BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

Wild Mushroom Classification Using Bioinformatics Techniques

Supervisor
Lect. dr. Mircea Ioan-Gabriel

Author
Tomșa Alexandru Eduard

2024

ABSTRACT

Mushroom foraging is a popular activity, although it involves great risks for the forager, with thousands of people being poisoned every year due to mistakes made in the mushroom identification process. Although there are tools available that can perform mushroom classification, they are not flawless, and should not be relied on. This thesis aims to improve their accuracy, but it is far from offering a complete solution, due to the very nature of the problem and the limitations of current computer vision technology. The first chapter acts as an introduction to the problem and our motivation for solving it. The second chapter expands on it by presenting the problem in greater detail and showing other scientists' attempts at solving it. The third chapter presents my own attempt at solving it, based on the research conducted by others before me. The fourth chapter presents the complete technical solution I developed, a complete application that can be used by anybody. Lastly, the final chapter shows the conclusions that can be drawn from this thesis and where mushroom identification technology could go next.

Contents

Chapter 1

Introduction

Although wild mushrooms can serve as excellent ingredients in many delicious meals, picking and identifying them represents a challenge that can pose great risks to human health, and some species are known to cause liver failure, seizures, renal failure, gastroenteritis and other unpleasant or lethal adverse effects [JHP14]. What's more, 1,675 poisoning cases were reported in France in 1998, and more recently, the ANSES has reported 1,269 cases in 2021 (of which 41 were severe and 4 resulted in deaths) and 1,923 cases in the second half of 2023 (of which 37 cases were severe and 2 resulted in deaths) [ANS23]. It is not difficult to imagine that this data applies to many other countries and that mushroom poisoning affects many thousands of people. The primary cause of these poisonings seems to be people mistaking poisonous mushrooms for edible ones, or trusting tools that gave mistaken results.

Given the advent of Artificial Intelligence and especially Computer Vision in recent years, more and more software solutions are showing up in order to solve various tasks, including those involving image classification, which mushroom identification easily fits into.

The aim of this thesis is to explore previous methods of identifying mushrooms relying on images, to see how they compare against each other on a real set of mushroom images and then, potentially, how they can be improved. The solution I ended up using involves a mix of Convolutional Neural Networks and Vision Transformers, for the purpose of best extracting information straight from the image and skipping the step of manual extraction of data from the image, which could introduce great errors. In order to better showcase this, I shall also be presenting a mobile application of my own design that uses the best solution I was able to develop, so that anybody could clearly see and evaluate my research.

This research is NOT, however, meant to replace vigilance and careful observation while foraging, and the solution presented here is NOT flawless. For reasons specific to the problem of mushroom identification, it is still impossible to get a perfect verdict based only on image recognition, and no tools that claim to be able to do this should be trusted. This being said, this research can hopefully assist in unveiling some of the mysteries surrounding fungi and lead us closer to a tool that is good enough to protect the general population from unpleasant incidents.

This thesis is organized into 4 chapters, each of them detailing a different part of the process that lead to building my solution.

The first chapter presents a glance into previous attempts at solving the problem by scholars who tackled it before, as well as an explanation for why such an overview is necessary before we start our work.

The second chapter will proceed to explain the solution I propose for tackling mushroom identification, present the articles I have taken inspiration from ([MR22a], [PXSJ23], [MR22b]) and present my own implementation of said solution, as well as various variations of it. It will then show the experiments I have performed using these models and the results I obtained.

The third chapter will explain the development of the actual mobile application, as well as the server it relies on in order to function and the way both of them are structured, exploring the architectural choices and technologies employed in order to bring it to completion.

Lastly, the fourth chapter will focus on the conclusions that can be drawn from this thesis, as well as suggestions for possible improvements that could be made in order to better the models and new potential functionalities for the application.

My original contributions include:

- FungID, a mobile application capable of determining the species of a mushroom based on an image submitted by the user.
- An implementation of the M-ViT network (possibly the first public one) using PyTorch and one variation of it that I call M-ViTv2.
- I have attempted to use the M-ViT network in order to determine a mushroom's species, not only to determine whether it is edible or not.

Generative AI Instruments

No generative AI instruments have been used in the development of this thesis.

Chapter 2

Related Work

The field of mushroom recognition in digital pictures falls under the broader field of Fine-grained Image Recognition / Fine-grained visual categorization (FGVC), [AOU+23][ZFZL19] a branch of Object Recognition which attempts to distinguish between various sub-categories of a more general category (e.g. different species of plants [ZSC+19]). Convolutional neural networks (CNNs) are among the most developed and most utilized techniques for solving FGVC problems, but they are not perfect. It is still very possible for FGVC tasks to imply details that are too subtle to be well represented by CNNs [ZFZL19]. In recent years, efforts have been made to branch away and try new methods of solving these problems, some of which will be discussed below.

However, I believe it bears mentioning that mushroom recognition as a field goes beyond the scope of Image Recognition as a whole and falls into the even larger domain of Mycology. While we, in Computer Vision, are limited to classifying mushrooms based on their morphological features (shape, dimensions, colors, consistency etc. [PR23]) extracted from pictures, mycologists abandoned this approach in the early 1990s in favour of the more precise method of molecular identification [KNS+20]. Even so, mushroom identification is far from a solved problem, as it is estimated that "only 4% of the upwards of 3.8 M estimated number of fungal species have been described". [KNS+20]

The previous paragraph, however, raises a different question. Why should we bother with mushroom recognition in digital pictures if better approaches have already been put to use? The answer is that these two approaches have different goals. Molecular identification might be the preferred approach for mycology research, ensuring a higher classification accuracy, but it is ill-suited for the general public and (in my opinion) too time-consuming and resource-consuming to be viable outside of a laboratory. Image-based classification, on the other hand, can be performed nearly instantly, and the only technology it requires is a device capable of taking pictures and either processing that picture itself, or connecting to a server capable of processing it, requirements that can be satisfied by modern mobile devices that a vast portion of the population possesses.

2.1 Previous Attempts At Solving The Problem

Now that we have established that trying to classify mushrooms based on digital pictures is a worthwhile pursuit, we ought to look at previous attempts at solving the problem to gauge exactly where we are starting from.

Over the years, a great number of technologies have been employed for this purpose, some of which not even relying on modern Object Recognition techniques. Such techniques that will be explored in the following sections include: Convolutional Neural Networks, Vision Transformers, K-Nearest Neighbor classification, attention mechanisms and other such variations and combinations of the aforementioned methods.

2.1.1 The K-Nearest Neighbor Approach

The authors of [RAP+18], for instance, decided to use a mix of K-Nearest Neighbor classification, Canny Edge Detection and feature extraction using the Zoning method, obtaining an accuracy of 90%, a reasonable result, were it not for the dataset consisting of 40 images of 8 different species of mushrooms. Moreover, as seen in Figure 2.1, the feature extraction seems to be performed on a black and white version of the image, which loses much of the visual information that can be used for classification.

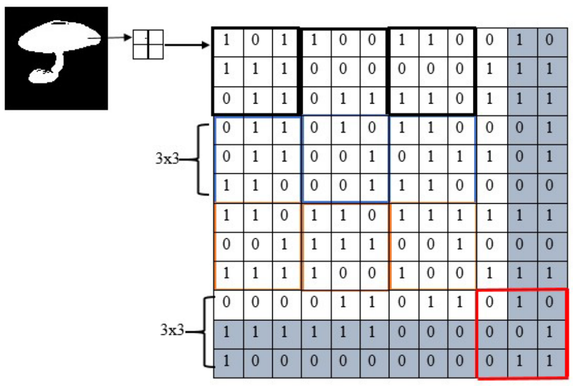


Figure 2.1: Feature extraction using the Zoning method [RAP+18]

2.1.2 The Attention CNN & Genetic Information Approach

There have also been attempts to solve the problem using more modern Object Recognition techniques, namely CNNs and Vision Transformers (ViTs). Thus, the authors of [LXZ$^+$22] propose MushroomNet, a model based on the traditional CNN technique with the addition of various attention mechanisms (used to compensate for the interference of the environment and the large variety of mushrooms, which lower the performance of regular CNNs). MushroomNet has a low number of parameters, and the fact that it is based on a lightweight CNN (MobileNet-V3) make it suitable for mobile devices and a good choice for field researchers. Even more, the researchers employed the use of genetic distance as the representation space, and the genetic distance between each pair of mushroom species was used as the embedding of said representation space. A CNN is used to extract and map the image's features to the feature space of a fully connected layer. Finally, the model was tested on two datasets, one local and one public. On the local dataset, the model performed well, with 73.8% accuracy, coming in behind EfficientNet_B0, with 75.4% accuracy and DenseNet121, with 76.4% accuracy, while also being more suited for mobile classification tasks. On the public dataset, however, the model got the top performance on both the validation and test datasets, with 82.0% and 83.9% accuracy respectively, implying that the model has strong generalization ability [LXZ$^+$22]. It should be mentioned, however, that the dataset the model has been trained on didn't contain mushroom species, as it was mentioned, but rather mushroom genera (18 of them), so it is unknown how well the model can tell actual species apart.

2.1.3 The ViT-Mushroom Method

The ViT-Mushroom network [Wan22] is one of the first attempts at mushroom classification making use of Vision Transformers. Thus, this network uses ViT-L/32 as its backbone model and attempts to classify 9528 images of mushrooms into 11 species (though as with the previous article, these are actually mushroom genera). The structure of the model can be seen in Figure 2.2. To get a better performance, the model has been pretrained on the ImageNet-21k dataset, so that it may learn relevant but generic information about object classification that can then be repurposed for mushroom classification (in a process known as transfer learning). This has been done in order to overcome the problem of insufficient labeled data.

In practice, the model competed with several state-of-the-art image classification models [Wan22], namely ResNet-34, VGG-16, Inception-V3, Inception-ResNet-V2 and Xception, all of which were similarly fine-tuned according to transfer learning principles on the ImageNet dataset. The results revealed that the ViT-Mushroom model was better than all the others on this task when judged by any metric, hav-

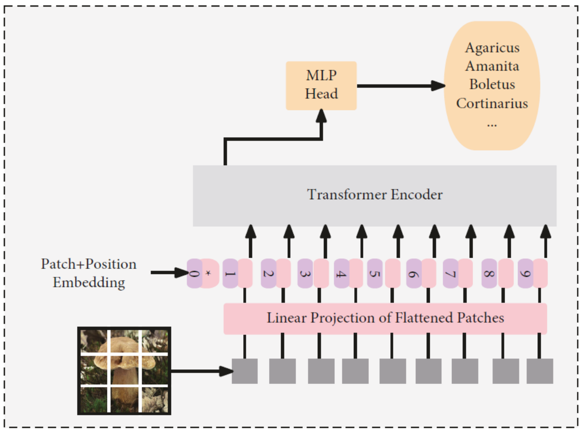


Figure 2.2: The ViT-Mushroom Pipeline [Wan22]

ing an accuracy of 95.97%, around 3% higher than that of Xception, the second best model, which obtained 92.95%. This pattern seems to also hold for the other 3 metrics (Precision, Sensitivity and F1-Score) both as Weighted Averages and as Macro Averages. The other networks seem to hardly be able to pass 85% by any metric, with VGG-16 performing the worst by far (81.31% accuracy), likely due to its simplistic structure and lack of modern elements (like an attention mechanism or a residual network).

Moreover, this model is validated with the use of t-SNE [Wan22], a method of visualizing high-dimensional data by scaling it down. Its goal is to essentially preserve the distribution of clusters when scaling down the data. Making scatter plots of this scaled-down data showed the clear separation between classes determined by the ViT-Mushroom model.

These results reveal the potential in the ViT architecture, which will continue to be leveraged by future proposed solutions.

2.1.4 The Bilinear CNNs Method

The authors of [WLX+21] came up with another network, being the first (to my knowledge) to try a design based on Bilinear Convolutional Neural Networks (B-CNN), which has been improved with the addition of an attention mechanism. The way it works is by using two CNNs (called A and B) which are used to learn two features at each position of the image, multiply the outputs (using an outer product) and then pass it through a classification layer. A conceptual diagram of this can be seen in Figure 2.3.

CNN A is used to locate the features in the input image, while CNN B extracts the features from the feature regions discovered by A. This ensures that the local detection and feature extraction tasks can be finalized successfully.

As for the attention mechanism, this network's creators opted for Mixed at-

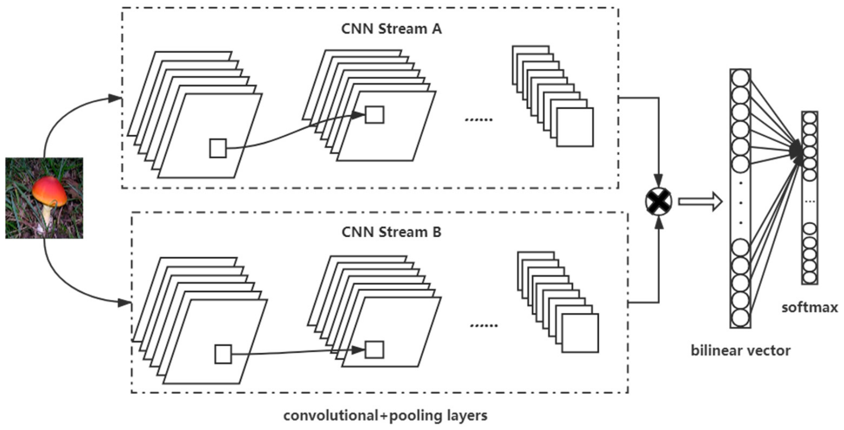


Figure 2.3: Bilinear CNN - Conceptual Diagram [WLX+21]

tention [WLX+21] (which combines multiple attention mechanisms). Specifically, they chose to use the Convolutional Attention Mechanism Module (CBAM) which is essentially the channel attention module (CAM) connected to the spatial attention module (SAM). This attention mechanism is simply applied to each of the two CNNs' output before the outer product is performed.

Finally, the network EfficientNet-B4 (ENB4) was chosen as CNN after comparing the network depth, width and input image resolution of the entire EfficientNet series (B0 through B7), due to having "fewer model parameters, but higher accuracy" [WLX+21].

In practice, this model was tested on a dataset consisting of 3219 images of mushrooms in one of 7 species in the Amanita genus. The model was tested against models such as the normal EfficientNet-B4, ResNet-50, VGG-16, as well as various ways of making bilinear models using these (e.g. VGG-16 + VGG-16, VGG-16 + ResNet-50), with or without the CBAM module.

The results show that in terms of model size, all variations of ENB4 have both vastly fewer parameters and vastly smaller sizes than the other networks [WLX+21]. While the bilinear ENB4 variants did have an increase in model size (130 MB instead of 82 MB), they also have almost 2 million less parameters than the simple ENB4 network (19.6M/19.7M instead of 21.4M). It should be noted that the next smallest network, VGG-16, has more than 3 times the number of parameters and twice the size (268 MB).

In terms of classification metrics, the proposed method comes out on top in all categories [WLX+21], with an accuracy of 95.2%, a precision of 94.5% and a recall of 94.6%, closely followed by the same model without the CBAM module and the simple ENB4 model with CBAM added (a less than 2% difference in all metrics). The VGG-16 model (the next smallest one) achieved the worst results overall (89.66% accuracy, 90.03% precision and 88.01% recall), making the proposed model's performance that much more impressive.

2.1.5 The Deep Sparse Dictionary Learning Method

Despite CNNs having proved themselves capable on image recognition problems, the feature maps they generate tend to have a large amount of redundancy and their means of information processing is not efficient. In order to combat this, we have a new mushroom classification model from the authors of [Guo24], using an improved method of dictionary learning to get a sparse representation of the original data that is relevant for our purposes.

If you use an effective data dictionary, it is possible to compress the redundant information to the point where you are only left with information that has a certain value [Guo24]. It is possible to combine a deep neural network with dictionary learning so that the deep underlying representation of the original image becomes an input to the dictionary learning model, allowing for better feature descriptions.

This paper proposes the Deep sparse dictionary learning (DSDL) [Guo24] method for mushroom classification, meant to solve the spatial redundancy problem in traditional CNNs, as well as to avoid the overfitting that came with a larger number of network parameters (and especially when using a smaller dataset). A stack autoencoder is used to extract "relatively deeper features", then the model recreates the deep features using a structured dictionary based on different categories of discriminant features.

The classic way of doing sparse representation classification requires using the original training sample as a dictionary and then approximating the sparse representation matrix to the ideal sparse representation, leaving only the sparse coefficient to be determined in a simple process [Guo24]. The trouble here is that the representation coefficient can't reach the ideal sparse structure due to the simple structure of the dictionary, which becomes a real issue when trying to classify complex images. In order to better this process and obtain better results, some methods of dictionary learning have been suggested that not only learn the sparse representation coefficients, but also the dictionaries themselves.

The true innovation of this paper is the use of the stack autoencoder (AE), an architecture for unsupervised training made by stacking autoencoders, each of which are made up of an encoder (which converts input data into hidden layer feature representations) and a decoder (which maps the hidden layer representation back to the original data) [Guo24]. The stacked autoencoder (SAE) essentially forms a deep network where each AE's output acts as input for the next. The way the SAE is trained essentially comes down to training every layer individually, in turn, while the values of the other layers stay fixed, which doesn't lead to optimal parameters

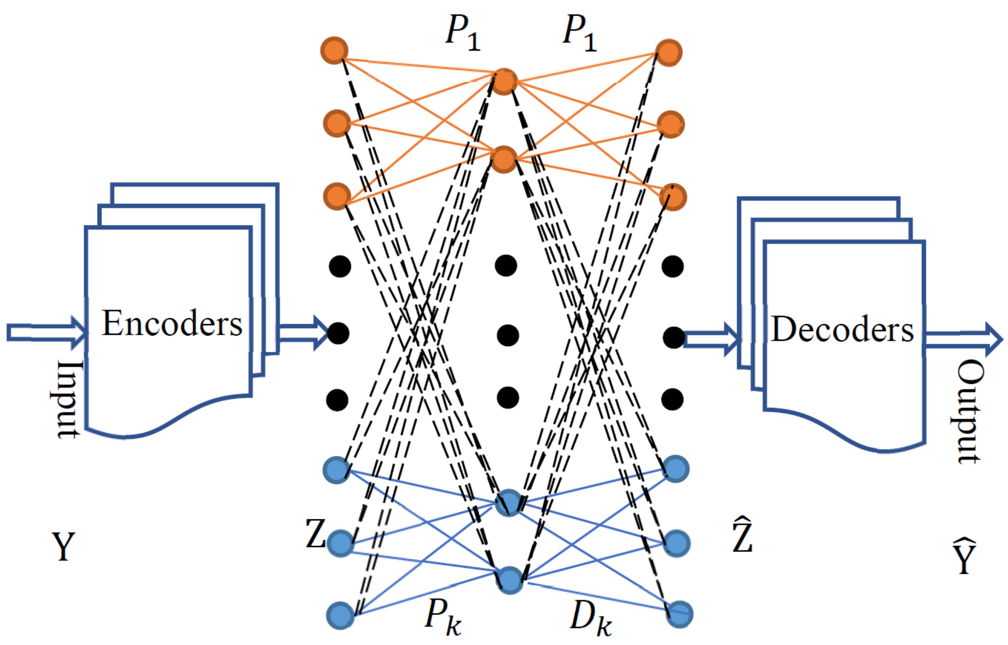


Figure 2.4: The structure of the DSDL model[1][Guo24]

for the entire network. Thus, after this training step, all network parameters need to be adjusted using the backpropagation algorithm. It should be noted that training each individual layer is mandatory, as using backpropagation on the initial randomly initialized weights will only make them converge to the local optimal, which isn't the case if they are individually brought to a state of near-convergence first.

Finally, the actual structure of the DSDL model consists of 2 parts (which can be visualized in Figure 2.4):

1. The stack autoencoder, accepting an input image Y and converting it into a depth feature Z (through the encoders) and accepting a feature Ẑ and outputting a reconstruction of it, Ŷ (through the decoders).

2. Discriminating dictionary pair learning (placed between the encoders and decoders of the SAE). The first dictionary, P, gets a low-dimensional representation of the input, while the second dictionary, D, trains the construction of the image.

In practice, the model has been tried on a dataset composed of 9533 images of mushrooms belonging to 12 different species [Guo24]. The dataset reportedly is quite unbalanced, the number of images for each category ranging from 400 all the way to 1563. It also has inconsistent image resolutions, though this has been alleviated by preprocessing, bringing all images to the same resolution (300 x 384) and

[1]I find it quite likely that the P_1 on the right side of the diagram was supposed to be a D_1 to match the one below it.

then applying image augmentation techniques (e.g.: random cropping, color distortion, image rotation).

The model has been run in parallel with other models rooted in dictionary learning [Guo24]. These include SRC, BSRC, CVX, LC-KSVD, SVM-DDPL and SLatDPL (the full names and functionality of these models or the ones not mentioned are beyond the scope of the paper and can be found in the bibliography of the original article). The results show the DLSL model having a slight edge over the others, with an accuracy of 97.64%, which is 0.56% higher than that of SVM-DDPL and 0.69% more than that of SLatDPL. This demonstrates the effectiveness of using the autoencoder for the task of deep dictionary learning. What's more, the model performed better than traditional dictionary learning algorithms (accuracy being between 0.81% - 10.92% higher, depending on the model), further solidifying the model and the features it extracts as suitable for the task of mushroom classification.

2.1.6 The R-CNN method

The authors of [KBK22] proposed a new mushroom classification and detection network based on Region Convolutional Neural Networks (R-CNNs). The idea behind R-CNNs is that while regular CNNs can identify objects in images, they cannot actually detect the object's location inside the image or draw its bounding box. R-CNNs get around this problem while also being relatively simple and easy to modify.

The way R-CNNs work is by extracting many (normally 2000) *region proposals* (of different shapes and sizes) from the input image, associating each with a ground-truth class label and a bounding box [ZLLS23]. Each region proposal then has its features extracted by a pretrained CNN which has been truncated before the output layer. Using the extracted features and the class of each region, multiple support vector machines (SVMs) need to be trained to classify objects, each SVM corresponding to one specific class. Likewise, the extracted features and labeled bounding box of each region proposal will be used to train a linear regression model in order to predict the ground-truth bounding box. This architecture can be visualized in Figure 2.5

Despite the use of a pretrained CNN, this method is slow, as it requires thousands of CNN forward propagations, making it infeasible to use R-CNNs widely.

The model chosen as CNN in this article is a modified version of the AlexNet network [KBK22]. In order to reduce training time, it was built using transfer learning on the pretrained AlexNet (trained on the ImageNet dataset with over a million images and 1000 categories), though it also removed the fourth and fifth convolution layers from the original model and added the GoogLeNet inception module. The final model can be seen in Figure 2.6.

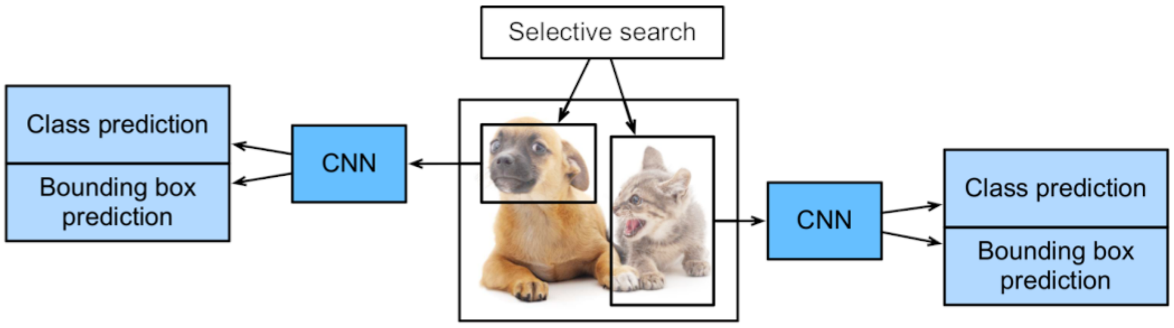


Figure 2.5: The R-CNN architecture [KBK22]

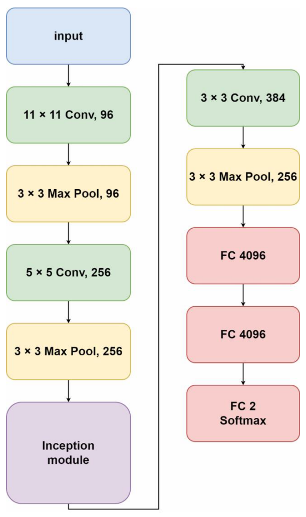


Figure 2.6: The AlexNet-based model [KBK22]

The dataset used started out with 623 mushroom images (all with a size of 227 $\times$ 227 $\times$ 3) belonging to 5 species, divided into 2 parts, namely poisonous mushrooms and edible mushrooms [KBK22]. However, this set was deemed insufficient to train a model without risking overfitting, and so the authors used data augmentation techniques to increase the size of the set to 2000 images. During the training, these images were divided into 10 equal groups and used according to the K-fold cross-validation method (divided into training and test sets using a ratio of 90:10).

Two experiments were run in the making of the cited paper ([KBK22]). Both experiments involved testing the model against the base AlexNet model, as well as ResNet-50 and GoogLeNet. The first experiment did the comparison with the models as regular CNNs (a classification task), while the second involved testing the models as part of an R-CNN (a detection task).

For the first experiment, all models achieved a high and stable training accuracy, as well as test accuracy: ResNet-50 got 99.50% in 40 iterations (5 min 50s training time), GoogLeNet got 99.50% in 125 iterations (2 min 20s training time), AlexNet got 99.00% in 160 iterations (1 min 42s training time), and the proposed model got 98.50% in 340 iterations (1 min 10s training time).

For the second experiment: ResNet-50 got 96.50% in 13 min 44s training time, GoogLeNet got 96.50% in 7 min 28s training time, the proposed model got 95.50% 4 min 37s training time, and AlexNet got 95.00% 5 min 07s training time.

Looking at the results, we can tell that the proposed model doesn't achieve the best accuracy, but the performance sacrifice when compared to the best model is relatively small (1% and 1.5% accuracy on the classification and detection tasks respectively) compared to the training time gained (a decrease in training time of roughly 5× on the classification task and 3× on the detection task). It remains to be seen how this model would scale on bigger and more diverse datasets, though if a short training time is ever a factor, then this model might be a good candidate.

2.1.7 The Danish Citizen-Science Project

One of the most frequent problems that plague computer vision projects is the lack of high quality training data, and models trying to solve the problem of mushroom classification are no exception. Even in previously reviewed papers, it can be seen how some of the datasets used were not ideal to test the proposed models. That's why the paper [PŠM+22] is impressive, as it not only provides a new way of addressing mushroom classification (not only by using image data, but also metadata), but it also demonstrates another way of gathering data: citizen-science projects. In this case, the Atlas of Danish Fungi, a project supported by over 4000 volunteers which contains approximately 1 million high quality observations of fungi.

Naturally, the source of data being random volunteers does raise the question of its credibility, but the atlas assures us that each observation is recorded together with a reliability score based on the rarity of the reported species, its geographical distribution, its seasonality as well as the reporting user's previous proposal precision. What's more, other users can either agree with a proposed classification or suggest an alternative. The submission score (1-100) must reach at least 80 before the identification is approved by the community. Even then, the entire process is observed by a small group of experts which can overrule any decision made by the community. This is what guarantees the quality of the data.

The First Generation of FungiVision

The first proposed method for mushroom classification is FungiVision [PŠM+22], the model that got the best results during the FGVCx Fungi'18 recognition challenge, and was later applied on the Atlas of Danish Fungi. This proposed approach is an ensemble of 6 CNN architectures based on either Inception-v4 or Inception-ResNet-v2, pretrained on either ImageNet-1k or LifeCLEF 2018. These are first tested without any other modifications, using one feed forward pass per image (central crop, 80%). Of the tested models, the one that will be considered for further experiments is the one based on Inception-v4 pretrained on ImageNet-1k, as that's the one that got the the best validation accuracy (48.8%).

Another experiment involved considering 14 different image augmentations applied at test time [PŠM+22], namely the original image, an 80% central crop, a 60% central crop + 4 corner crops and mirrored versions of the mentioned augmentations. All of these were then resized to square inputs using bilinear interpolation and used to test both the best model from the first experiment and the entire ensemble, using either 1 or 14 crops, and the predictions using multiple crops or multiple models being combined either by averaging (sum) or by choosing the most common prediction (mode). To make matters more complex, the authors decided that their model should be robust to different species distributions, based on seasonality, location and altitude, for example, and so they have decided to adjust the predictions of the CNNs to new class priors, assuming that the uniform distribution is given, as it was the case during the FGVCx competition (more details can be seen in the original paper).

To sum up the results, the adjustment to new class priors improved the accuracy for all models by at least 3.6% [PŠM+22] (in the case of the best model from the first experiment). In the case of the best performing model (the entire ensemble + 14 crops combined with sum), the accuracy increased from 54.2% to 60.3%. However, the contest's dataset was divided into a public section (containing about 70% of the data) and a private section (saved for the final evaluation to avoid bias towards the test images' performance), and the model that had the best preliminary performance on the public set was the entire ensemble with 14 crops, predictions adjusted to new class priors and the accumulation of predictions done by the mode. The submitted model outperformed all others in Top3 accuracy for both the public (79.23% accuracy) and private (78.80% accuracy) leaderboards.

Going further, the experts behind the Atlas of Danish Fungi were impressed with the model [PŠM+22], to the point where it has been added to the system. An evaluation conducted on the DanishFungi 2021 dataset (based on data from the atlas) reveals that of all mushroom samples that have been submitted for automatic recognition, only 7.18% haven't been approved by the community or an expert, a much better performance than most non-expert users in the system. Even more, 69.28% of the approved species were the top accuracy result given by the system, while 12.28% matched the second result and 10.54% were based on the top 3-5 suggestions. It could be said that the model had a Top1 accuracy of 69.28% on its own and 92.82% together with the community. The model cannot be considered to be fully reliable, but it does serve as a valuable learning tool for non-experts.

State-of-the-art NN Classifiers

In an attempt to upgrade the previously discussed model, the authors explored new CNN architectures, settling on SE-ResNeXt-101-32x4d, EfficientNet-B3 and EfficientNetV2-L and even some Vision Transformer architectures, namely ViT-Base/16 and ViT-Large/16. All of the models were initialized from publicly available ImageNet-1k checkpoints and were fine-tuned starting from there using images that were augmented using random flips, random resized crops and random contrast and brightness changes (more details in the paper [PŠM+22]).

Viewing the results of running the CNNs on the DF20 dataset (sourced from Atlas of Danish Fungi) with a resolution of 384 × 384, we see that SE-ResNeXt-101 got an accuracy of 78.72% [PŠM+22], while EfficientNetV2-L achieved the lower accuracy of 77.83%. On a resolution of 224 × 224, however, the roles are reversed, EfficientNetV2-L overperforming the other model by 1.22%.

When comparing these results with those of the ViT models, we see that on a resolution of 384 × 384, the best transformer (ViT-Large/16 with 80.45% accuracy) outperforms the best CNN by 1.73%. Interestingly enough, neither CNN nor ViT architectures seemed to take a massive lead on the 224 × 224 resolution test case, and seem evenly matched.

Adding Metadata To The Models

The last great piece of technical innovation presented in the article ([PŠM+22]) is the use of metadata in the classification task. Metadata refers to information outside of the image itself, such as the date of observation, the vegetation type, the substrate and habitat etc. and its use here is inspired by this same common practice in mycology. The way this works is by making the assumption that the visual appearance of a species does not depend on the metadata. While this is not always true, this approach allows us not to have to model the dependence of visual appearance and metadata. It results in a simplistic model, yet even so it reduces error rates (the actual implementation of this mechanism goes beyond the scope of this paper).

Applying this mechanism on top of the previously discussed ViT models lead to a significant increase in both Top1 and Top3 accuracy [PŠM+22]. The three pieces of metadata considered were the habitat, the month of observation and the substrate, of which the habitat proved to be the most relevant. However, using all three options simultaneously resulted in the performance of the ViT-Large/16 model on the DF20 dataset to increase by 2.95% in Top 1 and 1.92% in Top3, while ViT-Base/16 gained 3.81% in Top1 and 2.84% in Top3.

Impact on Fungi Recognition Moving Forward

Ultimately, however, I consider the biggest takeaway from this article not to be the technical results of the models, but rather the societal impact that these models had. Ever since the integration of the FungiVision into the Atlas of Danish Fungi, the rate at which people could learn about mushroom classification has greatly increased [PŠM+22], which in turn leads more people to contribute to the project, creating more high quality training data for the models, resulting in ”a virtuous cycle that helps both communities”. It ultimately presents proof that there is a demand for solutions to the problem of mushroom classification, and that computer vision models such as these can meet that demand, helping others while helping themselves.

2.1.8 The Entropy-guided Method

In the mushroom classification community, there are a number of competitions where AI models are tested against each other to see which ones can more accurately use the data they are given and operate within the assigned requirements in order to get the best mushroom classification. One such competition is the FungiCLEF [PŠCM], whose 2023 edition gave rise to the model we will be discussing here.

An Overview of The Competition

While automatic species recognition generally requires efficient prediction on limited resources [PŠCM], in practice, identification doesn’t solely depend on visual data of the specimen, but also on other data available to the observer. This challenge aims at providing a benchmark for combining visual information with other types of observations by providing data with rich metadata and precise annotations to all the participants. To achieve this, the training dataset used is the DanishFungi 2020 (DF20), containing 295,938 images (from 177.170 observations) representing 1,604 species. The validation and test datasets, on the other hand, were assembled from all observations submitted to the Atlas of Danish Fungi in 2021, using only data that had expert-validated labels, spanning all the types of substrate and habitat. Both of these are quite similar in structure, having similar numbers of observations ($\sim$30,131), images ($\sim$60,000) and species ($\sim$2,700), some known ($\sim$1,084) and some unknown ($\sim$1,600).

In contrast to previous editions of this challenge, the 2023 edition was created with the desire to create models could feasibly run on a mobile device [PŠM+22], which is why a new restriction was set in place, all models had to be under 1 GB. As a further complication, the contest essentially poses an open-set recognition problem, as the submitted models are expected to be able to tell the difference between known species and never-before-seen species. To this end, several scenarios were proposed, each with a different cost function for the decisions made by the model, in order to promote different characteristics of the model.

The different scenarios (or tracks, as they will be called going forward) are as follows:

1. **"Standard classification with an 'unknown' class"** [PŠM+22]: Essentially, a normal classification problem where all mushrooms of unknown species should be classified as unknown. All misclassified specimens are penalised the same.

2. **"Cost for confusing edible species for poisonous and vice versa"** [PŠM+22]: The cost function is defined so that edible species mistaken for poisonous species are assigned one cost (c_{ESC}) and poisonous species mistaken for edible ones are assigned another cost (c_{PSC}). For the benchmark, they are set like so: $c_{ESC} = 1$ and $c_{PSC} = 100$, as eating a poisonous mushroom is much riskier than ignoring a potentially safe mushroom.

3. **"A user-focused loss composed of both the classification error and the poisonous/edible confusion"** [PŠM+22]: Useful for when you are interested in both determining whether a mushroom is poisonous or not and species classification, it simply adds up the previous 2 cost functions.

4. **"Cost for missing 'unknown' species is higher; misclassifying for 'unknown' is cheaper than confusing species"** [PŠM+22]: Useful especially when collecting data for biological research. Missing the observation of a new species comes at a higher cost ($\alpha > 1$) and labeling a known species as "unknown" comes at a lower cost ($0 < \beta < 1$) than misclassifying it for another species. For the benchmark, the values used are $\alpha = 10$ and $\beta = 0.1$.

The Top Performing Model

The model that performed the best on both the private and the public test datasets is the one introduced in the meng18 [RJL+] paper, using entropy to solve the open-set recognition problem. This model works by using Metaformer, a hybrid architecture that combines convolutional blocks (in the first three stages) with Transformer blocks (in the last two stages) to fuse metadata and image data using relative attention. The final model is an ensemble of MetaFormer-0 and MetaFormer-2 (this was permitted as it didn't go over 1 GB in model size). The complete architecture can be visualized in Figure 2.7.

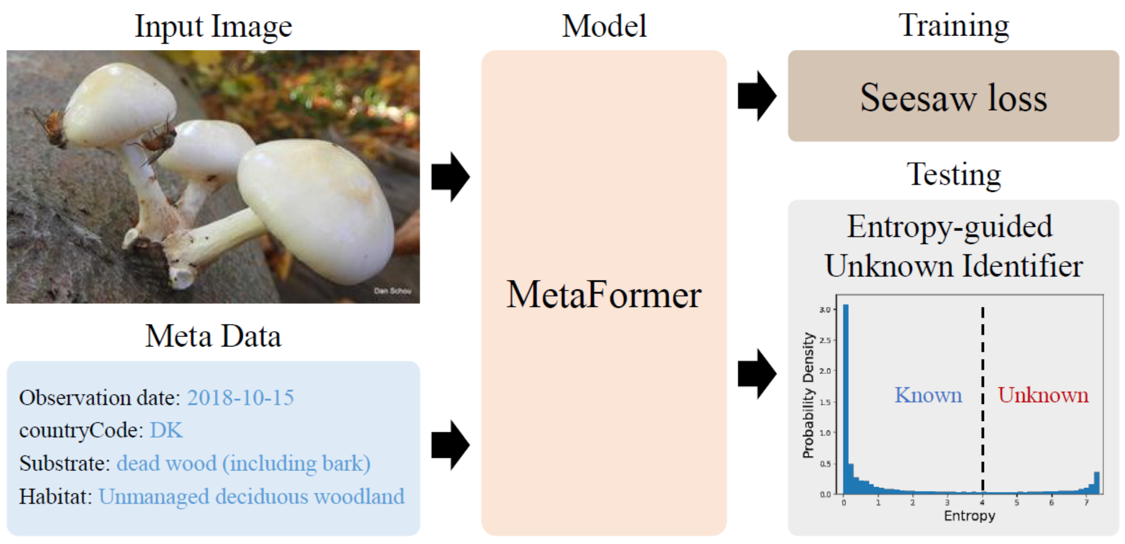

Input Image
Model
Training
Seesaw loss
Testing
Entropy-guided Unknown Identifier
Meta Data
Observation date: 2018-10-15
countryCode: DK
Substrate: dead wood (including bark)
Habitat: Unmanaged deciduous woodland
MetaFormer
Known
Unknown
Probability Density
Entropy

Figure 2.7: The Metaformer-based model [RJL+]

Dataset Imbalance

When observing the number of entries for each class in the dataset, it can be seen that it follows a long-tailed distribution [RJL+]. The most populated class has 1,913 images and the least populated one only has 31 images. This would lead the training process to be dominated by the head categories (with more images) and lead to serious misclassification for the tail categories (with less images). This can be combated by the use of seesaw loss, a modified variant of cross-entropy loss that can ”dynamically decrease the gradients of negative samples for each category”.

Data Augmentation

The sequence of operations performed on the data during training in order to augment it is pretty common [RJL+], including, in order, a random cropping on the image (between 8% and 100%), a resize using bicubic interpolation and a horizontal flip (with a probability of 50%). Then a RandAugment is used with the setting of the Swin Transformer, which tweaks image options such as brightness, contrast and sharpness, but also applies geometric transformations like shear, rotation and translation. Lastly, random erasing is used to mask out a region of the image 25% of the time. During testing, the augmentation consists of multi scale & ten crop.

Entropy-guided Unknown Identifier

Because we are trying to solve an open-set problem, the model has to be able to separate the known species of mushrooms from the unknown [RJL+]. The literature offers several ways of doing this, mostly involving a threshold τ, either for the final probability distribution of the classes (Maximum Softmax Probability), or for the final logits (Maximum Logit Score) that determines the level of uncertainty in the model. However, on this dataset, it has been noticed that there is no clear threshold that could separate the predictions with a high probability from those with a low probability. The solution to this is to use entropy as a metric instead of probabilities or logits, which gives us a much clearer cutoff to use as τ. The model will have a higher confidence for known mushroom species, which will mean a lower entropy. Consequently, an unknown species will translate to a higher uncertainty and so, a higher entropy.

Enhanced identification of poisonous species

With the dangers associated to eating poisonous mushrooms as high as they are, the authors of the model sought to decrease the risk of them slipping undetected. Thus, they came up with an additional loss function, called poisonous classification loss, so that should a poisonous mushroom be classified as edible, the corresponding cost should increase. This loss is then used as extra supervision for the model.

Results

The results show 5 metrics used in the competition, the F1-Score, and the losses of the four tracks previously mentioned [RJL+]. On the public dataset, this model achieved an F1-Score of 58.95, while the losses were 0.2072 on Track 1, 0.1742 on Track 2, 0.3814 on Track 3 and 1.4762 on Track 4, while on the private dataset, it achieved an F1-Score of 58.38 and losses of 0.2409 on Track 1, 0.1269 on Track 2, 0.3702 on Track 3 and 1.771 on Track 4.

The model significantly outperformed the second place [RJL+]. For instance, on the public dataset, this model's F1-Score was 2.68 higher than the next best one, indicating that it is the best at solving the classification problem overall. The model also proved itself to be better at identifying unknown species, as it has nearly half the loss of the next best model on Track 1, as well as Track 4 (the ones focusing on the unknown species). Finally, the lower cost obtained on Track 2 (the next best one being 0.2133) suggests that it can better discriminate between poisonous and edible mushroom species.

Ultimately, these results, together with the similar results obtained on the test dataset are what gave this architecture the edge and the first place in the competition.

Chapter 3

Proposed Approach

Looking at the papers summarized in Chapter 2, one can notice certain trends. For instance, it can be noticed that some of the models with the best performance are those that combine different architectures, like the models proposing hybrid CNN-ViT models, or even pairing CNNs together or pairing a CNN with a different mechanism. This makes it clear that the problem of mushroom classification cannot be solved by the simplest of models. This can be used to inform our choice of model moving forward.

In this chapter I will introduce the model M-ViT [PXSJ23], which my further experiments are based on, explain my custom implementation of the model (altered to perform species-level classification, possibly the first public one, as my searches have not revealed any others) using PyTorch, present my Flask API that will allow third parties to integrate the model into their application by simply submitting an image for classification without any hassle and then showcase my experiments with the model in order to see if I can improve its performance in any way.

3.1 Model Overview

The base model used in this paper is the M-ViT network, introduced by Peng et. al [PXSJ23] as a successor to the already existing MobileViT network [MR22a]. It should be noted that despite the very similar name, this network is different from the MViT (Multiscale Vision Transformer) network introduced in [FXM^{+}21]. M-ViT was created specifically to be a competent and mobile-friendly model for mushroom classification. This section is dedicated to explaining how it functions at its core, and to do that, it will first have to explain how MobileViT functions, as well.

3.1.1 The MobileViT Network

MobileViT is a hybrid model, meaning it combines Convolutional Neural Networks and Vision Transformers into one model [MR22a]. This idea is not a novel one and has been attempted before (e.g: [CDC+22], [XSM+21]), but to understand why this is needed, we must first understand each of the strengths and weaknesses of the two methods separately, and how we might combine the two.

CNNs - Pros and Cons

CNNs are networks that have the standard convolutional layer at their core. Due to this layer being computationally expensive, some methods have been proposed that aim to make it more ”light-weight and mobile-friendly” [MR22a]. Light-weight CNNs are networks that make use of some of these methods, and as a result they are more easy to train and versatile, even being able to replace heavy-weight backbones in task-specific models, which helps to reduce latency and model size. Beneficial as they are, CNNs have one major disadvantage, which is that they are spatially local, being completely unable to learn global information, as ViT-based networks do.

ViTs - Pros and Cons

The way Vision Transformers work is by dividing an image into a sequence of non-overlapping patches and then learning inter-patch representations using multi-headed self-attention [MR22a]. They have a tendency to require a greater number of parameters in order to increase performance, which comes at the cost of model size and latency, making them ill-suited for use on resource-constrained devices. Even if you reduced the size of the model to match the resource constraints of mobile devices, it would end up performing worse than light-weight CNNs. ViT-based networks are also very prone to overfitting, so they need extensive data augmentation and L2 regularization to prevent it. It’s quite likely that their need for more parameters stems from their lack of an image-specific inductive bias, which is inherent in CNNs. [XSM+21].

Introducing MobileViT

While analyzing the previous paragraphs, the need for a network that can combine the benefits of both CNNs (”spatial inductive biases and less sensitivity to data augmentation”) and ViTs (”input-adaptive weighing and global processing”) becomes apparent [MR22a]. This need is met by MobileViT, while also managing to be mobile-friendly in the process. This is done through the introduction of the MobileViT block (Figure 3.1), which effectively encodes both local and global information

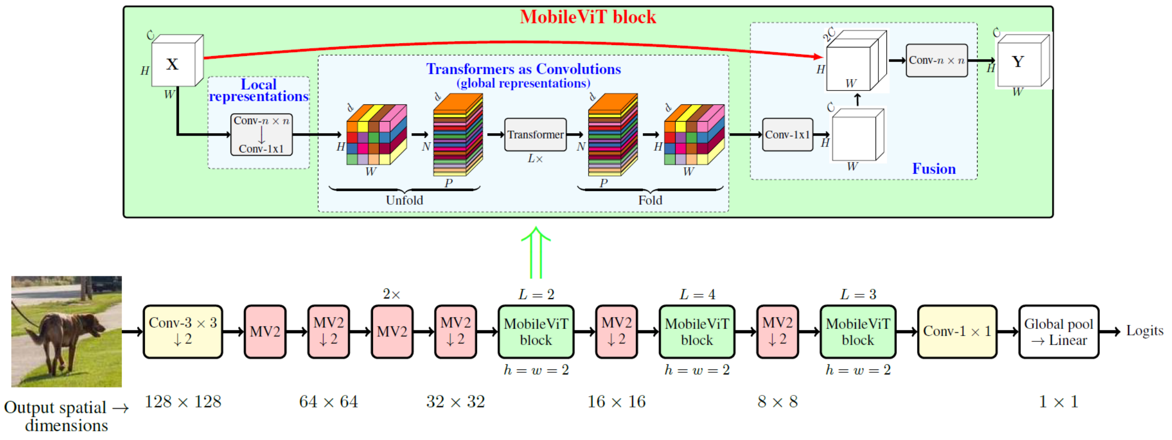


Figure 3.1: MobileViT network [MR22a]

into a tensor, by using a different perspective from other ViT variants to learn global representations. Convolutions normally consist of three operations: unfolding, local processing and folding. The change made by MobileViT is the replacement of local processing in convolutions with global processing using transformers, thus giving it both CNN and ViT-like properties. This is why it is said that the MobileViT block can be viewed as "*transformers as convolutions*". [MR22a]

To explain it briefly, the MobileViT [MR22a] network takes the image input $X \in \mathbb{R}^{H\times W\times C}$ (where W and H are the image's width and height respectively, and C is the number of channels) and passes it through a $n \times n$ convolution layer (which encodes the image's local information), then through a 1×1 (or point-wise) convolution (which simply projects the tensor to a d-dimensional space, where d > C), resulting in the tensor $X_L \in \mathbb{R}^{H\times W\times d}$. Going further, X_L is unfolded into N non-overlapping patches $X_U \in \mathbb{R}^{P\times N\times d}$ (where $P = wh$ is the total number of pixels in a patch and $N = \frac{HW}{P}$ is the total number of patches in the image, and $w \leq n$ and $h \leq n$ are the width and height of each patch respectively). For every pixel in a patch, transformers are used to encode inter-patch relationships, thus obtaining $X_G \in \mathbb{R}^{P\times N\times d}$.

$$X_G(p) = \text{Transformer}(X_U(p)), 1 \leq p \leq P \tag{3.1}$$

Unlike a normal Vision Transformer, MobileViT doesn't lose the patch order or the spatial order of pixels [MR22a], so X_G can be folded to obtain $X_F \in \mathbb{R}^{H\times W\times d}$. X_F is then projected back to a c-dimensional space using a point-wise convolution and concatenated with the input X. The concatenated features are then fused using an $n \times n$ convolutional layer. It should be noted that, as X_U encodes local information thanks to the $n \times n$ convolution and X_G encodes global information thanks to the transformer, X_G also encodes information from all the pixels in X. As such, it can be said that MobileViT has a receptive field of $H \times W$.

All of this being said, there is one more question that arises from this explanation. As mentioned previously, hybrid networks that combine CNNs and ViTs are not a new invention and have been tried before [MR22a], yet they have always been rather heavy-weight. What is it that allows MobileViT to do the same thing while remaining light-weight? The belief of the creators of the network is that it all stems from the way global information is encoded: ”Previous works convert the spatial information into latent by learning a linear combination of pixels. The global information is then encoded by learning inter-patch representation using transformers.” This would lead the network to lose its image-specific inductive bias, making it require more capacity to learn. The key information here is that MobileViT gets to keep that CNN-like inductive bias, allowing us to create models that are more ”shallow and narrow”. This is also what allows it to be more efficient than regular ViT models, despite it having a higher computational cost for calculating multi-headed self-attention ($O(N^2Pd)$ in MobileViT vs $O(N^2d)$ in ViTs).

Finally, one more difference between MobileViT [MR22a] and regular ViTs is that ViTs generally learn multi-scale representations by fine-tuning, due to them needing to interpolate positional embeddings based on the input size, while MobileViT has no such requirements. Due to the CNN-like properties mentioned above, MobileViT doesn’t require any positional embeddings and so it may benefit from multi-scale inputs directly during training (an approach which has been proven beneficial for CNNs).

3.1.2 The M-ViT Network

The M-ViT [PXSJ23] network improves upon MobileViT for the exact purpose of more accurately classifying mushrooms. The creators assure us that after a thorough search of the literature, they reached the conclusion that their work was the first to attempt to use a hybrid CNN and ViT model for mushroom classification. Previous methods of mushroom classification did not fully consider the minor differences between different classes of mushrooms, nor the significant differences within classes (a characteristic of the problem being part of the field of FGVC). For this purpose, M-ViT fuses multidimensional attention in order to feature map global high-level information (like semantic information), which helps with identifying relevant objects and with reducing the interference caused by the image’s background. At the same time, having the model make use of convolutions is beneficial, as it improves its generalization ability by making use of low-level local information (e.g.: texture) [PXSJ23].

M-ViT uses MobileViT as its backbone [PXSJ23] with a migration learning approach, making use of the idea of splitting and fusion. First, the attention mechanism SE (Squeeze-and-Excitation) is added to the MV2 block (taken from the Mo-

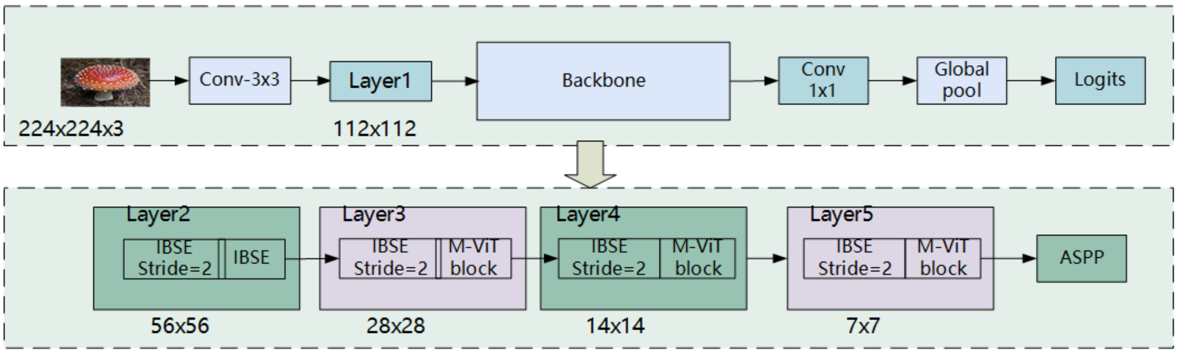


Figure 3.2: The M-ViT pipeline [PXSJ23]

bileNetv2 model) of the network to enhance the representation of picture channels. The structure of the Block module also gets changed so that now it contains an improved multidimensional attention module, whose resulting features need to be spliced together with the channels of the original feature map into 3C (3 channels). Lastly, the ASPP module ensures a more long-distance relationship. This method performs multi-scale feature fusion on the input image through the use of dilated convolution. Figure 3.2 shows the overall structure of the network.

As for the MV2 Block, an ”inverted bottleneck” structure is used (more commonly known as an ”Inverted Residual Block” or ”MBConv Block” [SHZ+19]), where the input is first widened using a 1×1 convolution, followed by a 3×3 depth-wise convolution, which will reduce the number of parameters. Finally, the Block ends with another 1×1 convolution to once again reduce the number of channels. In this model [PXSJ23], the residual structure also has the SE attention mechanism added to it in order to enhance the picture's channel characterization ability. The structure of the modified MV2 block can be seen in Figure 3.3.

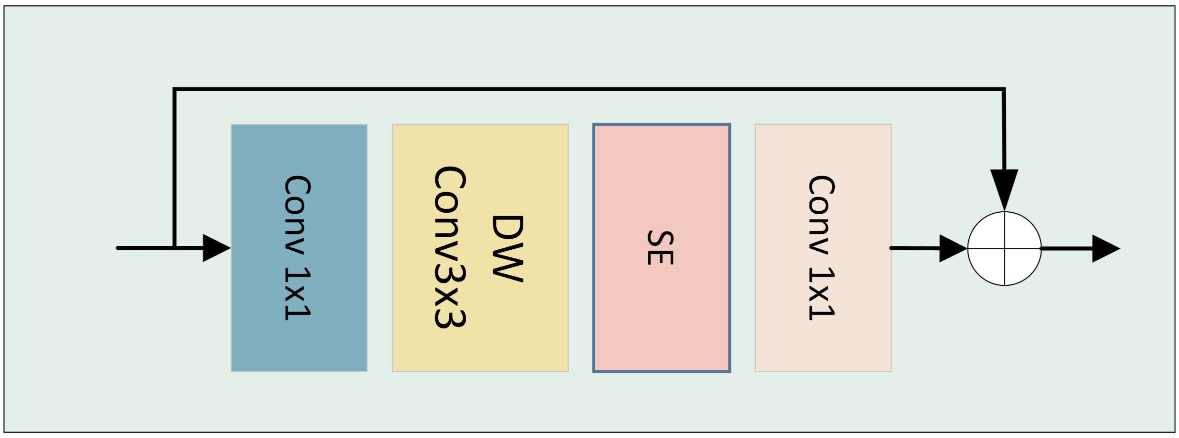


Figure 3.3: MV2 Block [PXSJ23]

Mathematically, the MV2 module [PXSJ23] can be written as in the equation 3.2:

$$MV2(x) = x + \text{Proj}(SE(DWConv(Norm(x)))) \tag{3.2}$$

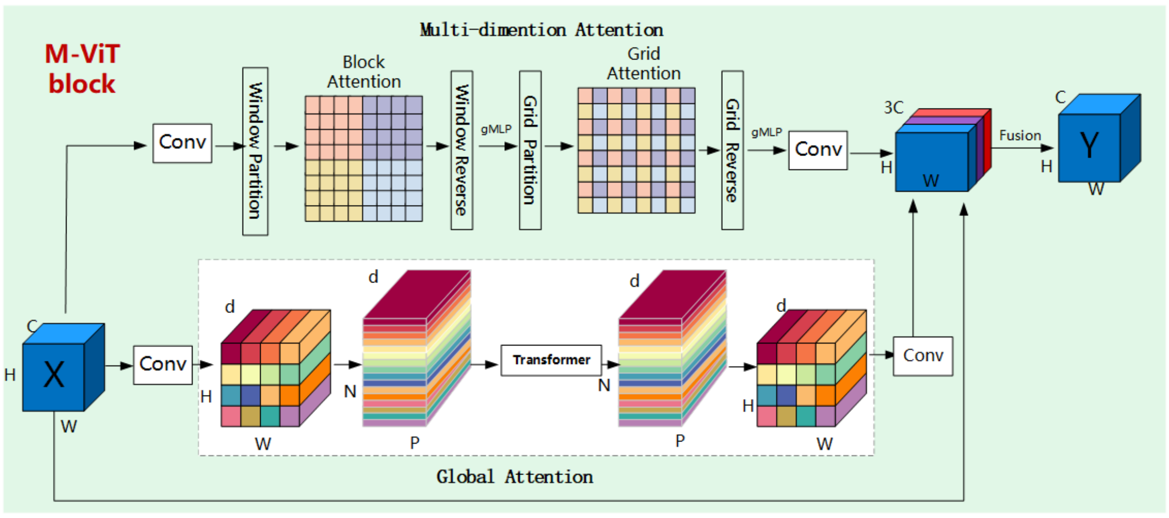


Figure 3.4: The M-ViT Block [PXSJ23]

Proj and *Norm* both represent point-wise convolutions, and *DWConv* is a convolution with the kernel size 3×3 and its input feature dimension equal to that of the output. As stated above, it will reduce the number of parameters. *SE* is a standard Squeeze-and-Excitation module.

Figure 3.4 details the overall structure of the M-ViT block [PXSJ23]. It can be seen that in order to learn more detailed global and local information, the network uses global attention, fused in parallel with multidimensional attention. Other common modules are also present, such as LayerNorm, gMLP, Transformer and jump connection. Depth-wise convolutions are also added in front of the attention module in order to replace the location encoding layer used in Transformer, since deep convolutions work as conditional position encoding (CPE).

To go into more details, the input feature map is passed through both of the aforementioned attention modules and then fused together [PXSJ23] with the output of the modules to obtain the feature map of three channels (3C), after which the feature map gets restored using Fusion with a convolution layer with a 3×3 kernel. The "Conv" blocks on the diagram represent 1×1 convolutions and the Window partition and Grid partition blocks are ways of dividing the input data into 4×4 chunks. Block attention simply performs self-attention in the window, while grid attention executes on the pixels of the entire window and grid space (both of these mechanisms are explained in more detail later). The global attention module in the original MobileViT network is insufficient for extracting local information, which is why this network has the MDA (multidimensional attention) module to enhance feature extraction.

The gMLP [LDSL21] (gated Multilayer Perceptron) block brings an MLP-alternative to the Transformer model that functions without self-attention, using instead only channel projections and spatial projections. Essentially, while attention mechanisms

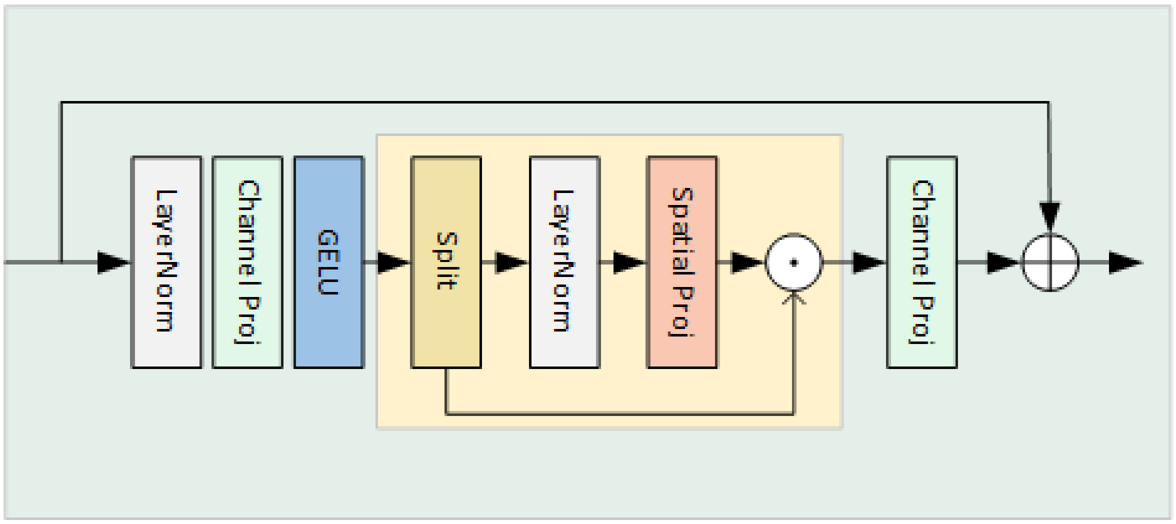


Figure 3.5: The gMLP Block[PXSJ23]

change dynamically based on the inputs, an MLP only learns a fixed weights matrix, making it statically parameterized. Thus, the gMLP resulted from the attempt to see if an MLP (which can learn arbitrary functions) could reach a Transformer's effectiveness, or whether the inductive bias brought by self-attention was required. The results, briefly, show that the gMLP could achieve similar results to those of a Transformer. In more detail, its structure is shown by the equation 3.3 [PXSJ23]:

$$\mathrm{gMLP}(x) = x + \mathrm{proj}(\mathrm{SGU}(\mathrm{GELU}(\mathrm{proj}(\mathrm{norm}(x))))) \tag{3.3}$$

$$\mathrm{SGU}(x) = \mathrm{proj}(\mathrm{norm}(\mathrm{split}(x))) \tag{3.4}$$

In the formulae 3.3 and 3.4, SGU is the acronym for the Spatial Gating Unit, *proj* refers to linear projection and *norm* refers to regularization. The gMLP block can be seen in Figure 3.5.

The multidimensional attention module takes a different approach from the global attention one [PXSJ23]. It decomposes the problem of attention (the spatial axis) into a local (block attention) and a global (grid attention) variant. Its structure can be seen in Figure 3.6.

To explain each type of attention in more detail:

1. Block Attention [PXSJ23]: Considering the input to be the feature map $X \in \mathbb{R}^{H\times W\times C}$, it first gets split into non-overlapping windows, each with the size of $P \times P$, resulting in a tensor with the shape $\left(\frac{H}{P} \times \frac{W}{P}, P \times P, C\right)$. This is done with the $\mathrm{block}(\cdot)$ operation defined in Equation 3.5 and the for doing the reverse we define the operation $\mathrm{unblock}(\cdot)$.

$$\text{Block} : (H, W, C) \rightarrow \left(\frac{H}{P} \times P, \frac{W}{P} \times P, C\right) \rightarrow \left(\frac{HW}{P^2}, P^2, C\right) \tag{3.5}$$

The Equation 3.7 shows the steps of calculating the output of the Block Attention mechanism.

$$\text{Block-SA}(x) = (x + \text{Unblock}(\text{RelAttention}(\text{Block}(\text{DWConv}(\text{LN}(x)))))) \tag{3.6}$$

$$\text{BlockAttention}(x) = \text{Block-SA}(x) + \text{gMLP}(\mathbf{LN}(\text{Block-SA}(x))) \tag{3.7}$$

The Block-SA Block in Figure 3.6 actually divides the input into Windows and then performs the RelAttention operation, as per Equations 3.5 and 3.8.

$$\text{RelAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) V \tag{3.8}$$

2. Grid Attention [PXSJ23]: It functions similarly to Block Attention, converting the input features into tensors with the shape $\left(G \times G, \frac{H}{G} \times \frac{W}{G}, C\right)$, with a window of adaptive size of $\frac{H}{G} \times \frac{W}{G}$, and then after further processing, a window size of $G \times G$. This is done using the $Grid(\cdot)$ operation, defined in Equation 3.9. After this, the RelAttention (3.8) mechanism can be applied on the newly formed grid. The difference between Grid Attention and Block Attention is that the grid dimension has to be placed on the assumed spatial axis, which requires us to use some additional transposes. To revert the grid operation performed in the beginning, we will also define the $UnGrid(\cdot)$ operation.

$$\text{Grid} : (H, W, C) \rightarrow \left(G \times \frac{H}{G}, G \times \frac{W}{G}, C\right) \rightarrow \left(G^2, \frac{HW}{G^2}, C\right) \rightarrow \left(\frac{HW}{G^2}, G^2, C\right) \tag{3.9}$$

The process of computing Grid Attention is described by the Equation 3.11. Grid-SA is the function that actually splits the input into windows and performs the RelAttention operation, similar to Block-SA.

$$\text{Grid-SA}(x) = (x + \text{UnGrid}(\text{RelAttention}(\text{Grid}(\text{DWConv}(\text{LN}(x)))))) \tag{3.10}$$

$$\text{GridAttention} = \text{Grid-SA} + \text{gMLP}(\text{LN}(\text{Grid-SA})) \tag{3.11}$$

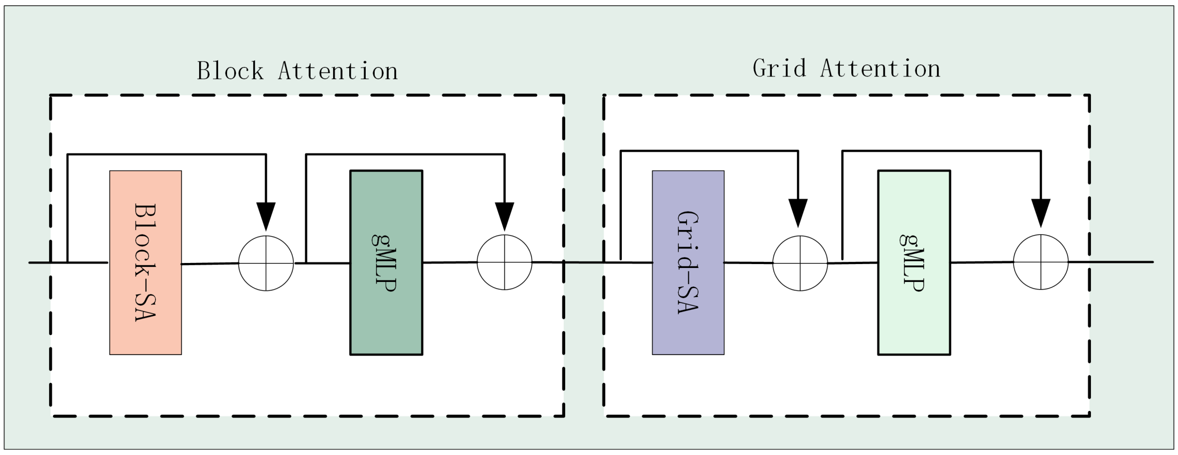


Figure 3.6: The MDA Block [PXSJ23]

3.1.3 The MobileViTv2 Network

Although MobileViT is an efficient network, it still suffers from one major bottleneck, one that most ViT-based networks suffer from, the multi-headed self-attention (MHA) [MR22b]. MHA is at the heart of MobileViT, allowing the tokens to interact and being the sole reason the model can even learn global information, but its complexity is $O(k^2)$, which is to say it is quadratic with respect to the number of tokens k, as each token must perform the attention calculation with respect to every other token. Besides this, MHA requires computationally expensive operations (e.g. batch-wise matrix multiplication and softmax).

MobileViTv2 is the network that resulted from the attempt at making MHA more easily computable. They key to this network is getting a way of computing self-attention with linear complexity, which in this case is done by using *separable self-attention* [MR22b].

The main idea behind separable self-attention is to compute context scores with respect to a latent token L [MR22b]. The scores obtained like this can then be used to re-weight the input tokens and generate a context vector. This approach reduces complexity by a factor k, as tokens no longer need to compute k calculations, each with respect to k tokens. Moreover, this paper presents a method for computing attention through element-wise operations, which further reduces the processing demands on resource-constrained devices.

MobileViTv2 is the network obtained by swapping MHA in the regular MobileViT network with a separable self-attention mechanism. This network also gives up on the skip connection and fusion block that the original used (Figure 3.1), for performance considerations [MR22b].

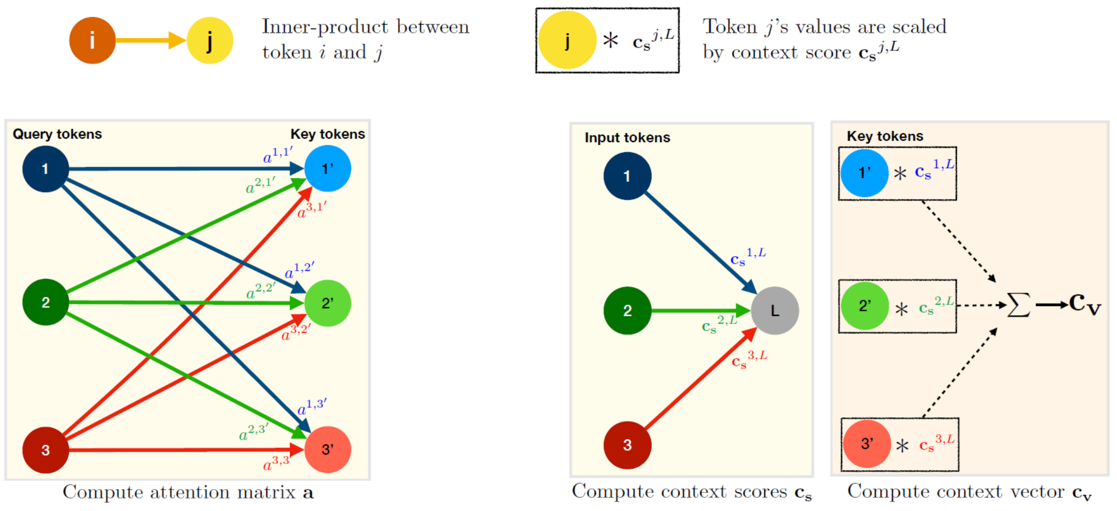

Inner-product between token i and j
Token j's values are scaled by context score $\mathbf{c_s}^{j,L}$
Query tokens
Key tokens
Input tokens
Compute attention matrix $\mathbf{a}$
Compute context scores $\mathbf{c_s}$
Compute context vector $\mathbf{c_v}$

Figure 3.7: Self-attention in transformers (on the left) vs separable self-attention (on the right) [MR22b]

3.1.4 The M-ViTv2 Network

Seeing how the M-ViT network was developed on top of MobileViT and so was MobileViTv2, and seeing how both of them have been reported to have better performance than the original MobileViT ([PXSJ23], [MR22b]), I began to wonder what would happen if one took the modifications that turned MobileViT into M-ViT and applied them to MobileViTv2. It seemed to me that such a model could show promise in solving the problem at hand, so I proceeded to implement it. Despite this new model being nothing more than a merger of M-ViT and MobileViTv2, I will begin referring to it as M-ViTv2 for the rest of this thesis, for the sake of simplicity.

3.2 Implementation Details

This section will explore the implementation of the model described in the previous section, as well as the technologies that were used to achieve this implementation and the way it is being made accessible for others.

3.2.1 Technology Stack

Given how a model such as M-ViT is challenging to implement from scratch, much less in an efficient manner, I have made use of the PyTorch library to facilitate the implementation and have integrated the model itself in a Flask application in order to expose it as an API that anybody could use, without any restrictions. This section will explain why these technologies were used and to what effect.

PyTorch

PyTorch is an open-source deep learning framework used by data scientists around the world, known for its flexibility and ease of use [Nvind]. It is not only a great choice for prototyping, but also a great choice for developing fully fledged models for applications dealing with image recognition and natural language processing, such as the problem we are attempting to solve here. It has a very intuitive Python frontend, meaning that it uses a popular imperative language while also harnessing the full power of the GPU.

Besides its performance, PyTorch developers also benefit from the framework's popularity, which gave rise to a thriving community of developers, making it almost trivial to seek answers to most problems you have [Nvind]. This makes PyTorch an all-around great framework for the development tasks required by the models presented in this thesis.

Flask

Flask is a web framework for Python designed to be lightweight and flexible [Intnd]. It's made in a way that allows for an easy start building web applications, while being powerful enough for more complex use cases.

Its simplicity made it an easy choice for my purposes, as I didn't need much more than a simple way of allowing my main application (described in Chapter 3) to use the AI model developed in PyTorch, and as the main application is not developed in Python, connecting them through a REST API was the easiest option, in great part thanks to Flask.

3.2.2 REST API

The REST API developed in Flask to expose the developed model was crucial to using this model in practice, yet its structure is not very complex. This section will briefly present it, together with its validations and tests.

Endpoints

The API exposes only one endpoint, `../classifications/identify`.

It accepts POST requests that contain an image in either PNG or JPEG format, and returns a JSON response shaped like so: `{"classificationResult": "..."}`. It will contain either the image label, predicted by the AI model, together with an HTTP OK code, or an error message together with an error code, depending on what went wrong. The API has checks that reject requests that either do not contain files, have files declared but none actually attached, or have files with the wrong format, together with a catch-all error message in case something unexpected happens.

Tests

Given the simplicity of the API, not many tests can be written. That being said, the endpoint presented earlier has been Unit Tested to ensure it truly is resilient (to a reasonable degree) against missing files or non-image files. These tests do not involve the AI model in any way, its use being entirely mocked out, in order to preserve simplicity and speed.

3.3 Experiments

This section is dedicated to showing the results of the experiments I have run, the dataset and conditions the model was trained on, as well as how it compares to other variations of itself.

3.3.1 Performance Metrics and Model Training

The model was developed using Python 3.11 and PyTorch 2.1.2. The input image size was $224 \times 224 \times 3$ (a resolution of 224×224 and 3 color channels) and the images were brought to this size first by resizing them to a resolution of 256×256 and then applying a random crop of 224×224. Then, data augmentation techniques were applied in order to increase dara variety, namely random vertical and horizontal flipping (both with a probability of 50%), as well as random rotations of 45°- 90°. All of these transformations were applied using Torchvision transforms. The model was trained on only one Nvidia GeForce RTX 3090. AdamW was used as optimizer, with a batch size of 32, a learning rate of 5×10^{-4}, and a weight decay, just like in the original paper [PXSJ23]. The training process ran for 100 epochs. In order to assess the model's performance, I have evaluated it based on its Accuracy, Precision, Recall and F1-Score. During this process, the dataset was split according to a ratio of 8:2 into a training set and a validation set.

3.3.2 The MO106 Dataset

All of the experiments in this section were performed on the MO106 dataset, containing 29,000 images of mushrooms belonging to 106 different species. The data from the set comes from two sources:

1. The 2018 FGVCx Fungi Classification Challenge data-set [KC21] - used during the challenge with the same name. It contained many more images, but many

of them were unsuitable for learning about mushrooms, and so only those in which you can see the whole mushroom or its gills have remained.

2. The Mushroom Observer website [KC21] - where people (not necessarily experts) can upload and label images of mushrooms and add various metadata, including the certainty of the classification, as determined by the community. Despite this, many images were unsuitable and the creators of the set evaluated them meticulously, picking only images from species that had a large image-count and ensuring that every image contained a clearly visible mushroom.

In the end, the dataset was left with the 29,100 images in 106 classes. The largest class has 581 elements, the smallest has 105 and the average is 275. The images resolutions vary between 97×130 and 640×640.

3.3.3 Results

Overall, four models were trained on the MO106 dataset according to the details mentioned in Subsection 3.3.1. These models were MobileViT, MobileViTv2, M-ViT and M-ViTv2. Due to computational constraints, these models were trained from scratch, without being pretrained on the ImageNet dataset. However, to truly be able to compare these models with the true state-of-the-art performance of modern mushroom classification techniques, an attempt was made at training the MobileViT and MovileViTv2 networks starting from the pretrained weights provided by the timm library[Wig19].

Model	Accuracy (%)	Precision	Recall	F1-Score
MobileViT	19.11	0.180	0.191	0.176
MobileViTv2	90.87	0.908	0.980	0.908
M-ViT	94.11	0.941	0.941	0.941
M-ViTv2	96.48	0.964	0.964	0.964

Table 3.1: Training metrics

Table 3.1 shows us the models' metrics during training. Although the results shown here need to be taken with a grain of salt, as they cannot reveal a model's overfitting, they can still give us some useful information about how the models progressed. For instance, it shows that the two M-ViT models achieved the highest accuracy, which would make theoretical sense, as the M-ViT model was designed specifically for the problem of mushroom classification.

We also see that the MobileViT was not capable of learning much meaningful information, not even by rote memorization (which is consistent with its learning

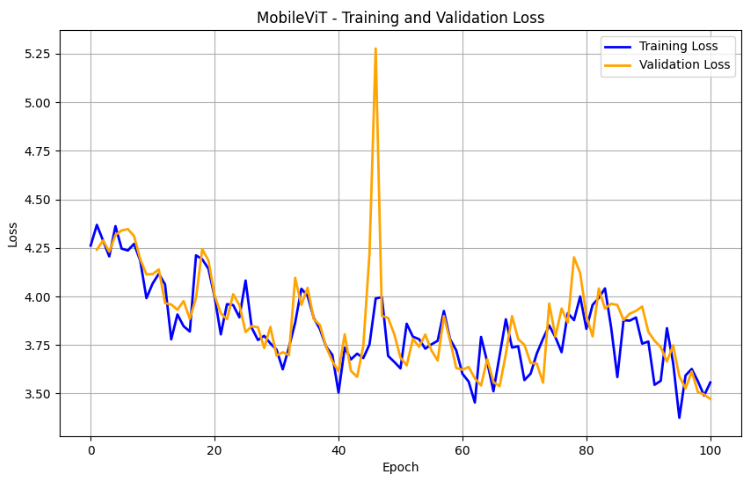


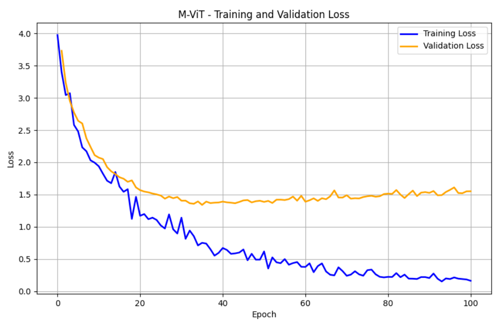
Figure 3.8: MobileViT's loss curve, not showing any meaningful decrease in loss over time.

curve seen in Figure 3.8). This is quite the puzzling result, as not only is it the base that the rest of the models are built on, but it has also achieved better performance on similar classification problems (e.g. Peng et. al [PXSJ23]). One might assume that the model simply isn't complex enough to adapt to this problem. Generally, MobileViT comes in a few variants, each implemented with a set of standard values for all its layers. Even the variant with the greatest complexity is much smaller than the fixed MobileViTv2 variants. This cannot be the issue, however, given that this model will later prove itself capable on this same problem, as seen in Table 3.3.

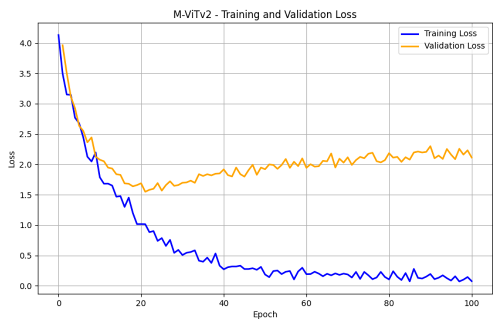
Model	**Accuracy (%)**	**Precision**	**Recall**	**F1-Score**
MobileViT	17.71	0.178	0.177	0.160
MobileViTv2	65.00	0.665	0.650	0.650
M-ViT	68.80	0.699	0.688	0.686
M-ViTv2	66.80	0.686	0.668	0.670

Table 3.2: Validation metrics

Table 3.2, on the other hand, shows us the models' validation metrics, which are much more reliable. Unsurprisingly, we see that MobileViT still didn't perform well, but the result isn't much worse. We also see that M-ViT and M-ViTv2 are still the better options, although now it's clear that the original M-ViT is marginally better at solving this task (supported by their learning curves in Figure 3.9). The fact that all metrics seem to have similar values for each model suggests that all models are pretty well-rounded, not leading to an disproportionate amount of false positives or false negatives, and although the numbers might not seem impressive (all metrics, excluding the MobileViT model, being in the range of $65\% \sim 70\%$), the fact that they



(a) M-ViT's loss curve



(b) M-ViTv2's loss curve

Figure 3.9: Two figures showing how the M-ViT networks' loss converge towards a lower value.

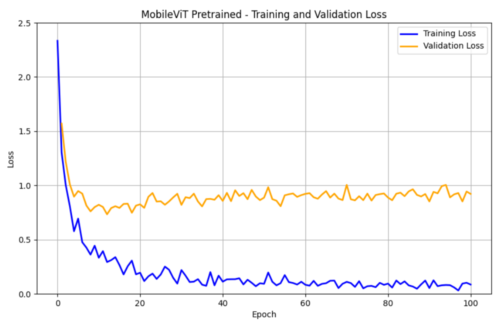
can achieve such performance on 106 classes shows that the models have a good capacity for learning mushroom features.

All that being said, this begs the question: Is this all these models can do? Alas, no, there is a way to push their performance further. I mentioned earlier how I have tried training MobileViT and MobileViTv2 on the MO106 starting from their weights pretrained on the ImageNet dataset. The results of that training can be seen in Table 3.3.

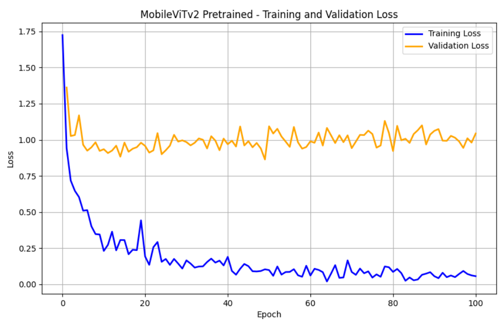
	Model	Accuracy (%)	Precision	Recall	F1-Score
Training	MobileViT	97.61	0.976	0.976	0.976
	MobileViTv2	98.22	0.982	0.982	0.982
Validation	MobileViT	80.98	0.819	0.809	0.809
	MobileViTv2	80.46	0.815	0.804	0.804

Table 3.3: Metrics for MobileViT and MobileViT with pretrained weights

Here we see a massive boost in performance, the pretrained MobileViTv2 achieving an accuracy of 80.56% and MobileViT getting an accuracy of 80.46%, which is 15.56% and 62.71% higher than the one they had obtained previously, an even more puzzling result, as these are the same models as earlier, trained on the same dataset split between training and validation the very same way and using the same code. We can, thus, deduce, that pretraining the models on a larger dataset and then training them on a mushroom dataset significantly improves their performance. However, judging by the rate of convergence of their loss functions, seen in Figure 3.10, we can deduce that this is the maximum performance they can reach. They converged on the near final loss around epoch 20 and haven't made any real improvements since. Nevertheless, they easily got the best results, both on the training and the validation dataset. Whether this is the limit of the models or the limit of the dataset, I cannot say for sure, but it is worth mentioning that both models got nearly



(a) MobileViT loss curve



(b) M-ViTv2's loss curve

Figure 3.10: Pretrained MobileViT and MobileViTv2's losses converging quickly towards.

the same performance (with a difference of 0.1%), so there must be a factor limiting their performance.

To end things, I thought it would be a good idea to put things into perspective and compare the accuracies of all the models presented here and the way they progressed, in order to get a better picture of the whole situation. This can be seen in Figure 3.11 and it shows that the models stayed mostly consistent throughout their training and mostly settled gradually into their final performance over time. It also confirms the superiority of the models pretrained on ImageNet and the very similar performances of M-ViT, M-ViTv2 and MobileViTv2 in the situation where no pretraining was performed.

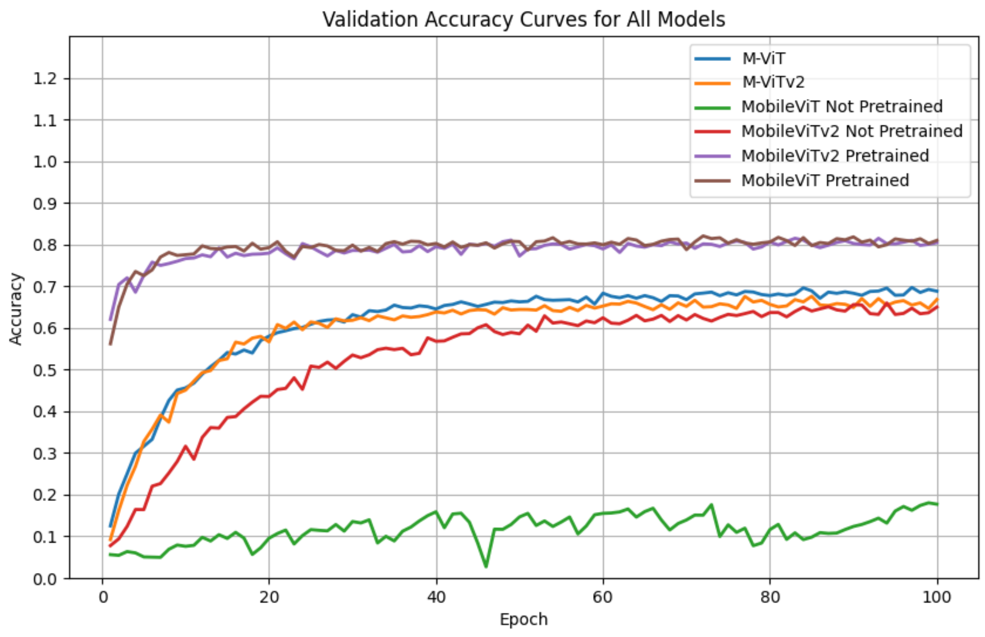


Figure 3.11: The validation accuracy curve of every model presented in this section.

Chapter 4

Practical Application

To showcase the effectiveness of the previous research in a more user-friendly and easily accessible way, I have integrated it into a mobile application called **FungID**. This chapter will focus on the way it was designed, tested, as well as how it can be operated from the user's perspective.

4.1 Application Architecture

This section will be dedicated to dissecting the application from an architectural point of view, going through things such as Class Diagrams, Use Case Diagrams, Sequence Diagrams and Component Diagrams.

4.1.1 Use Case Diagram

As can be seen in Figure 4.1, FungID has 5 main use cases:

1. Logout - the simplest use case, simply disconnecting a user from the application and not allowing further interaction with it until another user logs in.

2. Register - the use case meant to allow a new user to join the application, creating an account stored on the server's database, and allowing anybody access to the use cases that involve persisting data on the server.

3. Login - the use case that allows a user to access their account from any instance of the application, should the server be available. It serves as the true entry point to the application, and must be performed before gaining access to the View Identification History, Identify Mushroom or View Previous Identification use cases.

4. View Identification History - the use case that serves as the home page of the application, showing a user the list of all mushrooms they have submitted for

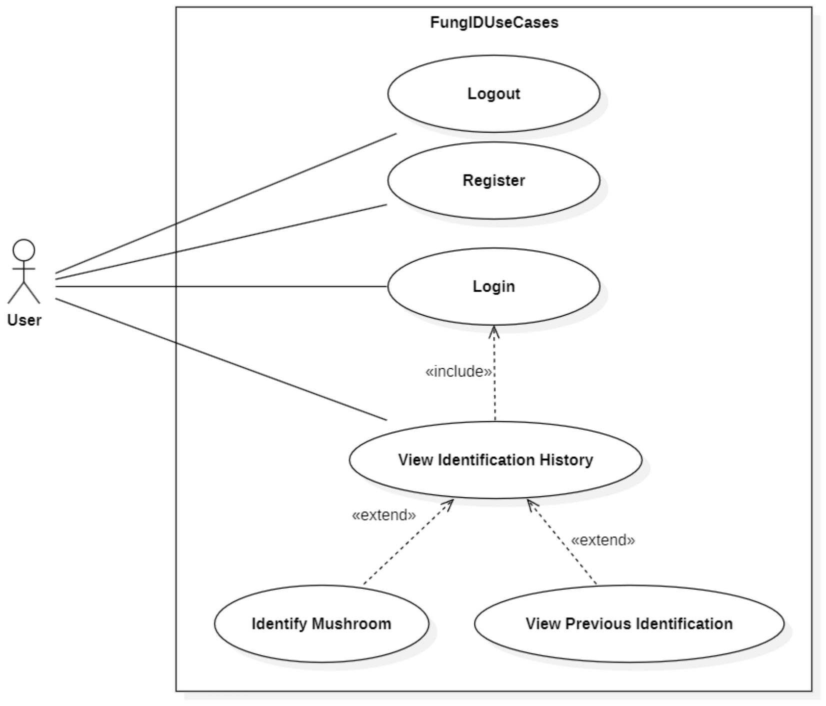


Figure 4.1: FungID - Use Case Diagram

classification using the app. It's also the base use case for the Identify Mushroom and View Previous Identification use cases.

5. Identify Mushroom - the use case representing the main functionality of the app. Starting from the list of previously classified mushrooms, it allows any user to submit new photos for classification.

6. View Previous Identification - the use case allowing users to review images submitted for classification in the past by selecting them from the list.

4.1.2 Login - Sequence Diagram

The flow of logging into FungID can be seen in Figure 4.2. Essentially, the user first types in the login information for an account they have previously registered, then this information gets sent from the UI layer down to the AuthService class, which can contact the backend of the application for authentication (no accounts are stored locally) via an HTTP request. Then, the server passes the information from the controller to the service and then to the repository, which looks up the login information in the database.

If it finds a matching account, it sends the account information back to the controller, which will send an OK response back to the client, together with an authentication token (JWT) so that they may perform their other requests without the app

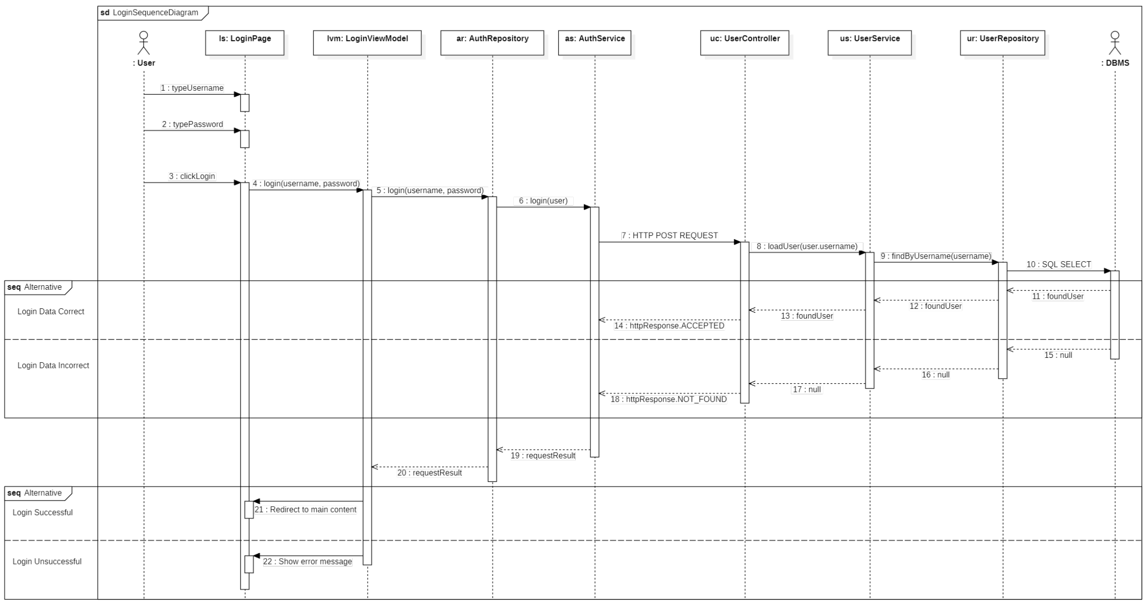


Figure 4.2: Login - Sequence Diagram

storing sensitive information. Upon receiving the response, the app will allow the user access to the other use cases.

If no matching account is found, then the repository returns a null back to the controller, which will send a NOT_FOUND response back to the application, which will then inform the user that the login information they provided is wrong.

4.1.3 Register - Sequence Diagram

Registration follows a flow similar to logging in. The user enters the information needed to create a new account and submits it, after which the information will be passed from the UI down to the AuthService which will contact the server via an HTTP request in order to confirm that there is no account with that information and to create a new one. Upon receiving the request, the UserController will pass the information to its service and then repository which will check for any existing accounts with that information in the database.

Should any account be found, that will be reported up to the UserController, which will send a CONFLICT HTTP response, prompting the application to display an error message for the user telling them they need to select different credentials.

If no matching account is found, however, this is reported back to the UserController, which will delegate the UserService and then the UserRepository the responsibility of creating a new account with the given information. Upon this creation being confirmed as successful, the UserController will send a CREATED HTTP response, together with an authentication token, thus allowing the mobile application receiving it to perform an automatic Login and redirect the user to the main page.

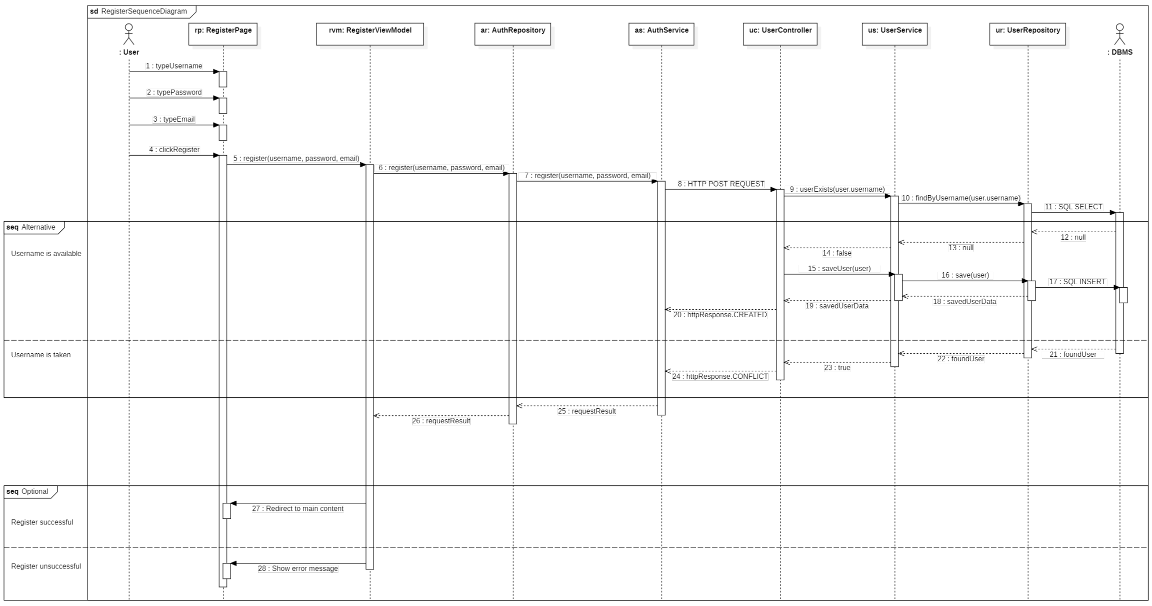


Figure 4.3: Register - Sequence Diagram

4.1.4 View Identification History - Sequence Diagram

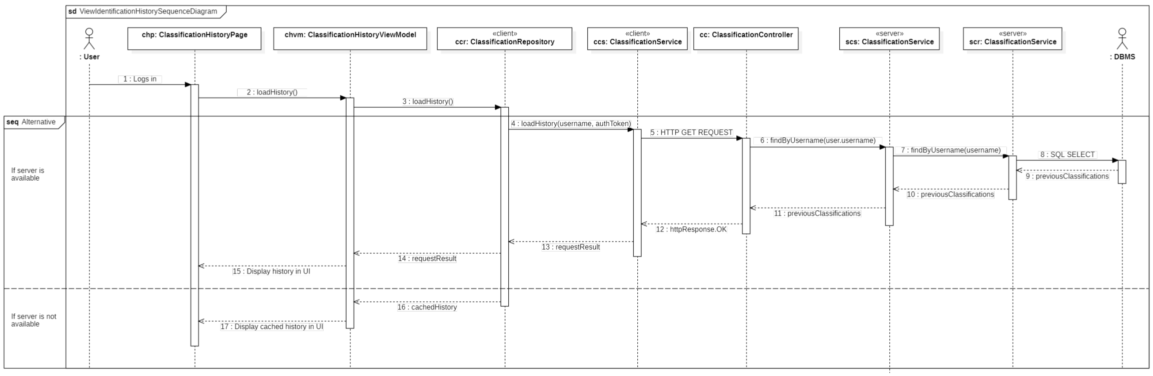

sd ViewIdentificationHistorySequenceDiagram
: User
chp: ClassificationHistoryPage
chvm: ClassificationHistoryViewModel
«client»
ccr: ClassificationRepository
«client»
ccs: ClassificationService
cc: ClassificationController
«server»
scs: ClassificationService
«server»
scr: ClassificationService
: DBMS
1 : Logs in
2 : loadHistory()
3 : loadHistory()
seq Alternative
If server is available
4 : loadHistory(username, authToken)
5 : HTTP GET REQUEST
6 : findByUsername(user.username)
7 : findByUsername(username)
8 : SQL SELECT
9 : previousClassifications
10 : previousClassifications
11 : previousClassifications
12 : httpResponse.OK
13 : requestResult
14 : requestResult
15 : Display history in UI
If server is not available
16 : cachedHistory
17 : Display cached history in UI

Figure 4.4: View Identification History - Sequence Diagram

Viewing the identification history is a use case triggered automatically by logging in. Upon a successful login, the user is redirected to the main page where the ClassificationHistoryPage will display a list of the mushrooms they have classified using the app. To do this, the application's UI will request the list from the ClassificationRepository, which will tell the ClassificationService to request it via HTTP request from the server, if it is available, sending along the user's authentication token. Upon receiving the request, the server's ClassificationController will extract the user's username from the authentication token, and forward it to the ClassificationRepository, which will extract all previous classifications of the user from the database and return it to the controller, which will return it to the application with an OK response. The application will then cache this list in the repository for later use, and display it on the ClassificationHistoryPage.

However, should the server be unreachable for any reason, the app's ClassificationRepository will check if it has any list of classifications already cached and if so, it will return that one to be displayed. This is the first use case that has an offline component.

4.1.5 View Previous Identification - Sequence Diagram

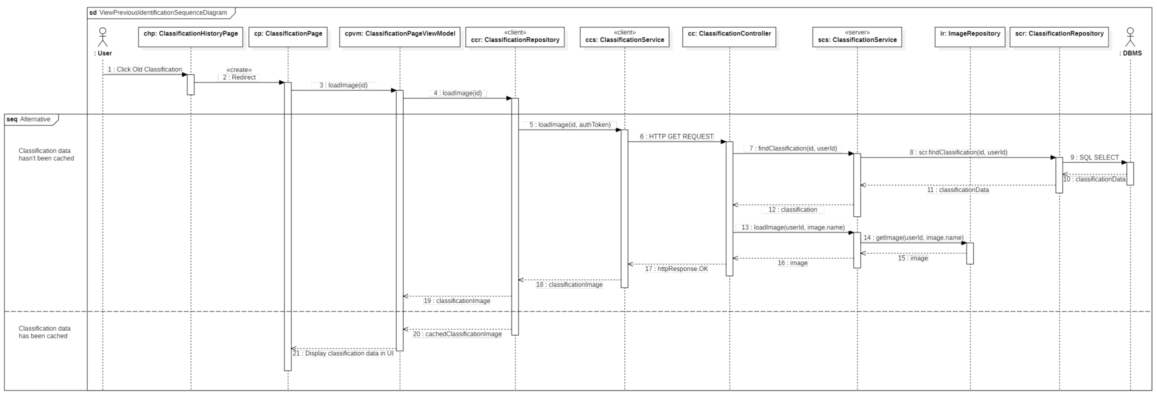

sd ViewPreviousIdentificationSequenceDiagram
: User
chp: ClassificationHistoryPage
cp: ClassificationPage
cpvm: ClassificationPageViewModel
«client» ccr: ClassificationRepository
«client» ccs: ClassificationService
cc: ClassificationController
«server» scs: ClassificationService
ir: ImageRepository
scr: ClassificationRepository
: DBMS
1 : Click Old Classification
«create» 2 : Redirect
3 : loadImage(id)
4 : loadImage(id)
seq Alternative
Classification data hasn't been cached
5 : loadImage(id, authToken)
6 : HTTP GET REQUEST
7 : findClassification(id, userId)
8 : scr.findClassification(id, userId)
9 : SQL SELECT
10 : classificationData
11 : classificationData
12 : classification
13 : loadImage(userId, image.name)
14 : getImage(userId, image.name)
15 : image
16 : image
17 : httpResponse.OK
18 : classificationImage
19 : classificationImage
Classification data has been cached
20 : cachedClassificationImage
21 : Display classification data in UI

Figure 4.5: View Previous Identification - Sequence Diagram

Viewing a previous mushroom photo sent for identification is initiated from the ClassificationHistoryPage by tapping on the element in the list you want to see more about. Then, the user will be redirected to a ClassificationPage displaying the classification data, grabbed together with the list of classifications. But in order to display the photo itself, the ClassificationPage will request it from the ClassificationRepository, which will either return it if it has been cached previously, or forward the request to the ClassificationService, which will make an HTTP request to the server for it, together with the user's authentication token. The server's ClassificationController will receive the request and forward it to the ClassificationService, which will retrieve the classification's data from the database (necessary in order to retrieve the image) and then it will ask the ImageRepository to retrieve the image from the server's storage. After successfully retrieving the image, the ClassificationController will send it back to the app with an OK HTTP response, which will then cache it in the ClassificationRepository for later access, and display it instead of the placeholder image in the ClassificationPage.

4.1.6 Identify Mushroom - Sequence Diagram

Identifying a mushroom is the most important use case in the application, the one everything revolves around. It is triggered by the user from the list of previously

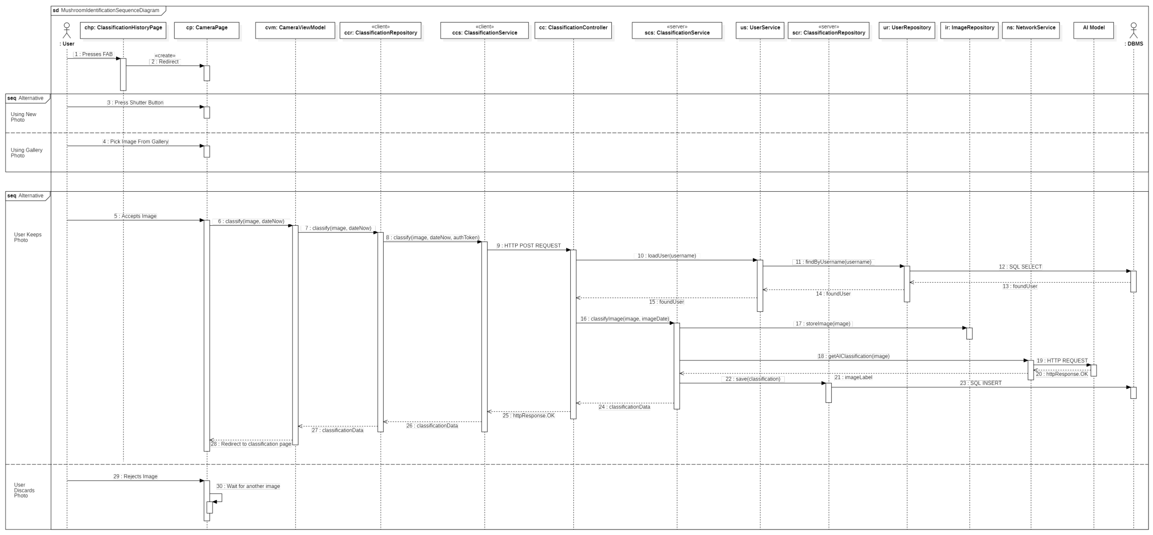


Figure 4.6: Identify Mushroom - Sequence Diagram

identified mushrooms by pressing the button, which redirects him to the CameraPage, where he has the choice of either snapping a new photo using the camera functionality, or picking a photo from his gallery. In either case, he then has the choice of proceeding with the selected image or rejecting it and picking a new one. If he decides to proceed, the image, as well as the time and date it was taken at get sent from the UI to the ClassificationRepository and to the ClassificationService, which sends them via HTTP request to the server's ClassificationController together with the user's authentication token. The server will first assign the UserService the task of identifying the user based on their token, which will pass the task on to the UserRepository, which will finally search for them in the database and send it back to the controller. Then, the ClassificationController will task the ClassificationService with identifying the mushroom in the image. In order, the service will store the image on the server through the ImageRepository, contact the AI Model's server through the NetworkService and get the prediction for the image and then tell the ClassificationRepository to store it in the database, together with the image's path on the server. With the classification finalized, the ClassificationController will send an HTTP OK response together with the predicted label. The application will receive this and have the ClassificationRepository update the local cache to contain the label. Then, the user will be redirected to the mushroom's ClassificationPage, containing the predicted label.

4.1.7 Component Diagram

The component diagram above shows a high-level overview of the entire application divided into its primary systems and components. Thus, the main parts of the

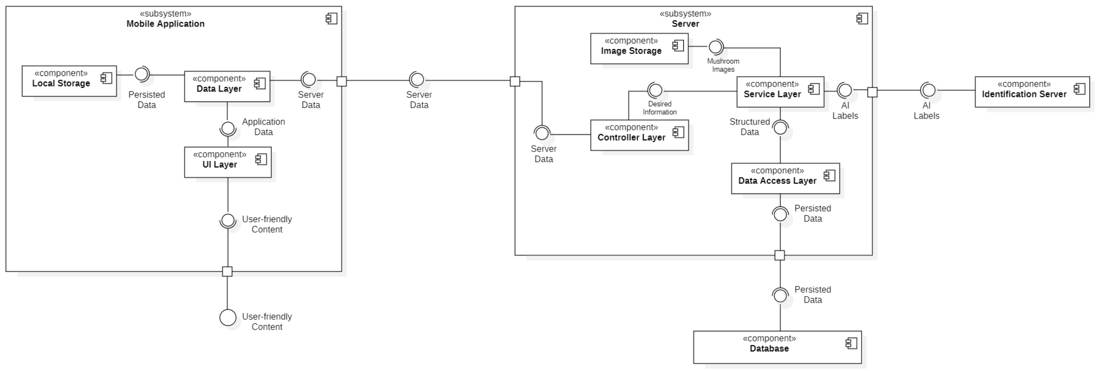


Figure 4.7: FungID - Component Diagram

application are the following.

The Database

The system's main source of truth and unit of persistent storage. Storing everything from user accounts to mushroom identifications, it connects only to the server, which uses it to answer and authorize requests.

The Identification Server

The component responsible for using Artificial Intelligence methods in order to determine the species of a mushroom sent as an image.

The Server

The subsystem taking care of most of the business logic in the system. It sorts out the requests from the mobile app, communicates with the database and the identification server in order to get the required information and put it exactly in the format requested by the app. It itself can be split further into several components:

1. **The Controller Layer** - A component whose job includes communicating with the mobile application, handling requests and formatting responses, coordinating the other components in the Server.

2. **The Service Layer** - Taking commands directly from the Controller Layer, this component aggregates information from other data sources, namely the Data Access Layer, the Identification Server and the Image Storage, sending to the controllers only what is necessary to answer a request.

3. **The Data Access Layer** - This component handles querying the database for any purpose, whether it's to validate the existence of a user account or to retrieve information. This filters the great amount of data it has access to and returns only what's needed to the Service Layer above.

4. **The Image Storage** - A component similar to the Data Access Layer, acting as a repository for storing and accessing images of mushrooms as needed.

The Mobile Application

The subsystem that the user will interact with directly, acting as a bridge between them and the rest of the system. It itself can be broken down into:

1. **The Local Storage** - A component that only persists information so long as a user is logged in. It stores the cached list of classifications done by the user, the images of all mushrooms sent for classification (only those that were viewed from the current device, in order to save storage and bandwidth) as well as the current user's authentication token.

2. **Data Layer** - The component handling most of the business logic in the mobile app. It both aggregates data from the Local Storage and handles communications with the server, acting according to the commands given by the UI Layer.

3. **The UI Layer** - The component that interacts directly with the user. It has the job of following the user's instructions and getting done the tasks they give, but also that of receiving the information from the Data Layer and displaying in a human readable way.

4.1.8 Architectural Patterns

Much like any project, this application's development was also governed by certain general rules, meant to speed up development and prevent issues from showing up as development progressed. This section is dedicated to presenting those general rules, also known as architectural patterns.

Mobile Application - Architectural Patterns

When developing Android apps, you are technically free to choose any kind of architecture you want. In practice, however, one of the most recommended architectures is the Model-View-ViewModel (MVVM).

Similar to others like it, MVVM's purpose is "to provide a clean separation of concerns between the user interface controls and their logic" [Micnd], and it does

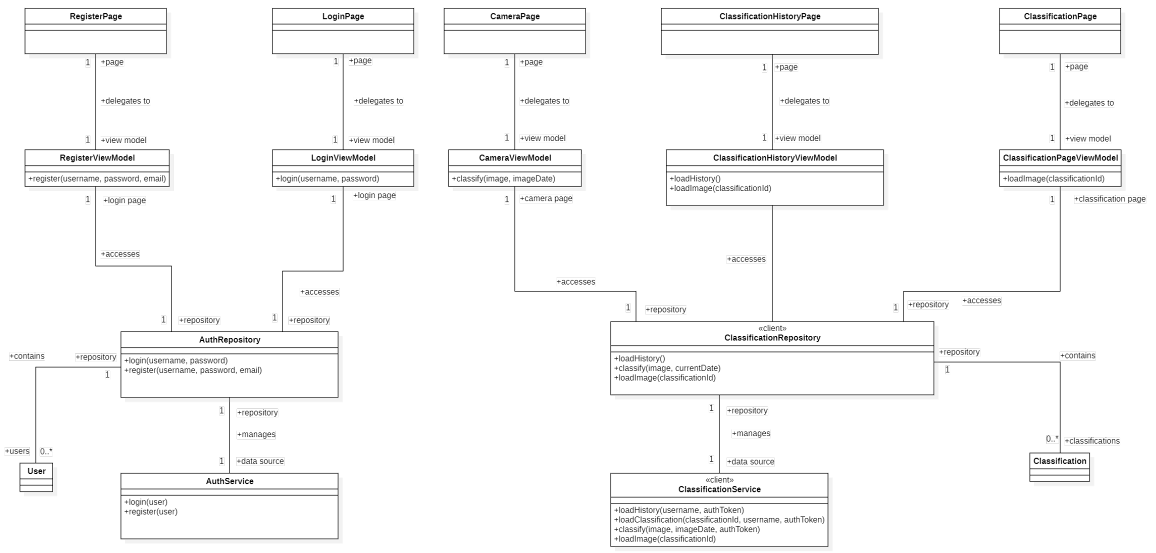

RegisterPage
LoginPage
CameraPage
ClassificationHistoryPage
ClassificationPage
1 +page
+delegates to
1 +view model
RegisterViewModel
+register(username, password, email)
LoginViewModel
+login(username, password)
CameraViewModel
+classify(image, imageDate)
ClassificationHistoryViewModel
+loadHistory()
+loadImage(classificationId)
ClassificationPageViewModel
+loadImage(classificationId)
1 +login page
1 +camera page
1 +classification page
+accesses
1 +repository
AuthRepository
+login(username, password)
+register(username, password, email)
+contains
+repository 1
+users 0..*
User
+manages
1 +data source
AuthService
+login(user)
+register(user)
«client»
ClassificationRepository
+loadHistory()
+classify(image, currentDate)
+loadImage(classificationId)
«client»
ClassificationService
+loadHistory(username, authToken)
+loadClassification(classificationId, username, authToken)
+classify(image, imageDate, authToken)
+loadImage(classificationId)
0..* +classifications
Classification

Figure 4.8: FungID Frontend - Class Diagram

this by using three main components, the View, the Model and the ViewModel. The View, much like in other patterns, is responsible for defining the appearance and overall structure of the things the user can see. The Model is application-specific, but in general refers to an object that contains a data model along with business logic, like a repository [Micnd]. Lastly, the ViewModel acts as an intermediary between the View and the Model, such that the View will call methods on the ViewModel, and the ViewModel will obtain data from the Model in a format the View can handle. Moreover, while the View has to call methods on the ViewModel, the ViewModel simply updates the state of the View when its own state is updated. [Micnd]

Besides MVVM, another very useful pattern is the Repository pattern, meant to act like ”an in-memory domain object collection” [FRF+]. It’s extremely useful for mediating between the UI and data layer of the application, streamlining data access operations, among other things.

Figure 4.8 shows how these patterns are applied, showing the way repositories interact with domain entities as well as external data sources and how ViewModels are the way for Views to interact with the application’s business logic.

Server - Architectural Patterns

The entire server’s architecture is based on one pattern, namely the Layered Architecture pattern. This essentially splits your application into multiple layers that isolate the most important design aspects [Eva04]. Most commonly, these layers are the Presentation Layer, the Application layer, the Domain Layer and the Infrastructure Layer. In this application, the Presentation Layer is represented by the controllers, the Application Layer is represented by the services and the Infrastructure Layer

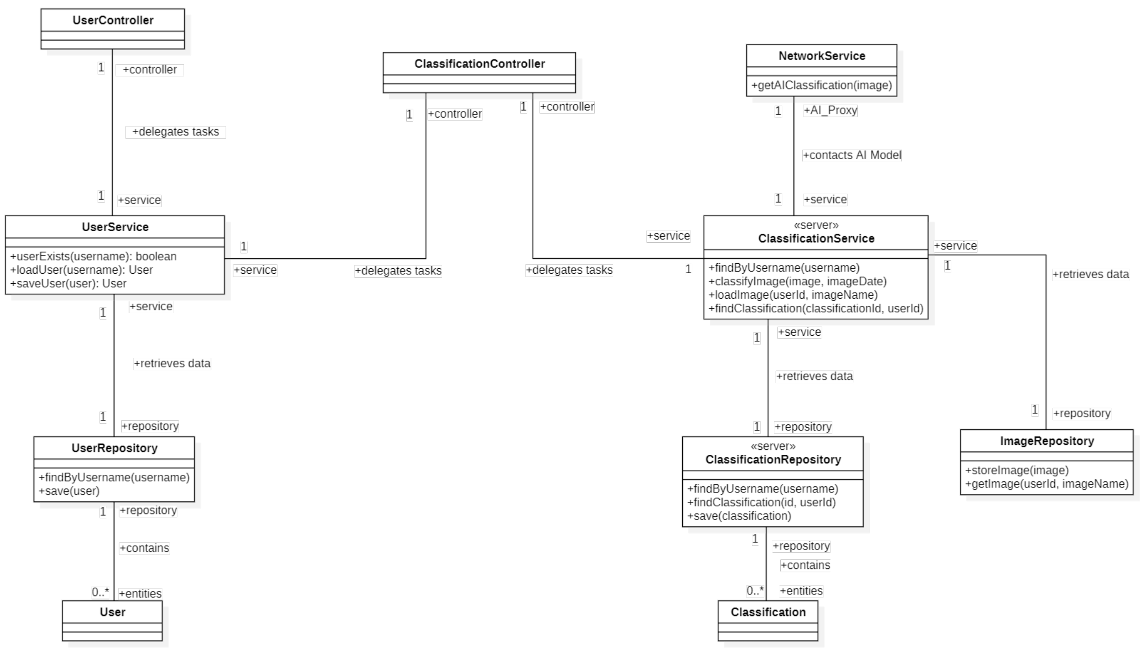


Figure 4.9: FungID Backend - Class Diagram

(though it can refer to more things in general) is represented by the repositories.

The Presentation Layer's job is to present information to an external actor (in this case it's the mobile application) and follow their commands [Eva04].

The Application Layer is responsible for the business logic tasks that are truly meaningful. In itself, it contains no business knowledge, though it coordinates tasks and delegates work in the next layer down [Eva04].

The Infrastructure Layer is represented by repositories, and thus what has been said earlier about the Repository pattern applies here also.

The Domain Layer represents concepts from the business domain. It contains and uses the state of the business, and is overall the "heart of business software" [Eva04].

This way of organizing architecture offers a great separation of concerns and much cleaner design for each individual layer. In isolation they become much easier to maintain, as they not only develop at different speeds, but also react to different needs. [Eva04]

Figure 4.9 clearly shows the way Layered Architecture is used in the server and the way the classes are associated to one another, forming a hierarchy.

4.2 Implementation

Now that we have discussed the application's architecture at length, it's time to discuss its actual implementation, as well. This section will be dedicated to explaining the choice of technologies and programming languages used, as well as give an overview of the way the app was tested.

4.2.1 Technology Stack

The development of this project necessitated a multitude of technologies and programming languages, some of them being Kotlin, Jetpack Compose, Java, Spring Boot and PostgreSQL, all chosen for reasons that will be described below.

Kotlin & Jetpack Compose

To explain why I used Kotlin for this project, I first have to explain why FungID is even on a mobile platform. In short, it simply made sense for a foraging app like this to be supported on a mobile device. Few people are likely to go looking for mushrooms carrying a laptop or desktop with them, and my intention was always to create something with great ease of access. Starting from there, the decision to use Kotlin was a simple one, as it is the recommended programming language for developing Android apps [Win19]. Google has made it one of their priorities to improve Kotlin's capabilities, and it's now a very capable tool, especially used together with other Android Jetpack libraries that offer navigation, lifecycle aware components, camera functionality and more.

One of the main Jetpack libraries available is Jetpack Compose, "a modern UI toolkit for Android built using a Kotlin domain-specific language (DSL)" [Win19]. It makes integrating the UI with the rest of the application absolutely seamless, and despite being different from other ways of structuring UI (e.g.: HTML, XML etc.), it offers great ease of development.

Java & Spring Boot

There are many reasons to use Java for a server like the one FungID has, some of them being its ease of use, its stability and maturity, the number of frameworks it allows you to adopt for your project (one of which being Spring), its statically-typed nature and the fact that it is widely used in the industry for building business apps [Olu23].

Another reason for using Java is access to the Spring framework, allowing for faster development thanks to their many libraries that handle most tasks, such as persistency and security. Spring also has an enormous community with tons of resources, ensuring that almost any problem you have has a solution out there [Sprnd]. On top of this, using Spring Boot spares you the many configurations

Spring would otherwise need, making it easy to focus on the essential functionalities of your app.

PostgreSQL

PostgreSQL is an open-source, object-relational database management system. [Posnd] It's very easy to learn to use, due to it being similar to others like it, very easy to set up and very reliable. It's very much needed in FungID, as several of the app's use cases rely on persistency to keep track of user's accounts and to store their data and restore it to them, even as they change their devices. User accounts are also vital in order to track and be able to restrict the API's usage, and to not be forced to store data for people (guests) who would never be able to retrieve it due to not being registered, so it is plain why persistency is not optional. PostgreSQL is also easily expandable, allowing for the addition of new data types, operators, functions and more, making it one of the best management systems out there.

4.2.2 Tests

In order to ensure a high app quality, the application has been tested extensively. Thus, the application's backend has a test battery of 107 tests, covering the entire business logic. Overall, the code base has a code coverage of 88%, with all but the simplest classes being fully tested.

And to ensure the tests' effectiveness, all classes (that supported it) had both Unit Tests as well as Integration Tests performed on them, as well as several End-To-End tests, all making use of JUnit 5 and Mockito.

4.3 User Interface

The previous two sections presented the application on a technical level, but it is important to also explain the application from the user's perspective. Thus, this section will be dedicated to showing how all of the use cases previously described can actually be performed from inside the app.

4.3.1 Login Flows

The first screen a user sees when opening the application for the first time is the Login screen, seen in Figure 4.10. Here is where the user must introduce their login credentials in order to access the application. The only other way to get past the login screen is to have already logged in previously, case in which the application will remember that and automatically skip the process, keeping your data saved. If the

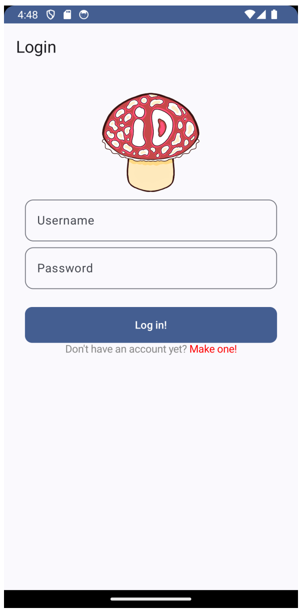


Figure 4.10: The login screen

user does not have an account, they can tap the interactive text ”Make one!” which will redirect them to the Register flow. If the user inputs valid login credentials, they will be redirected to the main page. Otherwise, they will be shown an error message, depending on the way their input was wrong (e.g.: empty fields, unreachable server, no account matching the credentials).

4.3.2 Register Flows

If the user tapped the ”Make one!” button in the Login screen, as described in the previous section, they will be redirected to the Register window, seen in Figure 4.11.

Here, the user must enter the required data in order to create an account. There are several requirements that must be met for the account to be created. The username and email must not be already used for an account, the two entered passwords must match, and no field must be left empty. If any of those conditions are left unmet, or if the server is unreachable, the user will get a detailed error message. If the user has met all of those conditions and their account is created successfully, they will also be logged in automatically, redirecting them to the main page.

4.3.3 Image Classification Flows

After logging in and gaining access to the main application, the user will see the main page, containing the list of previously classified mushrooms (empty at first, as can be seen in Figure 4.12). This page offers several options. First, the button in the top bar can be used to log out. This not only blocks your access to the application,

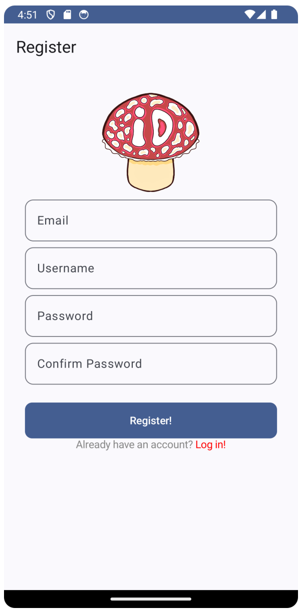


Figure 4.11: The register screen

but also wipes any trace of the mushrooms sent previously for classification, as well their labels or your account details.

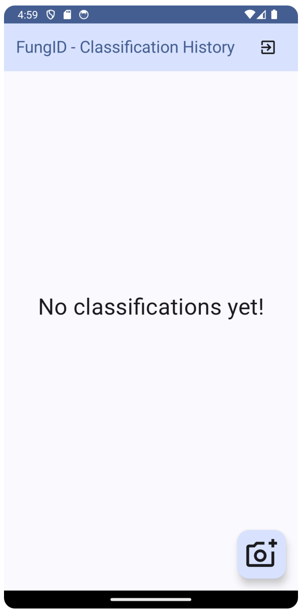


Figure 4.12: The empty main page

Then, if the list weren't empty, it could be used to view any individual classification's image and label. Finally, the camera button in the bottom right corner can be used to submit a new image for classification. After pressing it, the user will be redirected to the application's camera page, seen in Figure 4.13.

Once here, the user can either take a picture using their phone's camera by pressing the shutter button on the bottom center, or to select an image from their phone's gallery using the button on the bottom right.

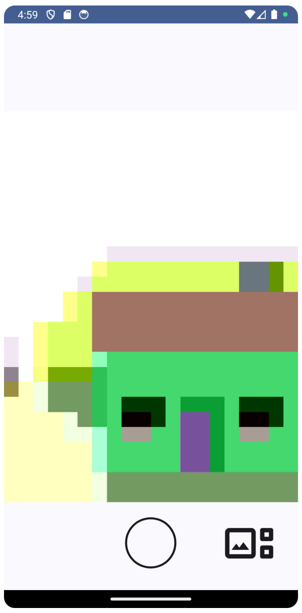
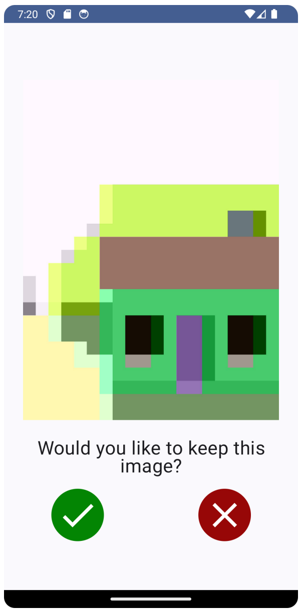


Figure 4.13: The camera page

Once they do that, they will be shown the snapped image (it might not be perfect, as the app optimizes for image quality, not shooting speed, so if the phone moved too much, there is a real chance of getting a blurry image) and are allowed to choose whether or not to submit that image for classification (Figure 4.14). If they choose no, the image will simply be discarded and they will return to the default camera page. If they choose yes, however, the image will be sent to the server for classification and once the result returns, a new entry will be added to the list on the main page and the user will be redirected to its classification page (seen in Figure 4.15).



Would you like to keep this image?

Figure 4.14: The image evaluation screen

The classification page is pretty simple. It only shows the image we just submit-

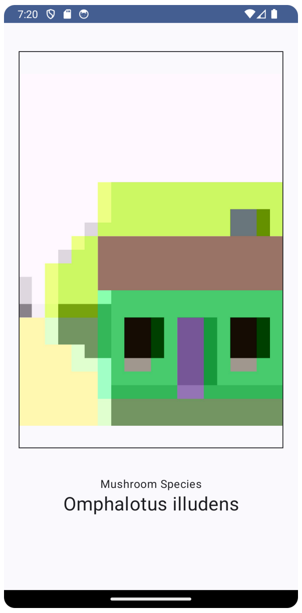


Figure 4.15: The classification page

ted for classification and the AI determined label for it, and if we go back from it we will be taken not to the camera page, but to the main page, where we can see that the list of classifications has one element (Figure 4.16). Tapping on that element reopens the classification page. It bears mentioning that if the camera page cannot contact the server for whatever reason, a snackbar will appear notifyind the user of that. Just the same, when the user opens the app, the main page will try to synchronize its content with the server, but if it cannot reach it, a snackbar will notify the user and contain a button to retry at any point.

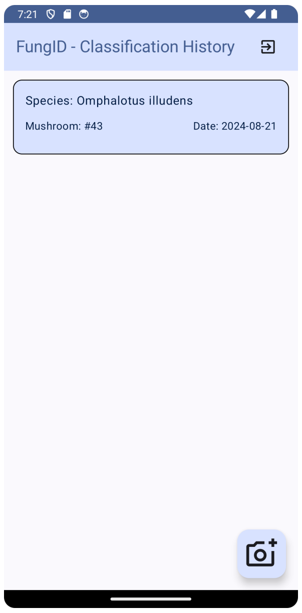


Figure 4.16: The main page containing one element

Chapter 5

Conclusions

In the end, I have been able to get an accuracy of roughly 68.8% on the proposed problem by using the M-ViT model trained from scratch, and my M-ViTv2 model performed similarly. I have also shown that it is more advantageous to use models that have been pretrained on the ImageNet dataset, as both MobileViT and MobileViTv2 were able to get an accuracy of close to 81% in that situation. These results are very good considering the complexity of the problem, mostly because it falls in the field of Fine-Grained visual categorization (FGVC) and is a classification problem with 106 classes, but they are absolutely not sufficient to justify the app being used by humans to determine if a mushroom is fit for human consumption.

The FungID application offers a good user experience, and it gets the job of interfacing with the AI model well enough, while also allowing users to look back on previous classifications, which is truly the only thing a foraging app needs to do.

If I were to continue this project, however, there are a few directions I would take it. Firstly, while interacting with the community built around mushroom classification, I have noticed their belief that mushroom classification based on images is possible, but one image is not enough. You would have much better chances of classifying mushrooms correctly if you allowed multiple characteristics of the mushroom to be visible at once. I recommend a three image perspective: one image showing the mushroom's cap from above, another one showing its underside directly from below and another one showing its side profile. In my review of the literature, I have not found any models that do this, possibly because there don't seem to be any datasets that contain images split like this, and building one from scratch would take considerable effort. I would like to point out, however, that there are databases of high quality images that could be used to synthesize such a dataset, if anybody is interested in this idea (e.g: the Atlas of Danish Fungi mentioned in [PŠM$^+$22], the Mushroom Observer mentioned in [KC21]).

Another area of possible research is trying to get the model to include metadata in its classification model, besides image data. There are already models that do this, some of which were discussed during Section 2.1, and they benefit enormously from it, as you cannot capture the location, the time of the year, the type of soil, the type of forest or other such details in an image alone.

Ultimately, I don't believe mushroom classification based on factors like this will ever be perfect, but I see no reason why it couldn't keep being improved until the point where it rivals a human expert's ability to distinguish mushrooms.

Regarding the application, while FungID gets its job done, it would be nice to see more miscellaneous functionalities implemented in it. My personal favourites include a functionality that would let you build your own atlas of discovered mushrooms composed of images you found. You could be able to keep track of the location where they were found and see descriptions of them based on the determined mushroom species. Perhaps another feature could allow you to share this with the rest of the FungID users, allowing the community to build a map of mushrooms around the world, as mushrooms tend to grow in the same areas year after year. With this information available, the application could even show the user what kind of mushrooms can be found in the area that they haven't discovered yet. Of course, this would involve much better infrastructure, in order to ensure scalability and perhaps better security, to make sure that the servers would be resilient against attacks and various exploits.

If implemented, this application could serve to unite the people in learning more about mushrooms in general, as other projects have already managed in certain countries ([PŠM+22]). I also speculate that a higher degree of knowledge about fungi could also act to reduce the number of cases of mushroom poisoning, though to my knowledge there is no data supporting this fact, so this remains to be seen until/if somebody ever decides to take a project like this to a larger scale.

Bibliography

[ANS23] ANSES. Mushroom season: Poisoning on the rise! https://www.anses.fr/en/content/wild-mushroom-season (Accessed: 24 August 2024), October 2023.

[AOU$^+$23] Usman Ali, Seungmin Oh, Tai-Won Um, Minsoo Hann, and Jinsul Kim. Fine-Grained Image Recognition by Means of Integrating Transformer Encoder Blocks in a Robust Single-Stage Object Detector. *Applied Sciences*, 13(13):7589, June 2023.

[CDC$^+$22] Yinpeng Chen, Xiyang Dai, Dongdong Chen, Mengchen Liu, Xiaoyi Dong, Lu Yuan, and Zicheng Liu. Mobile-Former: Bridging MobileNet and Transformer, March 2022.

[Eva04] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, Boston, 2004.

[FRF$^+$] Martin Fowler, David Rice, Matthew Foemmel, Robert Mee, and Randy Stafford. Patterns of Enterprise Application Architecture.

[FXM$^+$21] Haoqi Fan, Bo Xiong, Karttikeya Mangalam, Yanghao Li, Zhicheng Yan, Jitendra Malik, and Christoph Feichtenhofer. Multiscale Vision Transformers, April 2021.

[Guo24] Xiyang Guo. Research on Mushroom Image Classification Algorithm Based on Deep Sparse Dictionary Learning. *Academic Journal of Science and Technology*, 9(1):235–240, January 2024.

[Intnd] Introduction to Web development using Flask. https://www.geeksforgeeks.org/python-introduction-to-web-development-using-flask/ (Accessed: 22 August 2024), n.d.

[JHP14] Woo-Sik Jo, Md. Akil Hossain, and Seung-Chun Park. Toxicological Profiles of Poisonous, Edible, and Medicinal Mushrooms. *Mycobiology*, 42(3):215–220, September 2014. https://www.tandfonline.com/doi/full/10.5941/MYCO.2014.42.3.215.

[KBK22] Wacharaphol Ketwongsa, Sophon Boonlue, and Urachart Kokaew. A New Deep Learning Model for the Classification of Poisonous and Edible Mushrooms Based on Improved AlexNet Convolutional Neural Network. *Applied Sciences*, 12(7):3409, March 2022.

[KC21] Norbert Kiss and Laszlo Czuni. Mushroom Image Classification with CNNs: A Case-Study of Different Learning Strategies. In *2021 12th International Symposium on Image and Signal Processing and Analysis (ISPA)*, pages 165–170, Zagreb, Croatia, September 2021. IEEE. https://ieeexplore.ieee.org/document/9552053/.

[KNS$^+$20] Urmas Kõljalg, Henrik R. Nilsson, Dmitry Schigel, Leho Tedersoo, Karl-Henrik Larsson, Tom W. May, Andy F. S. Taylor, Thomas Stjernegaard Jeppesen, Tobias Guldberg Frøslev, Björn D. Lindahl, Kadri Põldmaa, Irja Saar, Ave Suija, Anton Savchenko, Iryna Yatsiuk, Kristjan Adojaan, Filipp Ivanov, Timo Piirmann, Raivo Pöhönen, Allan Zirk, and Kessy Abarenkov. The Taxon Hypothesis Paradigm—On the Unambiguous Detection and Communication of Taxa. *Microorganisms*, 8(12):1910, November 2020.

[LDSL21] Hanxiao Liu, Zihang Dai, David R. So, and Quoc V. Le. Pay Attention to MLPs, June 2021.

[LXZ$^+$22] Wenbin Liao, Jiewen Xiao, Chengbo Zhao, Yonggong Han, ZhiJie Geng, Jianxin Wang, and Yihua Yang. Mushroom image recognition and distance generation based on attention-mechanism model and genetic information. June 2022.

[Micnd] Microsoft. The MVVM Pattern. https://learn.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10) (Accessed: 20 August 2024), n.d.

[MR22a] Sachin Mehta and Mohammad Rastegari. MobileViT: Light-weight, General-purpose, and Mobile-friendly Vision Transformer, March 2022.

[MR22b] Sachin Mehta and Mohammad Rastegari. Separable Self-attention for Mobile Vision Transformers. http://arxiv.org/abs/2206.02680, June 2022.

[Nvind] Nvidia. PyTorch. https://www.nvidia.com/en-us/glossary/pytorch/ (Accessed: 22 August 2024), n.d.

[Olu23] Shittu Olumide. What is Java Used For in 2023? The Java Programming Language and Java Platform Strengths. https://www.freecodecamp.org/news/what-is-java-used-for/ (Accessed: 21 August 2024), April 2023.

[Posnd] PostgreSQL. What Is PostgreSQL? https://www.postgresql.org/docs/current/intro-whatis.html (Accessed: 21 August 2024), n.d.

[PR23] Marcel Pârvu and Oana Roşca-Casian. *Ghid practic de micologie*. Presa Universitară Clujeană, Cluj-Napoca, ediția a 2-a edition, 2023.

[PŠCM] Lukáš Picek, Milan Šulc, Rail Chamidullin, and Jiří Matas. Overview of FungiCLEF 2023: Fungi Recognition Beyond 1/0 Cost.

[PŠM+22] Lukáš Picek, Milan Šulc, Jiří Matas, Jacob Heilmann-Clausen, Thomas S. Jeppesen, and Emil Lind. Automatic Fungi Recognition: Deep Learning Meets Mycology. *Sensors*, 22(2):633, January 2022.

[PXSJ23] Youju Peng, Yang Xu, Jin Shi, and Shiyi Jiang. Wild Mushroom Classification Based on Improved MobileViT Deep Learning. *Applied Sciences*, 13(8):4680, April 2023.

[RAP+18] R F Rahmat, T Aruan, S Purnamawati, S Faza, T Z Lini, and Onrizal. Fungus image identification using K-Nearest Neighbor. *IOP Conference Series: Materials Science and Engineering*, 420:012097, October 2018.

[RJL+] Huan Ren, Han Jiang, Wang Luo, Meng Meng, and Tianzhu Zhang. Entropy-guided Open-set Fine-grained Fungi Recognition.

[SHZ+19] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks, March 2019.

[Sprnd] Spring. Why Spring? https://spring.io/why-spring (Accessed: 21 August 2024), n.d.

[Wan22] Boyuan Wang. Automatic Mushroom Species Classification Model for Foodborne Disease Prevention Based on Vision Transformer. *Journal of Food Quality*, 2022:1–11, August 2022.

[Wig19] Ross Wightman. PyTorch Image Models. Hugging Face, 2019. https://github.com/huggingface/pytorch-image-models (Accessed: 20 August 2024).

[Win19] David Winer. Android's commitment to Kotlin. https://android-developers.googleblog.com/2019/12/androids-commitment-to-kotlin.html (Accessed: 21 August 2024), December 2019.

[WLX$^+$21] Peng Wang, Jiang Liu, Lijia Xu, Peng Huang, Xiong Luo, Yan Hu, and Zhiliang Kang. Classification of Amanita Species Based on Bilinear Networks with Attention Mechanism. *Agriculture*, 11(5):393, April 2021.

[XSM$^+$21] Tete Xiao, Mannat Singh, Eric Mintun, Trevor Darrell, Piotr Dollár, and Ross Girshick. Early Convolutions Help Transformers See Better, October 2021.

[ZFZL19] Heliang Zheng, Jianlong Fu, Zheng-Jun Zha, and Jiebo Luo. Looking for the Devil in the Details: Learning Trilinear Attention Sampling Network for Fine-grained Image Recognition, June 2019.

[ZLLS23] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning, 2023.

[ZSC$^+$19] Youxiang Zhu, Weiming Sun, Xiangying Cao, Chunyan Wang, Dongyang Wu, Yin Yang, and Ning Ye. TA-CNN: Two-way attention models in deep convolutional neural network for plant recognition. *Neurocomputing*, 365:191–200, November 2019.