A Comparative Study of Pre-trained Deep Learning Models for Grapevine Leaf Disease Detection

Lorenzo D'Antoni
lorenzo.dantoni@studenti.unipd.it

Alessandro Canel
alessandro.canel@studenti.unipd.it

Fateme Baghaei Saryazdi
fateme.baghaeisaryazdi@studenti.unipd.it

Abstract

We introduce a deep learning approach for grape leaf disease classification, using different pre-trained model architectures and optimization techniques. We used a dataset of grape leaf images, labeled with four classes based on their disease: Black Rot, ESCA, Leaf Blight and Healthy. We enhanced an existing Kaggle project initially utilizing the TensorFlow framework, then adapted it to the PyTorch library. We experimented with various models and parameters, and evaluated their performance using different metrics. Additionally, we devised a max-voting ensemble model obtaining high accuracy.

1. Introduction

Grapes are one of the most important and widely cultivated fruits in the world. The global wine market reached a size of USD 489.3 billion in 2021 and is projected to reach USD 825.5 billion by 2030 [13]. However, grape cultivation is threatened by various diseases that can affect the quality and yield of the grapes. Therefore, it is crucial to detect and identify grape diseases as early as possible, and to apply appropriate treatments.

Visual crop inspections by humans are time-consuming and challenging tasks. There are many scientific articles regarding the diagnosis of crop diseases based on plant images, which helps us comprehend how crucial it is. Because of the rapid growth of artificial intelligence techniques, computer vision and machine learning approaches have become widely used for plant disease identification and categorization.

Our study focuses on grape leaf disease classification using deep learning, identifying whether the leaf is infected by Black Rot, ESCA, Leaf Blight or is healthy (Figure 1).

Due to the lack of freely available annotated datasets, we aimed to implement and improve a *Kaggle* project [16] [15], the only source offering a dataset [14], although extremely limited and small. The project used the *TensorFlow* framework to compare various deep learning pre-trained model architectures for grapevine leaf disease classification. We decided to translate it into the *PyTorch* library in a more optimal and clear manner.

We experimented with model architectures and parameters, enhancing performance and simplifying training. We evaluated the performance of our models using various metrics and we also created a max-voting ensemble model that achieved a very good accuracy. We analyzed the results and discussed the strengths and weaknesses of our approach.

2. Related Work

Before 2016, crop disease classification using plant images relied on methods such as SVM, k-means clustering, and PCA [12]. Typically, k-means clustering was used to find the diseased region (segmentation). Subsequently, both color and texture features are extracted. Finally, a classification technique (SVM) was adopted to detect the type of leaf disease. However, these methods used manually engineered features, which were time-consuming, incomplete, and lacked generalization. They also required extensive preprocessing to deal with image variations and noise

Recently, deep learning has revolutionized computer vision with CNNs that automatically learn discriminative features from images, avoiding complicated preprocessing and preserving relevant information. They also capture hierarchical spatial features at different levels, extracting complex patterns and structures. They are computationally efficient (parameter sharing) and translation invariant.

Many scientific articles use CNNs to detect and classify plant diseases [2] [3]. However, for grape leaf diseases, there were no optimal and efficient CNN models for real-time detection until recently. Grape leaves can have multiple and diverse sick spots (even of different diseases), and environmental and occlusion factors complicate the detection and classification process.

Real-time detection is useful for early disease interven-

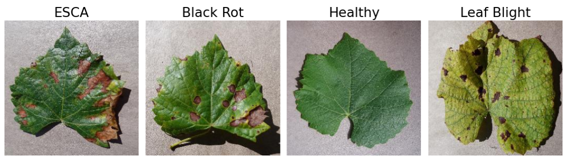

Figure 1. Grape leaf diseases.

tion and treatment, but it requires fast and efficient algorithms with high accuracy. These algorithms need to process large image data in real-time and often run on low-power devices with limited computational resources.

This research paper [21] designed a real-time detector for four grape leaf diseases using Faster R-CNN and ResNet. They enhanced the base model with state-of-the-art deep-learning modules, improving the multiscale diseased spot extraction and detection speed. They trained and tested the model on a private annotated dataset of grape leaves with simple and complex backgrounds. To prevent the overfitting problem, they also augmented the dataset using digital image processing.

Another research project [20] proposed a real-time detector of grape leaf diseases using an improved YOLOXS [4] model. They collected and used a dataset of 15 leaf diseases in natural environments. They aimed to improve the generalization and speed of the existing models, which mostly used laboratory images with simple backgrounds.

3. Dataset

Unfortunately, all the datasets used in the above papers are private and we have not been able to get in touch with the authors. Developing such real-time grape leaf disease detection models requires large annotated datasets of leaf images for training and evaluation. Collecting and annotating such datasets can be time-consuming and expensive, requiring collaboration with domain experts and farmers to ensure the availability of high-quality training data.

Due to limited time and resources, we simplified our objective to image classification of grape diseases, instead of detection and classification of the diseased regions on grape leaves.

We used one of the few online datasets [14] from a *Kaggle* project [16] [15] related to this paper [11]. This dataset has already been augmented with new variations of the original images, such as zooming, flipping, rotating, changing brightness, and shearing. These augmentations increase the size and diversity of the dataset, which helps the model learn better and recognize different patterns more effectively.

They combined 1344 images from another dataset with 1656 augmented images for each class, resulting in a total of 3000 images per class and 12000 images overall.

The original images were sourced from another *Kaggle* dataset [9]. This dataset has leaf images of four diseases (Black Rot, ESCA, Leaf Blight, and Healthy), with about 2000 images for training and 500 for testing per disease. The images are JPG files with a resolution of 256x256 pixels. The dataset has 7260 training images and 1805 testing images in total.

4. Method

We converted notebooks [16] and [15] from *TensorFlow* to *PyTorch*, based on the *Kaggle* project related to the previous paper. The project aims to find the best architectures for detecting four grape leaf diseases: Black Rot, ESCA, Healthy, Leaf Blight.

The authors created a baseline model to compare with more complex models, which were trained on *ImageNet* [1]. They compared the models using various metrics and visualizations, such as training and validation curves, accuracy scores, confusion matrices, classification metrics, and error visualization. They then combined the best three models into an ensemble model to boost performance.

4.1. Translation

Unlike *TensorFlow*, which has built-in APIs for training and validation when using *Keras*, *PyTorch* requires us to manually implement the training and evaluation loops from scratch. This gives us more flexibility, but also introduces some variability in the results. We also had to create a separate method to compute the accuracy based on the network outputs.

The original paper included two extra features in the training loop: early stopping with a patience of 8 based on the validation loss, and model checkpoints to save the best weights according to the validation accuracy. We had to implement these features from scratch as well in the training loop

For the pre-trained models, *TensorFlow* has an option to include or exclude the fully-connected layer at the top of the network. The authors used this option, but it is not as simple in *PyTorch*. Many architectures have a final sequence of layers named “classifier” or “fc”, which may include multiple dense layers, dropout layers, and pooling layers. We decided to keep them and only change the output of the final dense layer to match the number of classes. We also added a global average pooling layer before the classifier layers, as in the original paper. This layer averages each feature map over the spatial dimensions (height and width), resulting in a single value per feature map. This reduces the size of the feature maps and makes them easier for the next layers to process, while preserving important information.

4.2. Data Split

We resized all images to 224×224 pixels before loading them into *PyTorch* datasets and data loaders. Then, we

Dataset	Size (number of images)
Train	12000
Validation	903
Test	902

Table 1. Data partitioning.

converted the images to tensors of shape ($C \times H \times W$) with values between 0.0 and 1.0. This can improve the stability and convergence of the training process.

Next, we built three datasets: train, valid, and test. We did not agree with the authors' way of creating the validation set. They used the same test set for both validation and testing. However, the test set should be a separate dataset that is only used to measure the final performance of the trained model. If we use it to evaluate the performance of the model during training and to fine-tune hyperparameters, then the final evaluation of the model is biased.

The authors augmented the entire training set, so we could not use any of it as the validation set. It had no new data. We had two choices: either create new train and validation sets and augment only the train set, or split the test set in half and use one half as a smaller validation set. The first choice would have been ideal, but we picked the second one to replicate the original work without redoing everything. This allowed us to use the augmented set from *Kaggle*. But this had some drawbacks. There were few images, so the validation and test sets might not reflect the model's true performance on new data. This could mislead us about the model's quality and give unstable performance estimates. So, we should proceed with caution when interpreting the metrics' values and the graphs' patterns.

After the splitting, we got the dataset partitioning given in Table 1.

We used these datasets to train and test the baseline model. To used them with the pre-trained models, we needed to normalize the datasets with the *ImageNet* mean and standard deviation. The pre-trained models were trained on *ImageNet*, and *PyTorch* suggests doing this.

4.3. Models

4.3.1 Baseline CNN

We use this model as a baseline to measure the performance of more complex ones. Table 2 shows the architecture. It is a simple model that uses the same number of filters in every convolutional layer but it works well, as we will see later.

4.3.2 DenseNet

DenseNet (Densely Connected Convolutional Networks) [7] is a CNN architecture with dense connections between layers. In a dense block, each layer gets feature maps from all previous layers and passes them to all next layers. This helps reuse features and improve gradient flow.

Between dense blocks, transition layers reduce the size and increase the number of feature maps with convolution and pooling. This controls the model's complexity and cost.

This architecture has many variants with different numbers of layers: DenseNet-121, DenseNet-169, DenseNet-201, and DenseNet-264. The authors picked DenseNet-121, which balances complexity and performance well, and we agree because the task is simple.

4.3.3 EfficientNet

EfficientNet [19] is a CNN architecture with a new compound scaling method that scales network depth, width, and resolution together, improving performance and efficiency for different resource limits. With a new building block (MBConv), EfficientNet performs better on many image classification tasks with fewer parameters and FLOPs than other architectures.

This architecture has many models: EfficientNet-B0, EfficientNet-B1, B2, B3, B4, B5, B6, B7 and EfficientNet-Lite. B0 is the baseline model and the others are larger variants. Larger models come with increased size and cost, but they also offer enhanced performance

The authors used the EfficientNet-B7 model. We disagree since the model is too complex for the task (over 60 million parameters). The training was very slow, and the performance was bad because the training set is small, and bigger models can overfit to it. A smaller model does better because it is simpler and faster, as we will see next.

4.3.4 MobileNet

MobileNet [6] is a family of light CNN architectures that use depthwise separable convolutions as the main building block. These convolutions split a normal convolution into a depthwise convolution and a pointwise convolution, saving a lot of computation while preserving representation capacity. It also uses many design choices to make it efficient and effective on mobile and embedded systems, using less resources and memory while keeping good performance.

Some better versions were made to improve the original architecture: MobileNetV2, MobileNetV3, and MobileNetV3 Small. The authors used the MobileNetV2 version. We agree with that because the next version, MobileNetV3, is too sophisticated for our task.

4.3.5 ResNet

ResNet (Residual Network) [5] is a deep CNN architecture with residual blocks. These blocks have skip connections (shortcuts) that skip one or more convolutional layers, letting the network learn residual mappings, and making it eas-

Block (Conv-kernel_size)	in_ch, out_ch
Conv-6 + ReLU, BN, MP(2)	3, 32
Conv-5 + ReLU, BN, MP(2)	32, 32
Conv-4 + ReLU, BN, MP(2)	32, 32
Conv-3 + ReLU, BN, MP(2)	32, 32
Conv-3 + ReLU, BN, MP(2)	32, 32
Conv-3 + ReLU, BN, MP(2)	32, 32
Conv-3 + ReLU, BN, MP(2)	32, 32
Dropout(p=0.2)	
(flatten the tensor)	
Linear + ReLU	32, 512
Linear + ReLU	512, 512
Linear + ReLU	512, 4

Table 2. Original baseline CNN architecture. Each convolutional layer is followed by ReLU, BatchNorm, and MaxPool layers. The padding is set to “same” for all blocks.

ier to train very deep networks by avoiding the vanishing gradient problem.

This architecture has many variants with different numbers of layers: ResNet-18, ResNet-34, ResNet-50, ResNet-101 and ResNet-152. In deeper variants, like ResNet-50 and above, bottleneck blocks are used to save computation.

The authors used the ResNet-50 model, which has 23 million parameters. We think it is too complex for our goal, and a simpler model can do almost as well in less time, as we will see later, but it is still a good choice for training time.

4.3.6 VGG

VGG (Visual Geometry Group) [17] is a deep CNN architecture with simple and effective stacks of 3×3 convolutional layers followed by max-pooling layers to reduce spatial dimensions.

The main variants are VGG16 and VGG19, with 16 and 19 layers respectively. Although the computational cost is larger, deeper variants are better at capturing complex features. The authors used VGG16 with over 134 million parameters. We think this is too much for our task. The training of this model is very long and costly. So, we used a simpler model, VGG11. It still has many parameters (128 million), but it is less complex and costly, and trains faster.

4.3.7 Ensemble model

Finally, we designed a max-voting ensemble model. We picked the best three models, got their predictions, and chose the final prediction by the class (disease) with the most votes.

Block (Conv-kernel_size)	in_ch, out_ch
Conv-3 + ReLU, BN, MP(2)	3, 16
Conv-3 + ReLU, BN, MP(2)	16, 32
Conv-3 + ReLU, BN, MP(2)	32, 64
Conv-3 + ReLU, BN, MP(2)	64, 64
(flatten the tensor)	
Linear + ReLU	$64 * 14 * 14$, 512
Dropout(p=0.5)	
Linear + ReLU	512, 4

Table 3. Our baseline CNN architecture. Each convolutional layer is followed by ReLU, BatchNorm, and MaxPool layers. The padding is set to 1 for all blocks, ensuring that the spatial dimensions of the feature maps remain the same after convolution.

5. Experiments

We used the Adam [8] optimizer with default parameters, the cross-entropy loss function, 25 training epochs, batches of size 64, and 8 as the patience value in the early stopping strategy to prevent overfitting, as in the original article.

We evaluated the models using the accuracy score (correct predictions over total predictions), confusion matrix (predictions vs. actual labels in a table), classification metrics (precision, recall, and f1-score), and visualization of wrong predictions.

The original work did not apply any fine-tuning or optimization techniques. This might be because the models performed very well. But we think that improving (even a bit) the models will make the project more complete and efficient in training time, accuracy scores, and graph curves. We will show the outcomes and fine-tuning methods that we used next.

5.1. Experimental Setup and Environment

We ran the experiments on Windows 11 with low-level hardware: an NVIDIA GeForce GTX 1060 6GB GPU, Intel Core i5-8600 CPU, and 16GB of RAM. We used *PyTorch* 2.1.2, *Torchvision* 0.16.2, and *pytorch-cuda* 12.1.

5.2. Optimization techniques

We saw that overfitting and unstable training were problems in most models. So we decided to apply the following techniques: dropout, weights initialization (He initialization) of the final dense layers, AdamW optimizer and the one-cycle learning rate scheduler (OneCycleLR).

The He initialization [19] is the best way to initialize neural network layers with ReLU activation. It initializes the weights of layers with values sampled from a Gaussian distribution with zero mean and variance based on the number of input neurons. This technique empirically shows meaningful improvement in training stability and speed (helping mitigate the vanishing and exploding gra-

Model	Accuracy
Our VGG11	0.9979
Our VGG16	0.9948
ResNet50	0.9927
Our Baseline CNN	0.9927
Our EfficientNetB1	0.9927
Our ResNet50	0.9927
Baseline CNN	0.9917
VGG16	0.9906
Our ResNet18	0.9896
MobileNetV2	0.9833
DenseNet121	0.9816
Our MobileNetV2	0.9812
Our DenseNet121	0.9795
EfficientNetB7	0.9656

Table 4. This table shows the accuracy score of all the original models and our models.

dient problems).

AdamW [10] is an improved version of the Adam optimizer, made to fix issues with weight decay regularization. In adaptive gradient algorithms like Adam, the regularization term is added to the cost function, changing the gradients and causing problems with the moving averages of gradients and squares. By separating the weight decay term from the gradient update, AdamW makes regularization better, preventing overfitting, improving generalization, and boosting model performance. We used the default parameters values and set weight decay to 0.01.

The One Cycle Learning Rate Scheduler [18] (OneCycleLR) changes the learning rate in a cycle during training. It starts with a low learning rate and increases it to a high learning rate over some epochs to explore the parameters. Then it decreases it back to a low learning rate over the rest of the epochs. At the end of training it lowers the learning rate even more to help the model converge to a better solution. *fast.ai* courses suggest using this scheduler with AdamW, so we did that. We needed to specify the maximum learning rate; we used 0.01 because it worked well, but we could have employed a learning rate finder to find the best value. We utilized the default 0.0001 for the minimum learning rate.

5.3. Results

Given the limited size of the datasets, the accuracy scores may not show how well the models generalize and different runs may give different results. But we will use these scores to compare the models, knowing their limitations in representing overall model effectiveness.

The results of the experiments are presented in tables 4, 6, 5.

The baseline CNN (Table 2) implemented in the original

Model	Wrong Predictions
Our VGG11	2
Our VGG16	5
ResNet50	7
Our Baseline CNN	7
Our ResNet50	7
Baseline CNN	8
DenseNet121	8
Our DenseNet121	8
VGG16	9
Our EfficientNetB1	10
Our ResNet18	10
MobileNetV2	16
Our MobileNetV2	16
EfficientNetB7	37

Table 5. This table illustrates for each model how many wrong predictions it made.

Model	Number of parameters
Our VGG16	134,285,380
VGG16	134,276,932
Our VGG11	128,788,228
EfficientNetB7	63,797,204
ResNet50	23,516,228
Our ResNet50	23,516,228
Our ResNet18	11,178,564
DenseNet121	6,957,956
Our DenseNet121	6,957,956
Our EfficientNetB1	6,518,308
Our Baseline CNN	6,485,956
MobileNetV2	2,228,996
Our MobileNetV2	2,228,996
Baseline CNN	364,580

Table 6. This table shows how many parameters each model contains.

project and translated into *PyTorch* achieved high accuracy on the test set. It also had the least parameters of any model. We attempted to improve the model even further by creating a new network (Table 3) with 3×3 filters and increasing the number of channels in each convolutional layer. We used dropout (0.5), He initialization, AdamW, and OneCycleLR to prevent overfitting and optimize training. We got better performance and faster training.

The original DenseNet121 model did worse than the baseline CNN because it overfitted the training data.We attempted to reduce overfitting using the aforementioned strategies, but we did worse. So we chose to simply use the He initialization to optimize training stability and generalization performance.

The original EfficientNetB7 model had about 63 million parameters. The model was too complex for the task. The training was very long, and the accuracy score was much lower than the other models. So we used a simpler model, EfficientNetB1. It had fewer parameters than B7 (about 7 million) and fit the task better. We used our usual optimization techniques (He initialization, AdamW, and OneCycleLR) and got a very high accuracy score, comparable to the original baseline CNN.

We trained the original MobileNetV2 model quickly, thanks to its simplicity and low number of parameters. We knew it wouldn't perform excellently, since its main advantage is being efficient with limited computational resources and memory. However, it also suffered from slight overfitting, which we couldn't fix much with our usual optimization techniques.

The original ResNet50 model (with about 24 million parameters) achieved a high accuracy score, similar to our baseline CNN. But it also had some overfitting issues, which we tried to solve with our standard optimization techniques. Then we switched to ResNet18, a variant with fewer parameters (11 million parameters). After applying the same optimizations, we got a model with half the parameters, which trained faster and only lost a little performance compared to ResNet50.

The original VGG16 model (with its 134 million parameters) was too slow and complex for our task and small datasets. We first tried the VGG16 variant with batch normalization layers and the He initialization, which improved the learning stability and the accuracy score, but not the training time. So we switched to the VGG11 variant, which still had 129 million parameters, but trained much faster and performed even better than VGG16. We didn't bother to apply our regular optimization methods, because we lacked the time and the resources to fine-tune these models.

5.4. Ensemble and final considerations

We used the test dataset to compare the accuracy scores of different models. We then created a max-voting ensemble model that combined our best architectures: VGG11, baseline CNN (our), and EfficientNetB1. This ensemble achieved an accuracy of 99.56% misclassifying only four images.

Most of the errors occurred when the models had to distinguish between Black Rot and ESCA. Sometimes, even humans can't tell them apart. We can say that the ensemble model is very reliable when it predicts "Healthy" or "Leaf Blight". But when it predicts "Black Rot" or "ESCA", there is a small possibility that it is wrong.

We also noticed that the ranking of the models based on accuracy scores and the number of wrong predictions was different. But this is normal, because accuracy scores measure the average correctness of the models. A model may have high accuracy because it is very good at predicting some classes correctly, but it may also make many mistakes on other classes. On the other hand, a model with lower accuracy may spread its mistakes more evenly across different classes, leading to fewer overall wrong predictions.

6. Conclusion

We wanted to examine, implement, and improve the *Kaggle* project linked with this paper [11], which was the only source of a dataset for our task, though very small and limited. The project compared various pre-trained deep-learning models for classifying four grape diseases (Black Rot, ESCA, Leaf Blight, and Healthy) using image classification.

We converted the code from *TensorFlow* to *PyTorch*, adding all the algorithms and functions that *PyTorch* didn't have by default. We also corrected the original dataset splitting and applied several optimization techniques to fine-tune the models.

Our task was simple, and a small CNN model performed well after a few training epochs. We experimented with new models with fewer parameters to speed up the training and inference. We also created a max-voting ensemble model from our top three architectures, which achieved very high accuracy, misclassifying only four images. However, all models struggled to distinguish between Black Rot and ESCA. Some images were indeed very difficult to classify, even for humans.

To test the feasibility and usefulness of these models for grape farmers and experts in real-world scenarios (possibly in real-time), we need more comprehensive and robust classifiers. However, the scarcity of data in this field limits the power of the models. We hope that researchers will collaborate in the future, even if this is unlikely, to expand the available datasets and create a new standard and large dataset that can benefit everyone. In deep learning, data availability is more important than the models, as it often acts as a bottleneck.

References

[1] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[2] Konstantinos P. Ferentinos. Deep learning models for plant disease detection and diagnosis. *Computers and Electronics in Agriculture*, 145:311–318, 2018.

[3] Alvaro Fuentes, Sook Yoon, Sang Cheol Kim, and Dong Sun Park. A robust deep-learning-based detector for real-time tomato plant diseases and pests recognition. *Sensors*, 17(9), 2017.

[4] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. YOLOX: exceeding YOLO series in 2021. *CoRR*, abs/2107.08430, 2021.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[6] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[7] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[8] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[9] Pushpa Lama. Grape_disease, 2023. https://www.kaggle.com/datasets/pushpalama/grape-disease.

[10] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. *CoRR*, 2017.

[11] Rajarshi Mandal. Vineyard vigilance: Harnessing deep learning for grapevine disease detection. 2023.

[12] Pranjali B. Padol and Anjali A. Yadav. Svm classifier based grape leaf disease detection. In *2016 Conference on Advances in Signal Processing (CASP)*, 2016.

[13] Acumen Research and Consulting. Wine market size, share, growth, trends, analysis and forecast 2022 to 2030, 2022. Accessed on February 7, 2024.

[14] RM1000. Augmented grape disease detection dataset, 2023. https://www.kaggle.com/datasets/rm1000/augmented-grape-disease-detection-dataset.

[15] RM1000. Grapevine disease detection: Model evaluation, 2023. Accessed on February 7, 2024.

[16] RM1000. Grapevine disease detection: Model training, 2023. Accessed on February 7, 2024.

[17] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2015.

[18] Leslie N Smith and Nicholay Topin. Super-convergence: Very fast training of neural networks using large learning rates. In *International Conference on Machine Learning*. PMLR, 2018.

[19] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*. PMLR, 2019.

[20] Chaoxue Wang, Yuanzhao Wang, Gang Ma, Genqing Bian, and Chunsen Ma. Identification of grape diseases based on improved yoloxs. *Applied Sciences*, 13(10), 2023.

[21] Xiaoyue Xie, Yuan Ma, Liu Bin, Jinrong He, Shuqin Li, and Hongyan Wang. A deep-learning-based real-time detector for grape leaf diseases using improved convolutional neural networks. *Frontiers in Plant Science*, 11, 06 2020.