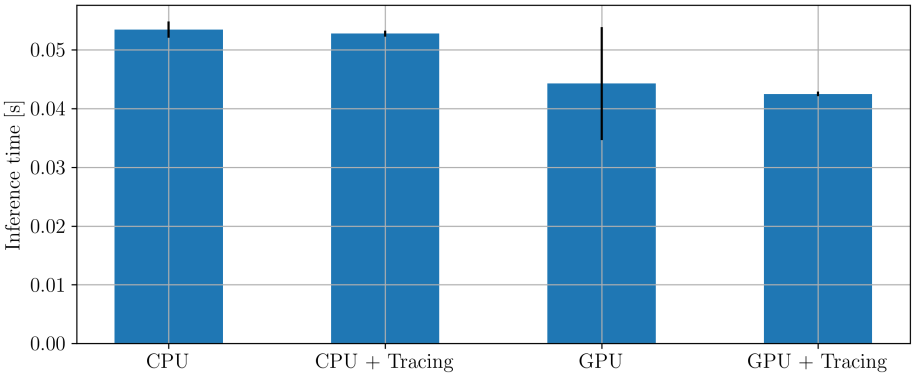

Abb. 4.3: Inference speed for custom CNN

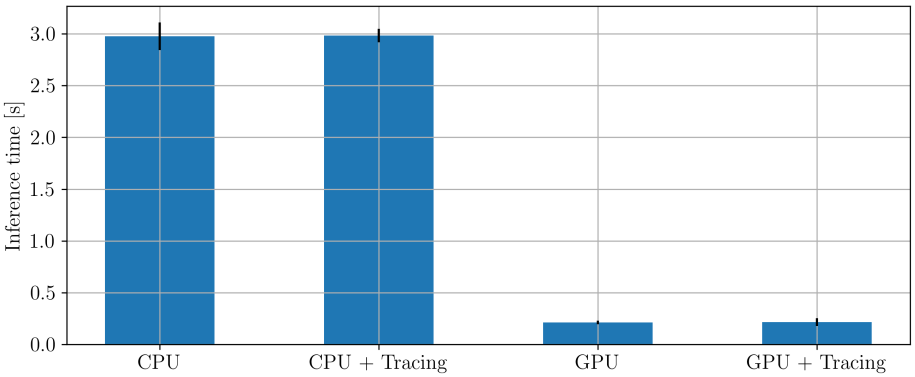

Abb. 4.4: Inference speed for custom CNN

5 Federated Learning

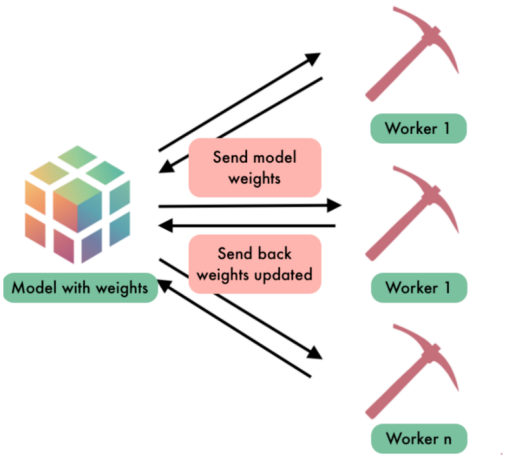

Abb. 5.1: Principal of federated learning, source: openmined

Federated Learning is based on the principle of decentralized data. This means that workers holding a share of a bigger don't have to transmit their data but get the model, perform a couple optimization steps on the weights and biases and only transmit the updated weights or the gradients. For further privacy these gradients can be averaged by a trusted aggregator so that no private information can leak to the model owner. This technique is for example used on mobile devices to improve the next word prediction for the keyboards. This approach of bringing the model to the client also cuts down on network latencies.

Training Speed

The time to train a simple network for 3 episodes with 2 to 4 workers is presented in figure 5.2. The results are surprising since every number of workers was significantly faster then the regular training process. The results can be reproduced by running `speed_federated_learning.py`.
A possible explanation could be that PySyft uses CPUs more efficient than the regular training process if every worker is trained in its own process. This would be supported by the fact that the training speed of federated models seems to correlate strongly with the available number of CPU cores on a machine.
The problem is that the documentation only covers the stable release but this projects PySyft version is taken from the development branch.

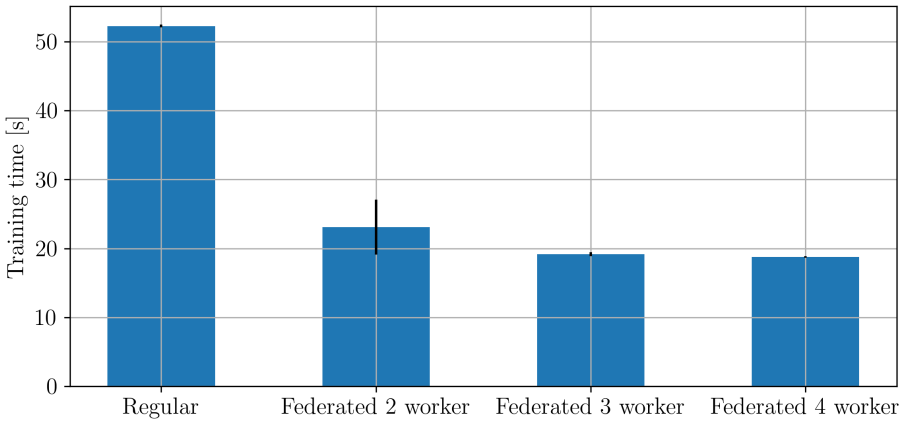

Abb. 5.2: Training times for 3 episodes with a small CNN

Imbalanced Distributions

A question of interest was how the training process is affected by unequal data distribution across workers. For the following examples it is assumed that unequal means about the same number of training samples but different distributions of the labels. The first experiment was carried out assuming that a noticeable increase in the models accuracy should take roughly twice as long for two workers. According to figure 6.4 this would mean $2 \cdot 60$ epochs if SGD is used with a learning rate of 0.1. To compensate statistical fluctuations every distribution was run three times with different random seeds.
The results of this experiment are presented in figure 5.3. It is clear that the initial distributions in this experiment were too unequal since only the one of three runs with an equal distribution showed signs of real progress within the first 150 episodes.
The second experiment was therefore reduced to only two different distributions but extended to 300 episodes. This experiment was carried out with two random seeds per distribution. The results in figure 5.4 indicated that all experiments eventually led to some training progress but barely beat the simple baseline presented in section 6.4.

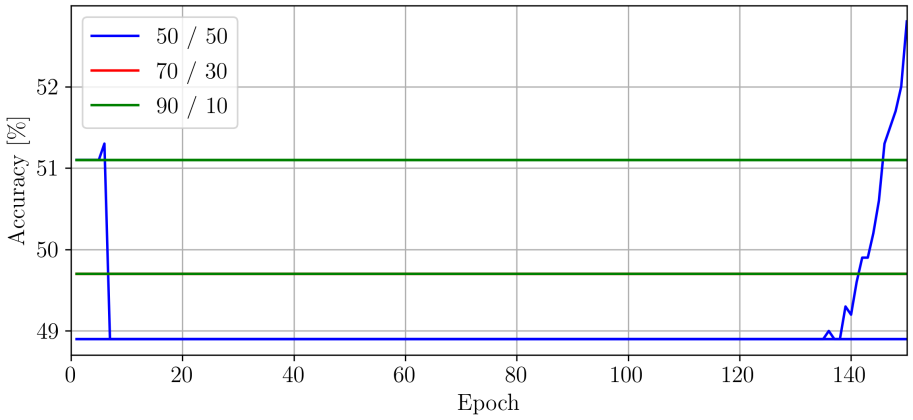

Abb. 5.3: Training two federated workers with each having imbalanced training data

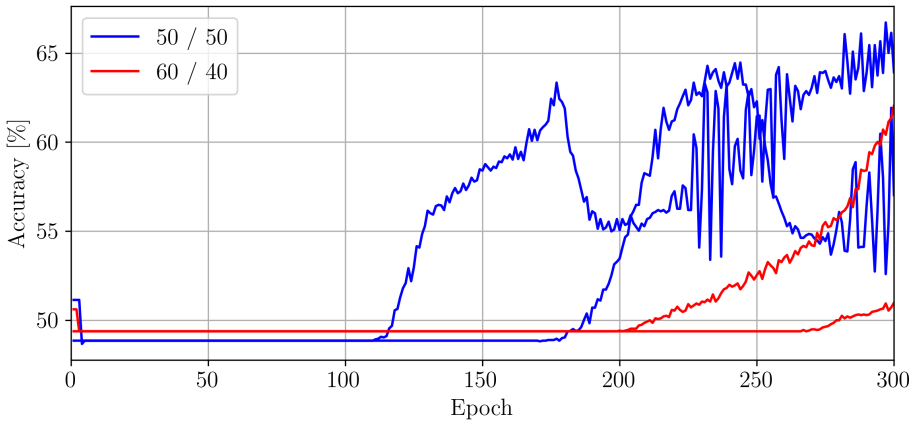

Abb. 5.4: Training two federated workers with each having slightly imbalanced training data for 300 episodes

6 Practices from the lecture

6.1 Programming language

I decided to use Python since it is basically the lingua franca in Machine Learning. A lack of raw performance can be compensated by great GPU support in various libraries like dask, numba, PyTorch and cupy and a versatile interface to Cython which runs at about the speed of C.

6.2 Version management

There are many different tools available but it was decided to go with git, since GitHub is widely used and the majority of the tools I used integrate nicely into it.
Sourcetree[1] is a free git client that works nicely with different repositories hosted by different providers like GitHub, GitLab and BitBucket and makes using git a lot more convenient for starters.

6.3 Profiler

Python offers an array of possible profilers to analyze your code. For other projects I have worked with Cpython and vprof but I decided to benchmark functions for this project with lineprofiler[2] and especially its `kernprof` functionality which only analyses functions with a @profile decorator. It generates either outputfiles or writes to the console directly and offers a very clean interface like shown in figure 6.1.
Another advantage of lineprofiler is that does not have to be run from the command line but can be included as a python function. This makes reproducing analysis very simple and allows for the same workflow under Windows and UNIX machines.

[1] `https://de.atlassian.com/software/sourcetree`
[2] `https://github.com/rkern/line_profiler`

```
File: C:/Users/ac130380/Documents/dmd_project/src/blob_detection.py
Function: analyse_image at line 227

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   227                                           def analyse_image(path, model):
   228                                               """
   229                                               Loads images in path, apply different blob detection algorithms, classify detected blobs
   230
   231                                               :param str path: path to images that should be analyzed
   232                                               :param model: PyTorch model to use for classification
   233                                               """
   234
   235         1       1657.0   1657.0      0.0      images = get_images(path)
   236         6         20.0      3.3      0.0      for image in images:
   237         5     142360.0  28472.0      0.8          img = imread(path + image)
   238         5      90956.0  18191.2      0.5          image_gray = rgb2gray(img)
   239         5         51.0     10.2      0.0          image_gray = image_gray.squeeze()
   240
   241         5      13028.0   2605.6      0.1          image_gray_inverse = 255 - image_gray
   242
   243         5         44.0      8.8      0.0          start = timeit.default_timer()
   244
   245         5    7013040.0 1402608.0     40.8           blobs = dog_blob_detection(image_gray_inverse)
   246         5    9906775.0 1981355.0     57.7           analyze_blobs(blobs, model, img)
   247
   248         5         57.0     11.4      0.0          end = timeit.default_timer()
   249         5        375.0     75.0      0.0          print("It took %.3f s to analyze the picture" % (end-start))


Process finished with exit code 0
```

Abb. 6.1: Lineprofiler analysis of a function

6.4 Testing Machine Learning Code

Testing code in ML is not trivial due to the inherent statistical behavior of Neural Networks meaning that quality of code being often measured in the percentage of correct predictions on given test cases. Recent advances led whole sub-fields of research on the robustness of these models to adversarial attacks and providing guarantees for specific behaviour but these have not found broad adoption in the community yet. An additional problem with Reluplex and similar approaches is the fact that its not feasible and arguably not possible to define all meaningful high dimensional test cases that would be needed to cover all desired behaviour. This is one of the reasons for slow adoption rates of NNs in the financial sector and autonomous driving. An additional problem is that there are whole new dimensions of failure cases that are not present in classical software development. Zayd Enam[3] wrote an excellent blog post on this:
Standard Software Engineering failure cases are either due algorithmic errors or having a bug in the implementation. Machine Learning has additional failure scenarios due to “faulty” models and problems with the data used to build the model. It is debatable if algorithmic problems and model issues are separate dimensions since both can be tackled to a substantial amount by hyperparameter tuning but the main argument is still valid.

Algorithmic Correctness and Model Issues

Baselines

It is hard to evaluate the performance of a model without any context. The simplest assumption for a classification problem with k classes it that with not adjusted weights about every k-th prediction would be right. To get an even better understanding of the problem it is a common practice to create a simple baseline with as few hyperparameters as possible. Commonly used algorithms are listed in the table 6.1 below: For this project it was decided to use logistic regression[4] since it is capable to

[3] `http://ai.stanford.edu/~zayd/why-is-machine-learning-hard.html`
[4] See baseline.py for the implementation

Abb. 6.2: Visualization of failure cases in ML, source: S. Zayd Enam

Regression	Nearest neighbor Linear regression Derivatives of Random Forests like XGBoost
Classification	Nearest neighbor Logistic Regression Derivatives of Random Forests like XGBoost
Time series	Moving average ARIMA Persistence model

Tab. 6.1: Baselines for different standard applications

work with high dimensional inputs that take a toll on decision trees. To avoid unnecessary implementation errors the implementation provided by Scikit-Learn[6] was used. The image size was set to 64 pixels since the dimension of the input scales quadratic with it and bigger images would render the baseline algorithm very inefficient. The dataloaders and especially the `create_dataloaders()` function created for the CNNs, were used to load the images and efficiently adjust their sizes.
The mean accuracy was found to be 58%[5] and should be regarded as a lower bound for more advanced models.

Hyperparameters

Sub-optimal hyperparameters can render a model useless even though supervised learning is usually fairly stable. It is therefore advised to run hyperparameter studies to rule out *incorrect* behaviour that results from a poorly chosen combination of hyperparameters and to validate the performance by carrying out the same experiment with multiple random seeds.
Figure 6.3 shows how the training process varies for the same optimizer settings in multiple consecutive runs. It also highlights how the setup with SGD and a learning rate of 10^{-3} does not seem to make much progress at all and stays at around 50% accuracy which is just as good as random guessing and worse than the established baseline. This was surprising since Adam[4] did well with

[5] run baseline.py to reproduce this result

the same learning rate and basically only differs in the second order momentum that is adaptively to the updates.
It is therefore advised to run experiments multiple times with the same settings to get a better understanding of the average performance and its deviations. It is also a common practice to run hyperparameter searches on an problem setting to map out the robustness of the model.

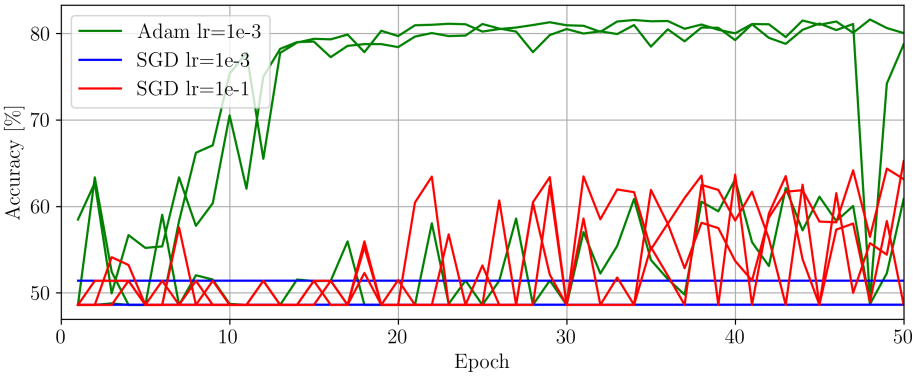

Abb. 6.3: Training a network with different optimizers

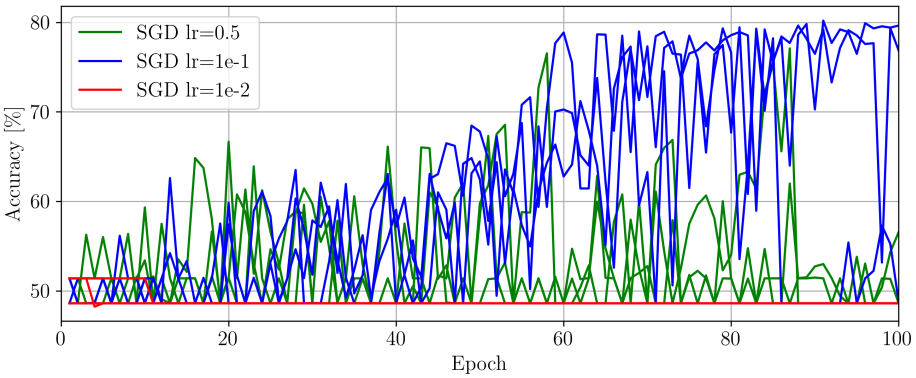

Abb. 6.4: Training a network with different optimizers

Errors associated with flukes in the data

Visualize reasoning behind decisions

There are lots of anecdotes and myths around models learning to classify images based on unintended commonalities in the training data. A famous example is an early example of detecting tanks in images[6]. It turns out the classifier did not catch on the tracks or the canon of the machines but on the fact that enemy tanks were predominately hidden behind trees. This is clearly an unintended behaviour and it's easy to come up with reasonable other examples that are vulnerable to this sort

[6] `https://www.jefftk.com/p/detecting-tanks`

Abb. 6.5: Visualizing the focus of a network with respect to a given label, source: lime

of deception like making a distinction between wolves and dogs based on snow in the background or the like. Lime[9] is a popular attempt to mitigate this by visualizing the features responsible for predictions given by the model. It's not perfect since this approach does not scale but it's a valuable tool to debug drops in performance of deployed computer vision models. This is especially true since Lime is model agnostic and can be used for every given model. Another huge advantage is that visualizing the reasons for a given prediction builds trust between the user and the systems. I had to create a class that takes in my PyTorch model and responds to the same API commands as Keras does in order to use the library. An additional hurdle is that Keras does not use the same formatting convention for batches of images and handles lists or nd arrays as inputs. I took care of those points by adding a reformatting operation and some casting to my predict function.

Using lime

It was not possible to reproduce the fine distinctions that Ribeiro et al.[9] show in their presentations. It is likely that this results from the wrapper not incorporating the full range of Keras[2] functionalities but preliminary results showed that about 10% of the predictions were based on the uniform black background as displayed in figure 6.6. This is why an optional lambda function to randomize the background was added to the dataloader in training. The right picture in figure 6.6 shows that this actually solves the problem described before.
Going through more explanations were alike figure 6.7 and showed that the majority of predictions is actually based on the cell and not on the background. Since training with randomization enabled is fairly time intensive it was decided to disable it by default and results , if not stated differently, were gathered without it.

Monitor data distributions

Monitoring the inputs to the model is an important task to ensure that performance specifications are met. This includes input levels as well as input labels since it is quite intuitive that networks which were trained to classify white shapes on darker background will show different performance

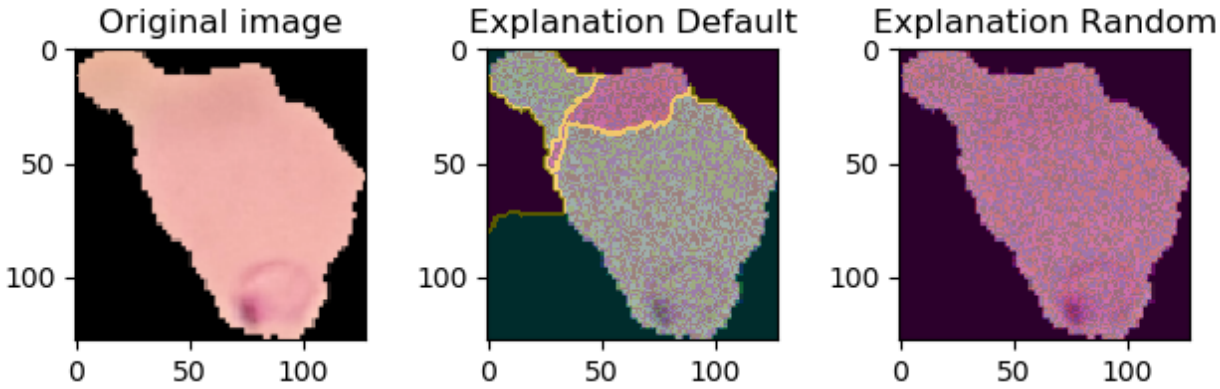

Abb. 6.6: Spotting unwanted patterns with lime

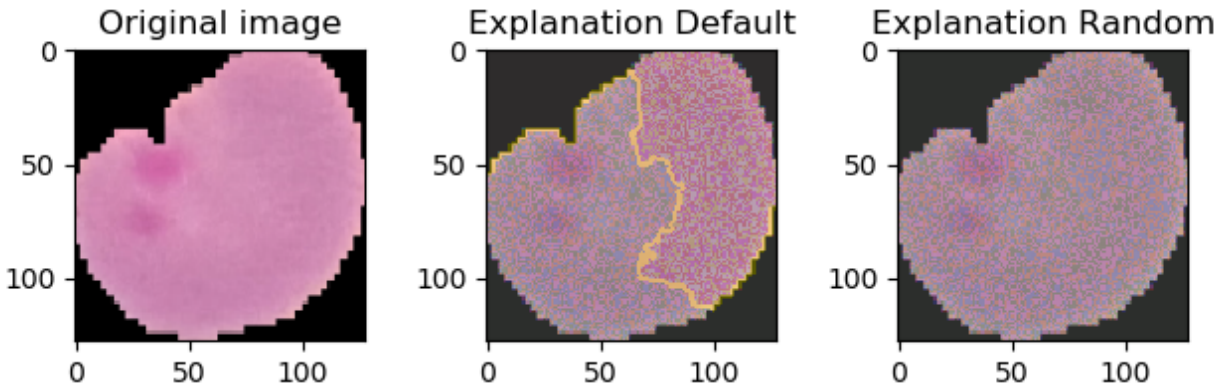

Abb. 6.7: The majority of explanations showed the desired behaviour

for dark images on light backgrounds. The same is true if there is a drift in the inputs means since this could potentially compromise the "logic gate" alike structures built by the activation functions. Considering changing label distributions one has to keep in mind that classification accuracies vary between different classes so it is clear that the performance drops if the ration of classes increases that the network struggles to detect correctly.
For images it is also possible to monitor the histograms of the pictures to determine if the inputs match the data the network was trained with. The input distribution for the networks used in this project is shown in figure 6.8. Possible similarity metrics could be for example the KL-divergence of the normalized input histograms. In real world applications filters can be applied to adjust the input images to the training data.
Another approach is to train a binary classifier on to distinguish between training data and inputs of the deployed model. That way warnings can be raised if the classifier "works too well", meaning that there is a divergence between the datasets that can't be neglected.

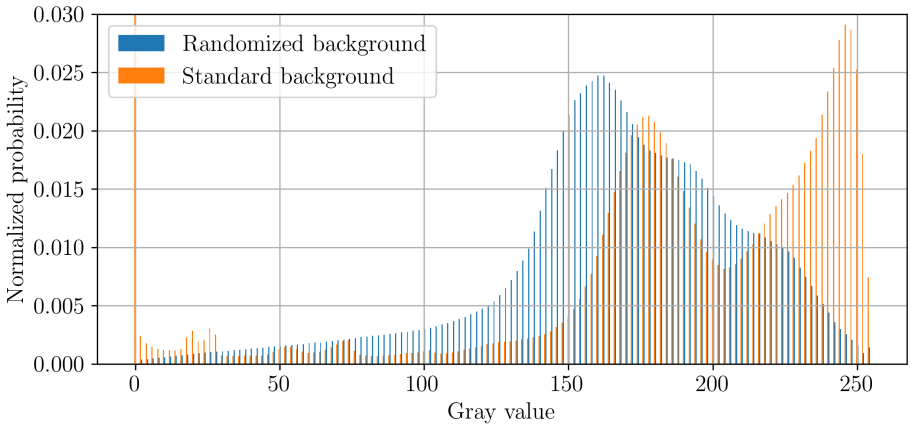

Abb. 6.8: Histograms of the training data with and without enabled background randomization

Implementation errors

Test the training process of the network

One of the most difficult problems in debugging ML models is that systems may fail silently and practitioners end up retreating to staring for a really long time at their code as the best way to detect what could have gone wrong. Nonetheless there are certain ground truths that can be tested in order to spot at least some mishaps:

- Usually, all parameters should be adjusted by the optimizer so constant parameters should make you suspicious
- The loss is never zero
- Activation functions like tangens hyperbolicus or sigmoid limit the output to a certain range – if the output exceeds that range it is clear that there is a mistake
- Especially for RNNs exploding or vanishing gradients are nasty to deal with. A good indicator for these types of failure cases are NaN or Inf values in the weight and/or bias tensors

6.5 Continuous Integration

Continous integration is widely used in bigger open source projects with multiple contributors. CI frameworks like Jenkins or Travis offer lots of functionalities out of the box that you would have to otherwise implement yourself. The key advantage over writing these functionalities locally yourself thought is that they are applied to every contributor of the project and can therefore be used to enforce quality standards.
The choice of Travis over Jenkins was mainly based on the fact that Jenkins needs a dedicated server to run on. This might be an advantage for other users since this ensures data security and gives the user lots of options regarding settings and plugins. The downside of Travis that enterprise plans charge the user does not apply to this project.
Some of Travis' features include mail notifications about builds, regularly running tests that replace cron jobs and the ability to define multiple environments to test the code in. Especially, the last feature is useful for broadly distributed code since it allows to test multiple permutations of Python and package versions on Travis' remote servers. Python has a similar tool called tox but running a lot of combinations can be cumbersome and time consuming.

6.6 Pre-commit hooks

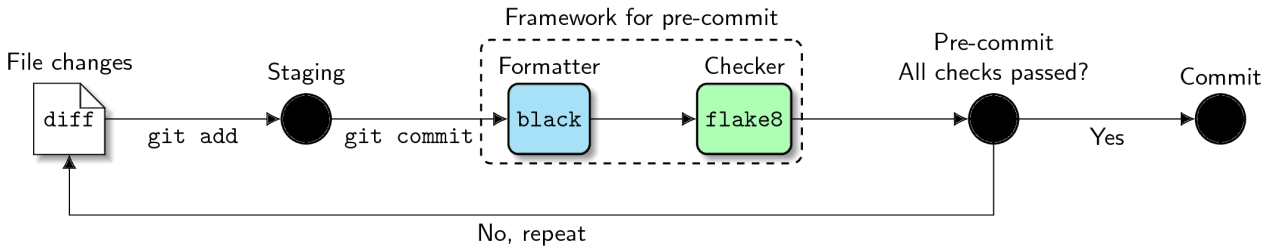

Abb. 6.9: Pre-commit framework

Pre-commit is a framework that integrates nicely into the git workflow. As shown in the figure above it runs an array of tests that make sure that best practices are being followed. Commits are only possible after all tests are passed or skipped since they did not apply to the changes made to the code (even though hooks can be confined to other stages as well). Pre-commit hooks are easily set up and can be copied to other repositories without any changes necessary. A disadvantage is that it is not possible to make a set of pre-commit hooks mandatory for contributing to a project directly. This leads to different standards in the contributed code. An additional disadvantage is that most of the tools are configured using the command line which makes them cumbersome to use on Windows.

Bandit

Bandit is a tool that checks your code for common security threats like hard-coded password strings or unreliable or insecure connections to the internet.

Black

Formatting Python correctly can increase the readability of source code significantly but it just so happens that code is rarely revised to get the formatting right after the functionalities have been developed. Black[7] is quick and deterministic way to to reformat code to comply with PEP8 and allows developers to worry more about content and less about style guides.

Flake8

Flake8 is a tool for static code analysis that marks bugs and enforced coding standards. Tools like theses are called linters. It was decided to use flake8 since it is easily integrated into the pre-commit framework and basically wraps pycodestyle and pyflakes together. It uses by default "PEP-0008 inspired" style checks but this can be modified to enforce different standards.

[7] `https://black.readthedocs.io/en/stable/`

6.7 3rd party checks

Codeclimate

Codeclimate [8]checks repositories for technical debt and bad practices and rates the quality of the code. Maintainability checks include among others checks for deeply nested loops, excessive numbers of arguments passed to- or returned from a function and the existence of recurring blocks of code that could be combined into a single function. Additionally, codeclimate can check the test coverage of the repository.
Eventually, it was decided to not use code climate since it uses a metric called cognitive complexity that was less expressive than the metrics used in Codacy.

Codacy

Codacy[9] shares a lot of the functionalities provided by codeclimate but has additionally a strong emphasis on pointing out security issues. It has a clear distinction between stylistic errors, duplicate code and code complexity which makes it easy to get a fast overview.

Snyk

Snyk[10] checks dependencies for vulnerabilities hidden in project dependencies, which is a useful feature for safety critical systems. Interestingly, the scientific computation library numpy is marked as a security threat. This is surprising since this is the backbone of the majority of all scientific repositories and is deeply integrated into PyTorch which makes it hard to replace.
The reason given for this rating is that numpy uses the "pickle Python module unsafely which allows remote attackers to execute arbitrary code via a crafted serialized object". It is clear that this poses a realistic threat if it were exploited but is not essential in the range of this project. If a system similar to this were deployed to real hospitals measures would have to be taken to either block this or exclude numpy from the project.

[8]`https://codeclimate.com`
[9]`https://www.codacy.com/`
[10]`https://snyk.io/`

6.8 Documentation

It is a good practice to write a documentation to ease the usage of code and make it easier for others to contribute. Since writing a separate documentation takes time away from developing functionalities it is helpful to generate documentations automatically from docstrings in the code. Outdated documentations are a potential hazard that should be avoided, online documentation have an advantage over local copies since currency can that way be controlled centrally. There are multiple tools that do that most like doxygen for C and Sphinx for Python.

Sphinx

Sphinx[11] was originally developed for the Python documentation and creates good looking documentations. It has a handy introduction video[12] to get first time users up to speed and functionalities like "quickstart" to create the needed folder structure. It uses reStructured Text but has interfaces to Markdown and HTML as well. It is recommended to use .rst files though since there is no unified standard on how to write and interpret Markdown files which can lead to unwanted results. Sphinx comes with useful tools like an alphabetic sorting of functions, a search menu and right out of the box

Readthedocs

Readthedocs (RTD) is a free hosting platform for technical documentations. It integrates nicely with Sphinx and optionally generates pdf versions of the websites for offline usage. RTD generates an environment with the libraries specified in requirements.txt . Since the memory limits don't allow for my whole environment to be installed there it was necessary to create a reduced requirements list and add mock classes for the not supported elements of the list. This insight was shared in a similar GitHub issue [13] . RTD creates a hook to GitHub and builds a new online documentation every time code is pushed to the remote repository. This makes sure that no deprecated versions are available. There is also a banner available showing the current build status of the documentation in my GitHub repository.
If multiple versions of a documentation exist it can be useful to add a robot.txt file to the repository to keep Google and other search engines from showing deprecated versions. My online documentation can be found at here.

[11] `http://www.sphinx-doc.org/en/master/`
[12] `https://docs.readthedocs.io/en/latest/intro/getting-started-with-sphinx.html`
[13] `https://github.com/rtfd/readthedocs.org/issues/5328`

7 Miscellaneous

7.1 Handling the training data

In order to use computational resources efficiently it was decided to create a dataloader to load the images of infected an healthy blood cells. Alternatively, one could have loaded each image individually, reformatted them and appended them to an array but that would be prone to unnecessary bugs and inefficient code. Additionally PyTorch dataloader class allows to sequentially load data that would be too large to fit into RAM and is therefore the tool of choice if scaling is important. To work with randomized results a function was added to the dataloader to randomize background pixels. Experiments with this feature take considerably longer than standard experiments. Especially inefficient is the fact that the randomization and background detection needs to be conducted every time a new batch is loaded. It is therefore advised to run the randomization once and save the resulting images in a new folder.

7.2 Cross Platform Compatibility

Python is principally available on all major platforms. There are potential pitfalls to render code written under Linux unusable on Windows and vice versa. Most notably it is a good practice to not use statements that include "\" or "/" since these are likely to fail. Python 3.4 introduced the pathlib library that can create objects that will automatically adept to the operation system the program is executed in. It also seemed like the import system works differently as well. Relative imports work well with both operating systems whereas absolute imports tend to fail under windows. Another observation is that PyCharm works well with absolute imports but does not offer functions like auto-completion and showing the arguments for custom functions if they are relative imports[1]. This is why it was decided in my project to generally use absolute imports since the majority of the work was carried out on Linux machines and put the import statements in try except blocks that react to ModuleNotFoundErrors [2].
OpenCV[1] is theoretically available for Linux and Windows machines but installing it on Windows is cumbersome and not advised.

[1] After reading through many threats I came up with a solution to the problem described before: sys.path.append(os.path.join(Path-to-root-directory")) changes the PATH variable correctly to use direct imports.
[2] `https://realpython.com/absolute-vs-relative-python-imports`

7.3 KD Tree

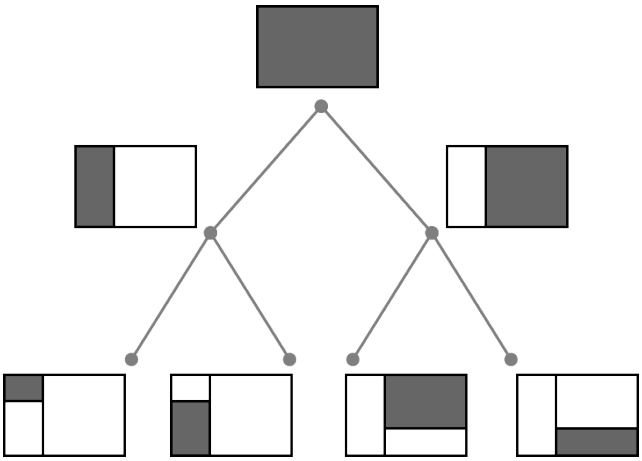

Abb. 7.1: Structure of a KD Tree,source

Finding the distance between the nearest blob and a center predicted by a blob detection algorithm was necessary to evaluate the performance of these. Scaling up the images while adding more and more blobs showed that the naive approach of calculating the distance in nested for loops running over the lists containing the real locations and the predicted ones respectively lacked computational efficiency. It was therefore decided to look into alternatives and KD Trees seemed to be the best fit for the task. This data structure splits an k-dimensional set of points perpendicular to an alternating axis. The criteria used to determine the split location is to separate the points in the set into equally sized children. An efficient way to do this is by using Quick Sort (or other sort algorithms): The set of points is sorted by the respective dimension, then the median is selected as location for the split. By performing this procedure recursively a binary tree is built as shown in figure 7.1. The time spans needed to find the nearest neighbor for two equally sized lists is shown in figure 7.2. It illustrates how KD Trees start outperform the naive approach significantly as the lists get longer.

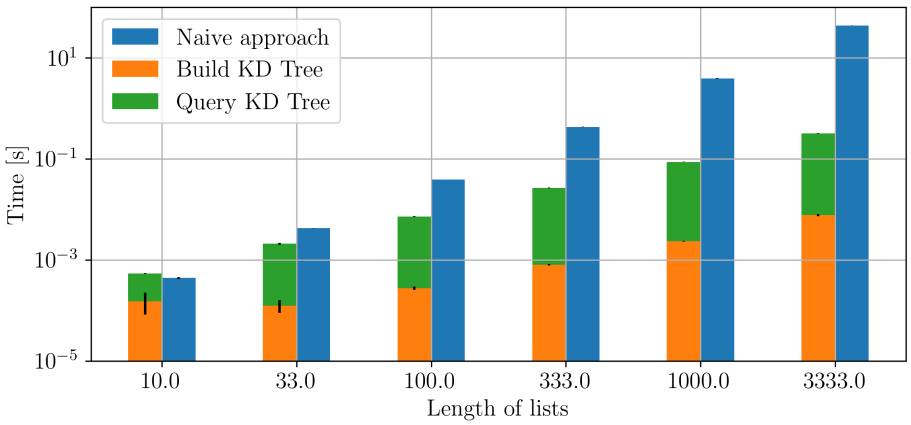

Abb. 7.2: Time to perform the nearest neighbor search for lists of different length

Literaturverzeichnis

[1] G. Bradski: *The OpenCV Library.* Dr. Dobb's Journal of Software Tools, 2000.

[2] François Chollet et al.: *Keras.* `https://keras.io`, 2015.

[3] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally und Kurt Keutzer: *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size.* CoRR, Vol. abs/1602.07360, 2016.

[4] Diederik P Kingma und Jimmy Ba: *Adam: A method for stochastic optimization.* arXiv preprint arXiv:1412.6980, 2014.

[5] Alex Krizhevsky, Ilya Sutskever und Geoffrey E. Hinton: *ImageNet Classification with Deep Convolutional Neural Networks.* In: Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, NIPS'12, USA, 2012, Curran Associates Inc., S. 1097–1105.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot und E. Duchesnay: *Scikit-learn: Machine Learning in Python.* Journal of Machine Learning Research, Vol. 12, S. 2825–2830, 2011.

[7] Joseph Redmon und Ali Farhadi: *YOLOv3: An Incremental Improvement.* CoRR, Vol. abs/1804.02767, 2018.

[8] Shaoqing Ren, Kaiming He, Ross Girshick und Jian Sun: *Faster r-cnn: Towards real-time object detection with region proposal networks.* In: Advances in neural information processing systems, 2015, S. 91–99.

[9] Marco Túlio Ribeiro, Sameer Singh und Carlos Guestrin: *"Why Should I Trust You?": Explaining the Predictions of Any Classifier.* CoRR, Vol. abs/1602.04938, 2016.

[10] Theo Ryffel, Andrew Trask, Morten Dahl, Bobby Wagner, Jason Mancuso, Daniel Rueckert und Jonathan Passerat-Palmbach: *A generic framework for privacy preserving deep learning.* CoRR, Vol. abs/1811.04017, 2018.

8 Appendix

8.1 Contributions

This section includes issues and attempted fixes that came up during the course of the project.

Travis

- Big installs like torch raised an error "The job exceeded the maximum log length, and has been terminated."
- `https://travis-ci.community/t/large-pip-installs-exceed-maximum-log-length/1599/12`
- `https://github.com/travis-ci/travis-build/pull/1685`
- I created a Pull Request - We discovered that this issue had been mitigated by another pull request but the default images used in Travis don't support the arguments used for that fix

PySyft

- Federating datasets doesn't work with subsets
  - `https://github.com/OpenMined/PySyft/issues/2071`
  - I wrote a small tool to convert Subsets back into Datasets that are compatible with the .federate((...)) method in PySyft
- Created an issue about the optimizer Adam causing errors in the training loop
  - `https://github.com/OpenMined/PySyft/issues/2070`
- Recent functionalities are not available in the PySyft version on pypi
  - `https://github.com/OpenMined/PySyft/issues/2080`

Readthedocs

- Inconsistent theme and display of docstrings and bypassing memory limitations RTD
- `https://github.com/rtfd/readthedocs.org/issues/5643`

8.2 Model summaries

8.2.1 Custom CNN

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 50, 42, 42]	1,400
Conv2d-2	[-1, 50, 14, 14]	22,550
Conv2d-3	[-1, 50, 2, 2]	22,550
Linear-4	[-1, 800]	160,800
Linear-5	[-1, 800]	640,800
Linear-6	[-1, 800]	640,800
Linear-7	[-1, 800]	640,800
Linear-8	[-1, 2]	1,602

Total params: 2,131,302
Trainable params: 2,131,302
Non-trainable params: 0

Input size (MB): 0.19
Forward/backward pass size (MB): 0.77
Params size (MB): 8.13
Estimated Total Size (MB): 9.09

8.2.2 Squeezenet

Layer (type)	Output Shape	Param \#
Conv2d-1	[-1, 96, 109, 109]	14,208
ReLU-2	[-1, 96, 109, 109]	0
MaxPool2d-3	[-1, 96, 54, 54]	0
Conv2d-4	[-1, 16, 54, 54]	1,552
ReLU-5	[-1, 16, 54, 54]	0
Conv2d-6	[-1, 64, 54, 54]	1,088
ReLU-7	[-1, 64, 54, 54]	0
Conv2d-8	[-1, 64, 54, 54]	9,280
ReLU-9	[-1, 64, 54, 54]	0
Fire-10	[-1, 128, 54, 54]	0
Conv2d-11	[-1, 16, 54, 54]	2,064
ReLU-12	[-1, 16, 54, 54]	0
Conv2d-13	[-1, 64, 54, 54]	1,088
ReLU-14	[-1, 64, 54, 54]	0
Conv2d-15	[-1, 64, 54, 54]	9,280
	...	
Conv2d-57	[-1, 256, 13, 13]	16,640
ReLU-58	[-1, 256, 13, 13]	0
Conv2d-59	[-1, 256, 13, 13]	147,712
ReLU-60	[-1, 256, 13, 13]	0
Fire-61	[-1, 512, 13, 13]	0
Dropout-62	[-1, 512, 13, 13]	0
Conv2d-63	[-1, 2, 13, 13]	1,026
ReLU-64	[-1, 2, 13, 13]	0
AdaptiveAvgPool2d-65	[-1, 2, 1, 1]	0

Total params: 736,450
Trainable params: 1,026
Non-trainable params: 735,424

Input size (MB): 0.57
Forward/backward pass size (MB): 89.22
Params size (MB): 2.81
Estimated Total Size (MB): 92.60