Expertise
and insight
for the future

Ali Bahrami

Classification of invasive species in Finland with deep learning

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

13 April 2021

Author Title Number of Pages Date	Ali Bahrami Classification of invasive species in Finland with deep learning 40 pages + 3 appendices 13 April 2021
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technologies
Professional Major	Software Engineering
Instructors	Peter Hjort, Senior Lecturer

In this project, an application was developed in order to classify whether a plant is lupine, hogweeds or impatiens, based on the image of the plant. These plants are considered as invasive species in Finland. The first step in order to remove an invasive plant from the environment is to recognize it, either by a human or a machine. The aim of this project was to create an application that can, with deep learning, recognize a plant in an image as lupine, hogweeds or impatiens. The application was developed by experimenting with different parameters in the learning algorithm and by observing their effects on results.

The algorithm is based on deep learning techniques. The main language to develop the code was Python. Most of the code is based on PyTorch and fastai libraries and the implementations of artificial intelligence theories. Google Research Colab was used to do the experiments in the cloud. The application was deployed to Heroku.

The application can recognize invasive plants and the success rate of the test results is over ninety percent. Since recognition is the first and vital step to take in order to remove invasive plants from an environment, the results of this project can be applied to robotics engineering in the future.

In conclusion, it is possible to use deep learning in order to clean nature from invasive plants. This means that the next step for this project is to apply the learning model to robots. However, robotics regulations and the ethical aspects concerning the use of robots should be studied first.

Keywords	Deep learning, invasive plants

Contents

Appendices

List of Abbreviations

AI	Artificial Intelligence
DL	Deep Learning
CNN	Convolutional Neural Network
RESNET	Residual Neural Network
NN	Neural Network
ANN	Artificial Neural Network
RNN	Recurrent Neural Network

1 Introduction

In Finland, around one thousand species have been identified as alien species and among them are some harmful plant species, bringing risks to their environment. The plant species that affect their environment negatively are considered as invasive ones. Among these species, lupine (Lupinus polyphyllus), hogweeds (Heracleum mantegazzianum) and impatiens (Impatiens glandulifera) are among the top five invasive species in Finland and continue to invade the forests of Finland. (Yle, 2015.)

The giant hogweed, illustrated in figure 1, is a toxic and tall plant, which endangers both local species and humans. One needs to cover hands and eyes before getting close to it due to sever damages to skin. There is also a risk of getting blind. It can spread vastly, and there is a high chance that the dormant seeds germinate easily.

Figure 1. Hogweed. Copied from Rahjola (Yle, 2015).

The impatiens or, to be more precise, the Impatiens glandulifera, which is illustrated in figure 2, is native to Himalayas. When established in an area, it will eventually dominate the land as the single species. If it grows along water resources, it can endanger the fish population too.

Figure 2. Impatiens. Copied from Flinck (Yle, 2015).

Lupine (Lupinus polyphyllus) is a flowering plant with large leaves growing next to the roads and rivers. The plant, which is illustrated in figure 3, is toxic to domesticated animals and butterflies since it has toxic alkaloids. Efforts to remove it from urban areas in Finland have been ineffective and they continue to grow in large groups.

Figure 3. The herbaceous lupine. Copied from Heikkinen (Yle, 2015).

What is a threat to Finland's nature is not necessarily a threat in other parts of the world. For example, there are cases where the impatiens is cultivated for its flowers and lupine for its seeds. The seeds of the lupine, for instance, can be used as an alternative to soybeans. Eighty five percent of lupine in the world is produced by cultivation in Australia (Ross, 2011). Lupine seeds enrich the soil with nitrogen, and it resists pests better than soybeans. Moreover, in Finland there are some local products out of lupine, such as Jalotempe Lupiini. However, despite different types of solutions and money spent to fight against the uncontrolled spread of the lupine, it seems that the plant is winning the battle in nature.

The effort in this project includes methods to classify lupine, hogweeds, and impatiens and to take the first steps to protect the endangering Finnish nature. In order to use machines to remove invasive species, there should be a mechanism to recognize plants and classify them. Since this is a classification problem, deep learning techniques can be instrumented for recognition of these species. The results of this approach along with the code used to produce it are explained in detail in the following sections.

2 Theoretical background

Machine learning solutions have evolved historically by numerous scientists. The first attempt to create a model that represents neurons was done by Warren S. McCulloch and Walter Pitts. It is possible to use propositional logic to describe the relations

among neurons since neurons have “all-or-none” behaviors in their activities (McCulloch & Walter Pitts , 1943).

Mark I perceptron machine, invented by Frank Rosenblatt, was the first implementation of the perceptron algorithm he created, based on an artificial neuron model that had weight variables. In his work on algorithms, he defines a system from which simple patterns could be recognized without previous knowledge of the shapes. The proposed machine consisted of three systems: sensory, association, response systems. (Rosenblatt, 1957.).

One of the first efforts to solve a problem with learning was done by Arthur Samuel. In his paper, “Some Studies in Machine Learning Using the Game of Checkers” (Samuel, 1959), he verified two different procedures that could be described as the process of learning if done by humans or animals. In his paper, he proposes that such a system includes characteristics that might be enumerated. Samuel (1959) intorduces the characteristics in the following way:

> The activity must not be deterministic in the practical sense. A definite goal must exist. The rules of the activity must be definite, and they should be known. There should be a background of knowledge concerning the activity against which the learning progress can be tested. The activity should be one that is familiar to a substantial body of people so that the behavior of the program can be made understandable to them.

He concludes from the tests that an effective learning technique must include a procedure to give the program a sense of direction. It must also contain a refined system for cataloguing and storing information. This is what is later used as a data gathering process and labelling the data.

Samuel, in his second paper (Samuel, 1962) , not only tries to answer to already negative perceptions of machine learning but also clarifies elements of such a system. He uses previous studies of perception and the structure of human neurons as a base for his proposal. He defines the idea of machine learning as follows (Samuel, 1962):

> Suppose we arrange for some automatic means of testing the effectiveness of any current weight assignment in terms of actual performance and provide a mechanism for altering the weight assignment so as to maximize the performance. We

> need not go into the details of such a procedure to see that it could be made entirely automatic and to see that a machine so programmed would "learn" from its experience.

From the previously mentioned works, one can conclude that the architecture of such a program consists of weights, an update for them and a mechanism to test it. However, it was not any time before the 1980s advances in processing power that such an architect could be comprehensively evaluated. In the second chapter of Parallel Distributed Processing, the research group proposed a general framework that can act in the same way as the human brain to solve a natural processing problem. Such a framework consists of the following (Rumelhart, et al., 1986):

- a set of processing units
- a state of activation
- an output function for each unit
- a pattern of connectivity among units
- a propagation rule for propagating patterns of activities through the network of connectivity
- an activation rule for combining the inputs impinging on a unit with the current state of that unit to produce a new level of activation for the unit
- a learning rule whereby patterns of connectivity are modified by experience
- an environment within which the system must operate

Today, such a framework is the base of deep learning solutions. One of the main differences is the number of layers, instrumented to solve problems. Models with more than one layer can avoid an early obstacle, discovered by Marvin Minsky. The problem was that XOR could not be taught to perceptron with only one layer (Marvin Minsky & Seymour Papert, 1969.). Adding another layer could overcome part of the challenges, one faces in creating an intelligence system. However, to solve natural processing problems with machine learning, one needs to add multiple layers, or in other words, use deep learning models. A deep learning model consists of the following (Howard, 2020.):

- functional architecture which is the model's structure of layers of nodes
- the input data without labels or independent variables
- the output results or prediction, which is calculated from input

To create such a model, there are a few phases to pass and a few parameters to set. The phases are gathering input data for the model, training the model with a selected learning algorithm, and choosing the right parameters for it, optimizing, and troubleshooting the model and finally, deployment of results. These phases of development along with the parameters are explained in the following sections in detail.

2.1 Data gathering

The first step to take for training a model was to prepare the right input. Depending on the learning algorithm and development strategy, which are discussed in next sections, the input might contain correct labels too. In this project, the images of the species were correct input data, and the labels were the names of the species accordingly. Image files for the same species were gathered into the same directory and each directory had the species name for the images. The name of the directory was then used as a label for the group of image files that represented the same species.

There are other methods which could be used in more advanced cases, for instance it is possible to label each category of plants in more details (flower, leaf). If there are more details for each category a map file or a table could be used. An example of such a map is a table with a column for the path of image files and a column or more for labels of the data. Each row then connects an image file to one or more related labels.

The amount of data and the quality of it are equally important for the learning model in the training and testing phase. This is an easier step to take if the training model has already been developed in other projects or studies. For this purpose, the first action should be to search for existing datasets and reuse them for new models.

A classic example of an appropriate dataset for a study is The MNIST database of handwritten digits. The MNIST contains a training set of 60,000 examples, and a test set of 10,000 examples. (LeCun, 1998.)Searching for datasets in the Google dataset search engine and Kaggle are two good options to start with. For computer vision datasets, ImageNet is also a good source to navigate for data gathering.

Data augmentation

Despite all the available datasets for classification problems, like the one which is the topic of this work, one might need to either create datasets from scratch or add more data for an optimized training phase. If finding enough data for input is not possible, then the data gathering could be improved by using a technique called data augmentation.

The methods of creating alternative versions of gathered data in a way that the meaning of the data is preserved while its appearance for the model is different is called data augmentation. (Howard, 2020.) Data augmentation should be done with care, to have better training and to keep the dataset valid. There are a few methods to do such a procedure. However, all these methods have their pros and cons since any modification of images will reduce the quality or meaning of the image. Rotation, flipping, perspective warping, brightness changes and contrast changes are examples of changes to the original data.

2.2 Training

The second phase in the development of a DL model is to train it based on gathered data. The training phase consists of choosing the appropriate learning method with an appropriate structure. The weights in the model are set to receive the desirable output from the gathered data.

2.2.1 Learning types

The improvements in results from the DL model are related to how the model learns from the gathered data. There are four different types of learning: unsupervised, supervised, reinforcement and semi-supervised learning. The semi-supervised learning is the practical choice in many cases. (Russel & Norvig, 2010.)

Unsupervised learning is to learn without feedback or to understand patterns in input data. In reinforcement learning a group of actions is taken by the model and it receives a reward or punishment. The model will choose which actions were related to rewards

or punishments. Supervised learning is when the model has valid pairs of input and output data and the mapping function between them is found by algorithm. Finally, semi-supervised learning is to learn from some pairs of input and output data which might not necessarily be correct. The model continues to recognize patterns in the rest of the data, which is without feedback, or in other words, from data which is not labeled (Russel & Norvig, 2010.).

2.2.2 Artificial neural networks

Artificial neural networks (ANNs or simply NNs) are layers of artificial neurons that are connected to each other via some mathematical operations. They include an input and output layer. Between the input and output layers there are hidden layers. The neural networks vary in the usage and behaviour based on how the nodes are connected to each other and what kind of mathematical operation is used in each layer. A deep neural network is a neural network which consists of many hidden layers in between the input and output layer. The hidden layers are modified and controlled by the learning algorithm. (Goodfellow , et al., 2016.)

A neural network can pass on feedback from the output layer to the input layer too. This kind of an NN is called a recurrent neural network or RNN. If the connections between nodes do not form a cycle, or in other words, if there is no feedback from the output layer to the input layer, then there is a feed forward network. (Goodfellow , et al., 2016.)

A CNN is a type of deep feed forwarding neural network that has widely been used for image recognition since its introduction (LeCun, et al., 1989). The operation in hidden layers is based on convolution which is a mathematical operation. Hence, if any layer of DL model uses the convolution operation, then it is a CNN. (Goodfellow , et al., 2016.)

The convolution for functions f and g is:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

In signal processing the variable t is normally used to represent time. The first function or f can be results from a sensor and the function g could be the average value based on previously obtained results. In a CNN, function f is usually considered as input or a tensor (multidimensional array) of input data, while function g is the kernel, or a tensor of variables that change through the learning algorithm, and the result of the convolutional operation is a feature map (Goodfellow , et al., 2016.)

If the kernel width, or the number of variables in the kernel is less than the input, then the convolution operates on a subset of the input for each node and is called sparse interaction. Images consist of thousands of pixels while the feature map could be a small subset of the image. An example of this is the borders of shapes in an image which consists of considerably fewer pixels. Hence, the kernel or variables in a convolutional model could consist of much less variables. This is beneficial in the performance of an algorithm and the algorithm would need less memory (Goodfellow , et al., 2016.)

Another important feature of a CNN is parameter sharing. Since the kernel is a subset of input, the calculation of output (the multiplication of weights) is done for several outputs with the same kernel variables. The use of the same parameters in calculation is called parameter sharing (Goodfellow , et al., 2016.)

Equivariance is the third feature in a CNN, which means that the same type of changes in input will be represented in the output. For instance, shifting the input will appear as shifting in output. However, not all types of transforms are translated into output in a CNN. An example of these transforms is rotation of input or change in the scale. The transforms need some other algorithms along with a CNN to make them appear in the output (Goodfellow , et al., 2016.)

From the previous layer to the next layer, a CNN transforms input with a convolutional operation and creates a series of linear activations. Then a nonlinear activation is calculated for the values and the result is passed to a pooling function. The result from the pooling function is passed to the next layer. A pooling function can be described as follows (Goodfellow, et al. 2016):

> A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs.

There are different kinds of pooling functions. Max pooling or average pooling are examples of those. The goal of a pooling function is to keep the important features in the input data and disregard the ones the changes of which will not affect the output.

It was not obvious until a recent study that CNN is an appropriate solution for many classification problems. Each layer of the model can learn features from images on top of each other, and eventually have a perception of the object, which functions like the human eye. The CNN, in fact, resembles the animal's visual cortex behavior. (Zeiler & Rob Fergus, 2013.)

The depth (number of layers) of a CNN plays an important role in the level of output features (low/mid/high). Deep CNNs have been instrumented in image recognition for this purpose. However, when the number of layers increases, another issue appears in the CNN model. Adding more layers will eventually affect the accuracy and result in more training errors. Since this is not an overfitting problem, a modification in the CNN was introduced by the Resnet models.

Resnet stands for deep residual learning for image recognition. It instruments techniques to overcome a few common issues in deeper convolutional neural networks. The structure in Resnet consists of residual connections which create escapes on layers and pass values to the next layers of the network. (He, et al., 2015.) Resnet solves overfitting and vanishing gradients problems while it reduces the overall time spent on training. There are different Resnet models, each named with a number at the end, which represents the number of layers in each model.

2.2.3 Regularization

The goal of any deep learning model is not only an optimized behavior with the training data but also optimized results from any source of data including new and unseen data. This is referred to as the generalization ability or the capability of the model to generalize from learning to perform well on unseen data. Modifications that are done to the learning algorithms which results in fewer errors in generalization are called regularizations. (Goodfellow , et al., 2016.)

In practice, these modifications should affect the training errors less but errors during testing phase more. There are a few regularization strategies in DL models, including dropout, activity regularization, weight constraint, data augmentation and early stopping.

In data augmentation for this project, images were cropped and resized into multiple new images. Practically, the meaning of an image is the same even if it has been modified by cropping and shifting. This feature is available in CNNs that are used for this work and is part of regularization techniques. (Goodfellow , et al., 2016.)

Regularization techniques can also omit the overfitting problem in learning. Overfitting is the status of learning when the model memorizes unrelated aspects of input data as features of it and cannot make generalization about new data.

2.2.4 Hyperparameters

One common approach in different DL algorithms is to choose some parameters that are supposed to be set appropriately to gain optimized performance and results. These parameters are called hyperparameters. (Goodfellow , et al., 2016.) Hyperparameters can be set either manually or through automations (Marc Claesen & Bart De Moor, 2015).

The amount of training data processed by a learning algorithm before updating the model's parameters is called batch size. The number of times that a learning algorithm has processed the whole dataset of training is called an epoch. An epoch consists of one or more batches. Iteration indicates the number of batches to complete one epoch in training. (SHARMA, 2017.)

The performance of the classifier can be measured with quantitative measures. Accuracy is the ratio of correct results compared to the total number of results. Error rate is the ratio of wrong results compared to the total result. (Goodfellow , et al., 2016.) While the accuracy or error rate could give an insight of how the model is performing on each epoch or in total, the algorithm needs an indicator of how the update of weights has affected the predictions compared to the labels. An indicator, which is a function, is called a loss function. (Howard, 2020.)

Choosing a loss function depends on what kind of output the model creates. For this project as the outputs are categories, a cross entropy loss function is chosen. The cross entropy is defined in the following way (Goodfellow , et al., 2016):

> [T]he average number of bits needed to encode data coming from a source with distribution p when we use model q.

Reducing the loss function to minimum is a mathematical optimization problem. The slope of a function f(x) is optimal when the derivative of the function is equal to zero. Hence, the problem is to find how much should be added to x to reach to minimize f(x).

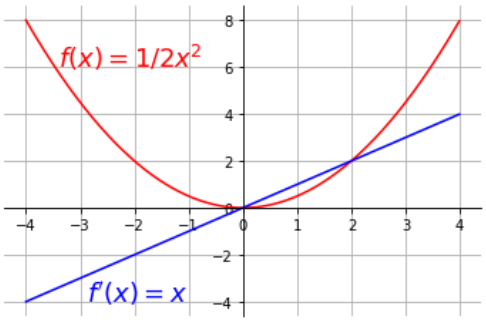

Figure 4. Function and its derivative

In figure 4, one can see that the minimum occurs at zero. If x<0, then the derivative of f(x) or f'(x) is negative, and to reach 0, small positive values should be added to x. On the other hand, if x>0, then the derivative is positive, and some negative value should be added to x to gain the minimum of function f(x). The technique or algorithm for calculating the sign and the amount of the step for x, to optimize f(x), is called gradient descent. (Goodfellow , et al., 2016.)

When there are multiple inputs, derivatives are calculated with respect to each input variable. The derivatives of multiple inputs are called a partial derivative. The Gradient is a vector of all the partial derivatives of a function. The minimum of a function occurs when the elements of a vector are all zeros. One denotation for gradient is $\nabla_x \mathrm{f}(x)$. The new x for a multiple input variable function f(x), when the results have a trend to a critical point, is evaluated with:

$$x_{new} = x - \epsilon \nabla_x \mathrm{f}(x)$$

In this formula, ϵ is the size of the step for updating x (weights of the model) in order to minimize the loss. In machine learning, ϵ is called the learning rate. (Goodfellow , et al., 2016.) The learning rate, in practice, is the vital hyperparameter in DL development and should be set appropriately in each algorithm. Since the manual task of finding the right learning rate is challenging, there have been some successful efforts in finding it via automations. For instance, there is a way to find a better learning rate for the model based on a study on Resnet networks. The proposed solution in the study was to find the possibilities for this parameter via one round of training. (Smith, 2015.)

Generalization error or test error is the number of errors for the model with new inputs. This is evaluated by using a different dataset from the training set, which is called the validation set. The validation set is used to evaluate hyperparameters and eventually the generalization error. (Goodfellow , et al., 2016.)

In practice, the division of the dataset into test, validation and train sets can be achieved by creating random subsets of the data. A common algorithm for creating these subsets is called k-fold cross-validation. This approach has better performance when the training dataset is relatively small. (Goodfellow , et al., 2016.)

2.2.5 Transfer learning

Many classification problems are already implemented by DL architectures. One way of benefiting from the architectures is to transfer the learning model and tune it for a new problem with an appropriate number of outputs. Tuning the model is the process of changing the structure of the last layer of the network based on the new inputs and new categories of results. Such a process is called transfer learning. (Howard, 2020.)

In the experimental part of this project, different Resnet models were fine-tuned, to have the new outputs, or categories, concerning this project. This method facilitated the experimental part as it was possible to concentrate on setting other parameters in the model, instead of creating the network from scratch.

2.3 Troubleshooting

There are many challenges in front of any deep learning model development. It has been a common experience for many to write code that has compilation errors or that gives wrong results. The model might also only work in specific cases. This poor performance in the model is due to four factors: implementation bugs, hyperparameter choices, model fit and dataset construction. (Tobin, 2021.)

The troubleshooting strategy should have a pessimistic approach. All aspects of the model should be restudied and considered as the source of the problem. Troubleshooting starts with investigating the input of the model or making sure the gathered datasets are appropriate. This step includes the verification of the quality and quantity of data as the structure of data might be one of the reasons for poor performances. Then the model's hyperparameters should be investigated and changed to achieve different results. (Tobin, 2021.)

The troubleshooting process should start with choosing the simplest possible model and the simplest implementation with the least values for batches and the least number of iterations. Then an evaluation of hyperparameters could be done for the small model. After achieving results without errors, the larger values for batches and epochs could be verified with increasing layers in the network. (Tobin, 2021.)

3 Tools and technologies

Along with all the tools that a developer uses for software development (programming languages, IDEs, and VS), two libraries were used for creating the model. PyTorch, developed by Facebook AI research team, is tensor library for deep learning. A PyTorch tensor is an n-dimensional array similar to NumPy array. The library not only provides mathematical functions for the tensors but also keeps important data such as gradients for deep learning development. (Johnson, 2021.)

The second library used was fastai. It is a Python library which has developed PyTorch concepts with a practical concept for deep learning developers. It provides all the

tools for developing a model. It is mostly used in Jupyter notebooks and it even has some features for presenting results (fast.ai, 2021). A lot of code in this work is based on methods in fastai and how to set the parameters. The fastai methods combined with PyTorch API facilitate focusing on technical details compared to development challenges.

Training the model on local environments needs expensive hardware and is a time-consuming process. Colab, Gradient and Sagemaker are good options for running Python code in the cloud. For instance, the code in this work has been tested and run in Colab by using Azure API for image grabbing.

Colaboratory, or "Colab" for short, is a cloud service by Google. Users can run Python code in Jupyter notebooks with free GPU and CPU options. It can be directly connected to Google drive too. (Google, 2021). Colab runs Jupyter Notebook with a free GPU and makes tests and tries very easy. One good feature is the direct access to GitHub to load notebooks from repository. The setup for the service does not support creation of graphical applications from the notebooks.

The code of this project is also available in the Lupo repository on GitHub (Bahrami, 2020). The Lupo code contains features to gather data and create an appropriate model for it. It is entirely written in Python.

The environment of Jupyter Notebook is not suitable for a production application. Voilà turns Jupyter notebooks into standalone web applications (QuantStack, 2019). The main purpose of Voilà is to transfer the interactive widgets of a notebook and disregard the other parts. For this project, due to some poor performance of Voilà in production, a new flask app was developed to use the trained model. (Bahrami, 2021.)

4 Strategy of development

In this project, the development was done from data gathering to deployment by experimenting different techniques in each phase and observing the results. Training was done through experimenting different hyperparameters and verifying their results. The goal

was to gain better performance and accuracy in each experiment compared to the previous one and eventually to deploy the model as a web application.

The development of the model started with the study of a simple classifier that could only recognize the input as one valid category among three categories. Then a model was developed that could predict if an input belongs to one of the three species or the input is not related to any of them. Different parameters in each model were changed to study their effects in the learning process.

5 Data gathering

5.1 Gathering images

For the species in this study, there was no previously available dataset that could be used. Hence, steps had to be taken to prepare a valid dataset containing images of the invasive plants for training. By searching for these species in search engines, one can web scrape from a good number of web results. Google, Bing and DuckDuckGo were tested for this purpose, and a repository with tools to use Bing API (Microsoft, 2021) and collect a dataset was developed (Bahrami, 2020). The tools in the repository contain basic web scraping methods and a hashing algorithm to remove duplicates and faulty images from results since web search contains all sort of results.

However, with later development of the model, the methods of fastai were also tried successfully to gather data. There were two reasons behind choosing fastai as a base for developing the model. The first reason was the availability of the results in the Jupyter notebook for the model. The second reason was the availability of data augmentation methods, which could give more data for training compared to the original collected results as the original collected images were multiplied.

```
from time import sleep
invasive_plant_types = {
    "Lupine": [
        "Lupinus polyphyllus",
        "Komealupiini",
        "lupiini",
        "Lupine",
```

```
    ],
    "Hogweed": [
        "Heracleum mantegazzianum",
        "Kaukasianjättiputki",
        "Hogweed",
    ],
    "Impatiens": [
        "Impatiens glandulifera",
        "Jättipalsami",
    ]
}

if not path.exists():
    path.mkdir()
    for item in invasive_plant_types.items():
        directory_name = item[0]
        destination_directory = (path/directory_name)
        destination_directory.mkdir(exist_ok=True)
        for entry in item[1]:
            results = search_images_bing(key, f'"{entry}"')
            download_images(destination_directory,
                            urls=results.attrgot('contentUrl'))
```

Listing 1. Python script for downloading images from the Bing engine copied from Howard (Howard, 2020).

After downloading the images, if the service in use is Colab, it is wise to move the files to a shared drive (for example Google Drive) to keep the files for later use. In addition, it is important to verify the files for similar or faulty ones.

```
# clean paths from failed cases
failed = verify_images(fns)
failed.map(Path.unlink);
```

Listing 2. Python script to verify and remove faulty images copied from Howard (Howard, 2020).

Another method used to clean the images was to check and remove similar photos by a hash algorithm. This will to some extent avoid the overfitting issue in the training phase.

```
# bypass limitation of search API by introducing relative keywords
def make_reference_hashes():
    print_status("referencing")
    reference_files = os.listdir(output_directory)
    for ref_file in reference_files:
        img = load_image(output_directory+ref_file)
        if not img:
            continue
        img_hash = imagehash.average_hash(img)
        reference_image_hashes.update({img_hash:ref_file})


def mark_duplicates():
```

```
    print_status("marking")
    input_files = os.listdir(input_directory)
    for in_file in input_files:
        img = load_image(input_directory + in_file)
        if not img:
            continue
        img_hash = imagehash.average_hash(img)
        ref_data = reference_image_hashes.get(img_hash, None)
        if not ref_data:
            shutil.move(input_directory + in_file, output_directory +
str(img_hash) + in_file )
            reference_image_hashes.update({img_hash:in_file})
        else:
           print_status(f"Image {in_file} is not unique. duplicate of
{ref_data}")
```

Listing 3. Python script to verify and remove similar images copied from Bahrami (Bahrami, 2020).

Finally, the gathered images could still have wrong labels and include images that are not at all related to the categories. API calls return related and unrelated results, and a simple (and faulty) training model can facilitate in determining the wrong images to complete the data cleaning phase. Along with the semi-trained model, one should verify images and clean the data from any unrelated ones that remain at the end. This process needs prior knowledge of the species and their visual features.

5.2 Creating DataLoaders

The first step in creating the cleaning phase model is to define an instance of DataLoaders. DataLoaders is a fastai class which stores DataLoader objects and contains the valid and train sets (Howard, 2020.). DataLoader, itself, is a PyTorch utility class which (PyTorch, 2021):

> combines a dataset and a sampler and provides an instance of Iterable over the given dataset.

To create DataLoaders, a DataBlock class was used. To create Datasets and DataLoaders, one should define a DataBlock. An example of such a definition is in listing 4.

```
invasive_plants = DataBlock(
    blocks=(ImageBlock, CategoryBlock),
    get_items=get_image_files,
```

```
    splitter=RandomSplitter(valid_pct=0.2, seed=42),
    batch_tfms=aug_transforms(),
    get_y=parent_label,
    item_tfms=RandomResizedCrop(224, min_scale=0.5))
dataloaders = invasive_plants.dataloaders(path)
```

Listing 4. Python script to define aDataBlock and DataLoaders copied from Howard (Howard, 2020).

Blocks defines the independent variables as image files, and the dependent variables, labels or targets as categories (the type of plants). Items are retrieved via the get_image_files call, and then images are selected with a random seed and a preservation of twenty percent of the whole data for the valid set. This means that twenty percent of all images will never be used for training the model. To set the labels, using the directory name where they are stored, a method parameter, get_y, is set to parent_label. Item transform (item_tfms) is a method that processes each individual piece of data (in this case an image) with the method defined for it. Here the method passed as a parameter does a random crop over each image and resizes the result to a 224-pixel square.

5.3 Training

After defining dependent and independent variables and the images used for the model, a learner is called to use them for training. This is where a trained model will be used and adjusted for the new data. The term for this operation is transfer learning.

```
learner = cnn_learner(dataloaders, resnet18, metrics=error_rate)
learner.fine_tune(3)
```

Listing 5. Python script to train a CNN model copied from Howard (Howard, 2020).

The call for cnn_learner creates a convolutional neural network (CNN) with the DataLoaders defined earlier. A resnet18 architecture and metrics of choice (here error_rate) were defined as the parameters of the learner. For the cleaning model, a resnet18 is chosen so the overall cleanup phase would take less time than the real training phase.

Fine tuning is where the transfer learning takes place, with the number of epochs for the process. This call will change the weights of the model, to adjust with the new data introduced to it. As the goal of this phase is just to help the cleaning phase, the results from

the model do not need to be optimized. Hence, the number of epochs could be a range of numbers that reduces errors to some extent, with the minimum training time.

5.4 Results

The following table represents the results of three epochs of training. The metrics defined in the process represent train and valid loss and an erorr_rate. These values are numbers between 0 and 1. The results have an error rate of 0.25 which is of course not the final goal of the classification but appropriate enough to help with the cleaning phase.

Table 1. Learner results with resnet18

epoch	train_loss	valid_loss	error_rate	time
0	0.554928	0.742238	0.285714	00:21
1	0.436820	0.719783	0.250000	00:20
2	0.360749	0.721695	0.250000	00:20

The learner model will then be used to clean and classify the rest of the images, by calling an ImageClassifierCleaner. ImageClassifierCleaner is an interface which loads the images of the trained and valid model with some functionalities to change their categories. After each call and assigning the images that are in the wrong category or removing the ones which are not at all related to the models, one can unlink them from the paths of images.

```
cleaner = ImageClassifierCleaner(learner)
for idx in cleaner.delete():
    try:
      cleaner.fns[idx].unlink()
      cleaner.fns.remove(cleaner.fns[idx])
      print(f"cleaned {idx}")
    except:
      print(f"index issue {idx}")
```

Listing 6. Cleaning the images after the first training copied from Howard (Howard, 2020).

However, investigating the files thoroughly is inevitable as this tool is limited to a few cases in each training. For example, if the images are too small or blurry, they will lead

to a poor performance in training if augmentation techniques are applied to them. After all, it is necessary to remove low resolution and small images. Another check point is if the images are representing one plant at a time. As the model is not multi-labeled at this stage, it will help the training model to classify correctly.

6 Training the model

The next step after the first (or more) rounds of data cleaning is to adjust the model for better learning. There are a few aspects that should be noticed for proceeding in this phase: the choice of model, the amount of data for it and at least the learning rate to use.

The number of images for the learner is limited by the API to around 150 images per category after all cleaning and multiple filtering of them for each category (the total number of images after multiple API calls and clean-up is 485). The first experiment is to train the model without changing the input.

```
invasive_plants = DataBlock(
    blocks=(ImageBlock, CategoryBlock),
    get_items=get_image_files,
    splitter=RandomSplitter(valid_pct=0.2, seed=42),
    get_y=parent_label,
    item_tfms=Resize(224)
    )
learner = cnn_learner(dataloaders, resnet18, metrics=error_rate)
learner.fine_tune(8)
```

Listing 7. DataBlock for training with the calls to train the model copied from Howard (Howard, 2020).

The results for the experiment are in the following table. The summary of DataLoaders and the model used for this training is in appendix 1. This summary is important to be verified before training to avoid later troubleshooting issues. The loss function used by the algorithm is Cross Entropy Loss which was explained in the previous sections.

Table 2. Training results without augmentation

epoch	train_loss	valid_loss	error_rate	time
0	0.389732	0.405033	0.206186	00:20

epoch	train_loss	valid_loss	error_rate	time
1	0.340786	0.350985	0.164948	00:20
2	0.268097	0.349396	0.134021	00:20
3	0.197201	0.390821	0.123711	00:20
4	0.154201	0.452041	0.154639	00:20
5	0.128261	0.485184	0.216495	00:20
6	0.106235	0.521717	0.247423	00:21
7	0.090579	0.539667	0.247423	00:20

From the table it is obvious that despite the reduction of training loss, has not been learning since the fifth epoch. Not only the overall performance of the model is poor at this stage, but it also looks like overfitting is occurring due to the rise of valid_loss and error_rate, despite the reduction of train loss. The confusion matrix is described in the following figure. The figure shows where the model made the least accurate prediction for each category.

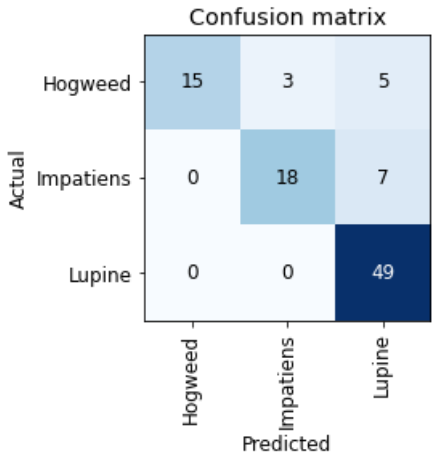

Figure 5. Confusion matrix

One way to multiply the number of images, is to use data augmentation in DataLoaders. There are different functions to use for data augmentation and here a padding with zero was chosen. The images for this second experiment were resized to 460 pixels.

```
invasive_plants = DataBlock(
 blocks=(ImageBlock, CategoryBlock),
 get_items=get_image_files,
 splitter=RandomSplitter(valid_pct=0.2, seed=42),
 get_y=parent_label,
 item_tfms=Resize(460,  ResizeMethod.Pad, pad_mode='zeros'),
 batch_tfms=aug_transforms(size=224, min_scale=0.2)
 )

learner = cnn_learner(dataloaders, resnet34, metrics=error_rate)
learner.fine_tune(10)
```

Listing 8. The transform augmentation parameters and learning call copied from Howard (Howard, 2020).

One good method of augmentation is to resize the image to a large scale and crop randomly and resize to smaller size again. However, any augmentation method affects the quality or meaning of the image to some extent. For example, it might be very challenging if there is a landscape image of a small flower like impatient and unrelated parts of it are cropped for training.

As shown in the listing 8, after resizing the image to 460 pixels, augmentation takes place, and the final size of the image will be 224 pixels. In addition, for this phase, one can choose a more layered model like resnet34 instead of resnet18.

Table 3. Results with data augmentation and resnet34

epoch	train_loss	valid_loss	error_rate	time
0	0.564891	0.419421	0.164948	00:28
1	0.461261	0.353335	0.164948	00:28
2	0.328840	0.324274	0.113402	00:28
3	0.265119	0.395922	0.092784	00:28
4	0.216781	0.383086	0.103093	00:27
5	0.177755	0.335392	0.113402	00:28
6	0.153971	0.317839	0.113402	00:28
7	0.156042	0.287527	0.092784	00:28
8	0.137733	0.284104	0.092784	00:28

epoch	train_loss	valid_loss	error_rate	time
9	0.125611	0.291761	0.092784	00:28

The summary of how the input data was defined is in appendix 2. Based on the results, it is obvious that the results have some improvement but still one can at least change the learning rate to obtain better performance.

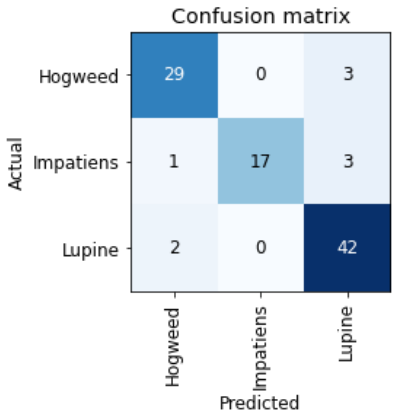

Figure 6. Confusion matrix

As discussed earlier, it is possible to find better learning rates automatically via one round of training.

```
lr_min, lr_steep = learner.lr_find()
print(f"Minimum/10: {lr_min:.2e}, steepest point: {lr_steep:.2e}")
```

Listing 9. Call for finding learning rate copied from Howard (Howard, 2020).

The results show that the model is learning between 1e-4 and 1e-1; hence, the model might achieve better accuracy by choosing a learning rate in between the values.

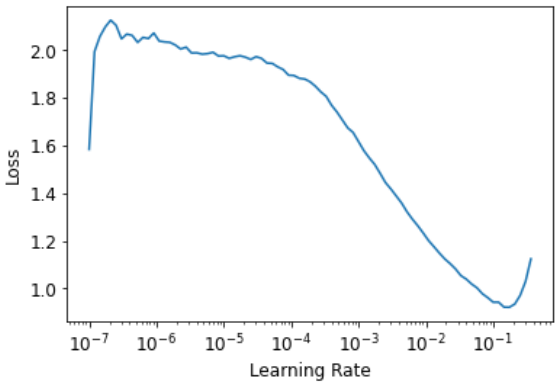

Figure 7. Learning rate with minimum/10: 1.74e-02, steepest point: 2.09e-03

Considering the logarithmic nature of the plot in figure 8, and 1e-1 being very close to the maximum rate for learning, a mean value between 1e-4 and 1e-1 is calculated. Next the model was trained with this base.

```
learning_rate_start=1e-04
learning_rate_end=0.5e-01
mean_value_for_learning_rate = math.exp((math.log(learn-
ing_rate_start)+math.log(learning_rate_end))/2)
learner.fine_tune(15,base_lr=mean_value_for_learning_rate)
```

Figure 8. Mean value for learning rate is 2.2e-3 copied from Howard (Howard, 2020).

The results have more improvements compared to previous trainings, considering the valid loss and error rates.

Table 4. Results with a specific learning rate

epoch	train_loss	valid_loss	error_rate	time
0	0.542325	0.414703	0.185567	00:27
1	0.418154	0.321694	0.154639	00:27
2	0.369949	0.320884	0.154639	00:27
3	0.301590	0.223288	0.113402	00:27
4	0.241585	0.241257	0.103093	00:27
5	0.197737	0.267305	0.134021	00:27

epoch	train_loss	valid_loss	error_rate	time
6	0.166188	0.167674	0.082474	00:27
7	0.143127	0.170643	0.072165	00:27
8	0.119068	0.178128	0.092784	00:28
9	0.101612	0.170208	0.103093	00:27
10	0.092394	0.164317	0.072165	00:28
11	0.080940	0.178523	0.072165	00:27
12	0.072939	0.171817	0.061856	00:27
13	0.067267	0.183079	0.082474	00:27
14	0.066202	0.178010	0.082474	00:27

Considering the metrics in the table, one can conclude that epoch 10 provides the best results on validation sets while the error rate is 0.07 which is better than in the previous experiment. Increasing epochs to 20 resulted in higher error rates and validation loss in another experiment that is not described in this work.

In the next experiment, the aim is to change the values of the last layer of the model for some rounds. Then based on the calculated learning rate and an early stop strategy, a second round of training is done by unfreezing all layers. The values for the learning rate were calculated from the first phase of training and they are shown in the following figure.

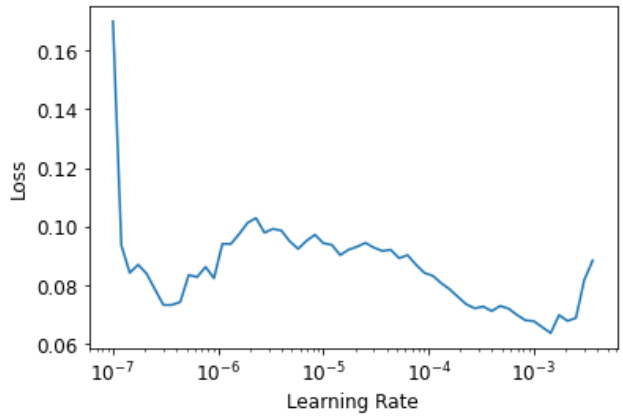

Figure 9. Calculation of learning rate before unfreezing the layers

One can see from the figure 10, that learning rate does not change in between 10e-6 and 10e-3 in a desired way. Hence the avarage and a max value of learning rate is calculated in the listing 10.

```
learner = cnn_learner(dataloaders, resnet34, metrics=error_rate)
learner.fit_one_cycle(4, lr_max= 2.2e-3)
learner.unfreeze()
learner.fit_one_cycle(20,
                      lr_max=slice(1e-5,1e-3),
                      cbs=EarlyStoppingCallback(moni-
tor='valid_loss', min_delta=0.01, patience=3))
```

Listing 10. Running the model with dynamic learning rate and early stop copied from Howard (Howard, 2020).

The result from listing 10 is shown in table 5. In this table one can see that in the first four epochs the model was run with a maximum learning rate of 2.2e-3.

Table 5. Training with dynamic learning rate and Resnet34

epoch	train_loss	valid_loss	error_rate	time
0	1.598841	0.693668	0.257732	00:26
1	1.050845	0.436337	0.144330	00:26
2	0.750245	0.412175	0.175258	00:27
3	0.571162	0.428932	0.185567	00:26
epoch	**train_loss**	**valid_loss**	**error_rate**	**time**
0	0.166017	0.405545	0.175258	00:26
1	0.150558	0.403779	0.175258	00:27
2	0.154601	0.393397	0.164948	00:27
3	0.152246	0.369193	0.164948	00:27
4	0.133295	0.767276	0.216495	00:27
5	0.116160	0.763293	0.195876	00:27
6	0.114661	0.596657	0.237113	00:27

However, the results from this experiment do not show improvements in error rate and valid loss. To have progress in the results, another set of experiments were done with new data, gathered from different sources and processed with tools written for this project. This is discussed in the following sections.

7 Improvements in training

7.1 Data gathering

As the amount of data in the training set is one important factor in learning, other ways to retrieve appropriate data could be used. One option is to use the DuckDuckGo search engine and Lupo tools to clean the final results.

```
def search_images_ddg(term, max_images=200):
    "Search for `term` with DuckDuckGo and re-
turn a unique urls of about `max_images` images"
    assert max_images<1000
    url = 'https://duckduckgo.com/'
    res = urlread(url,data={'q':term})
    searchObj = re.search(r'vqd=([\d-]+)\&', res)
    assert searchObj
    requestUrl = url + 'i.js'
    params = dict(l='us-
en', o='json', q=term, vqd=searchObj.group(1), f=',,,', p='1', v7exp='a')
    urls,data = set(),{'next':1}
    while len(urls)<max_images and 'next' in data:
        try:
            data = urljson(requestUrl,data=params)
            urls.update(L(data['results']).itemgot('image'))
            requestUrl = url + data['next']
        except (URLError,HTTPError): pass
        time.sleep(0.2)
    return L(urls)

for item in invasive_plant_types.items():
    directory_name = item[0]
    destination_directory = (path/directory_name)
    destination_directory.mkdir(exist_ok=True)
    counter = 0
    for entry in item[1]:
        urls =  search_images_ddg(entry)
        print(len(urls))

        for url in urls:
          img_name = str(counter) + ".jpg"
          try:
            download_url(url, destination_directory/img_name)
          except:
            pass
          counter = counter + 1
```

Listing 11. Code to retrieve images from DuckDuckGo (Howard, 2020).

Through this phase, about three thousand images were retrieved and cleaned up for training. The images were verified for representing each species close enough to camera and for having dimensions higher than the random crop size used for DataLoaders.

```
def parent_label_multi(o):
    return [Path(o).parent.name]
invasive_plants = DataBlock(
    blocks=(ImageBlock, MultiCategoryBlock),
    get_items=get_image_files,
    splitter=RandomSplitter(seed=57),
    get_y=parent_label_multi,
    item_tfms=Resize(460),
    batch_tfms=aug_transforms(size=224, min_scale=0.75))
dls = invasive_plants.dataloaders(path)
learner = cnn_learner(dls, resnet34, metrics=partial(accu-
racy_multi, thresh=0.2))
```

Listing 12. Defining multicategory dataset copied from Howard (Howard, 2020).

Here a different approach for dataloaders was chosen. The output or target of the model is not one label but multiple labels. The reason behind this change is to consider cases in which more than one of the species is present in the image or the case that none of them exist in the image. The result is an array of probabilities of all labels. Such a result can be filtered with a final function with different thresholds.

The first experiment is done with the unfreezing approach without previous training.

```
learner = cnn_learner(dls, resnet34, metrics=partial(accuracy_multi),pre-
trained=False)
learner.fit_one_cycle(10, lr_max= 1e-4)
learner.unfreeze()
learner.fit_one_cycle(20,
                      lr_max=slice(1e-5,1e-3),
                      cbs=EarlyStoppingCallback(moni-
tor='valid_loss', min_delta=0.01, patience=3))
```

Listing 13. Learning with a non-pretrained model copied from Howard (Howard, 2020).

It is important to note that the values for the learning rate were calculated by lr_find as shown in the next figure.

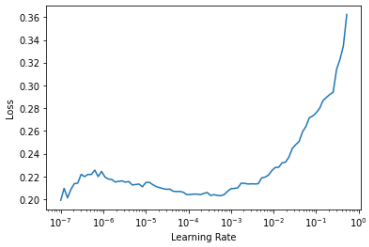

Figure 10. Learning rate finder results

From the next table one can conclude that more learning epochs are needed to reach more accuracy if the model is not trained earlier. In addition, it seems that the model gets overfit in the ninth epoch, since the early stop was triggered and stopped the algorithm.

Table 6. Results from none-pretrained model with early stopping

epoch	train_loss	valid_loss	accuracy_multi	time
0	1.023367	0.727853	0.552661	01:51
1	0.973791	0.679177	0.601468	01:52
2	0.901394	0.644051	0.626789	01:51
3	0.847724	0.638964	0.652844	01:49
4	0.795826	0.568082	0.724037	01:48
5	0.761432	0.533857	0.725138	01:50
6	0.719217	0.512053	0.746789	01:49
7	0.690814	0.514395	0.751193	01:50
8	0.677926	0.499088	0.760000	01:49
9	0.666584	0.499246	0.763303	01:50

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.644835	0.493133	0.777982	01:48
1	0.636601	0.476382	0.802569	01:48
2	0.620049	0.454817	0.824220	01:47
3	0.597268	0.444143	0.844037	01:47

epoch	train_loss	valid_loss	accuracy_multi	time
4	0.561061	0.364894	0.909725	01:47
5	0.510549	0.317716	0.926972	01:47
6	0.442570	0.274310	0.922569	01:48
7	0.384005	0.282441	0.905688	01:47
8	0.324080	0.201883	0.927706	01:47
9	0.278073	0.177700	0.931743	01:48
10	0.250769	0.177007	0.926973	01:48
11	0.227466	0.169773	0.930275	01:49
12	0.212655	0.187240	0.926239	01:49

In second experiment, a range of dropout probabilities was added to the layers of the model by using the ps parameter.

```
learner = cnn_learner(dls, resnet34, metrics=partial(accu-
racy_multi),ps=[0.001,0.01],pretrained=False)
learner.fit_one_cycle(10, lr_max= 3e-4)
learner.unfreeze()
learner.fit_one_cycle(40,
                      lr_max=slice(1e-5,1e-2),
                      cbs=EarlyStoppingCallback(moni-
tor='valid_loss', min_delta=0.01, patience=3))
```

Listing 14. Learner with dropouts copied from Howard (Howard, 2020).

The overall results are shown in the next table. The maximum learning rate for the first 10 epochs was 3e-4.

Table 7. Results from training with dropouts

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.785463	0.773172	0.496881	02:10
1	0.698718	0.695730	0.553028	01:23
2	0.610403	0.512827	0.795963	01:22
3	0.533659	0.440671	0.863119	01:22

epoch	train_loss	valid_loss	accuracy_multi	time
4	0.471950	0.398834	0.899450	01:23
5	0.418938	0.357665	0.928440	01:22
6	0.374647	0.386928	0.909725	01:23
7	0.338999	0.285277	0.939817	01:22
8	0.316869	0.267271	0.948624	01:22
9	0.298119	0.272708	0.950459	01:22

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.270823	0.238954	0.943119	01:22
1	0.233599	0.201863	0.927339	01:22
2	0.196451	0.144600	0.946055	01:23
3	0.167771	0.171031	0.935780	01:23
4	0.158691	0.148840	0.939450	01:22
5	0.161442	0.172049	0.929908	01:22

From the table, it is obvious that the model has some improvement compared to the previous experiment. However, the accuracy could have been better if the model had had an early stop. Hence, more regularization techniques could be experimented with. One parameter to experiment with in this section is weight decay.

```
learner = cnn_learner(dls, resnet34, metrics=partial(accu-
racy_multi),ps=[0.001,0.01],wd=1e-1,pretrained=False)
learner.fit_one_cycle(10, lr_max= 3e-4)
learner.unfreeze()
learner.fit_one_cycle(40,
                      lr_max=slice(1e-5,1e-2),
                      cbs=EarlyStoppingCallback(moni-
tor='valid_loss', min_delta=0.01, patience=3))
```

Listing 15. Learning with weight decay copied from Howard (Howard, 2020).

The results are seen in the following table. Maximum learning rate for the first 9 epochs was 3e-4.

Table 8. Results from training with weight decay

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.786125	0.741409	0.526972	01:22
1	0.703438	0.583544	0.686239	01:23
2	0.622131	0.505232	0.784220	01:23
3	0.547986	0.438360	0.874862	01:22
4	0.478398	0.389272	0.914128	01:23
5	0.428087	0.366146	0.910459	01:22
6	0.381806	0.333397	0.926973	01:22
7	0.343448	0.300285	0.938349	01:23
8	0.321054	0.275058	0.951193	01:22
9	0.303716	0.270988	0.949358	01:23

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.271678	0.205989	0.950826	01:23
1	0.229857	0.152087	0.950459	01:23
2	0.188903	0.131500	0.949358	01:23
3	0.170305	0.199249	0.916330	01:23
4	0.162861	0.174087	0.938349	01:23
5	0.162629	0.174415	0.935780	01:22

From the results, it seems that the best performance of the model is at epoch 2, and the change of weight decay parameter did have some minor improvement on the results. In the following experiment, a trained model is used with the fine-tuning technique and a new data loader.

```
def parent_label_multi(o):
    return [Path(o).parent.name]
invasive_plants = DataBlock(
    blocks=(ImageBlock, MultiCategoryBlock),
    get_items=get_image_files,
```

```
    splitter=RandomSplitter(seed=57),
    get_y=parent_label_multi,
    item_tfms=Resize(460),
    batch_tfms=aug_transforms(size=224, min_scale=0.75))
dls = invasive_plants.dataloaders(path)
learner = cnn_learner(dls, resnet34, metrics=partial(accu-
racy_multi, thresh=0.2))
learner.fine_tune(5, base_lr=3e-3,
                  cbs=EarlyStoppingCallback(moni-
tor='valid_loss', min_delta=0.01, patience=3)
)
```

Listing 16. Learner with pretrained model copied from Howard (Howard, 2020).

The results are seen in the following table. The learning rate was 3e-3.

Table 9. Results of trained model

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.413757	0.271777	0.630826	01:24
1	0.330806	0.184023	0.825321	01:23
2	0.243575	0.106881	0.965872	01:23
3	0.160268	0.068783	0.979083	01:23
4	0.113618	0.065171	0.975413	01:22
5	0.076197	0.065915	0.979083	01:23
6	0.052048	0.054003	0.982385	01:23
7	0.036738	0.052650	0.980917	01:23
8	0.028893	0.056828	0.982018	01:23
9	0.023527	0.055726	0.980917	01:23

As shown in the table, the accuracy of the model has increased significantly with taking new steps in data gathering and using multicategory. Epoch 6 is where the model has performed best. The prediction might increase if better threshold values for tuning are found.

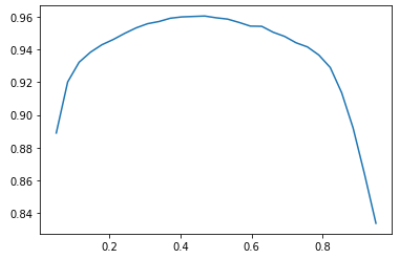

Figure 11. Finding better prediction threshold value

From the figure, one can conclude that taking a threshold of around 0.5 could increase the accuracy in the tuning process. However, one should consider the overfitting problem when changing the values of the threshold. If the train set consists of few sample data, it could eventually be a case of memorizing instead of learning, if the threshold is of high values.

From the structure of the model (appendices 1 and 2), one can conclude that the changes in the parameters affect the results in each experiment. For instance, the dropout parameter for the trained model is [0.25, 0.5], compared to the nontrained one which was [0.001, 0.01]. Changing the parameter in the model could also be a subject of new experiments.

8 Inference results for an online application

One of the methods to use the model in an application is to deploy the repository to Heroku. For this purpose, a small web application was developed in the lupo-app repository. (Bahrami, 2021.) The application which is deployed to Heruko is accessible from lupo-projekti.herokuapp.com. Since Heruko loads the environment and builds the image of the application, it is slow and just used for the purpose of demonstrating this project.

Inference is the process of using a deep learning model in an application. The application is just a simple representation of how the model is trained. However, for future of this project, there are more steps to take. Among them is to use the model in a drone.

9 Further implementation

Reaching higher accuracy in classifying plants can be achieved by expanding the labels in the model. A research on classification of plants by the entire plant, a flower frontal and lateral view, and a leaf top and backside view proves an increase level in accuracy compared to a model trained with entire plant images. (Rzanny, 2019.) For creating such a model, one should prepare five different packages of photos from each species, and each package should at least contain forty photographs. The images in the mentioned research were taken by Flora Capture, a smart phone application which is used for plant structure studies. (Floraincognita, 2021.)

In order to use the results of this study, one should take photos of the three species with the application and train the model likewise. Such a step can be taken during summer when the plants widely grow in nature. The trained model then might work better in a real environment since it can recognize species in their early stages, while growing and before seeds or even flowers.

Another round of experiments is planned for this project to take place by instrumenting continual learning. Continual learning is an effort to create better generalization results by changing the pipeline from the classic “training to deployment” to the instrumentation of generative models. (Lesort, 2020.) Since the subjects of this work are species in the nature, and the model is trained with datasets of static images, such an approach might improve the performance of the model in a real evolving environment.

Finally, the trained model will be instrumented for robotics engineering. Using drones for recognizing the invasive species is part of this phase. However, using real drones depends on many factors in the real environment. It is time consuming, expensive. Furthermore, the experiments cannot be fully supervised if real drones are used in the real world.

One approach to reach the goal of instrumenting drones is to use a simulation of flight in a forest environment. AirSim is a simulator for drones, cars and more, built on Unreal Engine. (Microsoft, 2021.) The goal of the project is to provide it as a platform for AI research to experiment with deep learning.

10 Conclusion

The invasive plants are threats to the nature. Three examples of the most invasive plants in Finland are lupine, hogweeds and impatiens. Removing them with the basic farming tools is a time-consuming and expensive process. Using automated machines to remove the invasive plants is only possible if the machines can recognize the plants correctly. Hence, the first step is to recognize and categorize the invasive plants.

This thesis describes an application / In this project/study, an application was developed that can recognize the mentioned plants in a given image. The application is deployed on the web for demonstration purposes. The application can also recognize if a plant belongs to none of the invasive plants, mentioned in this thesis. The overall training and test results are over ninety percent.

In order to develop the application, data was gathered from different sources and cleaned with automation and manually. The data was gathered by using different APIs. Then the results were cleaned first by comparing them with hash codes and testing if the images are without errors. After cleaning up similar or erroneous images by the script, a simple model was trained to categorize them. While the model was not trained well enough to be used for production, it could categorize most of the images in a good manner and save time. Finally, the results from the automated data gathering was verified manually to clean the datasets completely.

In the training phase, various techniques in training were tested to reach better results or observe their effects. It was obvious from the results that the most important parameter to set was the learning rate. Training on a pretrained model was also helpful and allowed more tests to be done in shorter periods.

The results were deployed to web as an application to test and demonstrate recognition of invasive species. First, a graphical version of the notebook was deployed. There were some incompatibilities between the libraries, and the user interface in the application did not work as it was expected. Hence, for using the trained model, another Python application was written and deployed successfully.

Image classification has been the subject of many academic and industrial projects. There are numerous ways to develop an application based on the already available results. However, there are many challenges in front of a developer to choose the right technology and many decisions to make before development starts.

The first challenge in this project was to create appropriate input data for training. Despite all the available tools, none of them was sufficient and ready to be used. Therefore, a set of tools was developed to gain images and clean the data from unrelated ones. Several hours were spent only on preparing data by manual verification of images. This process was repeated later, after the first data gathering failed to produce acceptable results for production purposes.

Experimenting with different methods on input data was challenging due to the limited free Colab environment used in this project. The GPU of the environment was not always available, and several disconnections could make an experiment fail. Consequently, the learner data of the model was lost multiple times during training and experimenting.

The experiments themselves were not always showing improvements, despite the theoretical reasons to achieve better results. The reason behind writing a section about troubleshooting in deep learning was to indicate what was done to achieve meaningful results from training. Many of these faulty experiments are not described in this thesis in order to keep the training report clean from unrelated issues in data, design and deployments.

Overfitting happened in many of the experiments. Some changes in architecture and in the algorithm were tested to overcome this issue. Early stopping, weight decay and dropout were used to prevent overfitting in some of the experiments. Data augmentation was used to create more data to avoid the same issue. Finally, different learning rates were calculated by the learning rate finder algorithm to prevent overfitting.

Different CNN models with or without previous training were tested to see the results in the training phase. Resnet models with different layers were part of the experiments too. In some experiments, due to the depth of network, overflow occurred. Faulty experiments

were not discussed in this work. However, whenever it was possible, modified codes from the faulty experiments were added to observe their effects on other experiments.

The trained models were also experimented with concerning how to unfreeze and change the weights of each layer and how to tune each layer to gain new results from the trained models. Changing the learning rate for deeper layers and having different learning rates for different layers were part of the experiments which were done with the trained models. Some very fast tests were done with these trained networks to see the effect of different techniques and as a result they are good choices for the proof of different concepts.

Through the experiments in this project, many aspects of a deep learning architecture were studied, and numerous experiments were done. The test results brought new ideas. Not all the tests that were carried out are discussed in this work, but they were the base for creating a path to determine a structure for training a model. The experiments, as mentioned in the previous section, will continue, until the findings of the project are implemented in a real-world environment.

Since online platforms and libraries are constantly changing, multiple developments were done for the demo application. In the first application, a representation of the notebook of the work was created. Because of changes in the libraries and inconsistency between the PyTorch, fastai and voila, the application was written once again entirely in flask and deployed.

From these experiments, one can conclude that along with the amount of input data, using a pretrained Resnet model with more layers and choosing the right learning rate were the most effective parameters. Any changes in them have a direct effect on how the model performed in experiments. Since the learning rate finder gives a deeper insight of how to choose the learning rate based on its estimations, it is a necessary tool for any DL developer to use.

Among the decisions in a classification problem, the ethical ones have not been discussed in this work. However, it is important to consider the consequence of instrumenting any classification model in real life. If such a model is used to clear out forest from

invasive plants, then the ethical use of such a system should be investigated in a good manner.

References

Bahrami, Ali. baherami/lupo. Online. Github. < https://github.com/baherami/lupo>. Accessed 11 May 2021.

Bahrami, Ali. baherami/lupo-app. Online. Github. < https://github.com/baherami/lupo-app>. Accessed 11 May 2021.

Claesen, Marc & De Moor, Bart. 2015. Hyperparameter Search in Machine Learning. The XI Metaheuristics International Conference, MIC:2015, pp. 141-145.fast.ai. About. Online. fast.ai. <https://www.fast.ai/about/>. Accessed 11 May 2021.

Floraincognita. *Flora Capture App.* Online. < https://floraincognita.com/apps/flora-capture-app/>. Accessed 11 May 2021.

Goodfellow, Ian; Bengio, Yoshua & Courville, Aaron. 2016. Deep Learning. Electronic book. MIT Press. <https://www.deeplearningbook.org/>. Accessed 11 May 2021.

Google. Welcome To Colaboratory. Online. Google. < https://colab.research.google.com/notebooks/intro.ipynb?utm_source=scs-index>. Accessed 11 May 2021.

Howard, Jeremy & Gugger, Sylvain. 2020. Deep Learning for Coders with Fastai and PyTorch. Electronic book. O'Reilly Media, Inc.. <https://course.fast.ai/> Accessed 13 May 2021.

Johnson, Justin. *Learning PyTorch with Examples.* Online. PyTorch. < https://pytorch.org/tutorials/beginner/pytorch_with_examples.html> Accessed 13 May 2021.

Kaiming, He; Xiangyu, Zhang; Shaoqing, Ren & Jian, Sun. 2015. Deep Residual Learning for Image Recognition. Proceeding IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770-778.

LeCun, Y.; Boser, B.; Denker, J. S.; Henderson D.; Howard R. E.; Hubbard W. & Jackel L. D.. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. Neural Computation, vol. 1, no. 4, pp. 541-551.

LeCun, Yann; Cortes, Corinna & Burges J.C. Christopher. 1998. THE MNIST DATABASE of handwritten digits. Online. <http://yann.lecun.com/exdb/mnist/>. Accessed 13 May 2021.

Lesort, T., 2020. Continual Learning: Tackling Catastrophic Forgetting in Deep Neural Networks with Replay Processes. Online. Institut Polytechnique de Paris. < https://arxiv.org/abs/2007.00487> Accessed 13 May 2021.

McCulloch, W. S. & Pitts Walter. 1943. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, no 5, pp 115-133.

Marvin Minsky & Seymour Papert, 1969. Perceptrons: an introduction to computational geometry. Cambridge, Massachausettes: MIT Press.

Microsoft. Welcome to AirSim. Online. Microsoft. <https://microsoft.github.io/AirSim/>. Accessed 13 May 2021.

Microsoft. Bing Image Search API. Online. Microsoft. < https://www.microsoft.com/en-us/bing/apis/bing-image-search-api>. Accessed 13 May 2021.

QuantStack. 2019. And Voilà!. Online. Jupyter. < https://blog.jupyter.org/and-voil%C3%A0-f6a2c08a4a93>. Accessed 13 May 2021.

Rosenblatt, Frank. 1957. The Perceptron, a perceiving and recognizing automaton. Bufallo, N.Y.: Cornell Aeronautical Laboratory, Inc..

Ross, Kate. 2011. Soy Substitute Edges Its Way Into European Meals. Online. <https://www.nytimes.com/2011/11/17/business/energy-environment/soy-substitute-edges-its-way-into-european-meals.html?pagewanted=all&_r=0>. The New York Times. Accessed 13 May 2021.

Rumelhart, David E.; McClelland, James L. & PDP Research Group. 1986. Parallel Distributed Processing, Volume 1. Cambridge, Massachausettes: MIT Press.

Russel Stuart J. & Norvig Peter. 2010. Artificial Intelligence: A Modern Approach. Englewood, New Jersey: Prentice Hall.

Rzanny, Michael; Mäder Patrick; Deggelmann, Alice; Chen, Minqian & Wäldchen Jana. 2019. Flowers, leaves or both? How to obtain suitable images for automated plant identification. Plant Methods 15, 77.

Samuel, A. L.. 1959. Some studies on machine learning, using the game of checkers. IBM Journal of Research and Development, vol. 3 no. 3, pp. 210-229. Samuel A.L.. 1962.

Samuel, A. L.. 1962. Artificial Intelligence: A Frontier of Automation. The ANNALS of the American Academy of Political and Social Science. vol. 340, issue 1, pp:10-20.

SHARMA, Saghar. 2017. Epoch vs Batch Size vs Iterations. Online. Towards Data Science. <https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9>. Accessed 13 May 2021.

Smith, L. N.. 2015. No More Pesky Learning Rate Guessing Games. Online. Arxiv org. <https://arxiv.org/abs/1506.01186v2>. Accessed 13 May 2021.

Tobin, Josh. 2019. Troubleshooting Deep Neural Networks. Online. <http://josh-tobin.com/troubleshooting-deep-neural-networks.html>. Accessed 13 May 2021.

Yle. 2015. Top five invasive species that pose a threat to Finnish nature. Online. Yle. <https://yle.fi/uutiset/osasto/news/top_five_invasive_species_that_pose_a_threat_to_finnish_nature/8146734>. Accessed 13 May 2021.

Zeiler, M. D. & Fergus, Rob. 2013. Visualizing and Understanding Convolutional Networks. Computer Vision ECCV 2014, pp 818-833.

Appendix 1- DataLoaders and model summary for the first experimenet

DataLoaders:

```
Setting-up type transforms pipelines
Collecting items from /content/gdrive/MyDrive/thesis/invasive_plants
Found 485 items
2 datasets of sizes 388,97
Setting up Pipeline: PILBase.create
Setting up Pipeline: parent_label -> Categorize -- {'vocab': None, 'sort':
True, 'add_na': False}

Building one sample
  Pipeline: PILBase.create
    starting from
      /content/gdrive/MyDrive/thesis/invasive_plants/Lupine/00000039.JPG
    applying PILBase.create gives
      PILImage mode=RGB size=1200x1600
  Pipeline: parent_label -> Categorize -- {'vocab': None, 'sort': True,
'add_na': False}
    starting from
      /content/gdrive/MyDrive/thesis/invasive_plants/Lupine/00000039.JPG
    applying parent_label gives
      Lupine
    applying Categorize -- {'vocab': None, 'sort': True, 'add_na': False}
gives
      TensorCategory(2)

Final sample: (PILImage mode=RGB size=1200x1600, TensorCategory(2))


Collecting items from /content/gdrive/MyDrive/thesis/invasive_plants
Found 485 items
2 datasets of sizes 388,97
Setting up Pipeline: PILBase.create
Setting up Pipeline: parent_label -> Categorize -- {'vocab': None, 'sort':
True, 'add_na': False}
Setting up after_item: Pipeline: Resize -- {'size': (224, 224), 'method':
'crop', 'pad_mode': 'reflection', 'resamples': (2, 0), 'p': 1.0} -> ToTensor
Setting up before_batch: Pipeline:
Setting up after_batch: Pipeline: IntToFloatTensor -- {'div': 255.0,
'div_mask': 1}

Building one batch
Applying item_tfms to the first sample:
  Pipeline: Resize -- {'size': (224, 224), 'method': 'crop', 'pad_mode': 're-
flection', 'resamples': (2, 0), 'p': 1.0} -> ToTensor
    starting from
      (PILImage mode=RGB size=1200x1600, TensorCategory(2))
    applying Resize -- {'size': (224, 224), 'method': 'crop', 'pad_mode': 're-
flection', 'resamples': (2, 0), 'p': 1.0} gives
      (PILImage mode=RGB size=224x224, TensorCategory(2))
    applying ToTensor gives
      (TensorImage of size 3x224x224, TensorCategory(2))

Adding the next 3 samples
```

```
No before_batch transform to apply

Collating items in a batch

Applying batch_tfms to the batch built
  Pipeline: IntToFloatTensor -- {'div': 255.0, 'div_mask': 1}
    starting from
      (TensorImage of size 4x3x224x224, TensorCategory([2, 0, 0, 2], de-
vice='cuda:0'))
    applying IntToFloatTensor -- {'div': 255.0, 'div_mask': 1} gives
      (TensorImage of size 4x3x224x224, TensorCategory([2, 0, 0, 2], de-
vice='cuda:0'))
```

Model:

```
Sequential(
  (0): Sequential(
    (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_run-
ning_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
    (4): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (5): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
```

```
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (6): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (7): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
```

```
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
  )
  (1): Sequential(
    (0): AdaptiveConcatPool2d(
      (ap): AdaptiveAvgPool2d(output_size=1)
      (mp): AdaptiveMaxPool2d(output_size=1)
    )
    (1): Flatten(full=False)
    (2): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_run-
ning_stats=True)
    (3): Dropout(p=0.25, inplace=False)
    (4): Linear(in_features=1024, out_features=512, bias=False)
    (5): ReLU(inplace=True)
    (6): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_run-
ning_stats=True)
    (7): Dropout(p=0.5, inplace=False)
    (8): Linear(in_features=512, out_features=3, bias=False)
  )
)
```

Appendix 2 - DataLoaders and model summary for the second experiment

```
Setting-up type transforms pipelines
Collecting items from /content/gdrive/MyDrive/thesis/invasive_plants
Found 485 items
2 datasets of sizes 388,97
Setting up Pipeline: PILBase.create
Setting up Pipeline: parent_label -> Categorize -- {'vocab': None, 'sort':
True, 'add_na': False}

Building one sample
  Pipeline: PILBase.create
    starting from
      /content/gdrive/MyDrive/thesis/invasive_plants/Lupine/00000039.JPG
    applying PILBase.create gives
      PILImage mode=RGB size=1200x1600
  Pipeline: parent_label -> Categorize -- {'vocab': None, 'sort': True,
'add_na': False}
    starting from
      /content/gdrive/MyDrive/thesis/invasive_plants/Lupine/00000039.JPG
    applying parent_label gives
      Lupine
    applying Categorize -- {'vocab': None, 'sort': True, 'add_na': False}
gives
      TensorCategory(2)

Final sample: (PILImage mode=RGB size=1200x1600, TensorCategory(2))


Collecting items from /content/gdrive/MyDrive/thesis/invasive_plants
Found 485 items
2 datasets of sizes 388,97
Setting up Pipeline: PILBase.create
Setting up Pipeline: parent_label -> Categorize -- {'vocab': None, 'sort':
True, 'add_na': False}
Setting up after_item: Pipeline: Resize -- {'size': (460, 460), 'method':
'pad', 'pad_mode': 'zeros', 'resamples': (2, 0), 'p': 1.0} -> ToTensor
Setting up before_batch: Pipeline:
Setting up after_batch: Pipeline: IntToFloatTensor -- {'div': 255.0,
'div_mask': 1} -> Flip -- {'size': None, 'mode': 'bilinear', 'pad_mode': 're-
flection', 'mode_mask': 'nearest', 'align_corners': True, 'p': 0.5} -> Random-
ResizedCropGPU -- {'size': (224, 224), 'min_scale': 0.2, 'ratio': (1, 1),
'mode': 'bilinear', 'valid_scale': 1.0, 'p': 1.0} -> Brightness --
{'max_lighting': 0.2, 'p': 1.0, 'draw': None, 'batch': False}

Building one batch
Applying item_tfms to the first sample:
  Pipeline: Resize -- {'size': (460, 460), 'method': 'pad', 'pad_mode': 'ze-
ros', 'resamples': (2, 0), 'p': 1.0} -> ToTensor
    starting from
      (PILImage mode=RGB size=1200x1600, TensorCategory(2))
    applying Resize -- {'size': (460, 460), 'method': 'pad', 'pad_mode': 'ze-
ros', 'resamples': (2, 0), 'p': 1.0} gives
      (PILImage mode=RGB size=460x460, TensorCategory(2))
    applying ToTensor gives
      (TensorImage of size 3x460x460, TensorCategory(2))
```

```
Adding the next 3 samples

No before_batch transform to apply

Collating items in a batch

Applying batch_tfms to the batch built
  Pipeline: IntToFloatTensor -- {'div': 255.0, 'div_mask': 1} -> Flip --
{'size': None, 'mode': 'bilinear', 'pad_mode': 'reflection', 'mode_mask':
'nearest', 'align_corners': True, 'p': 0.5} -> RandomResizedCropGPU --
{'size': (224, 224), 'min_scale': 0.2, 'ratio': (1, 1), 'mode': 'bilinear',
'valid_scale': 1.0, 'p': 1.0} -> Brightness -- {'max_lighting': 0.2, 'p': 1.0,
'draw': None, 'batch': False}
    starting from
      (TensorImage of size 4x3x460x460, TensorCategory([2, 0, 0, 2], de-
vice='cuda:0'))
    applying IntToFloatTensor -- {'div': 255.0, 'div_mask': 1} gives
      (TensorImage of size 4x3x460x460, TensorCategory([2, 0, 0, 2], de-
vice='cuda:0'))
    applying Flip -- {'size': None, 'mode': 'bilinear', 'pad_mode': 'reflec-
tion', 'mode_mask': 'nearest', 'align_corners': True, 'p': 0.5} gives
      (TensorImage of size 4x3x460x460, TensorCategory([2, 0, 0, 2], de-
vice='cuda:0'))
    applying RandomResizedCropGPU -- {'size': (224, 224), 'min_scale': 0.2,
'ratio': (1, 1), 'mode': 'bilinear', 'valid_scale': 1.0, 'p': 1.0} gives
      (TensorImage of size 4x3x224x224, TensorCategory([2, 0, 0, 2], de-
vice='cuda:0'))
    applying Brightness -- {'max_lighting': 0.2, 'p': 1.0, 'draw': None,
'batch': False} gives
      (TensorImage of size 4x3x224x224, TensorCategory([2, 0, 0, 2], de-
vice='cuda:0'))
```

Model:

```
Sequential(
  (0): Sequential(
    (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_run-
ning_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
    (4): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
```

```
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (5): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (3): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
```

```
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (6): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (3): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (4): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
```

```
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (5): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (7): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), pad-
ding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
  )
  (1): Sequential(
    (0): AdaptiveConcatPool2d(
      (ap): AdaptiveAvgPool2d(output_size=1)
      (mp): AdaptiveMaxPool2d(output_size=1)
```

```
    )
    (1): Flatten(full=False)
    (2): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_run-
ning_stats=True)
    (3): Dropout(p=0.25, inplace=False)
    (4): Linear(in_features=1024, out_features=512, bias=False)
    (5): ReLU(inplace=True)
    (6): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_run-
ning_stats=True)
    (7): Dropout(p=0.5, inplace=False)
    (8): Linear(in_features=512, out_features=5, bias=False)
  )
)
```

Appendix 3 - Application

```
from flask import Flask, render_template, request, redirect, url_for, abort
from fastai.vision.all import *

app = Flask(__name__)

app.logger.addHandler(logging.StreamHandler(sys.stdout))
app.logger.setLevel(logging.ERROR)
global learn_inf

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/', methods=['POST'])
def upload_files():
    uploaded_file = request.files['file']
    results = []
    img = PILImage.create(uploaded_file)
    global learn_inf
    labels, prediction, probability = learn_inf.predict(img)
    for idx,label in enumerate(learn_inf.dls.vocab):
        results.append([label, prediction[idx],f"{probability[idx]:.2f}"])
    return render_template('index.html', results=results)

def parent_label_multi(o):
    return [Path(o).parent.name]

if __name__ == '__main__':
    path = Path()
    global learn_inf
    learn_inf = load_learner(path/'invasive_plants_II/fmodel.pkl', cpu=True)
    port = int(os.environ.get('PORT', 5000))
    app.run(host = '0.0.0.0', port = port)
```