Cassava Leaf Diseases Classification

Kaggle project for Image Classification

February 15th, 2021

I. Definition

Project Overview

Convolutional Neural Networks (CNN) are one of the major machine learning (ML) algorithm for computer vision. This project shows a relative basic approach how CNN can be used for image classification. It includes the pipeline for model development: exploratory data analytics (EDA) and the iterative process of model engineering and model validation. The next step might be model deployment for closing the ML operations process on one end - the other end (data engineering) is already made by the competition platform Kaggle.

Problem Statement

The cassava leaf disease classification is a *Kaggle* competition. Kaggle is one of the most famous platforms for ML competitions and a well-known place to gain practical experience with ML.
The crop cassava is a highly important carbohydrate provider in Africa and widely spread amongst African smallholder farmers. Even though they can withstand harsh conditions, there are some viral leaf diseases affecting crop failures. The aim of this Kaggle project is to use data sciences for detecting infected plants (four different diseases) and giving the farmers the chance to separate them and consequently save their fields.[1]
The steps of this project are[2]:

1. EDA
2. Data processing (“making data ready for neural nets”)
3. Creating a simple model baseline
4. Overfit the dataset[3]
5. Regularize the model
6. Inference

[1] https://www.Kaggle.com/c/cassava-leaf-disease-classification
[2] The pipeline of model development is highly influenced by a blog post of Andrej Karpathy “A recipe for Training Neural Nets” (http://karpathy.github.io/2019/04/25/recipe/).
[3] Steps 4 and 5 are iterative and include model engineering and model evaluation.

Metrics

This Kaggle competition sets *accuracy* as metric. The accuracy is the number of correct predictions divided by the amount of predictions made. It is a suitable metric for classification problems, where the samples are quite equally distributed amongst all classes. Even though we use accuracy for being able to compare our results to other users' results, there might be better options for a productive system due to two reasons.

Firstly, as you can see in the following chapter, this dataset is imbalanced and therefore does not meet the requirement above. Imbalanced means that one or more labels are overrepresented and, logically, one or more labels are underrepresented. Here e.g. only 12% of the images show healthy plants. The cause is that a model misclassifying all healthy plants as infected can still reach an accuracy of 88% by consistently recommending the farmer to remove every plant. Does not seem to be helpful![4]

Secondly, accuracy penalizes false negative (here: infected plants classified as healthy) and false positive (here: healthy plants classified as infected) predictions equally. In contrast to that, farmers might prefer a system that focuses on detecting as many of the infected plants as possible, because just a few of them could ruin the entire harvest. This could be monitored by using recall as metric. *Recall* in our case is all correctly predicted and infected plants (true positives) divided by all infected plants (true positives and false negatives). To prevent the system from just predicting all plants as infected - this leads easily to 100% recall, but an useless system - we should also keep an eye on the false positive predictions (healthy plants that are predicted as infected and therefore destroyed). The *precision* is the percentage of all plants predicted as infected which are indeed diseased (true positives divided by true positives plus false positives).

One possible solution is taking the *F1 score*[5]. The more flexible and, therefore, recommended way is to predetermine a lower tolerance for the precision. We could e.g. allow 10% false positives (i.e. set the requirement of minimum 90% precision) and try to identify as many infected plants as possible (i.e. optimize recall).[6]

Monitoring the right metric is crucial for the benefit of a ML system and customers might not be familiar with all the details. However, there might be further options on

---

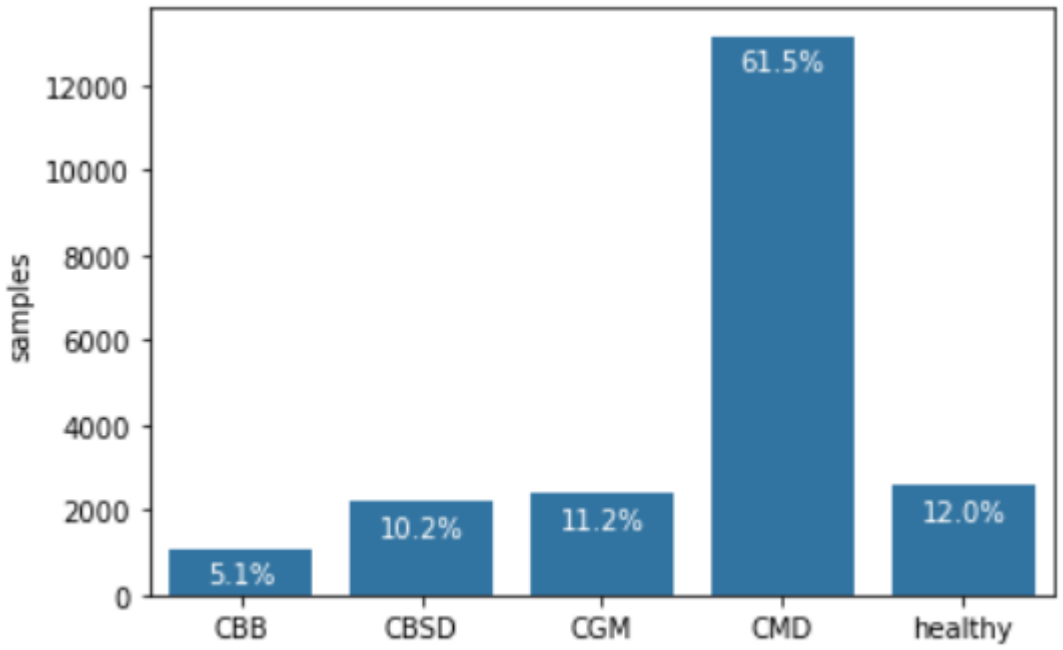
[4] Even though this project is a five class classification problem, we assume that the differentiation between healthy and infected might be of prime importance.
[5] F1 score = 2 * (precision * recall) / (precision + recall) (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html)
[6] deeplearning.ai - course by Andrew Ng and https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall

top of optimizing the ML model to meet customers' needs. In this case e.g., farmers could be motivated to input multiple images of different angles to support good predictions.[7]

II. Analysis

Data Exploration and Exploratory Visualization

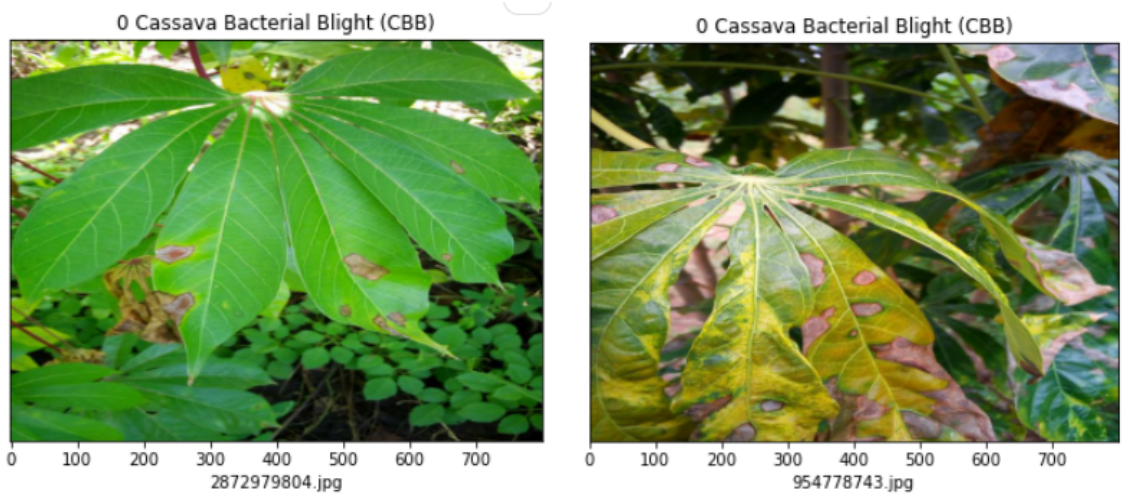
The project provides various data files and folders. The folder *train_images* contains 21.397 image files of type "jpg". There is only one "jpg" image file in folder test_images - the rest of the test data is hidden for the user and will be shown only by submitting the trained model to the Kaggle platform. The file *train.csv* keeps the information about the ground truth label for each image in *train_images*. The test_tfrecords and train_tfrecords contain the image data in a special Tensorflow data type. It speeds up training in Tensorflow, but will not be used in this solution.



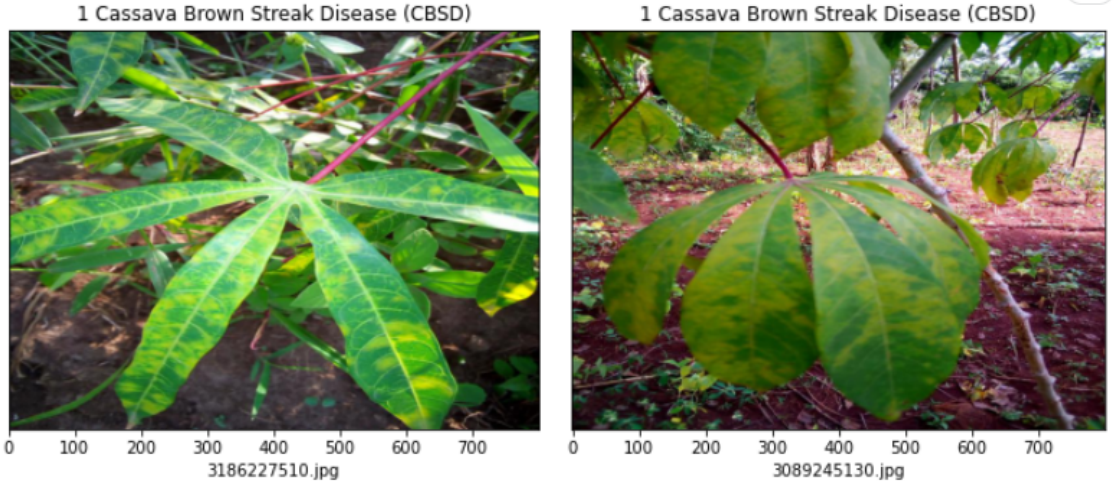
As the figure above clearly shows, the image data is highly biased towards the label 3 (disease CMD) and only 12% of the data shows healthy plants (label 4). This has a significant impact for training a model and, therefore, has to be taken into consideration. All pictures are of pixel size 800x600 (width x height).

[7] There are further ideas about monitoring the benefit of the model in the chapter "Benchmark".

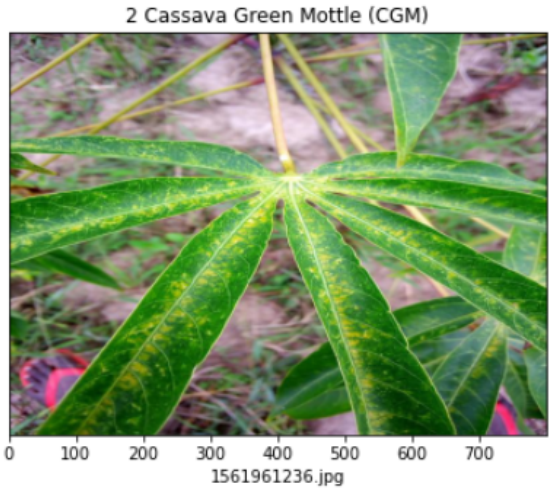
Label 0 - Cassava Bacterial Blight (CBB):
CBB apparently causes brown round marks which might appear especially at the edge of the leaves. Also some of the leaves are withered.

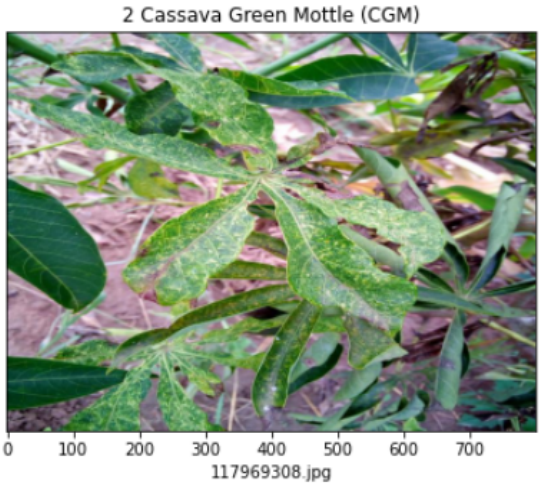


Label 1 - Cassava Brown Streak Disease (CBSD):
Even though the name contains “brown”, plants infected with CBSD seem to get extensive yellow marks in the regions of their streaks - at least this matches the name...

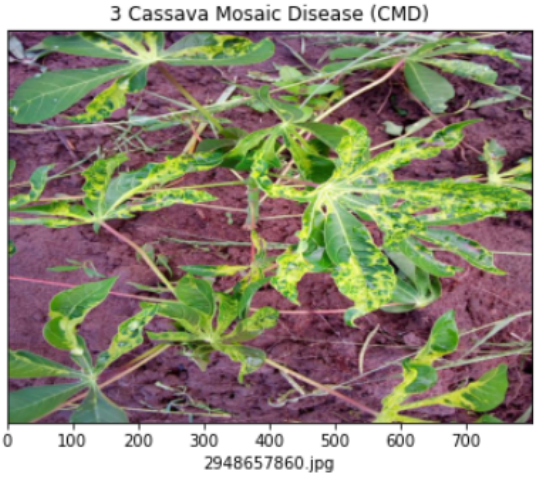


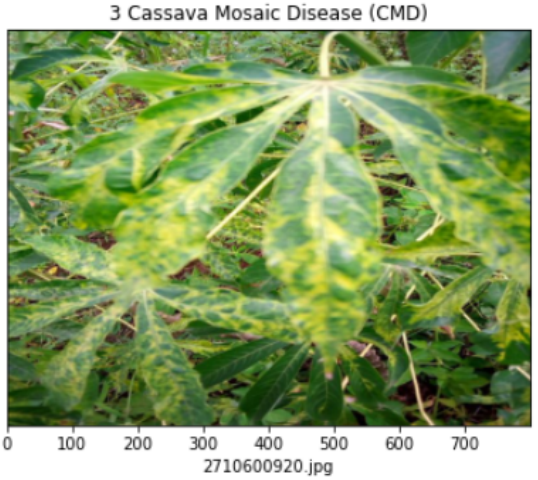
Label 2 - Cassava Green Mottle (CGM):
CGM looks quite similar to CBSD. While CBSD's marks are even and extensive, CGM seems to cause small yellow punctiform marks. However, models could have problems with distinguishing CGM from CBSD.



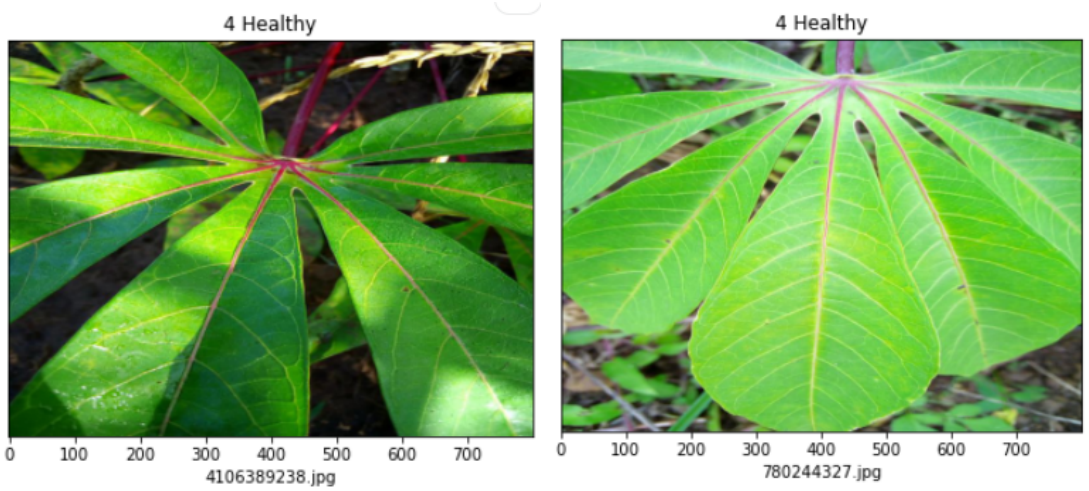


Label 3 - Cassava Mosaic Disease (CMD):
This disease affects not only the color (extensive, yellow marks) but also sometimes causes the shape (corrugated leaves).

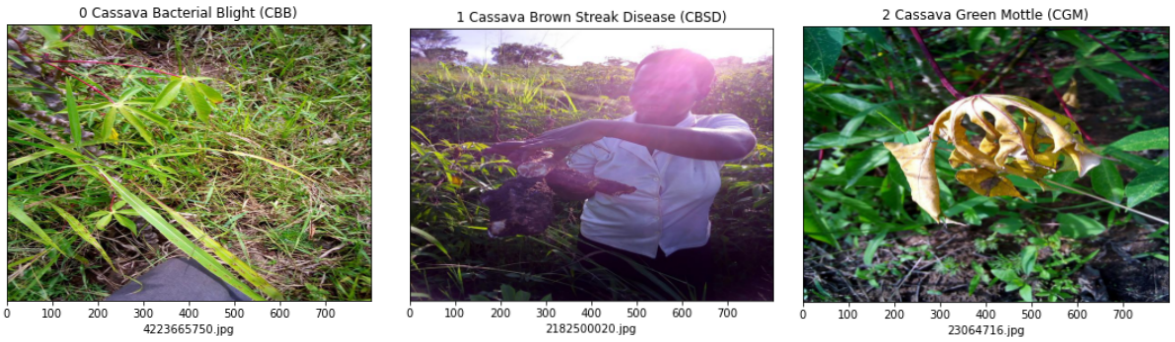




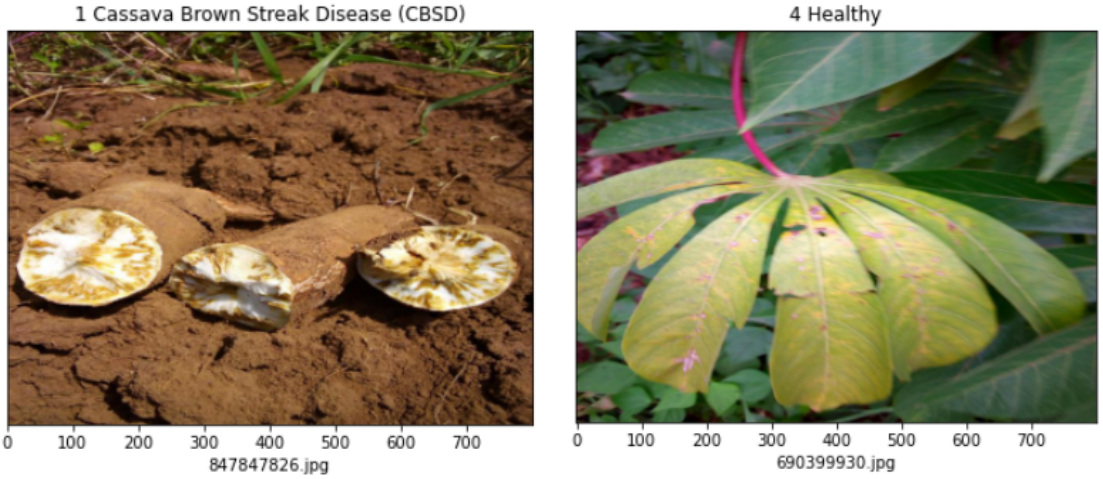
Label 4 - Healthy:
Healthy leaves just look … healthy. They might be all green with slight yellow streaks and do not have any marks.



Noisy Data and Mislabeling:
As the images below show, there is a lot of noisy data in this dataset. Pictures are taken far apart and, therefore, might not display enough details and focus on the leaves. Other ones even include people or show leaves that are so withered, that it is hard to believe even a specialist could label those images.
Another problem is that the brightness and contrast of the pictures vary a lot and many of them have a bad resolution. This could be addressed during the data augmentation process.



Of course, there might also be some mislabeled and wrong data. The examples below show a root and a not healthy looking leaf, probably mislabeled as “healthy”.

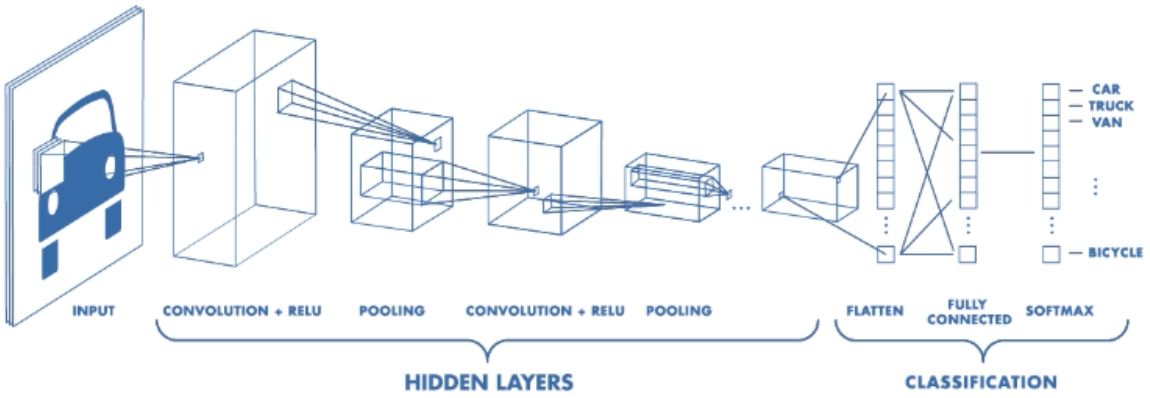


Even though it is not included in this project, mislabeled and wrong data should be removed from the dataset.

Algorithms and Techniques

As this project is an object detection task, Convolutional Neural Networks (CNN) should be the best practice. CNN are a subset of deep learning and only need minimal pre-processing of the input data.[8] The classical architecture of a CNN is a varying number of convolutional layers each with an activation function (mostly ReLUs) and each followed by a pooling layer. The neurons of the last pooling layer serve as the input of several fully-connected linear layer also each with an activation function. This architecture is shown in the image below.[9]

[8] https://deepai.org/machine-learning-glossary-and-terms/convolutional-neural-network
[9] https://www.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks--1489512765771.html



Transfer learning is one of the key breakthroughs of computer vision in the last years. Models are trained on big datasets (mostly the famous ImageNet) and can be reused with the learned parameters for almost all computer vision problems. The key idea is that the pretrained models learn to detect different kinds of basic structures, edges and forms which can be reused for other problems. This can be achieved e.g. by just retraining the last fully-connected layer and keeping the rest of the model's parameters constant.

Tools and Code Structure

All code is written on Kaggle Kernels, the free cloud computational environment of Kaggle. They supply free GPU acceleration and allow coding in notebooks as well as scripts. The project is divided into one utility script and notebooks for EDA, training and inference. The utility script includes e.g. the model and dataset classes, some useful functions and the engine class, which contains the training, evaluation and prediction loops. The modular code structure with the utility script keeps the notebooks simple and allows easier code recycling for future projects. The code is written in PyTorch.

ML is a highly iterative process and logging the configurations and training results is essential. This project used the tool *Weights and Biases*.

Links to Kaggle kernels:

https://www.kaggle.com/martinkolb25/cassava-eda-dataprep

https://www.kaggle.com/martinkolb25/cassava-training

https://www.kaggle.com/martinkolb25/cassava-inference

https://www.kaggle.com/martinkolb25/cassava-utils

Link to weights and biases project:

https://wandb.ai/mko25/cassava/overview?workspace=user-mko25

Benchmark

There is an older version of the same classification problem existing as a Kaggle competition. The best solution reached an accuracy of 92%. This project does not have the aim to reach or even beat this, because of the current knowledge of the author as a beginner. But it should be possible to get at least in the range over 85% accuracy.

As already told in the chapter “Metrics”, the author would not suggest evaluating the solution by calculating the accuracy if it would not have been a Kaggle competition. It is not the main goal of the farmer to have a computer vision system that identifies infected plants perfectly. The main goal should be to maximize the harvest. Predictions of a computer vision system lead to actions (destroy or not destroy plants). Maybe there is a possibility to calculate a metric which estimates the harvest of a farmer if his actions are based on our models prediction. The idea is that false positive predictions (healthy plants get destroyed) have another impact on the harvest than false negative predictions (infected plants remain in the plantation and contaminate other plants). The metric “estimated harvest” could also be compared to not-ML options (e.g. training of farmers; aid of biology specialists) and should also be at least higher than today’s harvest.

To conclude the previous thoughts, it’s important to define the correct metric that correctly meets the customers' needs and to know the corresponding benchmark to beat.

III. Methodology

Data Preprocessing

k-fold cross-validation

Cross-validation is a technique that holds back some of the labeled data from the training process, and thus helps to evaluate the model's success on data it has not seen during training. k-fold cross-validation stabilizes these evaluation metrics by splitting the data into k subsets of equal size. Instead of just training one model with one validation subset, k models are trained while each of the k subsets is used as validation data for one of these k models. During inference - the process of getting predictions for not labeled test data - we can average the outputs of those k models and get more stable results.

In this project we will use sklearn's StratifiedKFold which can handle imbalanced datasets easily. StratifiedKFold ensures that each fold has the same label distribution. However, we will only train one fold in the first steps to speed up training and allow the evaluation on totally unseen validation data.

Data augmentation

Another method of data preprocessing is the use of data augmentation. As we keep training the model[10] more and more, it will most likely succeed remembering the training data but will fail on data it has not seen (evaluation data and test data). The training metrics decrease but the evaluation metrics increase - this is called overfitting.

According to Andrej Karpathy, more data is the number one approach for preventing overfitting and helping a model to generalize better on unseen data. As it is difficult to collect more labeled data in this case, the second best solution is to create semi-new data out of existing data with data augmentations.[11] This is made e.g. by only selecting a random part of the image, choosing a random brightness or contrast, or flipping the image horizontally or vertically.

One important note: data augmentations should make sense! There is no benefit of adding augmentations that might not appear in real life. E.g. flipping an image in a dog breed classifier horizontally is more reasonable than flipping it vertically (upside-down). Another poor example would be some random rain added to x-ray images.

[10] At least if its capacity is high enough.

[11] http://karpathy.github.io/2019/04/25/recipe/

Implementation and Refinement

This approach is highly inspired by a blog post of Tesla's Director of AI Andrej Karpathy "A Recipe for Training Neural Networks"[12], which divides the model development process into six parts: EDA[13], Simple End-to-End Baseline, Overfitting, Regularization, Tuning and Ensembling.

Simple End-to-End Baseline

Deep convolutional models can be complex, which makes debugging difficult. The simple baseline model starts with an easy architecture without any fancy data augmentations, learning rate schedulers or k fold cross-validation. It has two convolutional layers, one pooling layer and three fully-connected layers. The simple model reaches an accuracy of about 64% which beats the easy guess "just predict class 3 for all" (61.2%) just a little bit. The training loss after five epochs is 0.94, the test loss 0.98. As the main reason for the simple model baseline is debugging, there are several tests made.

1. Input-independent baseline: set the input to all zero to check that the model extracts any information out of the input during training
2. Overfit one batch: try to reach zero training loss by training only on one batch
3. Verify decreasing training loss: increased the model capacity (bigger hidden layers) and saw that training loss goes down

Overfitting

This stage is highly iterative and has the goal to reach a training error as low as possible by overfitting the training data. We still use only 10% of the data and do not use any k fold cross-validation. The complexity of the model is increased only step-by-step, which helps to validate each experiment individually.

The first approach is a pretrained Resnet50 model with only the last (fully-connected) layer retrained on the cassava dataset with an Adam optimizer (learning rate 3e-4[14]). Both (Resnet50 and Adam) are popular choices for image classification problems. The training loss decreases to 0.7315 (compared to 0.9438 in the baseline model) and selecting a learning rate of 1e-3 improves the training process especially in the first epochs. As shown in the following figure, after three or four epochs the training error only decreases slowly and reaches 0.6752 at epoch 15. Even though it does not matter at this stage, the accuracy of the model is 70.6%.

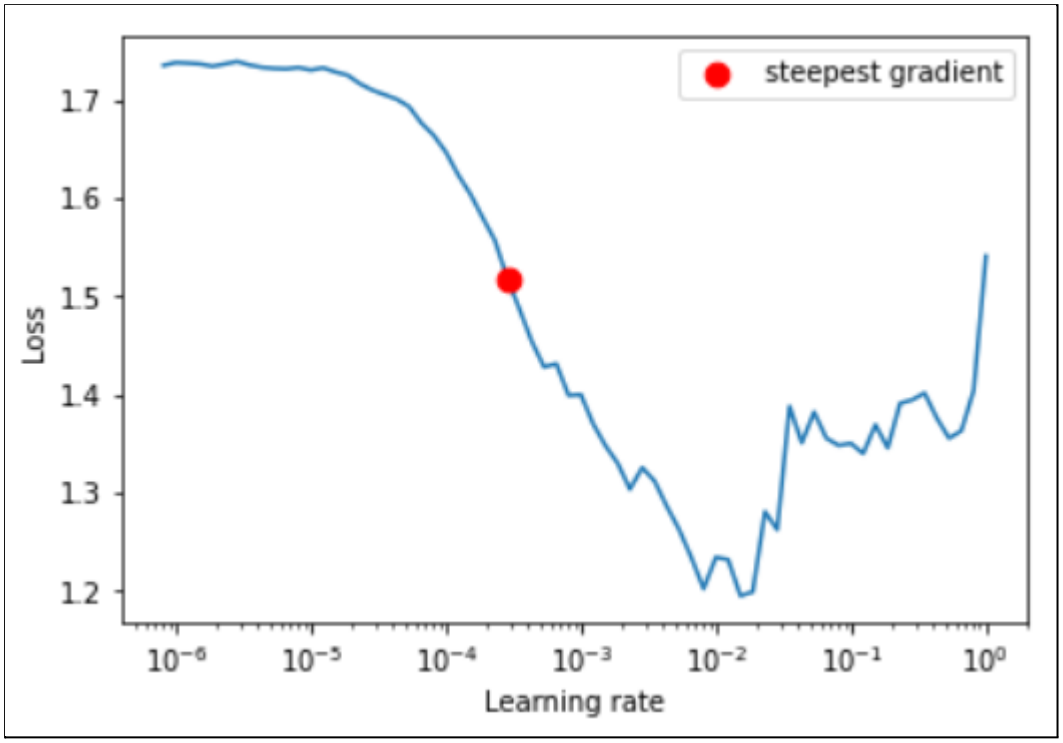
---

[12] http://karpathy.github.io/2019/04/25/recipe/
[13] EDA is explained in Chapter "II. Analysis"
[14] Andrej Karpathy made a humoristic tweet about 3e-4 being the best learning rate for (all) Adam optimizer (https://twitter.com/karpathy/status/801621764144971776).

As written in chapter “Algorithms and Techniques”, the key idea of transfer learning is only training a last fully-connected layer by keeping all parameters of the convolutional layers fixed as they were learned on ImageNet dataset. The figure below clarifies that it helps in this case to also retrain the last convolutional block of the model.[15] In contrast to the first layers the last convolutional block detects more complex patterns and, therefore, specifies more on the subject of the training data. By retraining also the last convolutional layer, we enable it to concentrate on leaf specific patterns instead of trying to identify e.g. structures like wheels or faces (on ImageNet data). This helps to reach a training loss of 0.3382 which is a big win!

[15] This convolutional block is named “layer4” in the Resnet 50 model.

The next training loops include some experiments on the input size of the images[16] and the learning rate. David Silva used the ideas of Leslie N. Smith for creating a learning rate finder for PyTorch.[17] The concept is to start the training on a very low learning rate and increase it little by little for each batch. The figure below plots the loss as a function of the learning rate. It suggests in this case to use a learning rate of 2.85e-4, because at this point the loss function has the steepest gradient.



By using an image size of 400 x 400 pixels and training with the determined “best” learning rate of 2.85e-4 the training loss reaches 0.3018.

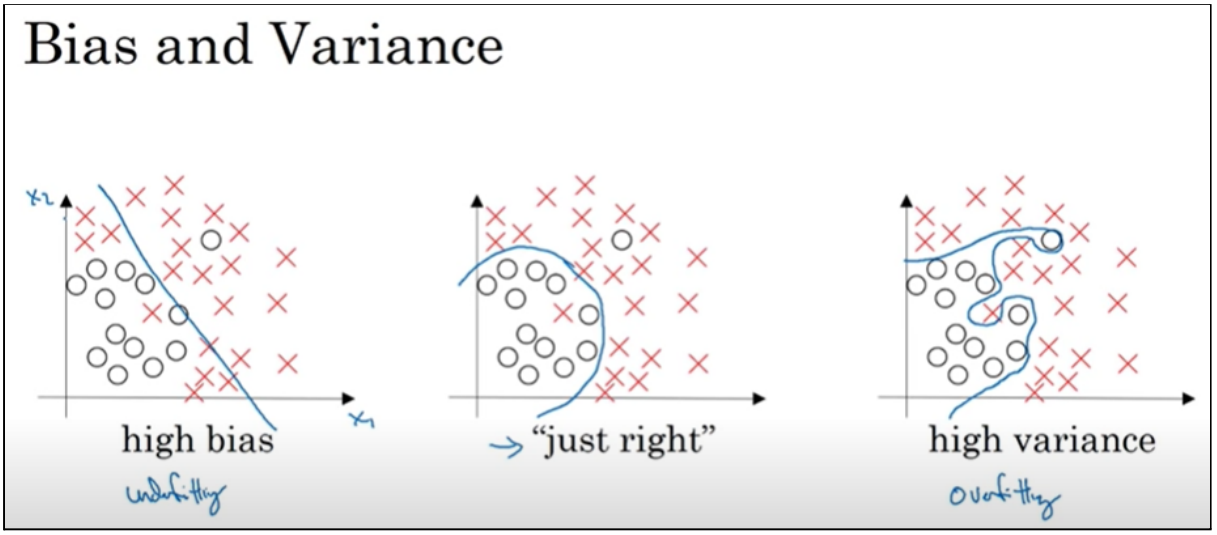
[16] This is possible because of the adaptive average pooling layer right before the fully-connected layer.
[17] https://pypi.org/project/torch-lr-finder/

As it is the goal of this stage, we clearly overfit the data, which can be seen in the following figure. While the training loss keeps improving there is no positive effect on unseen data of the validation dataset. However, the model still reaches an accuracy of about 84%, without any extra regularizations.[18]

So far we designed a model which is complex enough for fitting the training data, but fails on validation data. The figure below shows Andrew Ngs explanation of overfitting and underfitting with a simple ML problem with two variables (x1 and x2).[19] Our simple model baseline has low complexity (not deep; only a few parameters) and fails to extract more complex features out of the data. This is called underfitting or high bias. By training on a deeper and more complex model (Resnet 50) and those ideas explained above, the model at overfitting stage is able to classify in a more complex way. In the screenshot this is explained by the separation line which classifies every training sample perfectly and overfits (high variance). Why do we not

[18] There are some regularizations like dropout already used in the pretrained Resnet50 model. To be precise we could have deactivated them and tune them later in the regularization stage.
[19] Screenshot at 1:47 of the lecture video “Bias/Variance (C2W1L02)“ (https://www.youtube.com/watch?v=SjQyLhQIXSM)

want a perfectly drawn line between positives and negatives of the training data? It fails at generalizing on unseen data by overestimating the importance of outliers of the training data. The goal of the regularization step is to give up some of the training loss (i. e. simplify the “separation line” and therefore fail to classify outliers of the training data) and in that way improve the validation error by a better generalization.



Regularization

Karpathy’s top two solutions for regularization are adding more real data and creating more fake data. Most importantly we care about the ratio between the complexity of the model compared to the complexity of the data. As the prior training steps were made with only 10% of the dataset, we can easily decrease this ratio (and therefore regularize) by just using 100% of the data and thus increasing the complexity of the input data.

As written in chapter “III. Methodology”, data augmentation modifies the input images and prevents overfitting. One library which supports the data augmentation process in Python and is very popular these days in online competitions is called “data albumenations”.[20] In this project we only tried light data transforms, which seem to be accurate for this task:

- RandomResizedCrop: takes a random part of the image and resizes it to a specific image size
- Horizontal/Vertical Flip: flips the image with a given probability
- ShiftScaleRotate: shifts, scales and rotates the image in a random way

[20] https://albumentations.ai/docs/examples/pytorch_classification/

- RandomBrightnessContrast: modifies the brightness and the contrast randomly

The figures below show the effect of training on the whole dataset and adding data augmentations. While the lowest training loss increased (Overfitting stage, training all data, data augmentations: 0.3195 → 0.3627 → 0.4010) the validation loss decreased (0.6735 → 0.4057 → 0.3947) and the accuracy improved (84.20% → 86,54% → 87,41%).

Empirical studies have shown that neural networks tend to reach the optima of the loss function better when their learning rates are decreased after some epochs. This prevents it from oscillating around the optima by not getting really closer. One possible method for adapting this idea in PyTorch is to use a learning rate scheduler.[21] The simple learning rate scheduler StepLR decays the learning rate by gamma every step size epochs. We tried different options and got the best results for a step size of 7 and a gamma of 0.2. It helps improve the accuracy on validation data from 87,41% to 87,78%.

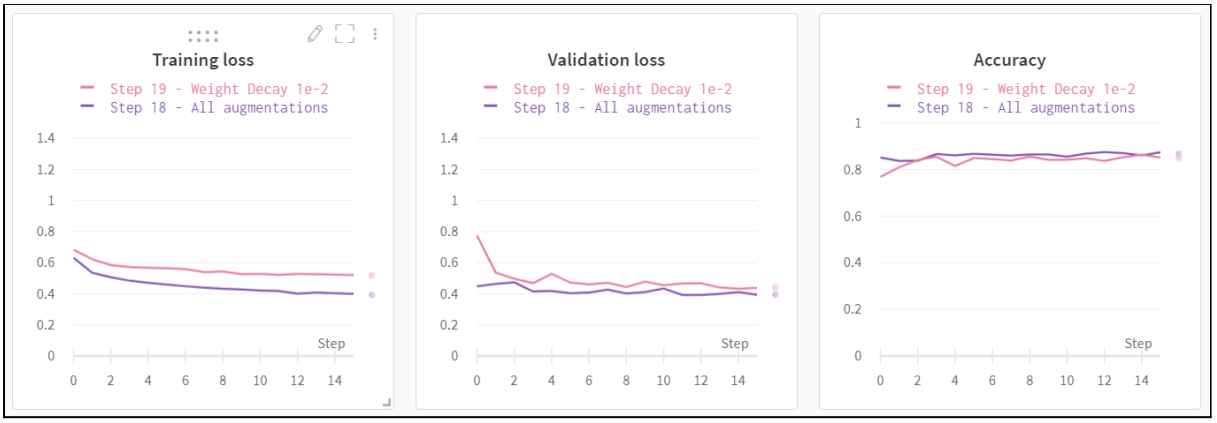
[21] https://pytorch.org/docs/stable/optim.html

The following figure shows the adjustment of the learning through the training process starting from 1e-4.

So far, all steps made can be seen somehow as a blueprint for image classification problems and do not need a lot of hyperparameter tuning. This means that their implementations are quite simple, robust and do not need much experience or/and GPU power. Jeremy Howard - the founder of fastai, a mostly high-level deep learning framework - suggests to always ask “why” before “how”, when it comes to optimizing an already good solution.[22]

Of course, there are a lot of other methods which can be tried for regularization as well overfitting. One possible method for regularizing the model is L2 regularization, which should not be explained here in detail but penalizes each weight of each layer. In this way it prevents the network to learn (= to choose) too big weights and, therefore, draws a smoother and less complex separation line - to keep the jargon of Andrew Ng’s bias/variance example above. Unfortunately, L2 regularization (or weight decay, which is the parameter that defines how much we penalize weights) does not seem to work in our project with the chosen parameter 1e-2. Even though we could try other parameters.

[22] This is his answer to a student’s question in one of his fastai course lessons. Unfortunately, the author can not find the exact lesson. (course link: https://www.fast.ai/)



Tuning

After overfitting and regularization steps we filled the toolbox with a lot of promising methods, proved their benefit on our losses and gained a model with an already quite satisfying accuracy. The next step - the tuning - tries to optimize the benefit of all those methods. There are different hyperparameter tuning processes like grid search, random search or bayesian methods that try to find the best combination of all parameters. Those methods might be GPU expensive (e.g. random search) or exceed the author's current skills (e.g. bayesian methods) and are not included in this draft version.

Ensembling

The idea of ensembling is to train different models and combine their outputs. If they are independent, they "learn" different errors and therefore "seeks the wisdom of crowds"[23] in making better generalizing predictions.

In this project we only ensemble the five Resnet50 models which were created during k fold cross-validation. Although they were trained with the same methods, those models differ due to randomness in the training process and data differences. This increased the accuracy on the test data (not validation data) from 86,75% for one Resnet50 model to 87,54% for the ensemble of all five Resnet50 models.

23

https://www.sciencedirect.com/topics/mathematics/ensemble-model#:~:text=Ensemble%20modeling%20is%20a%20process,the%20ensemble%20approach%20is%20used.

IV. Model Evaluation and Validation

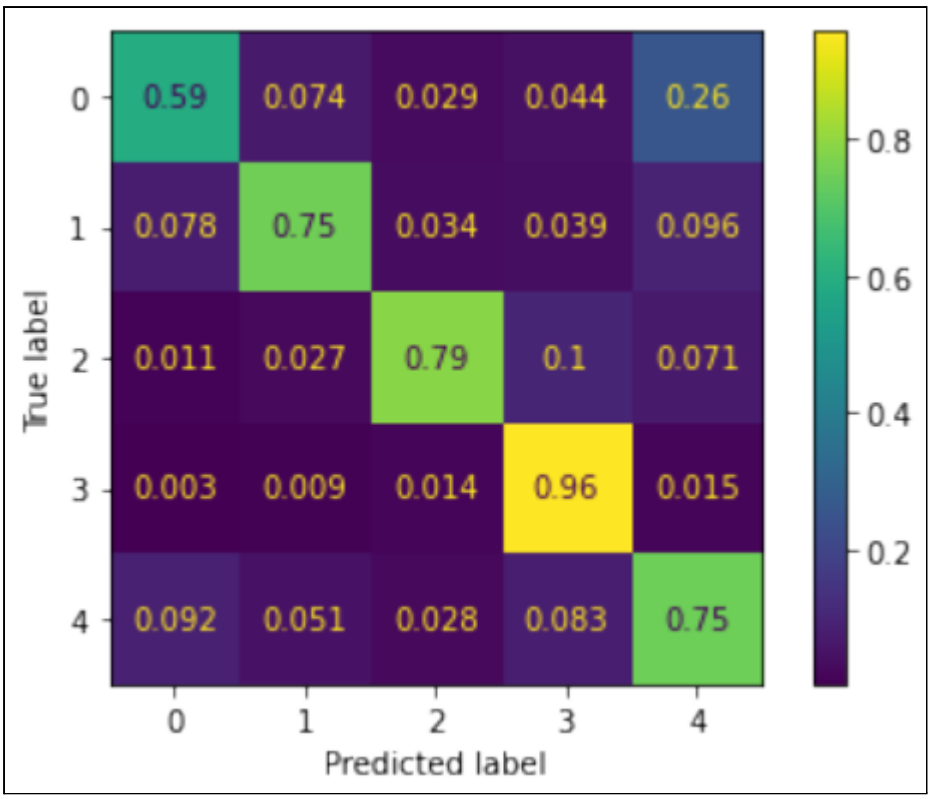
Final Result

The final result are five Resnet50 models

- created with stratified k fold cross-validation,
- simple data albumentations (RandomResizedCrop to 400x400, HorizontalFlip, VerticalFlip, ShiftScaleRotate, RandomBrightnessContrast, Normalize on ImageNet),
- trained 15 epochs on the convolutional block “layer4” and the classifier,
- with a learning rate scheduler LRStep with an initial learning rate of 1e-4, step size 7 and gamma 0.2.

Results of one model evaluated with confusion matrix

The confusion matrix below compares the predictions of one Resnet50 model on unseen, but labeled validation data to their true labels. It breaks down the model's validation accuracy into “class accuracies”. The figure below demonstrates that the model is really good in predicting class 3 (96% accuracy), but fails extremely at class 0 (59% accuracy). This can be explained with the bias problems in the data, where only 5% belongs to class 0 - compared to 61% class 3. This problem is even more problematic as 26% of the images of class 0 have been predicted as healthy and would not be removed by the farmer. The confusion matrix clarifies where the model’s problems are. Here e.g. it would not be super helpful to address any problems concerning class 3. In summary, the confusion matrix is a well-suited method for taking a deeper look in (mis-)classifications and could help improve the model a lot.



Results of the ensemble on the test data

The main goal of each Kaggle competition is to get a good ranking in the leaderboard against other users. In the cassava leaf disease competition, this ranking is based on the accuracy of a user's solution on unseen test data. Even though test and training (and validation) data should be from the same distribution, it is not guaranteed that a Kaggle competition meets this requirement. This becomes apparent when test accuracy and validation accuracy differ.

While our one-fold-model reached 87,78% accuracy on validation data, it only got 86,75% on test data. After resembling and combining all five models, our solution got more robust and achieved an accuracy of 87,54%, which results in rank 2611 of 3900.

V. Conclusion

Reflection

Although this only leads to a top 67% solution, it beats the self-defined goal of 85% accuracy. Also the gap to the first rank is only 3.78%, which is a big difference for Kaggle competitions but might be okay for real-life projects - especially if we consider the dissimilarity of the ML experiences.

It needs a lot of time and further knowledge for reducing the error rate (= 1 - accuracy) from 13% to 9% and should be well-justified before trying. In a lot of cases, the solution might already meet the customer's requirement or/and could also be improved by non-ML methods (e.g. training of farmers leads to less varying images and might help reduce error rate).

Furthermore, Kaggle competitions benefit a lot from contributions in forums. Users share results of their experiments and help each other a lot. This author of this project did not use any information on Kaggle forums and tried to design a solution without any domain-specific insights. Also, there are some Kaggle users, who publish their whole (good) solution. This leads to beginners just copy-and-paste it and climbing the leaderboard without any personal achievement.

Improvement

1. Build models which differ more and ensemble them:
   Instead of five Resnet50 models, we could start with other architectures (e.g. EfficientNet) and build models which are more independent from each other and, therefore, suit even better for ensembling.
2. Try other/further data augmentation:
   We only used classic data augmentation techniques with default (or random) parameters. We did not invest a lot of time in evaluating the augmented images and try other augmentations (and other parameters).
3. Hyperparameter tuning:
   Learning rate, architectural choices like how many fully-connected layers as classifier, learning rate scheduler and much more require some hyperparameters the engineer has to set. Although some experiments with different values were made, we did not do any extensive hyperparameter tuning. But this can squeeze out the juice. Random search picks random values for each hyperparameter, learns the model(s), loggs the results and picks new values, etc. Even though there are a lot of tools that support the hyperparameter tuning (e.g. weights&biases sweeps), Kaggle kernels and their auto-logoff prevent it from being user-friendly.
4. Not only Jeremy Howard suggests trying FP16 (half-precision floating-point format) instead of FP32 (single precision). It speeds up training, but also can regularize the network by adding noise.
5. Test time augmentations (TTA):
   The idea is to also transform the test data e.g. into five differently transformed images. The prediction for this image would be the aggregation of the predictions of the five transformed images. It might help, as it is used in a lot of image classification competitions on Kaggle.[24]

[24] https://arxiv.org/abs/2011.11156