Analysing the Characteristics of Neural Networks for the Recognition of Sugar Beets

Applied University of Hamm-Lippstadt

Bachelor thesis
for the attainment of the academic degree
Bachelor of Engineering

submitted by

Luca Brodo

Electronic Engineering
Mat.Nr.: 2180015
luca.brodo@stud.hshl.de

16. Januar 2022

First supervisor:	Prof. Dr. Stefan Henkler
Second supervisor:	Dipl-Ing Kristian Rother

Affidavit

I, Luca Brodo, herewith declare that I have composed the present paper and work by myself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The paper and work in the same or similar form has not been submitted to any examination body and has not been published. This paper was not yet, even in part, used in another examination or as a course performance.
Lippstadt, 16. Januar 2022

Brodo Luca

Abstract stays here

Table of Content

List of Figures

List of Tables

1 Motivation

The seemingly unstoppable growth of global population compounded by climate change is putting an enormous pressure on the agricultural sector. As estimated by the World Resources Institutes (WRI), the number of people on our planet will reach an astounding 10 billion by 2050 [AAUS+19] and, as a result, the agricultural production must double to keep up with the demand [SGSS16]. However, the agricultural sector is facing immense challenges due to plant diseases, pests and weed infestation and in these times, like never before, the switch to a more sustainable model seems inevitable. The main target of sustainable farming is to increase yield while reducing reliance on herbicides and pesticides; therefore trying to target treatments only to plants that specifically require it by monitoring key indicators of each individual crop, a technique usually referred to as precision farming. This technique is notoriously time and energy consuming if done manually and therefore usually discarded in favor of more traditional methodologies. [LHS+16a]
In the last years there has been, however, an increased focus on integrating cutting-edge technology with precision farming precision farming to improve quality and quantity of agricultural production, while at the same time lowering the inputs significantly [IRP+21]. This system is also known as "smart farming". This system is based on the adoption of autonomous robots, which could be both wheeled robots and Unmanned Aerial Vehicles (UAVs), and it has been enabled by the astonishing advancements in the field of the Internet of Things (IoT).
IoT is the key technology behind smart farming and allows to add value to the data collected by automated processes by ensuring data flow between different devices [IRP+21]. More importantly, IoT allows more cost-efficient and timely production and management practices, as showed by Glaroudis et al. in[GIC20], and at the same time the reduction of the inherent climate impact by enabling real-time reactions to infestations, such as weed, pest or diseases, and by enabling a more adequate use of resources such as water, pesticides or agro-chemicals.[IRP+21] In other words, IoT makes precision farming not only more efficient in terms of both money and resources, but even more sustainable than other traditional farming methods.
One of cultivations which will benefit immensely by the adoption of smart farming is the sugar beet. The *Beta vulgaris L*,depicted in Fig. 1.1, commonly referred as sugar beet, is ranked as the second most cultivated sugar crop all over the world next to sugarcane[BP20]. As showed by May in [May03], being a slow-growing crop early in the season [BP20], this plant seems to be a very poor competitor against weed, backed up also by Schweizer's researches, which reports that sugar beet root yields can be reduced by 26–100% by uncontrolled weed growth. [SD89]
An approach based on smart farming could reduce the effect of weeds infestations on sugar beet plantations and increment the yield and the sustainability of said plantation. As a matter of fact, even though tractors and hand labour are still vastly used for weed management, since the 50s, herbicides have been the most used method of weed

Fig. 1.1: Representation of a sugar beet plant [Mas91]

management in sugar beets farms ([CM10]) and, apart from the well known environmental damage, which the use herbicides leads to, another implication is the effect that they have on the sugar beet crops. Most of the selective herbicides have influence on sugar beet growth, with early symptoms showing on the leaves, and this can reduce the yield. [Pet04] To optimize and remove the effects of pesticides, the new frontier for weed management is to automatize the more traditional mechanical techniques ([RNSF20], [FMF$^+$14],[MPSG21]). Mechanical techniques and even techniques which use selective spraying of herbicides, however, require that the weed is localized before they can be applied. The localization of weed is a fairly researched topic and there are two main school of thoughts: Plant-based Classification and Weed Mapping. *Hasan et al.* in [HSD$^+$21] provide the definition for plant-based classification: *"localise every plant in an image and classify that image either as a crop or as a weed"*. This approach requires 3 main steps, which are detection, localisation and classification, and it is very suitable for real-time weed management techniques. [HSD$^+$21] Such approach works in combination with mechanical management techniques, as demonstrated by *Raja et. al.* in [RNSF20], where an automatic robot would detect and classify the weeds, which will be lately removed with a knife.
Hasan et al. in [HSD$^+$21] also provide the definition for weed mapping. According to the authors, such approach can be defined as the process of mapping the density of weeds in the field. Weed mapping is useful for site-specific weed management and it's mainly used to reduce the amount of herbicides utilized. [HSD$^+$21]
What both of those approaches have in common, however, is the need of Deep Learning techniques to map or recognize weeds. As a matter of fact, a lot of proposed solutions employ Convolutional Neural Networks in the recognition task([GFP$^+$20],[SIHvH18],[RAMP20]). Neural Networks, however, impose challenges to developer, from choosing which architecture to use to define effective training cycles to achieve pre-defined levels of accuracy. In addition, in real time applications, classification, or inference, time is of the highest importance since it effects the ability to respect deadlines.Furthermore, the values for these metrics obtained while testing do not fully reflect the reality when the Neural Network will be actually employed. Table 1.1 shows an example of different architectures trained to recognize weed

in a sugar beet plantations and, from this comparison alone, emerges evidently that the differences in performance amongst the networks.

Network	Accuracy	Training time	Classification time
	20 Epoch		
AlexNet	97.9%	9.0min	0.0038s
VGG-19	98.4%	37.4min	0.0130s
GoogLeNet	97.0%	23.8min	0.0033s
ResNet-50	96.2%	40.3min	0.0072s
ResNet-101	97.5%	106.6min	0.0118s
Inception-v3	90.8%	88.7min	0.0088s

Table 1.1: Example of comparison between different Neural Networks [SIHvH18]

Being able to predict the performances of Neural Networks before their deployment will allow engineers to precisely model the application to tailor them for the Neural Network they chose. We can make those predictions by empirically studying certain metrics and find correlations among them in order to able to be precise, scalable and predictable in time.
The purpose of this paper is to create a toolbox for the employment of Neural Networks in agriculture to optimize applications.
In order to achieve this goal, we will first analyse the main characteristics of Neural Networks in chapter number 2. This will allow us to pose the foundation for our study, since it is not only focused on those metrics, but also because identifying the right metrics is essential to optimize various applications.
Subsequently, in chapter number 3, we will study how we can precisely benchmark neural networks to obtain the measurements we need to find the correlations. Benchmarking is a notoriously insidious task, however at the end of our study, we will be able to create a tool to measure performances and analyse Neural Networks in a precise and reproducible way.
Finally, with the help of our tool, in chapter 4 we will profile Neural Networks with fairly different architectures to identify correlations between their metrics so that we will be able to optimize future applications.

2 Characteristics of Neural Networks

2.1 Introduction

In this chapter, we will explore the characteristics of NN models we want to measure and analyse. From now on, we will use the term *characteristics* to identify all the metrics of the model which will have an influence on its performances once the model is deployed. The term is therefore an umbrella term for any type of measurable property which we can use for our application. In the following sections we will then take a closer look to each one of them with the purpose of understanding them.

2.2 Learning Process and Training Time

Training time, as the name implies, is the time required to train a Neural Network, or any machine learning model. *Mitchell* in [Mit97] provides a general definition for the training, or learning, process : *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."* This definition is quite broad ([GBC16]) and it is out of the scope of this paper to give a formal definition for the experience E, task T, and performance measure P.
More practically, the process of training is the process of updating the weights of the neural network to so that the network is able to achieve the desired goal. [Mur16] The network weights are updated during back propagation by the following equation([Mur16]):

$$W_{new} = W - \eta \frac{dE}{dW} \tag{2.1}$$

The error, or "scoring", function is defined as the sum of squared differences between the network's output(equation n. 2.2) and the correct output and it is applicable when the output is vector, matrix, or tensor of continuous real values ([Mur16]).

$$E(W, b) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} ||h_{W,b}(x^{(i)}) - y^{(i)}||^2 \tag{2.2}$$

The learning experience can be broadly categorized into two main categories: supervised and unsupervised learning. Roughly speaking, given a dataset $\mathcal{D}$, an unsupervised learning algorithm aims to learn useful properties of this dataset. In the context of deep learning, we aim to learn the entire probability distribution $p(\mathcal{D})$ that generated the dataset or some useful properties of it. On the other hand, supervised learning implies the use of a dataset $\mathcal{D}$ and an associated value or label $\mathbb{Y}$ to teach the model to predict $\mathbb{Y}$ from $\mathbb{X} \subset \mathcal{D}$, usually by estimating $p(\mathbb{Y}|\mathbb{X})$([Mur16])

Usually the learning process proceeds in waves of mini-batches, which allow to avoid both over-fitting and under utilization of GPU's compute parallelism, therefore throughput is a key performance metrics to measure.[ZAZ+18] Finally, the learning process is a memory demanding one, as backward pass and weight updates, operations unique to training, need to save intermediate results in GPU memory (in some cases tens of gigabytes are required). [RGC+16] Training time is often an overlooked metrics of Neural Network compared to inference(2.3) or accuracy (2.5) ([ZAZ+18]), however, due to the recent growth in applications of Deep Learning technologies in different fields ([BTD+16], [HWT+15], [10.16], [AAB+15b]) training time is acquiring importance. [ZAZ+18]

2.3 Inference Time

In Machine Learning, therefore in Deep Learning as well, the term **inference time** is used to indicate the time required for the trained model to make predictions, regardless of the correctness of those.
Inference time has attracted a lot of attention in the research field, since execute already trained Neural Networks efficiently is inarguably a still open problem. [ZAZ+18]
Differently from Training Time (section 2.2), the memory footprint for inference is in the order of Mega-Bytes.([HLM+16]) As a matter of fact, inference is latency sensitive, but computationally less demanding. [ZAZ+18]

2.4 Uncertainty in Deep Neural Networks

In the previous section we observed that a benchmark should produce predictable results. Although this is true in general terms, we can not properly use the term "predictability"when discussing DNNs.
Due to the increased trust given by the advancements in recent years, DNNs are being applied in numerous different fields, as shown by the increasing number of peer review papers about Deep Learning (Fig. 2.1), even the higher-risk ones like medical image analysis or autonomous vehicle control. [GTA+21]

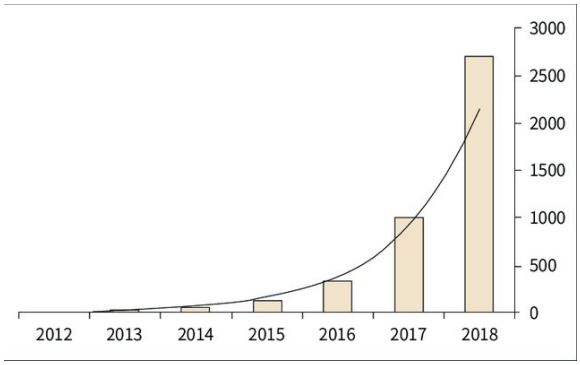


Fig. 2.1: Trend in publications about Deep Learning [SKD19]

In such high-risk scenarios, DNNs should not only be highly accurate, as a mistake in detecting an obstacle could cause catastrophic consequences for the passengers, but also be trustworthy enough so that the probability of the predicted values reflects the ground truth. DNNs, however, are subjected to different source of uncertainty and errors. *Galiwoski et al.* in [GTA+21] recognize five crucial factors which can cause uncertainty and errors in DNNs.

Those factors are:

I. the variability in real world situations.

II. the errors inherent to the measurement systems,

III. the errors in the architecture specification of the DNN,

IV. the errors in the training procedure of the DNN,

V. the errors caused by unknown data.

We can model a neural network as a function f, parametrized by weight θ, which maps a set of input $\mathbb{X}$ to a set of measurable outputs $\mathbb{Y}$, hence:

$$f\theta : \mathbb{X} \to \mathbb{Y} \qquad f\theta(x) = y \tag{2.3}$$

In case of supervised learning, we also model the training set as $\mathcal{D} \subset \mathbb{D} = \mathbb{X} \times \mathbb{Y}$, where $\mathbb{X}$ is an instance space and $\mathbb{Y}$ the set of outcomes that can be associated with an instance ([HW21]), containing N samples. A DNN trained in $\mathcal{D}$ can therefore predict a a corresponding target $f\theta(x^*) = y^*$.
During the data acquisition process, a measurement x and a target variable y are taken from space Ω to represent a real world situation ω. For example, ω could be a sugar beet, y the label "sugar beet" and x a picture of a sugar beet. The job of the DNN is therefore to predict the label from the image of the sugar beet (Eq n. 2.3). In a real world situation, however, measurements could be potentially different from the ones used for training and this could influence the prediction of y. The source of this difference could be different lighting condition, different environmental condition or general conditions not taken into account when training.A new measurement generally is not part of the training set, hence $x^* \notin \mathcal{D}$.These differences in real world scenarios compared to the training set are called *distribution shifts*([OFR+19]) and DNN are very sensitive to that. Moreover, source of noise could also be individuated in errors in labelling. This is the first factor that can cause uncertainty, i.e. **the variability in real world situations**. [GTA+21]
In addition, it has to be taken into account that measurement devices are also subjected to noise due to defects or imprecision. This kind of noise is the second factor, i.e. **tthe errors inherent to the measurement systems**. [GTA+21]
We previously modelled a neural network simply as a function $f\theta$. Although in most applications DNN are treated similarly to how we modelled them, i.e. as a black box which takes as input some data and outputs a prediction, they have a certain structure with certain parameters to take care of. These parameters can be for example the amount of layer, the parameters which need to be configured and the chosen activation function. These configurations are up to the modeller and are the third cause of uncertainty in DNNs, i.e. **the errors in the architecture specification of the DNN**. [GTA+21]

A further source of uncertainty in DNNs is also related to the very own nature of DNNs. DNNs are not deterministic, rather they are stochastic[1] in many ways : the order of data, the weight initialization or random regularization as augmentation or dropout. [GTA+21] Moreover, the stochastic gradient descent algorithm is widely use to optimize the learning process of DNNs. [Rud17]
Another version of the gradient descent algorithm which is also used is the mini-batch version, which selects random batches with aleatory size (called batch size) from the learning data-set to optimize the training process. [Rud17] This parameter, in addition to other ones like learning rate, and the number of training epochs, is also source of unpredictability in the DNN, as they could heavily change its performances
The stochastic nature of DNN and the choices of the aforementioned parameters are the forth cause of uncertainty in DNNs : **the errors in the training procedure of the DNN**
Previously we modelled the training set as $\mathcal{D} \subset \mathbb{D}$, hence as being a subset of a certain domain $\mathbb{D}$. However, the realm of input which could be fed to a neural network is not limited to this domain, rather the DNN is trained to solve tasks with inputs belonging to this domain. Another source of uncertainty can be therefore an input which the DNN is not trained for, even though it is able to process it. For instance, if a DNN is trained to classify dogs from images, an image as input could also depict a bird. The last factor which could cause uncertainty is therefore the **errors caused by unknown data**. Such unknown data, could also be caused by highly noisy measurement devices, like a broken camera, and some researches show how easy is to fool DNN with images that are meaningless for us humans.[NYC15] Fig.2.2 depicts some examples of picture which could fool even the most advanced DNN. The figures on the top row could also be produced by random noise from a malfunctioning camera and yet could be classified as something else with an accuracy $\geq$ 99.6%. [NYC15]

With the help of the five factors we studied before, we can further categorize uncertainty in two main categories: aleatoric and epistemic uncertainty. [HSHK21]

Aleatoric uncertainty is also usually referred to as data or statistical uncertainty) and it is generally used to identify the type of uncertainty cause by the nature of randomness, that is *the variability in the outcome of an experiment which is due to inherently random effects* [HSHK21]. Moreover, it also comprises the type of noise directly stemming from the noise in data used to train, validate, test and for inference, therefore referring to factor II [GTA+21] [BC18].
Aleatoric uncertainty is of the irreducible type, meaning it is impossible to get rid of.([KG17], [HSHK21]) For example, a model that predicts the output of a flip coin, even in case of a perfect model, is not able to produce the definite, correct answer for it, but only a prediction of the possibilities for the two outcomes. This is due to the fact that the data used for this model is of random, stochastic nature and this random component can not be reduced. [HSHK21]
Epistemic uncertainty, or systemic or epistemic uncertainty, on the other hand, refers to the type of uncertainty cause by ignorance or short comings of the model, hence not on any other underlying stochastic phenomenon. [BC18]. Usually, this is due to unknown or lack of coverage of the dataset used for training, hence covers factors I,III, IV and V.

[1] Stochastic is a synonym for randomness

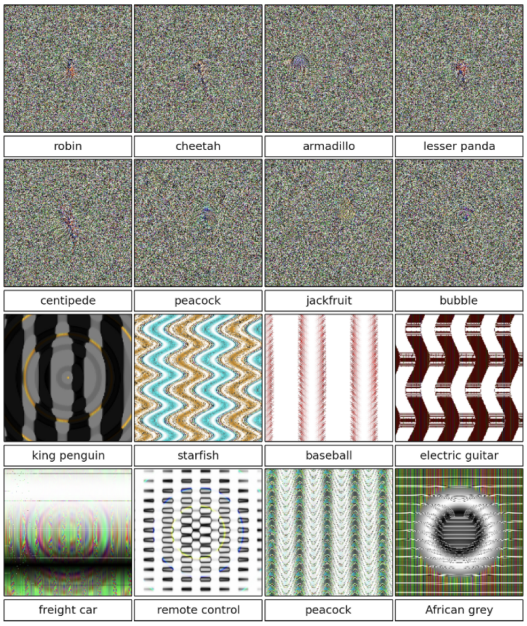


Fig. 2.2: This result highlights differences between how DNNs and humans recognize objects. Images are either directly (top) or indirectly (bottom) encoded [NYC15]

Epistemic uncertainty can be theoretically explained away by improving the architecture of the model and broadening the dataset. For example,let us consider a model which is able to determine whether a word is part of the Italian or English dictionary. If we feed the word 'macchina" to the model, given that we provided enough data to train and the model is perfect, we can be confident that it will predict the correct language, in this case Italian. However, if we feed it with the word "pasta", we can not be confident that the prediction is correct, i.e. we can only obtain the probability of this word to be part of one of the dictionaries **given the dataset we provided**. As a matter of fact, the word "pasta" is present in both dictionaries and the outcome depends on the probability of the word to appear in one of the dictionaries in our dataset. The first case is clear case of epistemic uncertainty, which we were able to completely remove with a perfect dataset; while the second case is a clear case of aleatoric uncertainty and it is impossible to remove. Fig. 2.3 helps visualize the difference between the two. In pictures (d), we can see that the aleatoric uncertainty increases on object boundaries and objects far from the camera. Pictures (e), on the other hand, shows how epistemic uncertainty increases for semantically and visually challenging pixels. For example, the bottom row shows a failure case when the model fails to segment the footpath due to high epistemic uncertainty, but low aleatoric uncertainty. [KG17]
Although this distinction is very helpful for further and more precise analysis on a DNN model, it is also important to note that the distinction between the two is highly dependent on the context: they are not absolute notions.([KD09]) Hence, changing the settings of the model will blur the distinction line between the two and therefore make their classification considerably more difficult.[HSHK21]
We mentioned previously that theoretically model uncertainty is 100% reducible. However, in case of real world data this is not the case. In addition to the probabilistic nature of the DNNs, the training dataset used for the model is very likely to be only a subset of all possible input data of the application, hence it is also very likely that unknown data for the

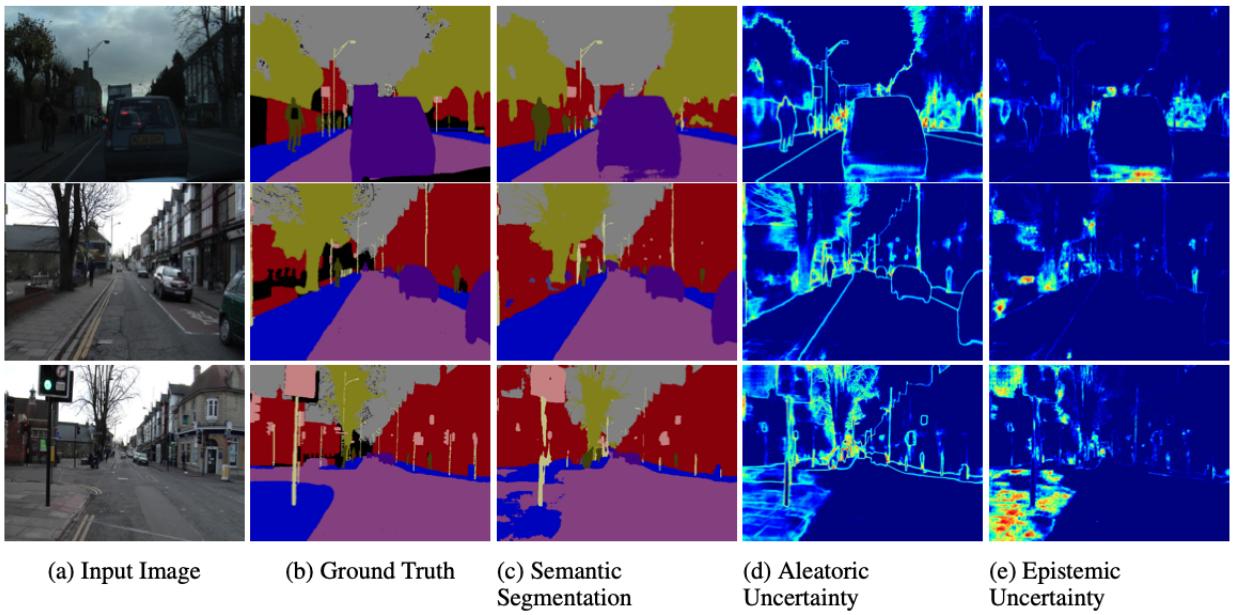


Fig. 2.3: Illustrating the difference between aleatoric and epistemic uncertainty for semantic segmentation on the CamVid dataset [BC18]

domain is unavoidable. In addition, represent exactly the uncertainty of the DNN is not possible, as the different sources of uncertainty can not be generally modelled accurately. [GTA+21]

2.5 Accuracy

Accuracy is usually one of the most looked after metrics when analysing Neural Network performances([HD19],[BCCN18]). Accuracy is usually the factor taken into account when the training process is evaluated, as could be a potential factor deciding when to stop it. Being able to predict accuracy, for example, could help detecting, and therefore terminate, unsuccessful training run [UKG+21].
Usually, the term "accuracy" refers to the *classification accuracy* of the neural network. The classification accuracy is the ratio of the number of correct predictions to the total number of input sample ([HW20]) and it is calculated with equation n. 2.4.

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions} \tag{2.4}$$

In case of a multi-class prediction problem, however, classification accuracy is meaningful only when each class contains an equal number of samples. For example, considering two classes A and B containing 98% and 2% of the samples respectively, the model can reach 98% accuracy by predicting exclusively samples from class A.
Another definition which is often used for accuracy in Neural Networks in the context of

binary classification is the one showed in equation n. 2.5.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.5}$$

Where TP = True Positives, TN = True Negatives, FP = False Positives, and FN = False Negatives.
Similarly to classification accuracy, however, this definition could be misleading as well. For example, considering a model that classified 100 tumors as malignant (the positive class) or benign (the negative class) as shown in table 2.1.

TP	FP	FN	TN
1	1	8	90

Table 2.1: Example for binary classification

Calculating the accuracy using equation n. 2.5. will give an accuracy of :

$$Accuracy = \frac{1 + 90}{1 + 90 + 1 + 80} = 0.91 \tag{2.6}$$

At first glance, this metric would show that the model is performing correctly (91 correct out of a 100). The dataset is composed of 91 tumors that are benign (90 TNs and 1 FP) and 9 that are malignant (1 TP and 8 FNs). The model is able to identify 90 out of 91 benign tumors correctly, but only 1 out of 9 as malignant. This is another case of class-imbalanced data set in which a model that would only classify tumors as benign would reach the same level of accuracy. [Goo10]

2.6 Precision and Recall

As we already delineated in the previous section, sometimes analysing only at accuracy might be misleading, especially when we are dealing with class-imbalanced data-sets. Accuracy does not distinguish between the number of correct labels of different classes. [SJS06]
In these cases, two other metrics we can observe are *precision* and *recall*.
Formally, *precision* is defined as equation n. 2.7 and it represents the number of detected anomalies that are actual real anomalies.[TLZ+19] In other words, it expresses the proportion of positive identifications that was correct.

$$Precision = \frac{TP}{TP + FP} \tag{2.7}$$

Where TP = True Positives and FP = False Positives
On the other hand, *recall* is defined as equation n. 2.8 and it represents the number of real anomalies that have been detected.[TLZ+19]

$$Recall = \frac{TP}{TP + FN} \tag{2.8}$$

Where TP = True Positives and FN = False Negatives

We can use the example in table 2.1 to calculate both precision and recall.

$$\begin{aligned} Precision &= \frac{TP}{TP + FP} \\ &= \frac{1}{1+1} \\ &= \frac{1}{2} \\ &= 0.5 \end{aligned} \tag{2.9}$$

Having a precision of 0.5, our model is correct 50% of the time when it predicts that a tumor is malignant.

$$\begin{aligned} Recall &= \frac{TP}{TP + FN} \\ &= \frac{1}{1+8} \\ &= \frac{1}{9} \\ &= 0.11 \end{aligned} \tag{2.10}$$

In other words, with a recall of 0.11, our model has been able to identify 11% of all malignant tumors.
Ideally, we would like to have a high recall percentage as well as a high precision one. Unfortunately, this is rarely the case since improving recall often comes at the expense of precision, since in order to increase the TP for the minority class, the number of FP is also often increased, resulting in reduced precision. [HM13]
Furthermore, as it was the case for accuracy, neither precision nor recall give a full insight on the performance of the model, since a model could have high precision with low recall, as shown in the example above. It is, therefore, common practice to combine both with a weighted harmonic mean as an F score [vR74]:

$$F_\beta = (1 + \beta^2)\frac{Precision * Recall}{\beta^2 * Precision + Recall} \tag{2.11}$$

In equation n. 2.11, the coefficient β represents the balance between Precision and Recall, with high values favouring Recall. Usually, the F-score is used with $\beta = 1$, so a perfect balance between Precision and Recall, and in this case it is called F1 score(Equation n. 2.12). [Der16]

$$F1 = (2)\frac{Precision * Recall}{Precision + Recall} \tag{2.12}$$

When evaluating the F score, however,the focus is given exclusively to true positives, false positives and false negatives, neglecting the true negative group, which is usually the majority class. In addition, using the F score, one is unable to distinguish low-recall from low-precision systems. [Der16]

2.7 Loss

When training a Neural Network, finding the perfect weights for the neurons is impossible, therefore the problem is usually modelled as an optimization problem. As an optimization problem, it is solved using an algorithm which aims to optimize the weights to make good predictions. Usually, Neural Networks are trained using the Stochastic Gradient Descent (SGD) and the weights are updated through back-propagation. In the context of an optimization algorithm, the function which evaluates a candidate solution is defined as the objective function and such function in Neural Networks evaluates how good a prediction is, hence the SGD is used to minimize this function, i.e. finding the solution with the lowest score. When we are minimizing the objective function, we are referring to it as the cost function, loss function, or error function.[GBC16]
The loss function has the fundamental job of faithfully distill all the aspects of the model, both good and bad, down into a single, scalar value, which allows candidate solutions to be compared. [RM99]
The choice of the loss function is highly influenced by the output layer of the Neural Network. The output layer defines the type of the solution we defined for the problem, i.e. if it is a regression problem of a classification problem. In other words, the way we represent the output determines the loss function. [GBC16]
The following are some of the best practices for each type of problem.

Regression problem
: For these type of problems, usually the output layer is one node with a linear activation unit, therefore the loss function to use is the Mean Squared Error(MSE).

Binary Classification problem
: The output layer is one node with a sigmoid activation unit and the loss function used is Cross-Entropy, or Logarithmic loss.

Multi-class Classification problem
: The output layer is composed of one node for each class using the soft-max activation function nd the loss function used is Cross-Entropy, or Logarithmic loss.

2.8 Over-fitting and Under-fitting

As we already discovered in section n. 2.2, the training routine is necessary for the model in order to make predictions on a new set of data. Therefore, the goal of the learning process is not to minimize the accuracy on the training set, but rather to maximize its predictive accuracy on the new data points. [Die95]
If the model works too hard to find the best fit for the training data there is the risk that it is going to fit the noise present on the data-set, hence it will learn the inevitable peculiarities and bias present in the training set, rather than finding a general predictive rule. In other words, the performance of the model will decrease when tested against an unknown data-set.[JK15] This phenomena is called *over-fitting.* [Die95]
Over-fitting is influenced by the training data fed to the model, as small data-sets are more prone to it, even though also large data-sets can be affected. [DSSM09]
Under-fitting, on the other hand, can be considered as the opposite process. This occurs when the model is too simple for the training data and it is incapable of learning the

variability of the data. [DSSM09]

One strategy to avoid over-fitting is to use a so-called ”early-stopping” approach. This approach consists simply of stopping the training before over-fitting can occur and it is one of the mostly used strategies to avoid this phenomena, as it is simple to understand and proven to be the superior one in many cases ([FHZ93], [Pre00]).
The key part of this approach is to divide the training set into two parts, namely the validation set and the training set, and to use those to find a criteria to stop the training process.
Fig. 2.4 shows the idealized evolution over time of the error on the training set and on a validation set not used for training, i.e. training and validation loss.[Pre00]

Fig. 2.4: Validation and training loss curve. The x axis is time, the y axis is the loss [Pre00]

Following the trend in Fig. 2.4, we can observe that the validation error decreases until a certain point together with the training loss before increasing indefinitely. We can use the validation error to approximate the generalization error and, therefore, stop the training in the moment where the validation loss is increasing again. We can summarize the ”early-stopping” approach in the following steps:

1. Split the training data into a training set and a validation set, for i.g. in a 4-to-1 proportion
2. Train over the training and evaluate the validation loss every arbitrary number of epoch
3. Stop training as so on as the error on the validation set is higher than it was last time
4. Use the weights the network had in that previous step as the result of the training run

Furthermore, depending on the data, there might be also situations in which the validation loss is not constantly bigger than the training loss, as shown in Fig. 2.4. As a matter of fact, this graph shows an idealized behaviour of the two curves. A real example for this curve can be seen in Fig. 2.5.

In such situations, it would be particularly difficult to choose an optimal stopping criteria, as the validation function as multiple local minima (16 in the pictures case). It is left to the reader to investigate strategies to optimize early-stopping in these situations.

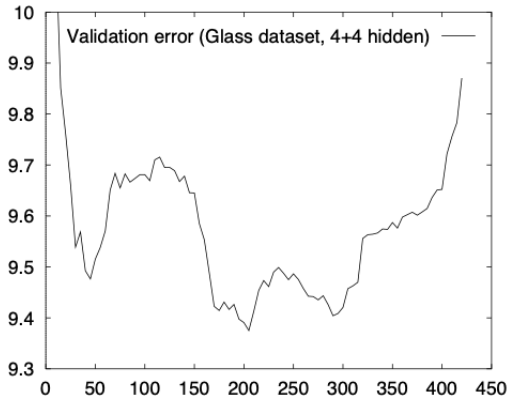


Fig. 2.5: Real Example of the Validation and training loss curve. The x axis is time, the y axis is the loss [Pre00]

Furthermore, there are also scenarios in which, depending on the data presented, the validation loss is at first less than the training loss, as shown in Fig. 2.6.

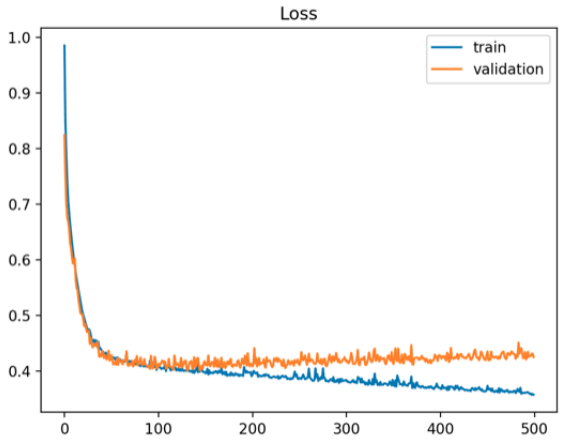


Fig. 2.6: Real Example of Validation and training loss curve. The x axis is time, the y axis is the loss

In these situation, as a rule of thumb, we can use the following statements to determine the status of the training and decide when to stop.

- validation loss >> training loss: over-fitting
- validation loss > training loss: some over-fitting
- validation loss < training loss: some under-fitting
- validation loss << training loss: under-fitting

2.9 Architecture

In the following sections we are going to briefly analyse the architectures of the models we are going to study to find characteristics and correlations.

For the sake of this paper, we are going to limit our analysis to some of the most common architectures for convolutional neural networks which have been proven by other studies ([SIHvH18], [YKU+20], [Suh18]) to achieve promising results in the settings of sugar beet recognition. The architecture chosen are listed below and are treated in more details in their respective sections.

- Resnet
- Alexnet
- VGG

2.9.1 Alexnet

Alexnet is a Convolutional Neural Network (CNN) architecture designed by Alex Krizhevsky in collaboration with Ilya Sutskever and Geoffrey Hinton. [KSH12]
The architecture is depicted in Fig. 2.7. The net is composed of eight layers: The first 5 are convolutional layer an convolutional and the remaining three are fully- connected. The output of the last fully-connected layer is fed to a 1000-way Softmax which produces a distribution over the 1000 class labels. [KSH12]

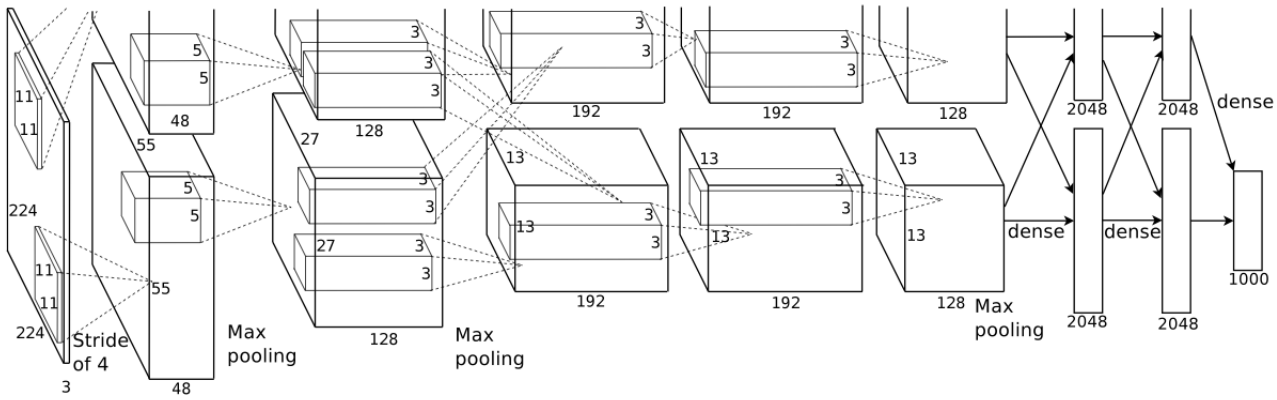


Fig. 2.7: Overview of the architecture of Alexnet [KSH12]

As described by *Krizhevsky et al.* in [KSH12], the layers are defined as it follows:

- The first convolutional layer receive as input a $224 \times 224 \times 3$ image and filters it with 96 kernels of size $11 \times 11 \times 3$ with a stride of 4 pixels[2]
- The second convolutional layer takes as input the response-normalized and pooled output of the first layer and filters it with 256 kernels of size $5 \times 5 \times 48$.
- The third layer has 384 kernels of size $3 \times 3 \times 256$ it is connected to the normalized and pooled output of the second
- The fourth convolutional layer has 384 kernels of size $3 \times 3 \times 192$ and it is connected to the third without pooling or normalization layers

[2]The stride is the distance between the receptive field centers of neighboring neurons in a kernel map

- The fifth convolutional layer has 256 kernels of size $3 \times 3 \times 192$ and as input it receives the output of the forth.
- The fully-connected layers are composed of 4096 neurons each

2.9.2 VGG Networks

VGG stands for Visual Geometry Group at Oxford University and have been developed by Simonyan and Zisserman in [SZ15] for the ILSVRC (Image Net Large Scale Visual Recognition Challenge) 2014 competition ([RDS+14]). The concept of VGG net is similar to Alexnet: as net increases, the number of convolutions and future maps increases as well. Rather than using 11×11 feature detectors or filters, however, the authors decided to use smaller 3 x 3 filters. The various architecture proposed in [SZ15] are showed in Fig. 2.8. In this paper, we are going to consider only the networks D and E, i.e. with 16 and 19 weight layers respectively. For the rest of the paper, we are going to refer to them as VGG16 and VGG19 respectively.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224×224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Fig. 2.8: Overview of the various architecture for VGG [SZ15]

2.9.3 Residual Neural Networks (Resnet)

Many studies ([SZ15], [SLJ+14]) reveal that the depth of the Neural Networks is crucial and for most visual recognition tasks the trend has been to stack more layers into neural networks and increase their depth. ([SZ15],[IS15],[GDDM14],[HZRS14])
Training deeper networks like VGG19 and VGG16 (section n.2.9.2), however, is not a trivial task, since,as they start converging, the accuracy starts to saturate and then degrades quickly. This phenomena is due to the gradient vanishing during back-propagation,

i.e. when the weights are updated, as it becomes smaller the more layers it goes through and vanishes before reaching the initial layer, hence they can not be updated.
He et al. in [HZRS15] tackle the problem of degradation with the introduction of a *deep residual learning* framework. At the core of the framework there is the so called "residual block", shown in Fig. 2.9.

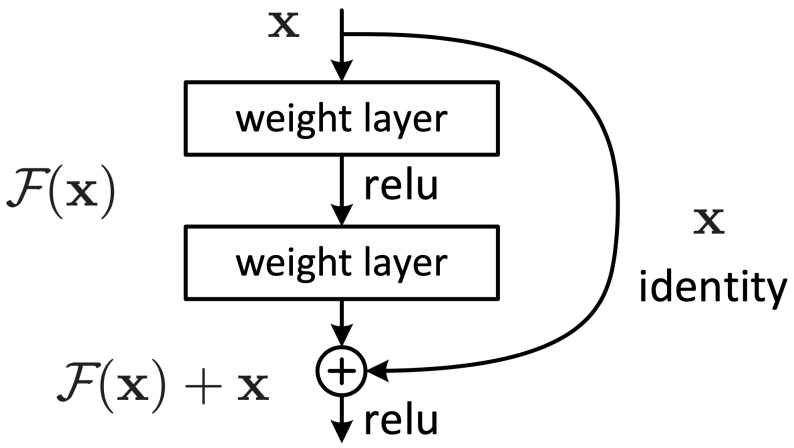


Fig. 2.9: Residual learning: a building block. [HZRS15]

Differently from plain convolutional networks, these blocks also have the so called "identity connection" between the input and the output layer. In plain convolutional networks, the output $\mathcal{H}(x)$ is obtained by:

$$\mathcal{H}(x) = \mathcal{F}(wx + b)$$
$$or$$
$$\mathcal{H}(x) = \mathcal{F}(x)$$

with w being the weights and b being the bias
With the introduction of residual blocks, the output function assumes the form showed in equation n. 2.13.

$$\mathcal{H}(x) = \mathcal{F}(x) + x \tag{2.13}$$

Reworking the equation lets us obtain the so called *residual function*:

$$\mathcal{F}(x) = \mathcal{H}(x) - x \tag{2.14}$$

The purpose of residual networks becomes to approximate the residual function, hence the difference between the input and the output.In other words, the aim is to learn the residual function in a way such that it approaches zero, hence making the identity value optimal.
The problem of the vanishing gradient is resolved by the identity function. During back-propagation the gradient can take this path and, since no weights are encountered, there won't be any change in the value computed by the gradient. This effectively allows the gradient to entirely skip the block and reach the initial layers and correct their weights.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Fig. 2.10: Overview of the Resnet Architecture. The building blocks are shown in brackets [HZRS15]

The architecture of residual networks we are going to use in this paper is showed in Fig. 2.10. For the rest of the paper, we are going to refer them as "Resnet" followed by the amount of layers. For example, Resnet18 refers to the residual network having the 18 layers described in Fig. 2.10.

3 Analysing the models

3.1 Introduction

Before discussing how to use the characteristics of a model to optimize complex application, it is vital we find a way to measure and evaluate models. As the whole application will have its foundations on those measurements, finding them precisely and reliably is our main goal. Moreover, we are expecting applications to be composed of devices with completely different characteristics, both in hardware and software. Although it is possible to predict some of the characteristics of NNs using their weights ([UKG+20]), in this paper we will rely upon methods to find the empirically. The output of our investigation will be a toolbox which we can easily run on any device to measure the characteristics we are mostly interested in.
In section 3.2 we will explore benchmarking in general terms to make an idea of benchmarks and to develop the requirements we will need to full-fill. In section 3.3 we will dive deep into measuring NNs' performances and to find the tools we can use for our own benchmark. Finally, in section 3.4 we will describe the benchmark and we will use it to measure the performances of some networks.

3.2 Benchmarking

Benchmarking is a widely used tool to evaluate the performance of a system, either software or hardware, whose main goal is to produce consistent and precise measurements of said systems. High precision and affidabilty in benchmarks, however, have been always tricky to achieve and the complexity of these tasks have reached a high complexity in later years, due to advanced processor designs and very intricate interaction between programs and operating systems.[BC18]
In addition, knowledge about the timing behavior of tasks, more specifically their worst-case execution times (WCETs) , is of the highest importance for building reliable and dependable real-time systems.[WDESP17]
According to *von Kistowski et al.* in [vKAH+15], benchmarks should follow certain key characteristics in order to be reliable:

- Relevance
- Reproducibility
- Fairness
- Verifiability
- Usability

In the next section, we will better define them and describe them thoroughly. Understanding these characteristics is essential to produce precise and reliable measurements. Subsequently, we will explore some common techniques used to benchmark software. In section 3.2.3, some of the main roadblocks and pitfalls of benchmarks are described to help the reader avoid them. In the same section, we will also point out some strategies to reduce measurement noise. Finally, in section 3.4, the benchmark used in this paper is described to facilitate its reproducibility.

3.2.1 Key characteristics for a correct benchmark

In this section, we will explore the main characteristics a benchmark should possess in order to be a good benchmark.

3.2.1.1 Relevance

Relevance is probably the most important criteria. Relevance measures how close the behavior of the benchmark relates to the behaviors it is trying to test. [vKAH+15]
Even if the benchmark is realized perfectly and respects all the other characteristics, if the results do not provide relevant information they are of no use. It is when the relevance of the benchmark is taken into account that the computer scientists or engineers need to make the first trade-offs and the first considerations. As a matter of fact, benchmarks that are designed to be highly relevant in a specific scenario tend to have a narrow applicability; vice-versa, benchmarks with a broader spectrum are usually less indicative in specific scenarios. [vKAH+15]
Relevance also relates to the scalability of the benchmark. For example, benchmarks designed to test the full ability of servers might use the full resources that the server offers, for i.e. multi-threading, therefore will not perform correctly in system that do not have this possibility. [vKAH+15]

3.2.1.2 Reproducibility

Reproducibility is the ability to consistently obtain similar, if not equivalent, results if the benchmark is tested on the same environment. [vKAH+15]
Reproducibility is tightly correlated to the ability of describing the environment in perfect details, so that it could be performed on the same environment and expect similar results. The hardware must be described perfectly and software versions must be included and documented, as well as every other configuration done on the system. Especially with the increase of complexity in modern hardware architecture and modern operating systems, variability in performance is introduced by several actor, including timing of thread scheduling, physical disk layout and user interaction. [vKAH+15]
Such variability can be addressed by running the benchmark for long enough periods of time in steady state, therefore without factors like user interaction.

3.2.1.3 Fairness

In order to create a perfect benchmark, fairness is a characteristic that should be taken into account. A fair benchmark is a benchmark that allow the results obtained in different systems with completely different hardware and software to be comparable. Benchmarks are simplistic and, by their own nature, include a certain degree of artificiality, so it is necessary to put some constrains in order to stop the programs to ëxploit"them. Those restrictions can be put, for example, in the software used.
Kistowski et al. in [vKAH+15] point out, for example, a benchmark realized in Java requires a virtual machine on top of the operating systems and the choice of virtual machine can heavily influence the results. This shows how limiting software and hardware components must be carefully limited to insure fairness in the results. However, putting too many limitations may actually hide some of the results, which might be relevant. [vKAH+15] Therefore it is essential to find a good balance amongst the different limitations.
Furthermore, portability plays also a huge role in a fair benchmark. If too many limitations are imposed, or if those limitations are too strict, a big pool of devices could be not suitable to run the benchmark. Similarly to the amount of limitations, portability is also a factor which requires a trade off. As a matter of fact, the portability of a benchmark strictly depends on the very nature of the devices which are set to be used.

3.2.1.4 Verifiability

A good benchmark should be able to provide trustworthy results and also results whose trustworthiness can be verified. To ensure the verifiability of the results, good benchmark usually run some self-verification tasks during run-time, for i.e. if hyper-threading is still active when it should not. Moreover, it is also the case that some benchmarks let users configure some parameters. Those configurations should be documented, as an undocumented alteration might define the result as not trustworthy. [vKAH+15] Finally, it is also good practice to include more metrics in the results as the ones strictly needed, since some red flags might be raised by inconsistencies in these metrics.

3.2.1.5 Usability

Usability is the characteristics of a benchmark which aims to remove roadblocks for users to run the benchmark in their environments. This feature is not limited to how easy is to run the benchmark,which should be high, but some of the aforementioned characteristics relate to this one. For example, a good description of the benchmark also improves the usability of it, since it is easier to replicate. In addition, if the benchmark is too complex or expensive for its purpose might limit a potential user and not allowing them to effectively reproduce the tests.

3.2.2 Benchmark Techniques and Methodologies

This section will delineate the different methodologies to measure the execution time of programs. It will contain techniques for both the *wall time* and the *CPU time*.
We are going to use the term *CPU time* to define the time spent by the CPU to process a piece of software, while we are going to use the term *Wall Time* to indicate the time

```
#include <time.h>

//...

clock_t start,finish;
double total;
start = clock();
// Insert function here
finish = clock();
total = (double) (finish - start) / (double) CLK_TCK
printf("Total = %f\n",total);
```

Fig. 3.1: Example of use for the function *clock()*

elapsed between the start and the end of the execution of a program.
Measuring the wall time of a program is the most straightforward, as it can also be done analogically using a manual stopwatch: simply start the time as the program starts and stop it as the program finishes. This method is undoubtedly imprecise and not very flexible, as it can only be applied to program whose needed measurements are just rough approximation.[Ste01]
The equivalent method in UNIX-like systems is by using the command *date*, which returns the system date and time. Due to the imprecise nature of such method it will not be described further.
A method along the lines of the one described above is by using functions like *clock()* within the program we are trying to benchmark. This function needs to be used within the code of the program, similarly to the snippet of code n.3.1 [Ste01], therefore it is limited to measure only small parts of the program.
In addition, such functions present two other problems. Since the function does not have a standardized implementation, the result may vary from device to device, which makes reproducibility impossible. Furthermore, these type of function needs to be executed as well, similarly to any other function in the code, hence they might introduce overhead and noise in the measurements.

A better way to measure the *wall time* of a program is to use the UNIX command *time*. The synopsis for this program is the following: [Ker10]
The *time* command runs the specified **command**, which can be running a specific program,

```
$ time [options] command [arguments...]
```

Fig. 3.2: Synopsis of the command *time*

and, as the program finishes, it returns the giving timing statistics about this program run. These statistic consist of *(i)* the elapsed real time between start and end of the program, hence the wall time, defined as **real**; *(ii)* the CPU time, excluding I/O blocking functions, time for preemption and the time used for other OS related functions, defined as **user**, or user time;*(iii)* and the time spent by the OS to execute tasks on behalf of the user, defined as **sys**, or system time. The last one includes time for preemption and I/O blocking functions, or any other function associated to the program. [Ste01] It is worth to note

that some shells have a built-in *time* command, hence may vary from the original one. To access the real one, it is suggested to specify its path-name (for i.g. /usr/bin/time).[Ker10]

Nevertheless, we can run a small C++ program to give an example of how to use this command and to analyze its output. The snippet of the code can be seen in Fig.3.3. The program we are going to use simply accumulates numbers from 0 to 10000000000 and prints the result on screen. Before we start accumulating, however, we add a small delay, namely 1 second, to check if this will be counted in the CPU time. Once the code has been compiled, we can run it as shown in Fig.3.2, hence by writing *time ./accumulate* in the shell [1].
From the result we can observe the three indications of time we described earlier (user, system and total time). In the case of the machine used for this paper, the user time was roughly 24,11 seconds, the user time roughly 0,05 seconds and the total time 25,254 seconds.
From these measurements, we can clearly observe that the total time is around one second bigger than the user time and the system time combined and it is due to the delay we introduced in line 6. If we comment out this line and we run the test once again, the total time now roughly equalizes the combination of user and system time (23,46s user 0,03s system, 23,578 total)

```
    #include <iostream>
2   #include <chrono>
    #include <thread>
4
    int main(){
6       std::this_thread::sleep_for(std::chrono::milliseconds(1000));
        double accumulate = 0 ;
8       for(double i= 0; i< 10000000000; i++)
            accumulate += i;
10      std::cout << "Result : "<< accumulate <<std::endl;
        return 0;
12  }
```

Fig. 3.3: Program used to test the command *time*

Even though the *time* command offers multiple information, the time calculated and showed might not be reliable. As pointed out by *Beyer et al.* in [BLW17], this command does not reliable include the CPU time of child processes. This is due to the fact that the Linux kernel is able to count the CPU time of a child process if that process terminated before the parent and the parent waited for it. If this happens, the CPU time of the child is lost and this will interfere with the precision of the measurements.[BLW17],

Beyer et al. in [BLW17] introduces also another another tool for benchmarks in their solution: control groups (cgroups). The Linux manual defines cgroups as ä Linux kernel feature which allow processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored". [Ker21] Cgroups therefore allow the user to assign processes to a group and every child process spawned by the assigned process will be tracked down.

[1] "accumulate" is the name of the program and we assume to be in the correct directory

Cgroups have been initially release with Linux 2.6.24, however, due to the problems with the initial version, starting from Linux 3.10, cgroups version2 have been introduced. In this paper, we will not describe the differences between the two versions, as it is not the purpose of this paper (we refer to the Linux manual for further details [2]) and the *controllers* we are interested in did not change in functionality. *Controllers* allow the user to easily control and measure a specific resource within each group. [BLW17]
The following *controllers* are the ones we are more interested in:

cpu can guarantee a minimum number of CPU resource, even when the system is busy. However, this does not limit the CPU use in case of a free CPU

cpuacct provides information about the accomulated CPU usage

cpuset restrics the number of CPU cores available to a cgroup. In a system with more than one CPU and nonuniform memory access (NUMA) it allows to restric also the process to a subset of the physical memory [BLW17]

freezer allows to suspend and resume all processes in the group

pids allows to limit the number of spawnable processes.

In addition to a more precise calculation of the CPU time, *cgroups* augment also the fairness and the verifiability of the benchmark. By limiting the resources of a more powerful machine, the benchmark can have similar results to a much less powerful one, hence increasing fairness. Although this approach may hinder usability, as it is more cumbersome to set up than using the *time* command, the benefits in precision make this trade-off worth it.

3.2.3 Common Practices in benchmarking

In section 3.2.2 we already discussed some of the issues which can be encountered when trying to design benchmarks. The pitfalls however are not limited to those challenges; on the contrary, it is notoriously difficult to obtain consistent results when designing benchmarks. In this section, we will explore some techniques to effectively reduce the non-determinism of lots of features, both hardware and software, which intend to increase performances. In most cases, since we have little to no control over them, the solution will be to disable these features. Although it will effectively get rid of the non-determinism, the environment we are going to create does not reflect how the application will run on a real application, however it would allow us to obtain consistent results.
The first element we are going to analyse is Turbo-boost. This feature will automatically make the processor core run faster than the marked frequency. *Acun et al.* in [AMK16] calculated an execution time difference of up to 16% amongst processors on the Turbo Boost-enabled supercomputers. Since we have little control over it, the only solution to limit this variation is to completely disable it. The commands to permanently disable it are shoun in Fig.3.4

Modern CPUs often employ a technique called Simultaneous multi-threading (SMT) to speed up execution. This technique makes it possible for two threads run simultaneously on the same core sharing execution resources like the cache or ALU.In case of Intel processor,

[2] `https://man7.org/linux/man-pages/man7/cgroups.7.html`

```
# Intel
echo 1 > /sys/devices/system/cpu/intel_pstate/no_turbo
# AMD
echo 0 > /sys/devices/system/cpu/cpufreq/boost
```

Fig. 3.4: Disable turbo boost in both AMD and Intel devices

the implementation of this technology is called Hyper-threading.
As a result of this implementation, it might happen that the sibling thread steals cache space for example from the workload we want to measure. To avoid these situations and limit the number of CPU-migrations, we will need to disable Hyper-threading by turning down the sibling thread in each core as shown in Fig.3.5.

```
# X being the CPU number
echo 0 > /sys/devices/system/cpu/cpuX/online
# To check the pair of CPU x
/sys/devices/system/cpu/cpuX/topology/thread_siblings_list
```

Fig. 3.5: Disable Hyper-Threading

Another technique we can use to reduce noise in our measurements is to bind a process to a certain CPU core. In Linux we can do that with the *taskset* tool. [Ker21]
Furthermore, we can also increase the process priority with the command *nice*. Processes with higher priority will get more CPU time since the Linux scheduler will favour them compared to processes with normal priority of 10. [Ker21]
A further step to take is to set the scaling governor policy to be *performance*. The scaling governor changes the clock speed of the CPUs on the fly to save battery power, because the lower the clock speed, the less power the CPU consumes. To set the policy to be *performance* one can use the method described in Fig. 3.6

```
for i in /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
do
  echo performance > $i
done
```

Fig. 3.6: Set the scaling governor policy to be *performance* for every cpu

3.3 Benchmarking Deep Neural Networks

Once we understood benchmarks and the characteristics they should possess, we can explore more deeply how we can benchmark Deep Neural Network(DNN).
As expressed by *Reddi et al.* in [RCK+20], "Designing ML benchmarks is different from designing traditional non-ML benchmarks". As a matter of fact, the spectrum of tasks NNs can perform is very broad, consequently there is a wide spectrum of requirements they need to fulfil, making the development of a reliable and relevant benchmark, which is at the same time universal, untreatable. In this paper, we are interested in measuring performances in

images classification tasks in the settings of sugar beet recognition. Moreover, although, benchmarks are usually designed to compare performances, the scope of our investigation is to precisely measure the characteristics of Neural Networks and to find possible correlations among them.
Both academic and industrial organizations have already developed numerous benchmark solutions to evaluate the behaviour of NNs under different workloads and on different devices. For example authors of [LHZ+20] and [ITK+19] have proposed benchmarks of NNs specifically for mobile devices. *Hendrycks et al.* in [HD19] developed a benchmark to assess the robustness of image classifiers under condition of perturbations.
Reddi et al. in [RCK+20]propose a benchmark, namely MLPerf, to evaluate inference of Machine Learning system under various workload. They divided workloads under high level tasks such as image classification and they provide a reference model for it.
On the other hand, *Zou et al.* in [ZAZ+18], among other contributions, proposed a benchmark suite to analyse NNs' training time of eight different state-of-the-art models under six different application domains. The metrics they collect during profiling are:

Throughput
Throughput is defined as the amount of input samples processed by the networks. This metric is relevant because, contrary to inference, is not latency sensitive.

GPU utilization
They define the GPU utilization as the fraction of time in which the GPU is busy and it is calculated as follows:

$$GPU\ Utilization = \frac{GPU\ Active\ Time \times 100}{total\ elapsed\ time} \tag{3.1}$$

FP32 utilization
This metric refers to the percentage of floating operations done of the viable one during training, since typically training DNNs is performed using single precision floating point operations (FP32). This metrics measure how effectively the GPU resources are use. It is calculated as:

$$FP32\ Utilization = \frac{actual\ flop\ counting\ during\ T \times 100}{FLOPS_{peak} \times T} \tag{3.2}$$

Where:

- $FLOPS_{peak}$ is the GPU theoretical peak analysis
- T is the period of time in seconds that the GPU is active

CPU utilization
This is calculated as the average utilization of each CPU core:

$$\frac{\sum_c total\ time\ of\ active\ core\ c \times 100}{CPU\ core\ count \times total\ elapsed\ time} \tag{3.3}$$

Memory consumption
The memory consumed by the NNs during training

Although the aforementioned benchmarks have brought many advancements on the field, they are not meant to be used to study the characteristics of the NNs, but rather to compare and assess NNs performance under different condition and workloads. Therefore, in this section, we will explore techniques to measure NNs performance and determine their behaviour for our own specific goal and under our conditions.

3.3.1 Benchmark Inference Time

As we already delineated in section n.2.3, inference time is one essential characteristics of DNN and some cases fast inference time is an important requirement, therefore speeding up calculations is of vital importance. To achieve faster inference time, and faster calculations in general, DNNs are trained and run on Graphics processing units (GPUs), which means exploiting their hardware acceleration capabilities and their asynchronous execution (multi-threading). ([CCG17], [GDP09], [CBB17], [PJY+13],[OJ04])
In order to measure inference time, we can apply the watchdog approach we discussed earlier in section 3.2.2 as shown in Fig. 3.7. As we already stated in previous section, multi-threading is a factor which must be taken into account when performing benchmarking, since it could be source of indeterminism. To prevent that, in the snippet of code in Fig.3.7, taken from the benchmark developed by *Bianco et al.* in [BCCN18], there is a call to the function *torch.cuda.synchronize()*. This function is used to synchronize the host and the device, i.e. GPU and CPU, so the time is recorded only once the processes running on the GPU are terminated. Although overcoming arguably one of the biggest issues, further inspection of the code reveal some potential errors.

```
def measure(model, x):

    # synchronize gpu time and measure fp
    torch.cuda.synchronize()
    t0 = time.time()
    with torch.no_grad():
        y_pred = model(x)
    torch.cuda.synchronize()
    elapsed_fp = time.time() - t0

    return elapsed_fp
```

Fig. 3.7: Wrong benchmark for inference time [BCCN18]

Although the function *time.time()* is insinuated to be better and more accurate than the UNIX equivalent in the python documentation[3], it calculates the CPU time, hence it is not accurate if the model is run on a GPU. One way to prevent this is to use a CUDA event. Pytorch provides a useful wrapper class for them, namely *torch.cuda.Event*[4]. Moreover, GPU initialization must be taken into account as well. When the GPU is not used, it enters a lower power state in which the GPU shuts down parts of the hardware. In this state, any program which invokes the GPU will cause the drive to load again or the GPU

[3] https://docs.python.org/3/library/time.html#time.time
[4] https://pytorch.org/docs/stable/generated/torch.cuda.Event.html

to initialize once again, processes which might need a significant amount of time. It is therefore necessary to run the model some times before measuring inference time. [Gei21] Finally, one must be careful with transferring data between the CPU and GPU. This is usually done accidentally, when the tensor is created on the CPU and then run on the GPU. This memory allocation process takes a considerable amount of time which will influence the measurements. [Gei21] With these corrections, a more correct measurement for inference can be found with the snippet of code in Fig. 3.8.

```
def measure(model, x):
    device = torch.device("cuda")
    model.to(device)
    dummy_input = torch.randn(1, 3,224,224, dtype=torch.float).to(device)
    start = torch.cuda.Event(enable_timing=True)
    end =  torch.cuda.Event(enable_timing=True)
    timings=np.zeros((repetitions,1))
    #GPU WARM UP
    for _ in range(10):
        _ = model(dummy_input)
    # MEASURE PERFORMANCE
    with torch.no_grad():
    for rep in range(300):
        start.record()
        _ = model(dummy_input)
        end.record()
        torch.cuda.synchronize()
        timings[rep] = start.elapsed_time(end)
    mean_syn = np.sum(timings) / repetitions
    std_syn = np.std(timings)
    print(mean_syn)
```

Fig. 3.8: Correct way to measure inference

In case the model needs to be run on a mobile device, Tensorflow Lite provide benchmarking tools to measure inference time both in warming up and steady state.[AAB+15a]
In case of an Android device, the benchmark can be run the command show in Fig. 3.9.
The result of this command is a log file which can be accessed by the command shown in

```
adb shell am start -S \
  -n org.tensorflow.lite.benchmark/.BenchmarkModelActivity \
  --es args '"--graph=/data/local/tmp/your_model.tflite \
              --num_threads=4"'
```

Fig. 3.9: Run benchmark on android with Tensorflow Lite [AAB+15a]

Fig.3.10.

Finally, since this paper is mainly focused on detection of weed and sugar beet plants from images, and since object detection is usually done with Convolutional Neural Networks (CNN) (Add references), it is also worth mentioning that inference time for these Neural

```
adb logcat | grep "Average inference"
```

Fig. 3.10: Command to access the inference in Tensorflow Lite[AAB+15a]

Networks can be calculated manually knowing the architecture of the model. Inference time is the time the model needs to forward propagating the image through each layer, therefore is the amount of computations needed to be performed at each steps. These operations are defined as Floating Point Operation, or FLOP. The amount of FLOPs is different for each layer and require different information to be computed.
In case of the convolutions layers, the number of flops is calculated with the following equation:

$$C\_FLOPs = 2 \times Nk \times Ks \times Os \tag{3.4}$$

where:

Nk - number of kernels

Ks - kernel shape (3x3,5x5, 7x7, etc.)

Os - Output shape

For the fully connected layers, the amount of FLOPs is defined as:

$$FCL\_FLOPs = 2 \times Is \times OS \tag{3.5}$$

where:

Is - Output shape

Os - Output shape

For the pooling layer, the number of FLOPs also depends whether there is a stride or not. In case of a stride-less pooling layer, the formula to use is the following:

$$PL\_FLOPs = H \times D \times W \tag{3.6}$$

where:

H - height of the image

D - depth of the image

W - width of the image

In case of a stride, equation n.3.6 becomes:

$$PL\_FLOPs = (H/S) \times D \times (W/S) \tag{3.7}$$

where:

S - stride

To better understand the use of the aforementioned equations, we can calculate the FLOPs of the neural network described in Fig.3.11.

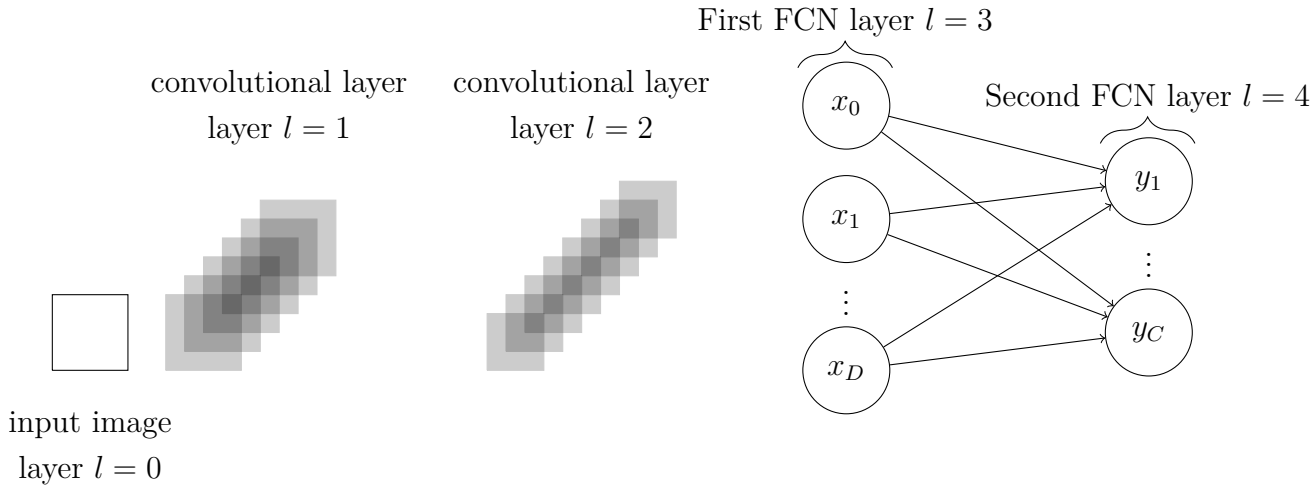


Fig. 3.11: Architecture of the convolutional neural network used as an example to calculate inference time. Layer $l = 0$ is the input layer, with dimension $28 \times 28 \times 1$; $l = 1$ is the first convolution layer with a kernel of 3x3x5 and an output shape of $26 \times 26 \times 5$; $l = 2$ is the second convolution layer with a kernel of $3 \times 3 \times 5$ and an output shape of $24 \times 24 \times 5$;$l = 3$ is the first fully connected layer (FCN) with an input size of $24 \times 24 \times 5$ and an output size of 128;$l = 4$ is the second FCN, and final layer, with an input size of 128 and an output size of 10

The total number of FLOPs is given by:

$$\begin{aligned} &FirstConvolution - 2x5x(3x3)x26x26 = 60,840FLOPs \\ &SecondConvolution - 2x5x(3x3x5)x24x24 = 259,200FLOPs \\ &FirstFCLayer - 2x(24x24x5)x128 = 737,280FLOPs \\ &SecondFCLayer - 2x128x10 = 2,560FLOPs \\ &FLOPs = 60,840 + 259,200 + 737,280 + 2,560 = 1,060,400 \end{aligned} \tag{3.8}$$

Supposing that the processor used performs at 1 GFLOPS (Floating Point Operations per Second), the inference time is given by:

$$\begin{aligned} InferenceTime &= \frac{FLOPs}{FLOPS} \\ &= \frac{1,060,400}{1,000,000,000} \\ &= 0.0010604 \\ &= 1,0604ms \end{aligned} \tag{3.9}$$

Equation n.3.9, however, only reveals a theoretical, ideal, value for inference time. As a matter of fact, even though the number of FLOPs remains constant, the amount of FLOPS in a processor might not.The number of flops is calculated as :

$$FLOPS = Number\_of\_Cores \times Average\_frequency \times Operations\_per\_cycle \tag{3.10}$$

Although the number of cores is an easy-to-find and a stable value, the average frequency

and operations per cycles are not. The operating frequency is usually a lower bound of the actual operating frequency and in modern processors may vary drastically due to technologies such as TurboBoost(Intel) or Turbo Core(AMD). For example, a modern Intel®Core™ i7-4500U Processor has a base frequency of 1.80 GHz, but it can reach 3.00 Ghz with TurboBoost.[5]. To have an estimation of the FLOPs of your machine, you can use Intel MKL benchmark suit, which solves linear system of equations to estimate them. [ZS21]

3.3.2 Benchmark Training Time

As we discussed already in section n 2.2, the training time is the time needed by the model to update its weights. Learning is usually done in *epoch*, which defines one cycle through the full training dataset. [AR20] The number of epochs is defined by the modeller and it can drastically influence the time needed for training. Logically, since more epoch means more cycles, hence more calculations, more time is needed to complete the training.
Some libraries already provide tools which measure training time, like for example *fast.ai*. When the wrapper function *fit_one_cycle()* of the class *Lerner* is invoked, at the end of every epoch, returns, in addition to the metrics defined by the user like accuracy or error rate, also the time required for that epoch. An example can be seen in table 3.1. Such results can also be saved in a csv file at the end of the process.

epoch	train loss	valid loss	accuracy	time
0	2.310583	0.764163	0.767841	28:27
1	1.031493	0.420149	0.874435	30:12
2	0.549505	0.350187	0.889792	31:15
3	0.369313	0.336940	0.891599	30:15

Table 3.1: Metrics from fast.ai fit_one_cycle() function. The time is in minutes

However, if more precision or more control is needed, training time can also be calculated using the watch dog approach, as shown in Fig. 3.12. Even though a bit of overhead is introduced with more function calls, usually this overhead does not impact the measurement significantly. Furthermore, if the training is done on a GPU, a similar discussion to the one we made in section 3.3.1. In the example in Fig. 3.12 is used, however CUDA event should be used for more precision, similarly to Fig.3.8.

3.3.3 Measuring Accuracy

As mentioned in section n. 2.5, accuracy is usually one of the most looked-after metric in a neural network. During the analysis of training time, we are going to use the accuracy provided for us by *fastai* at each epoch. *fastai* calculates the accuracy during validation by calculating the mean between two tensors, as shown in Fig. 3.13. [fas21]

[5]Intel Documentation

```
    time_elapsed = []
    for epoch in range(1, args.epochs + 1):
        # start time
        torch.cuda.synchronize()
        since = int(round(time.time()*1000))
        train(args, model, device, train_loader, optimizer, epoch)
        torch.cuda.synchronize()
        time_elapsed[epoch] = int(round(time.time()*1000)) - since
        print ('training time elapsed {}ms'.format(time_elapsed[epoch]))
    print ('training time elapsed '+ str(sum(time_elapsed)) + 'ms')
```

Fig. 3.12: Benchmark for training time

```
def accuracy(inp, targ, axis=-1):
    "Compute accuracy with `targ` when `pred` is bs * n_classes"
    pred,targ = flatten_check(inp.argmax(dim=axis), targ)
    return (pred == targ).float().mean()
```

Fig. 3.13: Fastai calculation for accuracy [fas21]

3.4 Benchmarking Tool for Neural Networks

Following the characteristics delineated in section n. 3.2.1 and the techniques we analysed in sections n. 3.2.2 and n. 3.3, we can build a tool which allows us to study the behaviour of Neural Networks.
Conceptually, the behaviour of the suite is shown in Fig. 3.14.

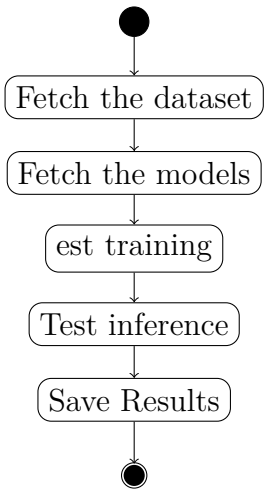


Fig. 3.14: Behaviour of the benchmarking suite on a conceptual level

The tool will allow us to study the behaviour of neural networks in order to find correlations amongst the metrics we analysed in chapter n. 2, therefore the main requirement it needs to respect is flexibility. Applications in the field of sugar beet recognition may be subjected to various requirements and different conditions, hence the tool must be able to reflect that. In other words, it should be flexible enough so that we are able to configure it in multiple ways to reflect the various condition, but at the same time it is also able to

produce results that are specific for our scope, i.e. finding the correlation between different metrics. Furthermore, we need the results to be verifiable and trustworthy. Being the foundation for further steps, we need to be able to trust the results and they need to be precise and accurate enough so that they can be reasoned about.
Finally, we are expecting the tool to be run on multiple devices, hence usability is also a factor to pay attention to. If the tool is too difficult to set up or run, it will lead to an undesirable waste of time and resources.
Flexibility is also a key factor for the choice of the data-set. For example, studies in the field of weed recognition use various techniques to collect the images for their data-sets. For examples, authors of [LY20] use the BOSCH's Bonirob system, which is a field robot, to collect various data about the plants, while authors of [BHC18] use Unmanned Aerial Vehicles (UAVs). Moreover, the way images are labelled is data-set dependent. Therefore, in order to insure flexibility, the user must have full control over the data-set and how it is represented in the tool.
As already discussed in section n. 2.9, the choice of the architecture is limited amongst Resnet, Alexnet and VGG. The models which can be analysed will be the ones listed below.

- Resnet18
- Resnet34
- Resnet50
- Resnet101
- Resnet152
- Alexnet
- VGG16
- VGG19

Both the indication regarding the dataset and the choice of the models will be indicated in an external configuration file by the user. The tool will read this file before starting the test to collect the necessary information to run. This configuration file helps to improve both flexibility and usability, in addition to automatically create a documentation of the run for reproducibility.
To further increase usability, a logging system will be implemented so that information about the run and potential errors are visible to the user and can be corrected.

The actual implementation of the tool is done using Python and revolves around the *fastai* library. This library is built on top of Pytorch, a very common library for machine learning, and offers powerful functions and high flexibility without a significant drop in performances. [HG20]
fastai suits perfectly the goals of the benchmarking tool as it is *"approachable and rapidly productive, while also being deeply hackable and configurable"*. [HG20]
Thanks to *fastai*, we also have the possibility to use publicly available implementation of the models listed above, hence removing possible differences in results coming from divergent implementations.

In the following sections, we are going to describe how the the training process and the inference time are analysed.

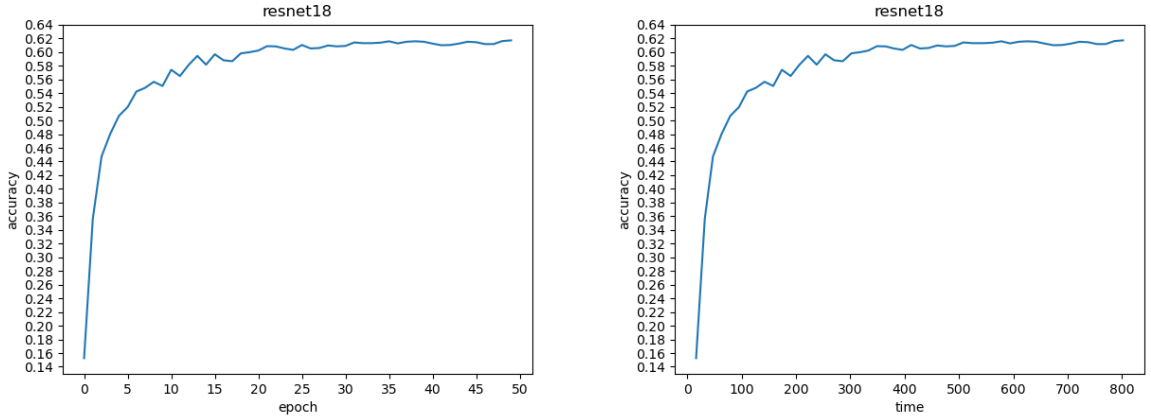
3.4.1 Analysing the training process

epoch	time	accuracy
0	00:16	15.25710374116897
1	00:16	35.62246263027191
2	00:15	44.72259879112243
3	00:16	48.03788959980011
4	00:16	50.67659020423889
5	00:16	51.96211338043213
6	00:15	54.22868728637695
7	00:16	54.769957065582275

Table 3.2: Example of training time results

Training time is the first characteristic of a models which can reveal some important information and can be used to optimize applications. Fastai provides 3 functions for training a model, namely *fit()*, *fit_ one_ cycle()* and *fine_ tune()*, and each of these functions has a different purpose.
The function *fit()* is simply a wrapper around the *train()* function of PyTorch and it represents a very basic training loop.[fas21]
On the other hand, *fit_ one_ cycle()* implies a phenomenon called *Super-Convergence* which allows fast training of Neural Networks exploiting the learning rate. [ST17] One of the key elements of super-convergence is training with the one-cycle policy developed by *Smith* in [Smi18]. [ST17]
Finally *fine_ tune()* can be seen as a particular combination of *fit_ one_ cycle()* and *(un)freeze* which works well in a lot of cases. *fine_ tune()* is geared mainly towards transfer learning, i.e. towards training pre-trained models with a new dataset, since it employs the *freeze()* function. Such function is used to freeze the first layers so that the weights do not get updated during further training.[fas21]
Since the three functions have different scopes and are beneficial in different kinds of applications, the benchmarking tool is configurable in a way that the preferred way of training can be chosen.
Once the models and the training method have been selected, the tool will start training each model accordingly for the number of epochs chosen by the user. Once the training is complete, the results are stored and then represented into a graph.
As suggested in section n. 3.3.2, there are two ways to record training time. For the purpose of this paper, we will use the first one, i.e. the one offered by *fastai*. Even though the second method (Fig. 3.12) is be more precise, it does not give actual indications about the time taken for each epoch. On the other hand, this method neglects the time taken to validate the training step, hence not considering this small delay for every epoch. As a result the total training time does not comprise this delay which, even though potentially insignificant, it increases as the number of epochs increases and it will influence the total

training time. However, using the first method, we also have information of the accuracy obtained at each epoch and thus allowing us to better reason about it. Table 3.2 shows an example of the information extracted from this process. Furthermore, this information is then extrapolated and graphed, as shown in Fig. 3.15a and Fig. 3.15b and saved as a serialized file.



(a) Example of the accuracy graphed against the number of epoch used to train for resnet18

(b) Example of the accuracy graphed against time taken to train in seconds for resnet18

Fig. 3.15: Example of graphs produced by the tool when analysing training time

3.4.2 Analysing inference time

As described in section n. 2.3 inference time is the time needed for the model to make predictions. We discussed in section n. 3.3.1 different ways to measure inference time, including how to calculate it considering the architecture of the Neural Network we chose. This last method, however, is not suitable for our purpose, since it is not precise enough to lead to correct predictions, therefore the method used for the benchmark tool is the one showed in Fig. 3.8, with the only difference that the tool will save the prediction result and calculate the total accuracy achieved. The results will be then graphed as shown in Fig.3.16.

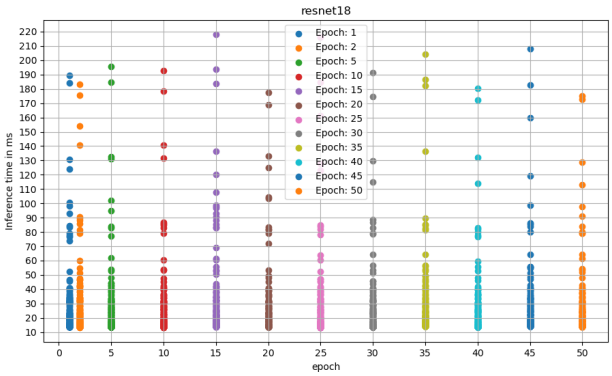


Fig. 3.16: Example of the graphs produced by the tool when analysing inference time

Fig. 3.16 shows an example of Resnet18 trained for various epochs and tested using 50

images which were not part of the training set. During the analysis of inference time, the tool will calculate accuracy as the percentage of corrected predictions over the whole set, as shown in equation n. 2.4. Since we are not going to treat any binary classification problem, we are going to avoid equation n. 2.5.
Inference time is analysed based on the number of epochs used for training the models. A complete overview of the process is shown in Fig. 3.17.
The user indicates a list containing all the epoch to use for training. For example, in Fig. 3.16 the epoch inserted were:

$$1, 2, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50$$

Once the model has been trained for the respective number of epoch and the inference has been tested, the results are saved and the model initialized. The process ends when the inference time of all the models have been trained for each epoch in the list.

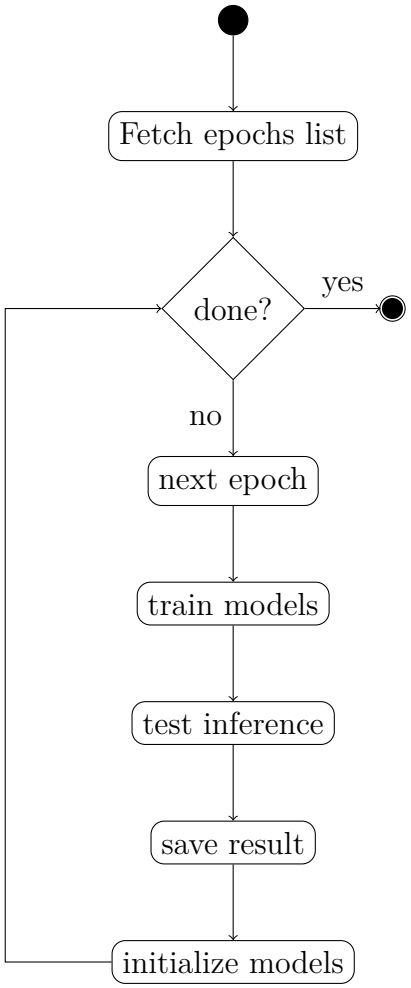


Fig. 3.17: Overview of the process to test inference

3.4.3 Further Analysis on the Metrics

Once the tests terminate the tool stores the information collected in a file on the disk. The information stored are:

- Epoch
- Training loss
- Validation loss
- Accuracy
- Training time for epoch
- Inference Time
- Prediction
- Ground Truth

The data, however, it is serialized before saved, therefore it can be visualized and further investigated with the use of a ready-to-use Jupyter notebook ([KRKP+16]). In addition to the raw visualization of the data, the notebook allows the user to elaborate information for further analysis. This notebook calculates the average training time needed for each epoch, the highest accuracy achieved during training and the highest accuracy during the prediction test. Furthermore, it also calculates precision (equation n. 2.7), recall (equation n. 2.8) and the F1-score (equation n. 2.12).
The notebook gives also the possibility to graph the trend of the single models during the test both over training time and epochs. In addition to the trend during training as discussed in section n. 3.4.1, it is also possible to visualize the validation loss and training loss over training time.

4 Analysis of the characteristics

4.1 Introduction

In this chapter, we will analyse and measure different characteristics from various neural network architecture with the purpose of finding useful correlations which we can use in a later stage. In order to achieve this goal, we will use our understandings from chapter 2 of each metric of Neural Network and the tool we developed in section 3.4.
In challenges such as the ImageNet classification challenge ([RDS+15]) the ultimate goal is to achieve the highest accuracy possible, neglecting other performance metrics like inference time. [CPC16]
Although accuracy is of high importance, in practical applications other metrics are to be considered as well, depending on the different requirements. As pointed out by *Canziani et al.* in [CPC16], metrics like inference time, parameters and operations count are hard constraints for the deployment of Neural Networks in practical applications. Furthermore, training time is often a time consuming process which highly depends on factors like the complexity of the task, size of the network and training set([PE89]).
Finding relations between these metrics and other factors as well, like influence of a given input feature to the prediction of the model ([HEKK19]), will allow us to optimize applications, saving time and resources in the process.

4.2 Environment use for the test

All the experiments have been run on the same machine running Ubuntu 20.04.3 LTS (Focal Fossa). For the specific of the machine, please refer to table 4.1

CPU	AMD EPYC 7452 32-Core Processor
CPU MHz	1499.324
CPU max MHz	2350,0000
CPU min MHz	1500,0000
Total memory	1056709772 kB
GPU	Nvidia A100-PCIE-40GB
Number of GPUs	8

Table 4.1: Specifics of the machine which run the experiments

4.3 Datasets and Training Methodology

It is assumed for all the experiments that, if no specification is made, the training of the models has been carried out by the *fit_one_cycle()* function present in fastai using the default learning rate. Furthermore, the models have not been pre-trained, hence no transferred learning is applied, and the models have been trained using full precision.
For each experiment we use rather different datasets, some of which are not related to the farming world. In the first experiment, we use the dataset proposed by *Vevaldi et al.* in [PVZJ12], which will be referred to as "the Pets dataset"for the rest of the paper. This dataset is directly accessible from the library and contains 37 category of pets, with roughly 200 pictures each.
For the second experiment, we are going to use the dataset proposed by *Giselsson et al.* in [GJJ+17], which we are going to refer to as the 'plant_seedlings_v2' dataset. This dataset contains ~1000 RGB images with a resolution of 10 pixels per mm divided in 12 different plant spices. The plants in the dataset are listed in table 4.2. This dataset contains pictures of one of the most common weed found in sugar beets plantations, a plant commonly referred to as "charlock". [CM10]
Even though the dataset is mainly focused on seedlings and it contains pictures of other plants as well, this will give us proper insights of the models' behaviours in a farming settings.

English	Latin
Maize	Zea mays L.
Common wheat	Tricicum aestivum L.
Sugar beet	Beta vulgaris var. altissima
Scentless Mayweed	Matricaria perforata Mérat
Common Chickweed	Stellaria media
Shepherd's Purse	Capsella bursa-pastoris
Cleavers	Galium aparine L.
Redshank	Polygonum persicaria L.
Charlock	Sinapis arvensis L.
Fat Hen	Chenopodium album L.
Small-flowered Cranesbill	Geranium pusillum
Field Pansy	Viola arvensis
Black-grass	Alopecurus myosuroides
Loose Silky-bent	Apera spica-venti

Table 4.2: Categories of the 'plant_seedlings_v2' dataset [GJJ+17]

4.4 First experiment

As mentioned in section 4.3, this experiment has been carried out using a data-set which is rather far from the agricultural field. However, it already gives us some insights of what

is to come.
The first metrics we are going to analyse are number of epochs, training time and accuracy. We will study those metrics to be able to recognize some patterns and use those to be able to find correlations between the three with the final aim of being able to predict one of them knowing the others. These prediction patterns can be used to save time during the learning process in future applications, as we can estimate the accuracy before starting the process.
The benchmarking tool run the test for each model using three different amounts of epochs, i.e. different training time, in order to simulate three different scenarios: one example with low training time,one with a medium training time and finally one with a very high training time. The tool run with ten, fifty and 100 epochs respectively.
The one scenario which yield more promising results and the one we are going to analyse first is the one with fifty epochs. Fig. 4.1 shows the results of each model's accuracy graphed against the number of epochs used for training, while Fig. 4.2 shows the results of each model's accuracy graphed against the necessary training time needed to reach that accuracy.

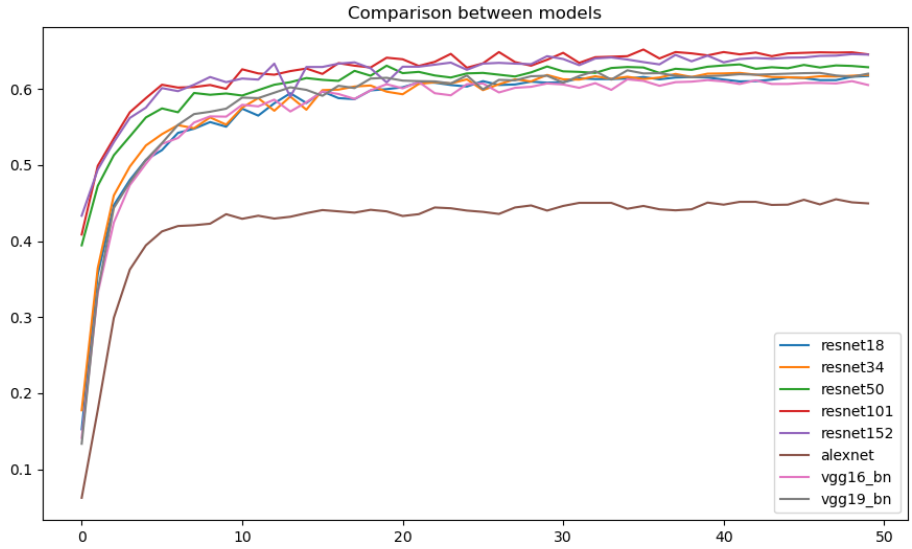


Fig. 4.1: Comparison between epoch/accuracy for each model. The x axis is the number of epoch, while the y axis is the accuracy achieved

As suspected, for each model, the accuracy grows logarithmically higher as the number of epochs increments, or as the training time increments. A closer inspection of Fig. 4.2 lets us derive other conclusions. Alexnet finishes training in considerably less time compared to the other networks(~6 minutes), reaching however the lowest accuracy overall(45%). We can observe this difference in time by looking at Fig.4.3 and Fig.4.4, which shows the behaviour of Alexnet, Resnet101, Resnet152 and VGG19 in the same settings.
As also shown in the previous graphs, Resnet101 reached overall the better accuracy at around 65% with a training time of ~62 minutes, second only to Resnet152, which needed ~90 minutes to reach an accuracy of ~64%. Finally, VGG19 took ~60 minutes to reach an accuracy of 61%.
In order to collect more information about the response of the model, we should take a closer look to how they performed individually.

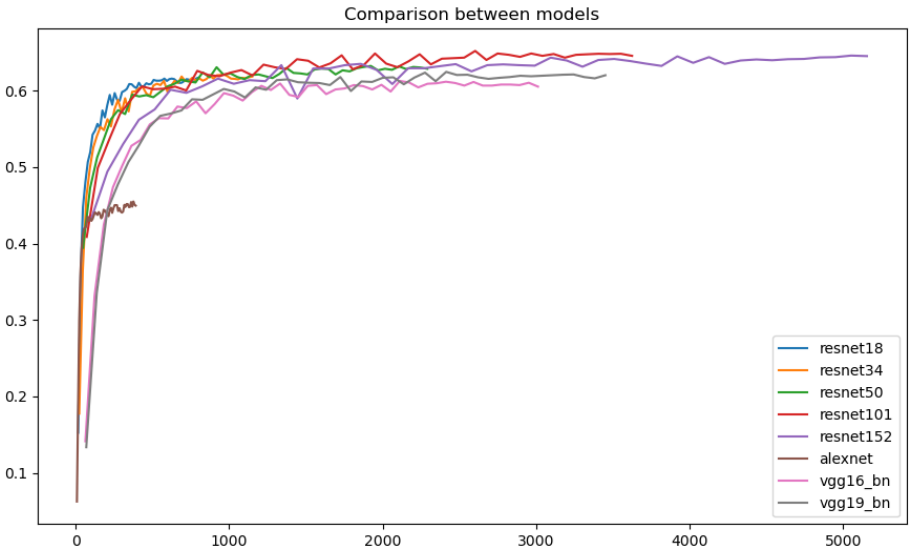


Fig. 4.2: Comparison between training time/accuracy for each model. The x axis is the training time in seconds, while the y axis is the accuracy achieved

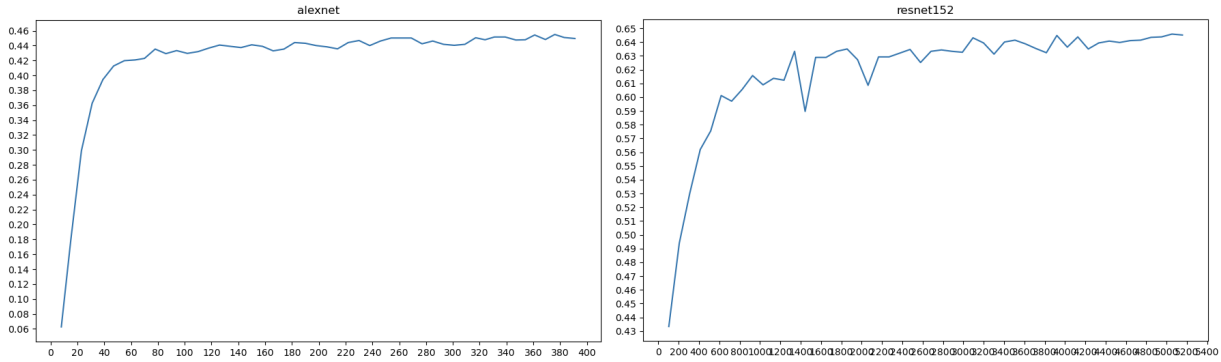


Fig. 4.3: Accuracy of Alexnet and Resnet152 against training time in seconds

Fig.4.3 and Fig.4.4 also show how stable each model were during training. The stability we are observing right now is how fluctuating each model has been during training regarding its accuracy. The less fluctuating it is, the better we able to predict the accuracy from training time or number of epoch, and vice-versa. From the results, we can see that Resnet152 and Resnet101 tend to fluctuate more compared to Alexnet or VGG19 (Fig. 4.4).
Such fluctuation, however, does not hide a trend which is in common amongst all models: after a certain number of epochs, the accuracy tends to stabilize and grow significantly slower. Fig. 4.1 can help us locate the point at which the accuracy stops increasing at a high rate at around 10 epochs and this is further proved by Fig. 4.5.

We can further analyse the behaviour of each models for the first 10 epochs by observing Fig. 4.6. This graph gives us a closer look to how the models have been trained and how the curve looks like. Differently from the previous graph, Resnet152 this time reached a higher accuracy, however it was also the models who took the most time to fully complete

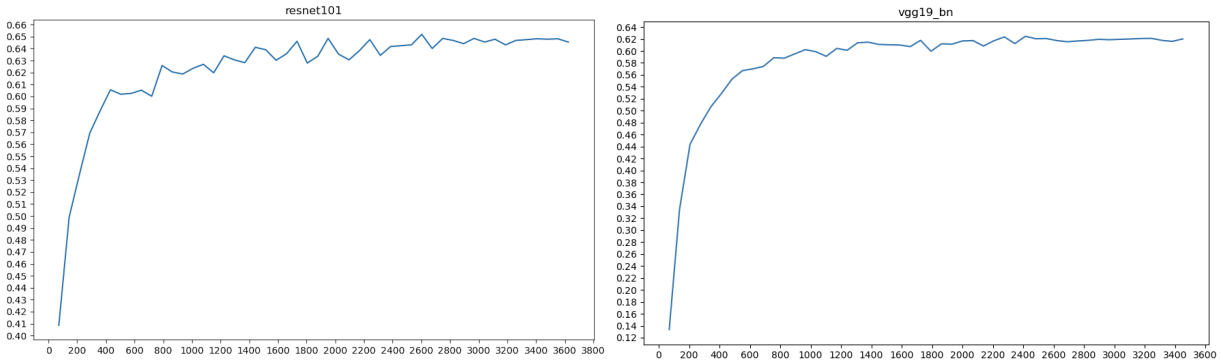


Fig. 4.4: Accuracy of Resnet101 and VGG9 against training time in seconds

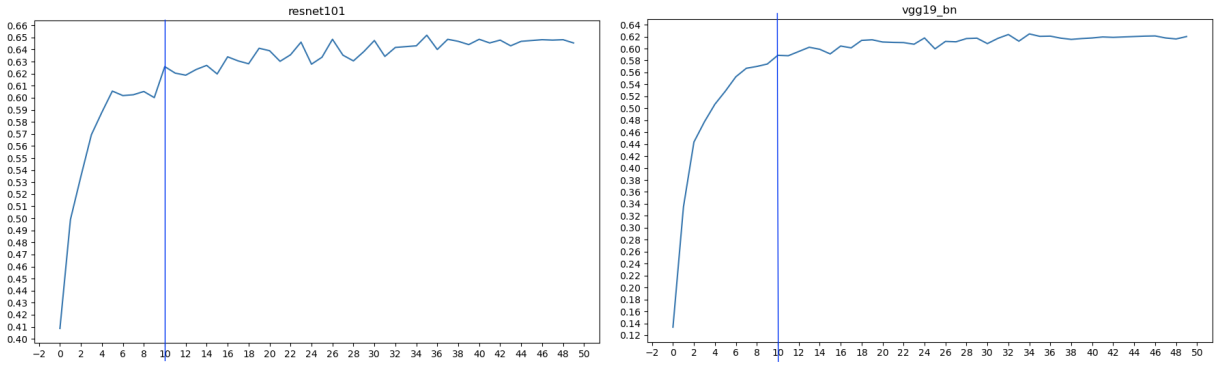


Fig. 4.5: Breaking point of Resnet101 and VGG19

the training.
On the other hand, from Fig. 4.6, we can clearly observe that, if compared with each other for the same training time, shallower networks like Resnet34 or Resnet50 achieved higher accuracy than deeper networks like Resnet152. This is obviously due to the fact that within the same training time shallower networks manage to complete more epochs, therefore complete more training cycles. As a matter of fact, if we were to compare models on an epoch base we will find that deeper networks will achieve better accuracy given the same number of epochs.
If we observe Fig. 4.2 before the 1000 seconds mark, we can see that Resnet18's curve starts to flatten reaching an accuracy of ~61%, while the others tend to reach smaller accuracy values. Around the 1000 seconds marks the behaviour of all the models starts to equalize and afterwards the accuracy of deeper networks will increase reaching higher values. As mentioned previously, this is due to the models being able to finish more epochs within the same time frame. In this case, the models reached to finish the training completely, as shown in 4.2. Models from different architectures do not follow this trend. Alexnet, as we already discussed above, does not manage to reach somewhat close to the same accuracy of the other models. VGG16 and VGG19 follow similar trends and both curves overlap multiple times. Even though VGG19 is considerably bigger than VGG16 ([SZ15]), they reach very similar accuracy even before the 1000 seconds marks with very similar training time.

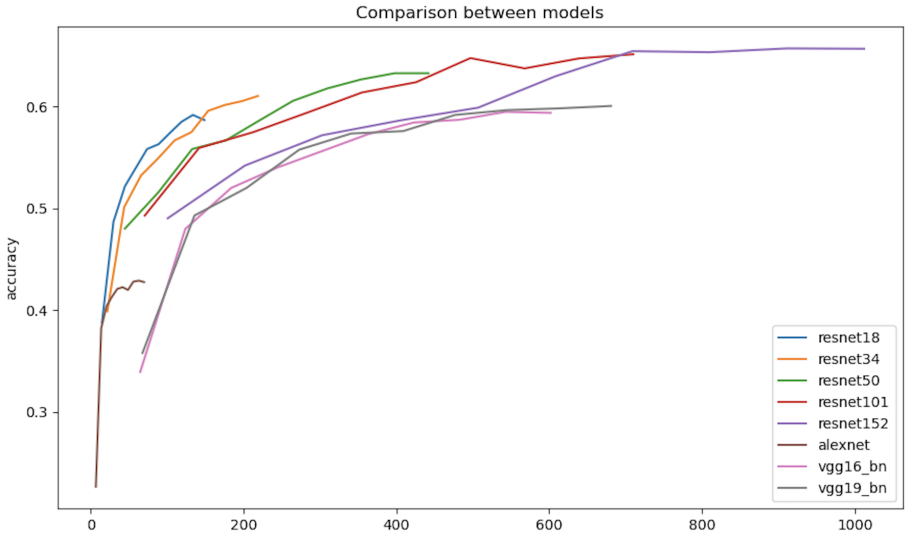


Fig. 4.6: Comparison between training time and accuracy for each model for 10 epochs. The x axis is the training time in seconds, while the y axis is the accuracy achieved

This behaviour is further highlighted in Fig. 4.7 which shows the behaviour of the models trained for 100 epochs. We can see that once again around 1000 seconds the curve of every model starts to flatten and the models using the Resnet architectures achieve similar accuracy. The highest accuracy is achieved by Resnet152, which also needed the most training time. Surprisingly, Resnet50 performed better than Resnet101 achieving better accuracy with less training time. VGG16 and VGG19 performed similarly displaying overlapping curves, with VGG19 once again requiring more training time.
More importantly, however,this graph confirms the results and the hypothesis we made previously.Furthermore, we can use all the data we acquired to calculate the average training time required for each epoch. The results are provided in table 4.3.

Model	Time (s)
Resnet18	15.01
Resnet34	22.0
Resnet50	45.02
Resnet101	72.11
Resnet152	102.02
Alexnet	7.01
VGG16	60.09
VGG19	68.97

Table 4.3: Average time for each epoch

Training for 100 epochs gives us also more complete insights regarding the future performance of our models. In other words, we can determine when the model starts to over-fit or under-fit and when to stop the training to avoid future poor performances.

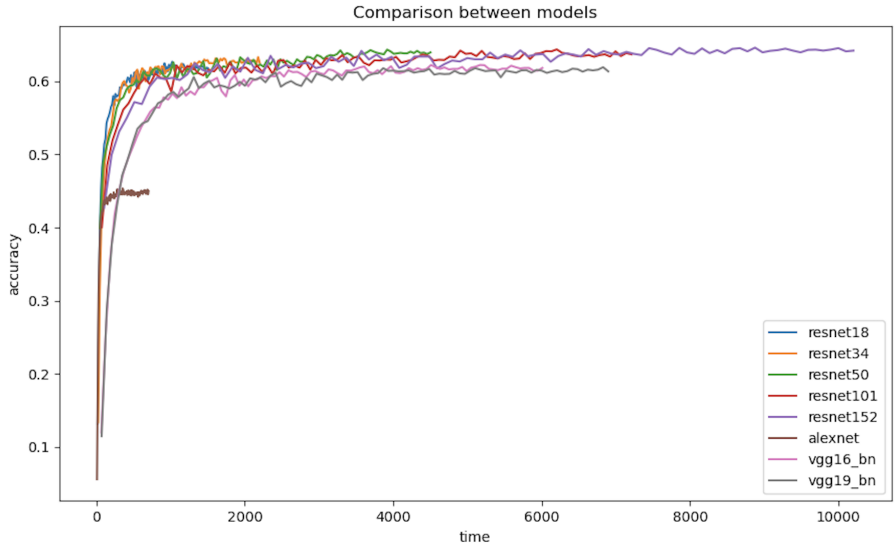
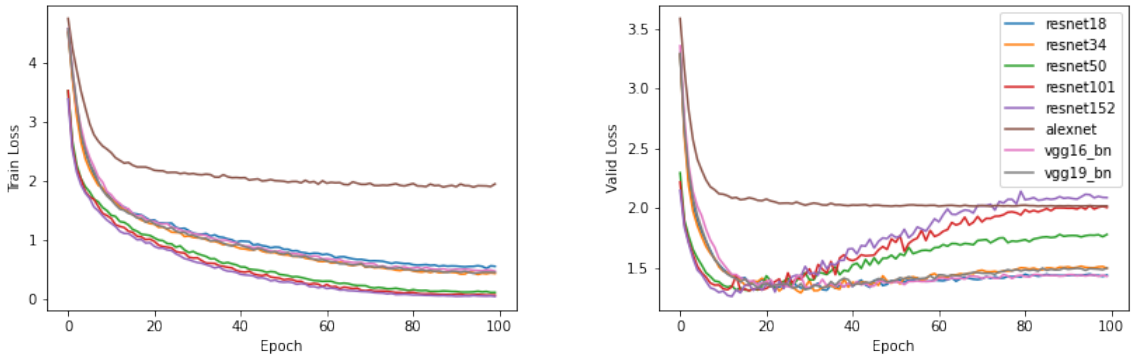


Fig. 4.7: Comparison between training time and accuracy for each model trained for 100 epochs. The x axis is the training time in seconds, while the y axis is the accuracy achieved



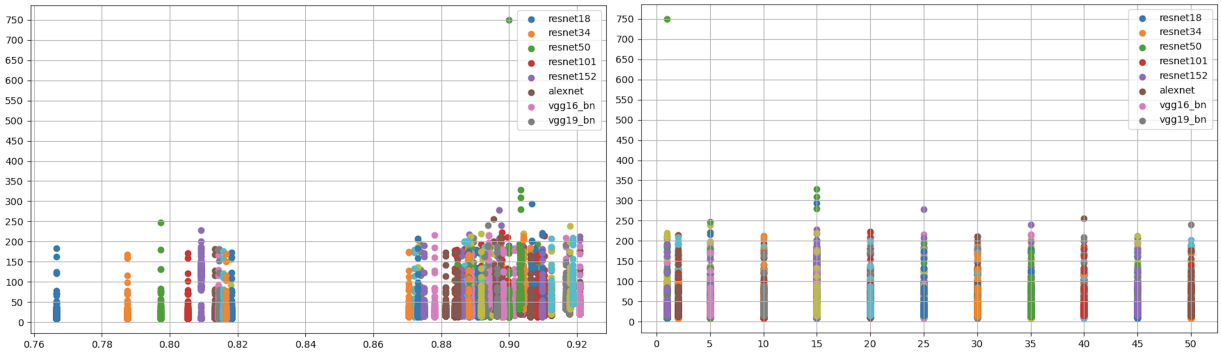
(a) Training Loss calculated over 100 epochs **(b)** Validity Loss calculated over 100 epochs

Fig. 4.8: Training loss and validity loss of all models calculated over 100 epochs

As shown in Fig. 4.8a, the train loss decreases at each epoch for each model. For Alexnet, the curve tends to flatten at around 15 epochs, while for the others it flattens at around 60. Rather than observing the training loss trend alone, however,which does not give us the possibility to comprehend correctly the response of the models, we should compare it to the trend of the validation loss shown in Fig. 4.8b. Alexnet remained stable for the duration of the training, with a validation loss comparable to the training loss. The other models, on the other hand, show a rather different behaviour. At around 20 epochs, the validation loss of deeper networks, i.e. Resnet152, Resnet101 and VGG19 starts to increment drastically. For shallower networks of the Resnet architecture, i.e. Resnet18, Resnet34 and Resnet50, and for VGG16 the validation loss decreased for the first 15 epochs and started to increment only after ~40.
In section n. 2.8 we defined over-fitting to be a situation in which the validation loss is much larger than training and from Fig. 4.8b we can see that, although after various number of epochs, most of the networks start to enter this condition as the validation loss increases and it becomes much larger than their training loss.We also discussed some

techniques to avoid this, like for i.g. Cross-Validation. For the purpose of this experiment, we only split the dataset 80-20, hence we used no cross-validation or augmentation on the data-set whatsoever.

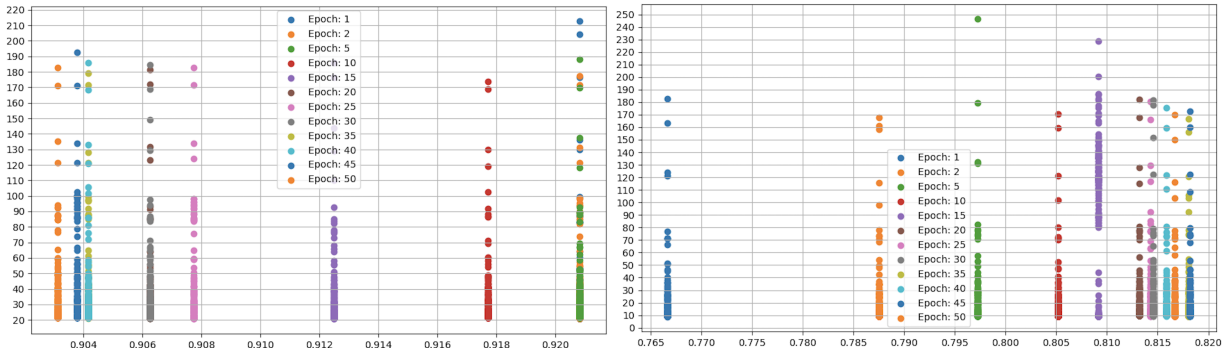
In addition to the training time, we can also use the benchmark tool we developed to measure and analyse the inference time of each model.
As we are mostly focused on sugar beet recognition, it is safe to assume use cases where field robots would scan the field to recognize the vegetation, similarly to the one proposed by *Lottes et al.* in [LHS+16b]. In such setting, the time taken to classify the image results in a soft deadline, as the time taken to scan the field is greatly influenced by it, therefore being able to estimate the needed inference time could help optimize this part of the application.
To measure inference time, we need to collect a dataset of related pictures which are not part of the training dataset to feed to each model. For our tests, we are going to use a dataset comprising of 200 random pictures. The pictures we are going to use are going be of different dimensions and different quality in order to see if we can recognize patterns. We can see the results in Fig. 4.9, which displays the training time in milliseconds graphed against the accuracy and the number of epoch used to train. From this figure, we can clearly see that the inference time for every model rarely is measured to be more than 230 milliseconds, with the exception of few outliers, and most of the models for most epochs have an accuracy between 87% and 92%. In addition, if we analyse the inference time based on the number of epoch (Fig. 4.9b) the models display similar responses.



(a) **(b)**

Fig. 4.9: Inference time measured for each model using the 200 pictures dataset discussed previously. The inference time is in milliseconds, while the accuracy for Fig. 4.9ais the percentage of correct predictions.

When we compare model individually like in Fig. 4.10, other similarities appear. As a matter of fact, when we analyse each model, we can observe that some pictures require considerably more time than others. For example, Fig. 4.10 shows the measurements obtained by model Resnet18 (4.10a) and Alexnet (4.10b) and from their response it appears that, at each epoch, there is a constant number of images which takes more time to be processed.
We can run the tool once again to identify the 10 images that took more time to be processed at each epoch in order to analyse them and find elements which can explain



(a) Resnet18 **(b)** Alexnet

Fig. 4.10: Inference time measured for model Resnet18 and Alexnet

such difference.
The first property of the image we are going to take a look is the size of the images. Fig. 4.11 shows the results obtained for each model.

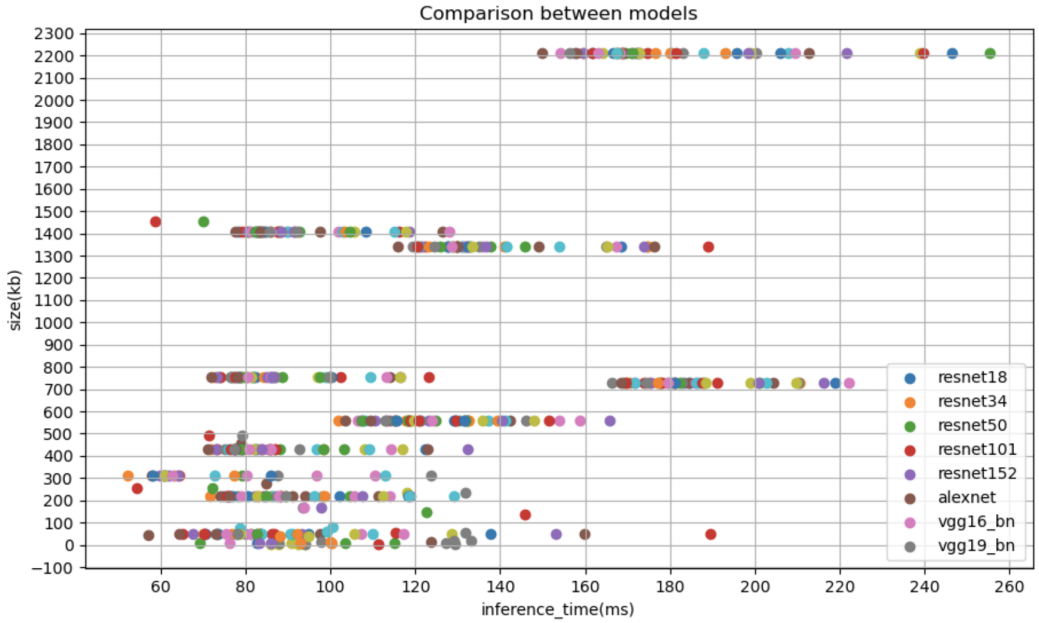
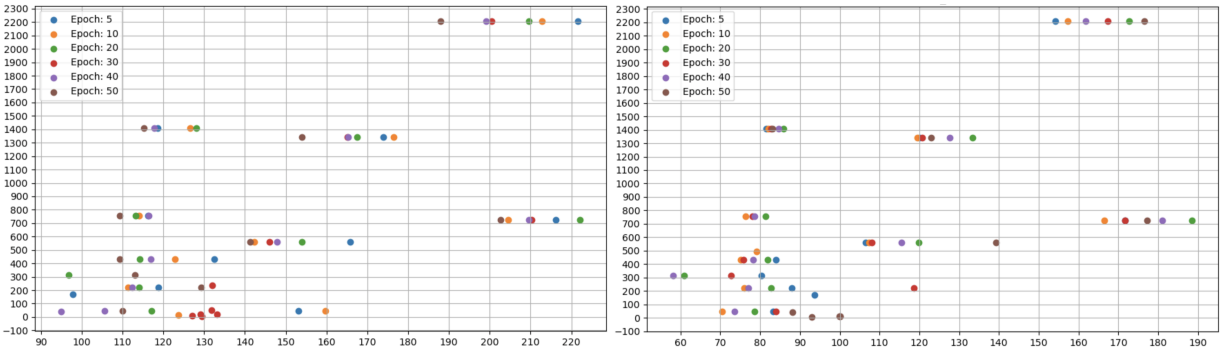


Fig. 4.11: This graph shows the size in kb of the ten slowest images over the time taken to be processed

From the graph we are able to spot some rather interesting behaviours. First of all, we would expect that for each epoch the slowest images would be the same. This hypothesis would be confirmed if the graph showed group of pictures of the same size having different inference time. However, this is only the case for sizes bigger than 500 kbs. As a matter of fact, we are not able to cluster pictures before 500 kbs under a certain inference time range as effectively as we can do for heavier pictures. We can conclude from this that regardless of the amount of training, pictures over 500kbs are going to be the slowest ones.
From a closer investigation of the individual models emerged some differences in the response of the single models.
For deeper networks the situation is similar to the discussion we made. As shown in Fig. 4.12, for both Resnet152 and VGG16, the response for pictures smaller than 500 kb is

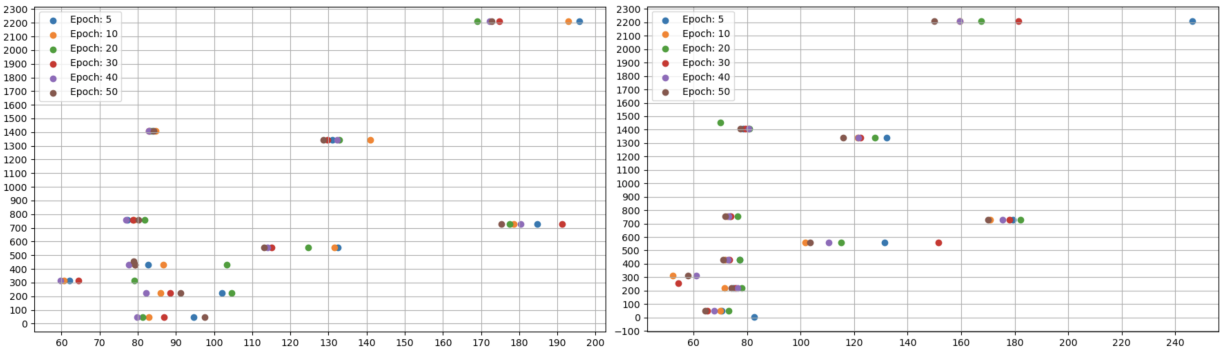
noisy, although Resnet152 show a more stable behaviour than VGG16. This implies that for these networks only the response with images over 500kbs displays similarities.



(a) Resnet152 **(b)** VGG16

Fig. 4.12: Inference time measured for model Resnet18 and Alexnet

For shallower networks, however, the situation is slightly more different. Fig. 4.13 shows the slowest images identified in the models Resnet18 (Fig. 4.13a) and Alexnet (Fig. 4.13b). Differently from what we concluded before, these models show a much more precise response for images smaller than 500 kbs. We are in fact able to cluster the images by size, with the exception of very few outliers. In addition, we can also point out which number of epoch would yield faster predictions for the slowest images. For Resnet18, we can observe that the model trained with 40 epochs is among the fastest for most sizes, with very few exceptions. For alexnet, on the other hand, the fastest model is the one trained for only 10 epochs. This conclusion, however, does not take into account the accuracy that those models achieved. Using Fig. 4.10a we can see that the same models, i.e. Resnet18 trained with 10 epochs, achieved one of the lowest accuracy rate over all (~90%) and from Fig. 4.10b we can extrapolate a similar conclusion for Alexnet trained with 10 epochs.(~80%). The best trade off between fast inference time and accuracy is achieved when both models have been trained with 50 epochs, however, in case of Resnet18, as we discussed before, this is also the number of epochs when the validation loss is bigger than the training loss, hence we found ourself in a situation of slight over-fitting.



(a) Resnet18 **(b)** Alexnet

Fig. 4.13: Inference time measured for model Resnet18 and Alexnet

Regardless of which number of epoch is better for this specific case, the purpose of our

exploration is to identify correlation between certain characteristics. From these graphs we are able to find a correlation and to reason about it in order to tailor future applications. In a real world implementation, if is is known that images with a certain size are going to be among the slowest is useful because it will influence the decision of which camera to mount on the devices sent to the fields. For example, knowing that images with a size of 700 kbs are going to take 200 to 220 ms to be classified will help model the time behaviour of these devices and give information to verify that they won't miss any deadline. Furthermore, if we are able to reason about the trade-off between inference time and accuracy we are also able to choose a suitable model for different requirements. For example, in case the highest accuracy possible is not a hard requirement, but we have a hard deadline to respect, we are able to identify which model and the amount of training necessary to respect those requirements.

A closer look to the list of slowest images reveals that there are pictures in common amongst all models (Fig. A.1). Using the tool we can investigate if those pictures present similarities that may explain why this is the case. The information collected from the pictures are shown in table 4.4.

Picture n.	Dimensions(x,y)	DPI(x,y)	size(kb)
1	(3888, 2187)	N/A	727
2	(3018, 2585)	(300,300)	2209
3	(3000, 2019)	(300,300)	1341
4	(2003, 2003)	N/A	558
5	(1373, 1012)	(72,72)	1409

Table 4.4: Information collected from the slowest pictures

As already demonstrated by Fig. 4.11, each of the five pictures has a size bigger than 500kb, i.e. they are amongst the heaviest pictures on the dataset. Furthermore, they are also amongst the pictures having the highest dimensions. Even though not available for all the pictures, table 4.4 also shows that the pictures have high DPI, which implies that these pictures have very high quality.

4.5 Second Experiment

In this section, we are going to analyse the performances of the models over the 'plant_ seedlings _v2' data-set. This dataset contains information taken from the agricultural field, we can therefore use the benchmarking tool to measure the metrics in a setting that reflects better a real-world use of the tool.
Similarly to the first experiment, the first metric we are going to analyse is training time. The tool run the test three times for 100, 200 and 50 epochs. We are going to start our analysis by studying the results obtained when trained for 100 epochs, which are shown in Fig. 4.14 and 4.15.
As already encountered in the first experiment, Alexnet is the model that achieved the lowest accuracy overall reaching ~86% after 99 epochs. The highest accuracy has been achieved by Resnet101, peaking at ~97% after 66 epochs. Moreover, VGG16 and VGG19

achieved the same peak accuracy at ~95%, however VGG16 required 13 epochs less (43 compared to the 56 needed for VGG19). Similarly, Resnet18 and Resnet34 achieved comparable top accuracies at 94.39% and 94.48% respectively, however Resnet34 required considerably less time to reach this number peaking after 62 epochs, while Resnet18 achieved this number when the training cycle was almost done, namely after 96 epochs. An overview of the performances of all the models can be seen in table 4.5.

Model	Top Accuracy (%)	Epochs needed	Average Time (s)	Total Time (s)
Resnet18	94.39	96	7	747
Resnet34	94.48	62	9	876
Resnet50	96.65	67	14	1439
Resnet101	97.01	66	21	2101
Resnet152	96.56	98	29	2851
Alexnet	85.88	63	7	705
VGG16	95.20	43	17	1743
VGG19	95.20	56	20	1952

Table 4.5: Performances of the models trained the 'plant_seedlings_v2' dataset

Fig. 4.14 shows the response of all models during the training based on the epoch used for training. Similarly to the first experiment, the response of the model show that within the first ten epochs the accuracy increases quickly to stabilize afterwards. However, contrary to the first experiment, here we can observe a more compact graph, meaning that the all the models except Alexnet achieved accuracies not far off from each other during the training.

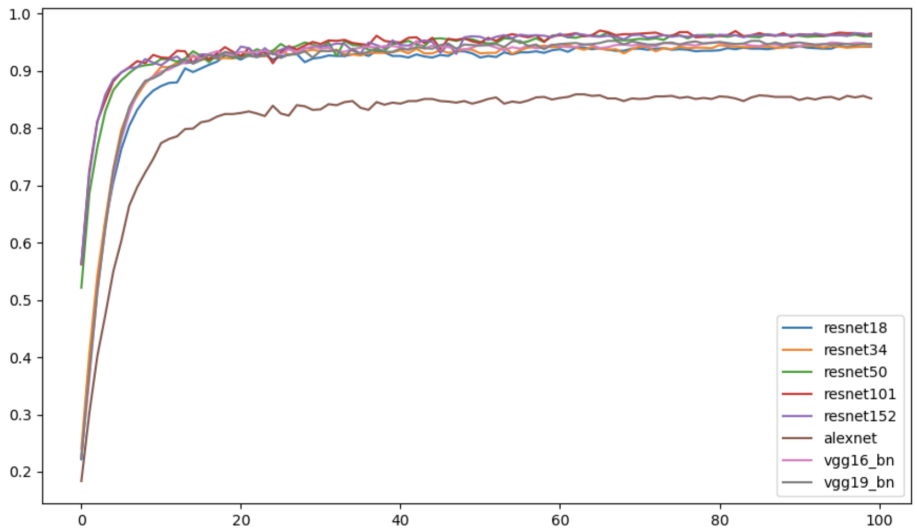


Fig. 4.14: Accuracy achieved when training for 100 epochs. The x axis is the number of epochs, while the y axis is the accuracy achieved

In Fig. 4.15, on the other hand, depicts the accuracy graphed over training time. Alexnet

took the least amount of time to finish the training cycle (747 seconds), however Resnet took only ~40 seconds more and reached a far better accuracy. Resnet152 took 47 Minutes and 30 Seconds (2851 seconds) to complete the test and is the model that required the most time.

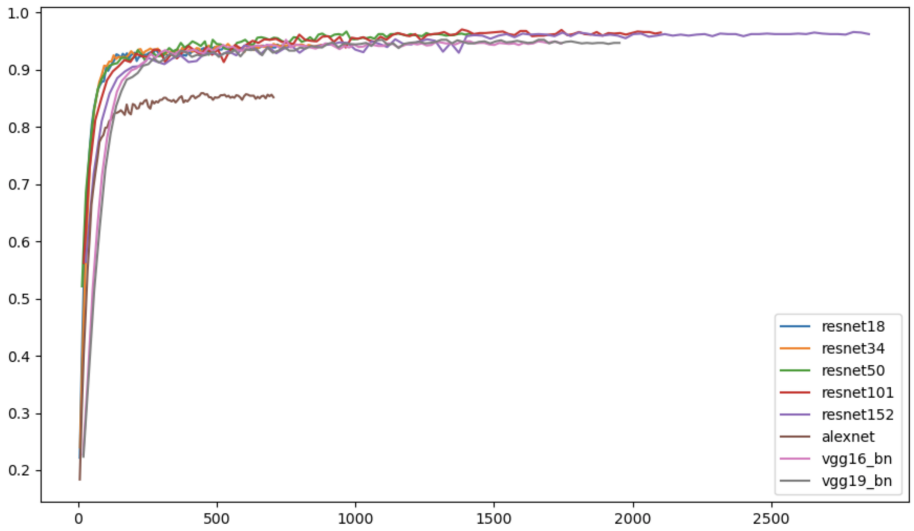


Fig. 4.15: Accuracy achieved when training for 100 epochs in relation with training time. The x axis is the training time in seconds, while the y axis is the accuracy achieved

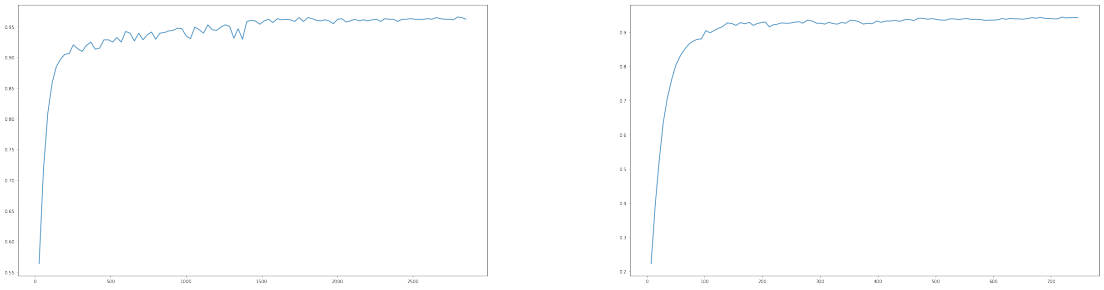


Fig. 4.16: Breaking point of Resnet101 and VGG19

4.6 Conclusion

5 Conclusion

5.1 Future work

For the purpose of this paper, only techniques related to measuring execution time have been described. However, benchmarks are also used to measure other metrics, such us energy consumption or memory use. It is left for future work to find better techniques to precisely measure and monitor them, as they are also of the highest importance especially for smaller devices.
In addition, this paper focused mostly on UNIX operating systems, therefore purposely neglecting other operating systems and micro-controllers. The study of measurements techniques is also left for future work.

- Talk about benchmark
- Talk about optimize application
- Finish conclusion

References

[10.16] *RecSys '16: Proceedings of the 10th ACM Conference on Recommender Systems*, New York, NY, USA, 2016. Association for Computing Machinery.

[AAB+15a] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[AAB+15b] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin, 2015.

[AAUS+19] Muhammad Ayaz, Mohammad Ammad-Uddin, Zubair Sharif, Ali Mansour, and El-Hadi M. Aggoune. Internet-of-Things (IoT)-Based Smart Agriculture: Toward Making the Fields Talk. *IEEE Access*, 7:129551–129583, 2019.

[AMK16] Bilge Acun, Phil Miller, and Laxmikant Kalé. Variation among processors under turbo boost in hpc systems. pages 1–12, 06 2016.

[AR20] Saahil Afaq and Smitha Rao. Significance of epochs on training a neural network. *International Journal of Scientific & Technology Research*, 9:485–488, 2020.

[BC18] Martin Becker and Samarjit Chakraborty. Measuring software performance on linux. *CoRR*, abs/1811.01412, 2018.

[BCCN18] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018.

[BHC18] M Dian Bah, Adel Hafiane, and Raphael Canals. Deep learning with unsupervised data labeling for weed detection in line crops in uav images. *Remote Sensing*, 10(11), 2018.

[BLW17] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21:1–29, 2017.

[BP20] Tamalika Bhadra and Swapan Paul. Weed management in sugar beet: A review. *Fundamental and Applied Agriculture*, 5(0):1, 2020.

[BTD+16] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars, 2016.

[CBB17] Qingqing Cao, Niranjan Balasubramanian, and Aruna Balasubramanian. Mobirnn: Efficient recurrent neural network execution on mobile gpu. In *Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications*, EMDL '17, page 1–6, New York, NY, USA, 2017. Association for Computing Machinery.

[CCG17] Emine Cengil, Ahmet Cinar, and Zafer Gueler. A gpu-based convolutional neural network approach for image classification. In *2017 International Artificial Intelligence and Data Processing Symposium (IDAP)*, pages 1–6, 2017.

[CM10] Franco Cioni and Gianfranco Maines. Weed Control in Sugarbeet. *Sugar Tech*, 12(3-4):243–255, December 2010.

[CPC16] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *CoRR*, abs/1605.07678, 2016.

[Der16] Leon Derczynski. Complementarity, F-score, and NLP evaluation. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 261–266, Portorož, Slovenia, May 2016. European Language Resources Association (ELRA).

[Die95] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM computing surveys (CSUR)*, 27(3):326–327, 1995.

[DSSM09] Eulanda M. Dos Santos, Robert Sabourin, and Patrick Maupin. Overfitting cautious selection of classifier ensembles with genetic algorithms. *Inf. Fusion*, 10(2):150–162, apr 2009.

[fas21] fast.ai. Fastai documentation. `https://docs.fast.ai/`, Nov 29, 2021. [Online; accessed 27-12-2021].

[FHZ93] William Finnoff, Ferdinand Hergert, and Hans Georg Zimmermann. Improving model selection by nonconvergent methods. *Neural Networks*, 6(6):771–783, 1993.

[FMF+14] C. Frasconi, L. Martelloni, M. Fontanelli, M. Raffaelli, L. Emmi, Michel Pirchio, and A. Peruzzi. Design and full realization of physical weed control (PWC) automated machine within the RHEA project. 2014.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[GDDM14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. 2014.

[GDP09] Alexander Guzhva, Sergey Dolenko, and Igor Persiantsev. Multifold acceleration of neural network computations using gpu. In Cesare Alippi, Marios Polycarpou, Christos Panayiotou, and Georgios Ellinas, editors, *Artificial Neural Networks – ICANN 2009*, pages 373–380, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Gei21] Amnon Geifman. The correct way to measure inference time of deep neural networks, 2021.

[GFP+20] Junfeng Gao, Andrew P. French, Michael P. Pound, Yong He, Tony P. Pridmore, and Jan G. Pieters. Deep convolutional neural networks for image-based Convolvulus sepium detection in sugar beet fields. *Plant Methods*, 16(1):29, December 2020.

[GIC20] Dimitrios Glaroudis, Athanasios Iossifides, and Periklis Chatzimisios. Survey, comparison and research challenges of IoT application protocols for smart farming. *Computer Networks*, 168:107037, February 2020.

[GJJ+17] Thomas Mosgaard Giselsson, Rasmus Nyholm Jørgensen, Peter Kryger Jensen, Mads Dyrmann, and Henrik Skov Midtiby. A public image database for benchmark of plant seedling classification algorithms, 2017.

[Goo10] Google. Machine learning crash course. `https://developers.google.com/machine-learning/crash-course/classification/accuracy`, 2020-02-10. [Online; accessed 27-12-2021].

[GTA+21] Jakob Gawlikowski, Cedrique Rovile Njieutcheu Tassi, Mohsin Ali, Jongseok Lee, Matthias Humt, Jianxiang Feng, Anna Kruspe, Rudolph Triebel, Peter Jung, Ribana Roscher, Muhammad Shahzad, Wen Yang, Richard Bamler, and Xiao Xiang Zhu. A survey of uncertainty in deep neural networks, 2021.

[HD19] Dan Hendrycks and Thomas Dietterich. Benchmarking neural network robustness to common corruptions and perturbations, 2019.

[HEKK19] Sara Hooker, Dumitru Erhan, Pieter-Jan Kindermans, and Been Kim. A benchmark for interpretability methods in deep neural networks, 2019.

[HG20] Jeremy Howard and Sylvain Gugger. Fastai: A layered api for deep learning. *Information*, 11(2), 2020.

[HLM+16] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network, 2016.

[HM13] Haibo He and Yunqian Ma. *Imbalanced Learning: Foundations, Algorithms, and Applications*. Wiley-IEEE Press, 1st edition, 2013.

[HSD+21] A. S. M. Mahmudul Hasan, Ferdous Sohel, Dean Diepeveen, Hamid Laga, and Michael G. K. Jones. A Survey of Deep Learning Techniques for Weed Detection from Images. *arXiv:2103.01415 [cs]*, March 2021. arXiv: 2103.01415.

[HSHK21] Denis Huseljic, Bernhard Sick, Marek Herde, and Daniel Kottke. Separation of aleatoric and epistemic uncertainty in deterministic deep neural networks. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 9172–9179, 2021.

[HW20] Ramy Hussein and Rabab Ward. Chapter 4 - energy-efficient eeg monitoring systems for wireless epileptic seizure detection. pages 69–85, 2020.

[HW21] Eyke Hüllermeier and Willem Waegeman. Aleatoric and epistemic uncertainty in machine learning: an introduction to concepts and methods. *Machine Learning*, 110, 03 2021.

[HWT+15] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, Fernando Mujica, Adam Coates, and Andrew Y. Ng. An empirical evaluation of deep learning on highway driving, 2015.

[HZRS14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *Lecture Notes in Computer Science*, page 346–361, 2014.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[IRP+21] Nahina Islam, Md Mamunur Rashid, Faezeh Pasandideh, Biplob Ray, Steven Moore, and Rajan Kadel. A Review of Applications and Communication Technologies for Internet of Things (IoT) and Unmanned Aerial Vehicle (UAV) Based Sustainable Smart Farming. *Sustainability*, 13(4):1821, February 2021.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 2015.

[ITK+19] Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu, and Luc Van Gool. Ai benchmark: All about deep learning on smartphones in 2019, 2019.

[JK15] H Jabbar and Rafiqul Zaman Khan. Methods to avoid over-fitting and under-fitting in supervised machine learning (comparative study). *Computer Science, Communication and Instrumentation Devices*, pages 163–172, 2015.

[KD09] Armen Der Kiureghian and Ove Ditlevsen. Aleatory or epistemic? does it matter? *Structural Safety*, 31(2):105–112, 2009. Risk Acceptance and Risk Communication.

[Ker10] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, USA, 1st edition, 2010.

[Ker21] Michael Kerrisk. Linux manual page, 2021.

[KG17] Alex Kendall and Yarin Gal. What uncertainties do we need in bayesian deep learning for computer vision? *CoRR*, abs/1703.04977, 2017.

[KRKP+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[LHS+16a] P. Lottes, M. Hoeferlin, S. Sander, M. Muter, P. Schulze, and Lammers C. Stachniss. An effective classification system for separating sugar beets and weeds for precision farming applications. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5157–5163, Stockholm, Sweden, May 2016. IEEE.

[LHS+16b] P. Lottes, M. Hoeferlin, S. Sander, M. Müter, P. Schulze, and Lammers C. Stachniss. An effective classification system for separating sugar beets and weeds for precision farming applications. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5157–5163, 2016.

[LHZ+20] Chunjie Luo, Xiwen He, Jianfeng Zhan, Lei Wang, Wanling Gao, and Jiahui Dai. Comparison and benchmarking of ai models and frameworks on mobile devices, 2020.

[LY20] Yuzhen Lu and Sierra Young. A survey of public datasets for computer vision tasks in precision agriculture. *Computers and Electronics in Agriculture*, 178:105760, November 2020.

[Mas91] Amédée Masclef. Sugar Beet, 1891.

[May03] M J May. Economic consequences for UK farmers of growing GM herbicide tolerant sugar beet. *Annals of Applied Biology*, 142(1):41–48, February 2003.

[Mit97] Tom M Mitchell. *Machine Learning*. New York McGraw-Hill, 1997.

[MPSG21] Jannis Machleb, Gerassimos G. Peteinatos, Markus Sökefeld, and Roland Gerhards. Sensor-Based Intrarow Mechanical Weed Control in Sugar Beets with Motorized Finger Weeders. *Agronomy*, 11(8):1517, July 2021.

[Mur16] John Murphy. An overview of convolutional neural network architectures for deep learning. *Microway Inc*, 2016.

[NYC15] Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images, 2015.

[OFR+19] Yaniv Ovadia, Emily Fertig, Jie Ren, Zachary Nado, D Sculley, Sebastian Nowozin, Joshua V. Dillon, Balaji Lakshminarayanan, and Jasper Snoek. Can you trust your model's uncertainty? evaluating predictive uncertainty under dataset shift, 2019.

[OJ04] Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.

[PE89] Perugini and Engeler. Neural network learning time: effects of network and training set size. In *International 1989 Joint Conference on Neural Networks*, pages 395–401 vol.2, 1989.

[Pet04] J. Petersen. A Review on Weed Control in Sug- arbeet. *Inderjit (Ed), Weed Biology and Man- agement.*, 2004.

[PJY+13] Thomas Paine, Hailin Jin, Jianchao Yang, Zhe Lin, and Thomas Huang. Gpu asynchronous stochastic gradient descent to speed up neural network training, 2013.

[Pre00] Lutz Prechelt. Early stopping - but when? 03 2000.

[PVZJ12] Omkar M. Parkhi, Andrea Vedaldi, Andrew Zisserman, and C. V. Jawahar. Cats and dogs. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

[RAMP20] W. Ramirez, P. Achanccaray, L. F. Mendoza, and M. A. C. Pacheco. DEEP CONVOLUTIONAL NEURAL NETWORKS FOR WEED DETECTION IN AGRICULTURAL CROPS USING OPTICAL AERIAL IMAGES. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLII-3/W12-2020:551–555, November 2020.

[RCK+20] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark, 2020.

[RDS+14] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[RGC+16] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design, 2016.

[RM99] Russell D Reed and Robert J Marks. *Neural smithing supervised learning in feedforward artificial neural networks*. 1999. OCLC: 1227498094.

[RNSF20] Rekha Raja, Thuy T. Nguyen, David C. Slaughter, and Steven A. Fennimore. Real-time robotic weed knife control system for tomato and lettuce based on geometric appearance of plant labels. *Biosystems Engineering*, 194:152–164, June 2020.

[Rud17] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[SD89] EE Schweizer and A.G Dexter. "Weed control in sugarbeets (Beta vulgaris) in North America". 1989.

[SGSS16] Arti Singh, Baskar Ganapathysubramanian, Asheesh Kumar Singh, and Soumik Sarkar. Machine Learning for High-Throughput Stress Phenotyping in Plants. *Trends in Plant Science*, 21(2):110–124, 2016.

[SIHvH18] Hyun K. Suh, Joris IJsselmuiden, Jan Willem Hofstee, and Eldert J. van Henten. Transfer learning for the classification of sugar beet and volunteer potato under field conditions. *Biosystems Engineering*, 174:50–65, October 2018.

[SJS06] Marina Sokolova, Nathalie Japkowicz, and Stan Szpakowicz. Beyond accuracy, f-score and roc: A family of discriminant measures for performance evaluation. volume Vol. 4304, pages 1015–1021, 01 2006.

[SKD19] Kyoung Song, Myeongchan Kim, and Synho Do. The latest trends in the use of deep learning in radiology illustrated through the stages of deep learning algorithm development. *Journal of the Korean Society of Radiology*, 80:202, 03 2019.

[SLJ+14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

[Smi18] Leslie N. Smith. A disciplined approach to neural network hyper-parameters: Part 1 - learning rate, batch size, momentum, and weight decay. *CoRR*, abs/1803.09820, 2018.

[ST17] Leslie N. Smith and Nicholay Topin. Super-convergence: Very fast training of residual networks using large learning rates. *CoRR*, abs/1708.07120, 2017.

[Ste01] David B. Stewart. Measuring execution time and real-time performance. 2001.

[Suh18] Hyun Suh. *Advanced classification of volunteer potato in a sugar beet field*. PhD thesis, 01 2018.

[SZ15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[TLZ+19] Nesime Tatbul, Tae Jun Lee, Stan Zdonik, Mejbah Alam, and Justin Gottschlich. Precision and recall for time series, 2019.

[UKG+20] Thomas Unterthiner, Daniel Keysers, Sylvain Gelly, Olivier Bousquet, and Ilya O. Tolstikhin. Predicting neural network accuracy from weights. *ArXiv*, abs/2002.11448, 2020.

[UKG+21] Thomas Unterthiner, Daniel Keysers, Sylvain Gelly, Olivier Bousquet, and Ilya Tolstikhin. Predicting neural network accuracy from weights, 2021.

[vKAH+15] Jóakim von Kistowski, Jeremy Arnold, Karl Huppler, Klaus-Dieter Lange, John Henning, and Paul Cao. How to build a benchmark. 02 2015.

[vR74] C. J. van Rijsbergen. Foundation of evaluation. *Journal of Documentation*, 30:365–373, 1974.

[WDESP17] Peter Wägemann, Tobias Distler, Christian Eichler, and Wolfgang Schröder-Preikschat. Benchmark generation for timing analysis. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 319–330, 2017.

[YKU+20] Jinhui Yi, Lukas Krusenbaum, Paula Unger, Hubert Hüging, Sabine J. Seidel, Gabriel Schaaf, and Juergen Gall. Deep learning for non-invasive diagnosis of nutrient deficiencies in sugar beet using rgb images. *Sensors*, 20(20), 2020.

[ZAZ+18] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. Benchmarking and analyzing deep neural network training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 88–100, 2018.

[ZS21] Gennady Fedorov Shaojuan Zhu and Abhinav Singh. Intel® oneapi math kernel library (onemkl) benchmarks suite, 2021.

A Appendix

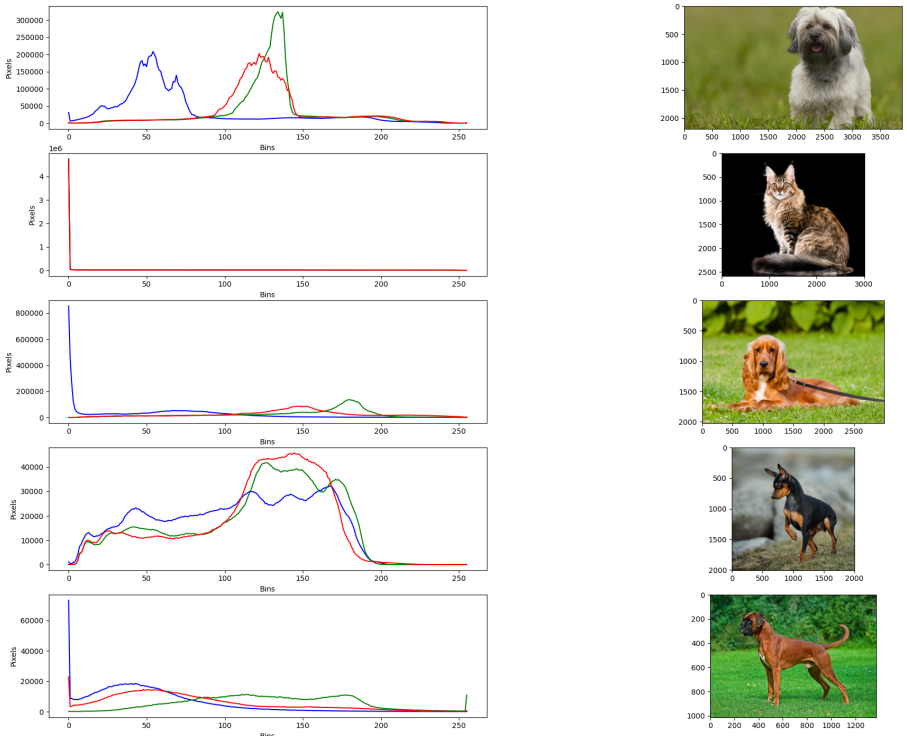


Fig. A.1: Histogram of the slowest files in common amongst all models

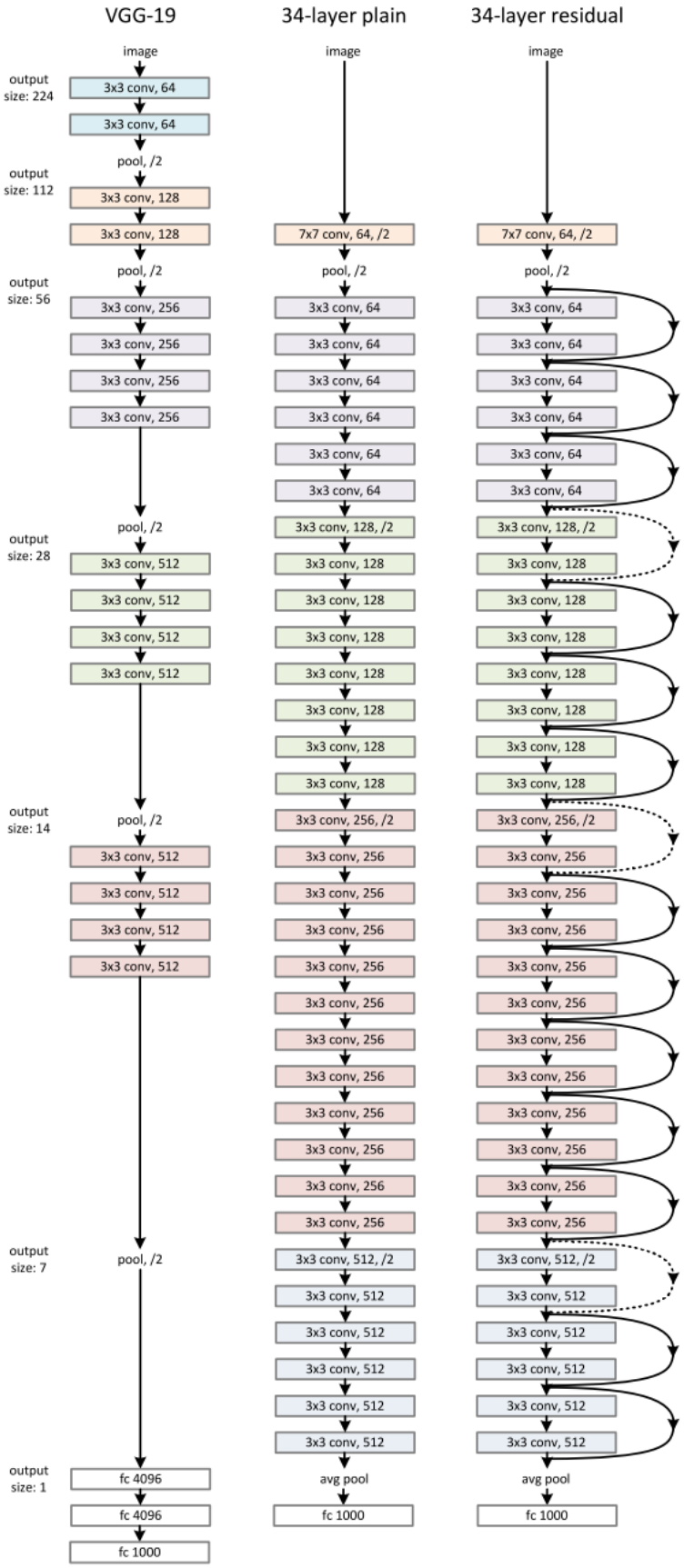


Fig. A.2: Left: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. Middle: a plain network with 34 parameter layers (3.6 billion FLOPs). Right: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions.[HZRS15]

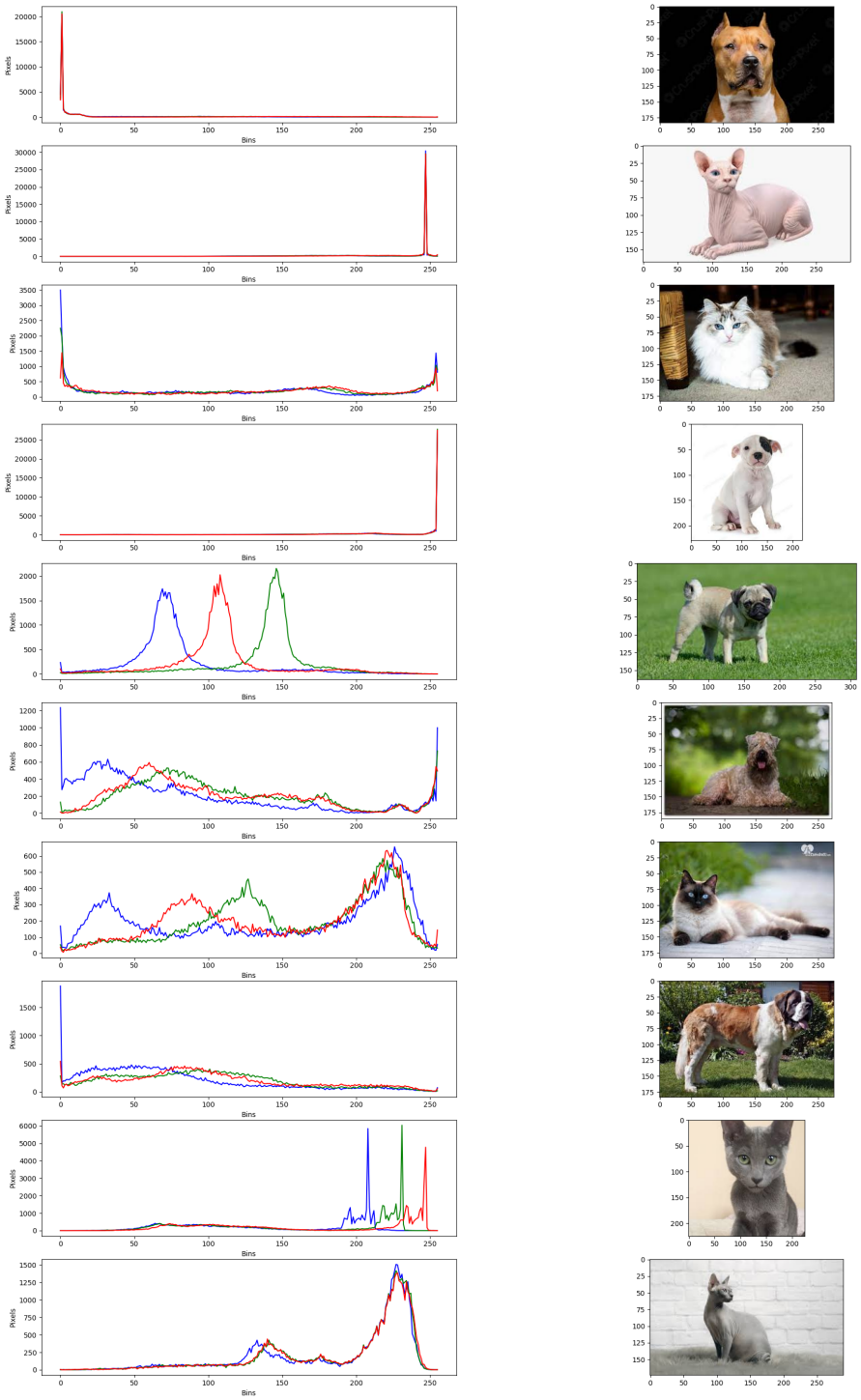


Fig. A.3: Histogram of the fastest files of Resnet 152

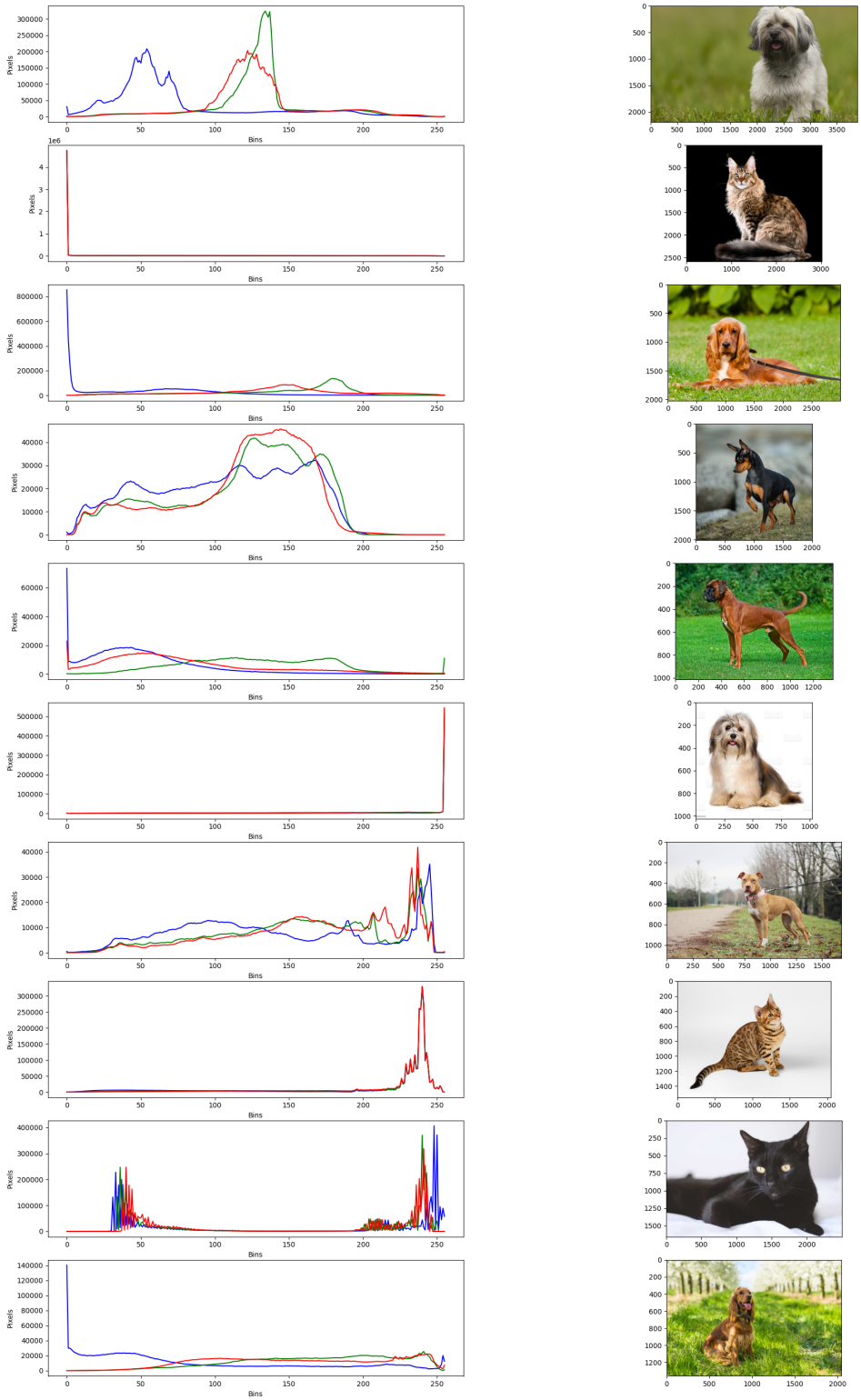


Fig. A.4: Histogram of the slowest files of Resnet152