UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

eXascale Infolab

UNIVERSITY OF FRIBOURG

MASTER THESIS

Single-Dataset Applications of Typhon and Parallel Transfer to Mitigate Overfitting and Improve Sample Efficiency Across Deep Learning

Author:
Christophe Broillet

Supervisors:
Dr. Giuseppe Cuccu
Prof. Dr. Philippe Cudré-Mauroux

August 13th, 2024

eXascale Infolab
Department of Informatics

Boulevard de Pérolles 90 • 1700 Fribourg • Switzerland
phone +41 (26) 300 84 65 • diuf-secr@unifr.ch • www3.unifr.ch/inf

Abstract

Christophe Broillet

Single-Dataset Applications of Typhon and Parallel Transfer to Mitigate Overfitting and Improve Sample Efficiency Across Deep Learning

Deep learning neural networks are powerful and efficient in various supervised learning tasks and applications, since they are able to approximate any complex functions to arbitrary precision. However, large networks require a considerable amount of data to be trained, due to the vanishing gradients problem. Indeed, the layers close to the input receive little, imprecise error feedback during backpropagation in such networks. The Typhon meta learning algorithm can mitigate this problem by training a model on multiple heterogeneous datasets, bridging transfer learning and multitask learning. Over those methods, Typhon has the advantage to learn (i) each task as a separate, independent target, and (ii) in a parallel manner. However, Typhon cannot be applied to a single dataset. This works proposes to extend the applicability of the Typhon algorithm to any single dataset, by introducing two new algorithms named Two Levels Typhon and Ultra Typhon, which treat classes or group of classes as independent tasks to solve. These two algorithms improve the sample efficiency over classical learning algorithms, resulting in higher performances and a better generalization. Moreover, they are resilient against overfitting. Overall, the Typhon family algorithms converge faster on deep learning training and can be used as a new alternative to mitigate overfitting.

Keywords: Machine Learning, Deep Learning, Overfitting, Moving Target, Transfer Learning, Multi-Task Learning, Sample Efficiency, Parallel Transfer, Typhon

Contents

Chapter 1

Introduction

This Chapter presents firstly a few foundations in various topics, including some mathematical background as well as machine learning concepts. Those topics are shown and explained in order to have the essential background knowledge to read the content of this work. At the end of this Chapter, the motivation beyond this work, and its main contributions are presented.

1.1 Functions and function approximation

A **function** is a mathematical object that describes a relationship between two sets. Let A and B be two sets, then a function f is a relationship that assigns one element $x \in A$ to exactly one element $y \in B$. This is commonly written as $f(x) = y$. To be fully defined, a function needs to assign all elements from the set A to an element to the set B, otherwise it is not a function. The set A is called the domain of the function while the set B is called the range of the function. Depending on how the elements of B are assigned, there are different types of function:

1. **Injective**, where all elements in B are being assigned at *most* once.
2. **Surjective**, where all elements in B are being assigned at *least* once.
3. **Bijective**, where all elements in B are being assigned *exactly* once (i.e. the function is injective and surjective).

Some elementary functions are for example $f(x) = x^2$, where $A = \mathbb{R}$ and $B = \mathbb{R}_+$, or $f(x) = \sin(x)$, where $A = \mathbb{R}$ and $B = [-1, 1]$. A function can be mathematically defined, i.e. it is possible to write a mathematical formulation of the function, as in the two examples above, but could also be so complex that it is nearly impossible to understand how it is constructed, nor writing it in a mathematical form. For example, let the set A be the set of all magnetic resonance images (MRIs) coming from scanners, and $B = \{\text{yes}, \text{no}\}$. Then a possible relationship, function between those two sets can be to assign `yes` or `no` to any MRI, given if it contains a cancerous tumor or not. This process or function is hard to compute, but it must exist, otherwise no radiologists, nor the human brain, would be able to make diagnoses about a cancer for the patients.

Some functions have defined parameters, such as the coefficients a, b, c in an example polynomial $f(x) = ax^2 + bx + c$, or the amplitude a and the frequency ω in a function describing a physical wave $f(x) = a \cdot \sin(\omega)$. Such functions are called **parametric functions**. The set of parameters is called the **parametrization** of the function. Parametric functions can be used to approximate other functions, by modifying their parametrization, for example as done in Taylor series [1] or Fourier series [2]. Finding the right parametrization of a (parametric) function that approximates a target function is the foundation of machine learning.

1.2 Supervised learning

The general goal of machine learning is to find and derive patterns as mathematical functions, or relationships between two sets. That is, to find the rules that govern the process of generating the data. Taking the previous example, this corresponds to how a radiologist is able to diagnose a `yes` or `no` by looking at the image. There are different ways of learning those relationships, called **learning paradigms**. The three main paradigms in machine learning are supervised learning, unsupervised learning and reinforcement learning. The latter is used in applications that lack access to labeled data, as often common in dynamic tasks such as continuous control. Unsupervised learning is used to find relationships or patterns *within* the data, for example if it can be separated into similar groups, or in order to compress it without losing information. Finally, supervised learning makes use of already labeled data in order to recognize patterns, such as in classification tasks. In this thesis, unsupervised learning and reinforcement learning are simply mentioned and not used. From now on, all learning techniques and concepts are related to supervised learning.

The set A contains all the **inputs**, **observations** or **samples**, denoted by x, and the set B consists of all the **labels**, denoted by y which are corresponding expected outputs. The goal of supervised learning is to derive, find, or approximate the **real underlying function** $f : A \to B, x \mapsto f(x)$, also denoted by $f(x) = y$ that generated the data. To accomplish this, the function f needs to be modeled or approximated by a **model** or **approximator** $\widehat{f}_\theta : \tilde{A} \to B$ with $\widehat{f}_\theta(x) = \hat{y} \approx y$, where $\tilde{A} \subset A$ is the **available data**. The output of the model $\hat{y}$ is called a **prediction**, and θ represents the **parametrization** of the model, e.g. some numerical coefficients, or architectures. All the pairs $\mathcal{D} = \{(x, y) : x \in \tilde{A}, b \in B\}$ are put together and form the **dataset**. There are different family of models, such as the linear models, the polynomial models, the neural networks, which are basically compositions of functions, (see Section 1.5), or other methods. The (supervised) learning algorithms see the examples (x, y) from the dataset and then update the parametrization θ of the model $\widehat{f}_\theta$ so that it better fits the examples, meaning the predictions $\hat{y}$ are closer to the real labels y.

In the real world, for example in medical applications, it would be helpful to have access to a complex function f that outputs `yes` or `no` given a MRI, if there is a cancerous tumor or not. Indeed, a lot of people could be saved from cancer if such function was directly linked to a scanner for example, thanks to the early diagnosis. To find such a function, supervised learning can be used as learning paradigm. To model a complex function f using supervised learning, the sets $\tilde{A}$ and B have to be known. More precisely, supervised learning needs pairs (x, y) with $x \in \tilde{A}$ and $y \in B$ so that it can approximate the real underlying function.

In supervised learning, the two most common categories of problems are the **classification** and the **regression** problems. Classification problems consist of classifying an input into a pre-defined set of classes, for example classifying images between `cat`, `dog` and `bird`. In this example, the inputs are the images and the classes or labels are `cat`, `dog` and `bird`. The classification algorithms output *discrete* predictions, that is to which class the input belongs to. The second category, the regression problems, are used when the output of the algorithm needs to be *continuous*, for example in weather prediction or mortgage rate prediction.

1.3 Mathematical optimization

The similarity and/or difference of the prediction $\widehat{f}_\theta(x) = \hat{y}$ with the real value, i.e. the label y, needs to be measured. That is, an objective is required in order to define what is a good prediction or a bad one. The objective is modeled by an objective function, called the **loss function**. Mathematically, a loss function is denoted by $\mathcal{L}(\widehat{f}_\theta(x), y)$. Examples of loss functions include cross entropy loss [3], mean absolute error or mean squared error [4], and hinge loss [5]. Note that usually the total loss function is the sum or the average of the loss function value for each example in the dataset. The loss used in this work (see Chapter 3) is the binary cross entropy, defined as $\mathcal{L}(\widehat{f}_\theta(x), y) = -y \cdot \log(\widehat{f}_\theta(x)) + (1-y) \cdot \log(1-\widehat{f}_\theta(x))$. The end goal of supervised learning algorithms turns out to be a mathematical optimization problem: minimize the loss function $\mathcal{L}$ w.r.t. the model's parametrization θ, on a given dataset $\mathcal{D}$. This is written mathematically in Equation 1.1.

$$\underset{\theta}{argmin}\, \mathcal{L}(\widehat{f}_\theta(x), y) \tag{1.1}$$

In Equation 1.1, the inputs x and labels y, as well as the loss function $\mathcal{L}$ are defined beforehand. Thus, the single object that can be modified in order to solve the optimization problem is to alter the predictions $\hat{y}$ such that they are as close as possible to the real values y. This is done by updating the parametrization of the model $\widehat{f}_\theta$, until desired precision. Updating the parametrization of the model is often referred as **training** the model.

1.4 Evaluation

As mentioned in Section 1.2 and Section 1.3, supervised learning algorithms solve an optimization problem minimizing an objective function, the loss function, given a dataset with observations and labels w.r.t. a certain model $\widehat{f}_\theta$. In general, finding a good model $\widehat{f}_\theta$ is hard, since in principle the observations in the available data $\tilde{A}$ do not contain all possible elements from the real A set. Thus the model needs to **generalize** as much as possible, to perform well on unseen data, i.e. on examples of the $A \setminus \tilde{A}$ set that the model has not be trained on. Trying to understand the type and how the data was generated helps to understand the real underlying process, thus helping to achieve better generalization.

To simulate this, **data splitting** is usually performed. The full dataset $\mathcal{D}$ is separated in two parts: one **training set**, and one **test set**, usually in a 80% - 20% manner. The training set is used to train the model, i.e. to optimize its parametrization in order to minimize the loss function, while the test set is used to evaluate the performance of the trained model $\widehat{f}_\theta$, i.e. how it is performing on unseen data, on which it has not been trained on. This informs about which trained model is more desirable for a specific application, and which model has achieved the best generalization.

Most algorithms include **hyperparameters** that modify their behavior by for example set a trade-off between sub-optimization problems the algorithms are solving. These hyperparameters need to be optimized as well, in order to get the best performance out of the model. This step is crucial when solving real-world problems. As the test set should not be seen during the optimization nor the training procedure, the training set is usually split further into two sets: (i) the proper training set and

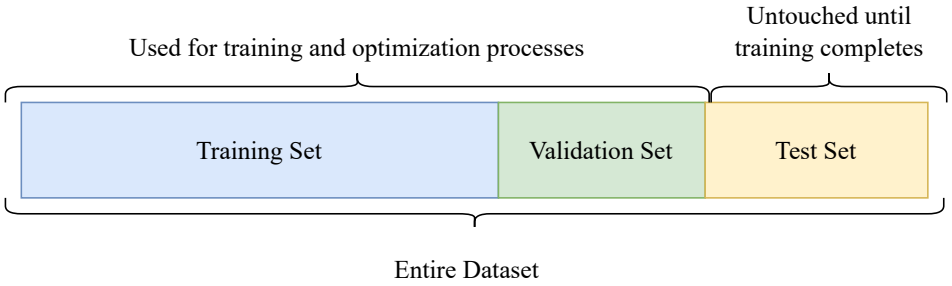


FIGURE 1.1: **Example of data splitting.** The entire dataset can be split into three parts: (i) a training set that is used in the training phase, i.e. to optimize the loss function, (ii) a validation set used to fine-tune the hyperparameters if any, and (iii) a test set to evaluate the final trained model after the optimization of the hyperparameters. Typical split size are 70% for the training set, 10% for the validation set and 20% for the test set. *k*-fold cross validation [6, 7] can also be used to make the results statistically significant.

(ii) the **validation set** which can be used to test different hyperparameters configurations without showing the real test set data. Figure 1.1 shows how a dataset is split into these sub-sets, each having a different role.

To complete the evaluation of the model, different measures called **metrics** can be computed to give different information on how the model is performing. Such metrics include for example accuracy, precision, recall, F1-Score [8], area under the ROC-curve [9], Dice-Sørensen coefficient [10, 11], Jaccard index [12, 13], or coefficient of determination [14, 15]. Depending on the application being solved, the choice of the metrics differs. For example, accuracy, precision and recall are often used in classification problems while the coefficient of determination is used to evaluate regression tasks. Metrics are also used in official publications, where authors can compare their results of their own proposed methods on the same metrics, thus having a fair comparison.

1.5 Neural networks

A modern and popular generic function approximator, ubiquitously used for deep learning are the **(artificial) neural networks**. Neural networks are universal function approximators, meaning that they are in principle able to approximate any function to arbitrary precision, depending on their architecture. The more complex is the neural network, the more complex the function it approximates can be. They are called neural networks because they are inspired by the biological behavior of the human brain: neurons are connected between each other by synaptic weights, and if they have reached a certain electric potential they get activated, propagating the signal to the next neurons. This way, different areas of the brain are activated depending on the task the human is doing. Artificial neural networks are *only* inspired by the human brain in the sense that they are a mathematical abstraction and simplification of the real human brain [16].

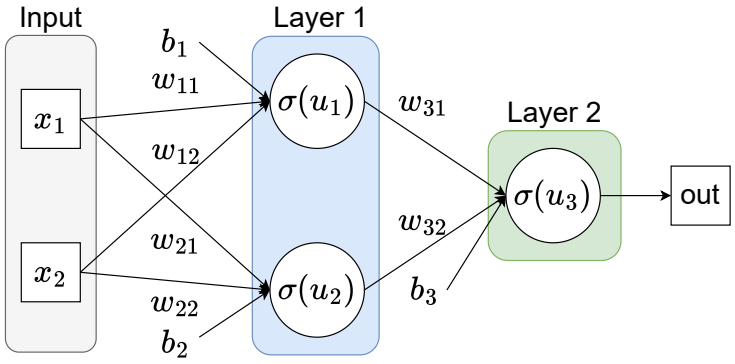


FIGURE 1.2: **Example of a (fully-connected) neural network with two layers.** The input is $x = (x_1, x_2)$, the weights w_{ij} and the biases b_i form the parametrization of the network. The intermediate outputs of a neuron, denoted by u_i, are computed by performing a linear combination of the weights w_{ij} and the outputs of the previous layer x_i, and add the bias b_i, that is $u_i = \sum_j x_j * w_{ij} + b_i$. An activation function σ is then applied to u_i to form the real output of the neuron $\sigma(u_i)$. The shape of the output of the last layer depends on the underlying function the network is approximating.

1.5.1 Architecture

The architecture of a neural network consists of different **layers**. Each of these layers applies a transformation to their input, which can be a matrix multiplication, or a convolution for example. All coefficients of those transformation, such as the coefficients in the matrix multiplications or the coefficients of the filters used in the convolutions, are called the **weights** or **parameters** of the network. They form the parametrization of the neural network, that needs to be optimized to minimize the loss function. At the end of each layer there can be also a non-linear activation function that transforms further the output of the layer, which becomes the true output of the layer. This output is then passed as input to the next layer. Thanks to the non-linear activation functions at the end of the layers, neural networks are able to approximate *any* function. In a mathematical point of view, neural networks can be seen as a sequence of complex (non-)linear function compositions. Figure 1.2 shows an example of a (fully-connected) neural network with two layers.

1.5.2 Classification

For a classification task, the number of neurons in the last layer usually corresponds to the number of classes to classify. Indeed, each output neuron can inform on the *classification score* for one specific class. The output of those neurons can be interpreted as a probability after applying the standard logistic or the softmax [17] activation function on the last layer of the neural network. In that way, for every input going through the neural network, probabilities to belong to each class will be assigned. The class corresponding to the highest probability is thus picked as the predicted class of the given input.

In order to use the labels y with neural networks, they have to be encoded. This is usually done either by **categorical encoding**, where each class is assigned to a single number, or by **one-hot encoding**, where each class is assigned to a `zero` n-dimensional array except a `one` at the position of the class. For example, suppose

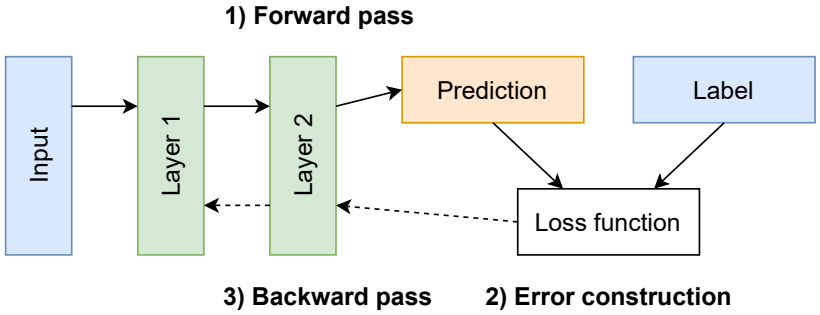


FIGURE 1.3: **Neural networks training.** Step 1: the input goes through the network, which computes the prediction. Step 2: the error, i.e. the value of the loss function, is computed with the prediction and the label. Step 3: the error is backpropagated in reverse order within the network, with the goal to compute all error gradients with respect to each single weight. With those gradients, an optimization algorithm is applied to update the weights of the network.

there are some (ordered) classes [`cat`, `dog`, `shark`, `crow`]. By using categorical encoding, the label for `dog` is 2 and the label for `crow` is 4, while by using one-hot encoding the label for `dog` is $[0, 1, 0, 0]$ and the label for `crow` is $[0, 0, 0, 1]$.

1.5.3 Training

The training of a neural network, i.e. the optimization of the loss function w.r.t. the parametrization of the network, is performed in two distinct steps, which are repeated until convergence, or by achieving desired precision. These steps are:

1. The gradients of the error w.r.t. each weight of the network are computed using the so-called **backpropagation algorithm** [18, 19, 20]. This is achieved in three sub-steps:

   1) The forward pass, where the input goes through the entire network and a prediction gets out as output.
   2) The loss function is applied to the prediction and the true label to compute the error.
   3) The backward pass, where the evaluated error goes back into the network to compute an error gradient for each single weight.

   This is summarized in Figure 1.3.

2. Once all gradients are available thanks to the backpropagation, any (first-order) **optimization algorithm** can be applied. This can be for example the stochastic gradient descent (SGD) [21, 22], the adaptive moment estimation (ADAM) [23] or the Broyden Fletcher Goldfarb Shanno (BFGS) algorithm [24]. Second-order methods such as Newton's method [25] could in theory also be applied, but the second-order derivatives can be very hard depending on the activation function and do need in general too much resources to be computed for neural networks.

1.6 Deep learning

A sub-set of machine learning that is used into many nowadays domains, such as computer vision or language modeling, is **deep learning** [17]. This is using deep neural networks, networks with many layers, to be able to approximate a very complex function. Because of their size, deep neural networks are heavily affected by error propagation. If the network makes a large mistake in the first layers, the error is then propagated through the network and becomes bad at the end. Also, the very first layers of the networks, which are the most important ones as they act as feature extraction since they are the closest to the input, will receive very little error feedback, i.e. error gradient, that was computed with the backpropagation algorithm. This makes deep neural network *data-hungry*, that a large amount of data is required to train them as the gradients received in the first layers are imprecise. This problem is known as the **vanishing gradients** problem.

Deep learning emerged in the early 2010's, due to a convergence of several factors:

1. The availability of larger dataset and the era of the *big data*. This is due to the progress in data storage and with the emergence of the open-source policy, where the access to data is made public. Large amount of data is often collected, stored and shared more easily.

2. The improvement on the hardware side for the computations. Since the training of neural networks is mainly composed of matrix multiplication operations, the latter can be parallelized using graphics processing units (GPUs). This speeds up the training process and enables the use of larger network architectures, that are able to approximate more complex underlying functions.

3. The development of new algorithms. Instead of the standard fully-connected architectures (see Figure 1.2), other network architectures were invented for specific applications. For example, convolutional neural networks [26, 19] for image processing or transformers [27] for text processing, which are used as architecture of large language models such as GPT [28].

Deep learning is powerful and useful but very data-hungry, which makes it hard to apply for some applications. Indeed, some fields have little, publicly available data. For example, in the medical field, data is subject to (i) the privacy of the patient and (ii) requires tremendous work from radiologists to correctly label the data, i.e. constructing the datasets, in order to train models with supervised learning. This lack of data can be summarized as the **data scarcity** problem.

1.7 Overfitting

Machine learning models, especially neural networks are universal function approximators, meaning that they are in principle able to approximate any function to arbitrary precision, depending on their architectures. The more complex is the network, the more complex is the function it can approximate. Thus in principle, enough complex networks could learn *perfectly* the dataset it is trained on, i.e. the training set. However this does not mean that the model will perform well on unseen data. To verify this, the performance of the model is evaluated on the test set (see Section 1.4). The outputs or predictions of the model are computed and compared to the real labels. Since the model does not have been trained on the data of the test set, it is a good measure on how the model is generalizing.

A good model is a model generalizing well, that is, it is capable of recognizing general patterns on unseen data. While being trained, the model learns general patterns, but also patterns specific to the training set which also includes any data selection bias and specific defects such as noise. However, the model should not learn such patterns, in which case the model will not generalize on unseen data. When the model starts to learn those *bad* patterns and loses its capacity to generalize, this is called **overfitting**. When training a model, the trade-off between learning the general patterns from the training set that can be applied to unseen data, without learning patterns that are too specific or only relevant to the training set, or biases and noise, should be handle carefully. Overfitting arises often when the model is too complex w.r.t. the real underlying function. Figure 1.4 shows an example of a model overfitting after some time of training.

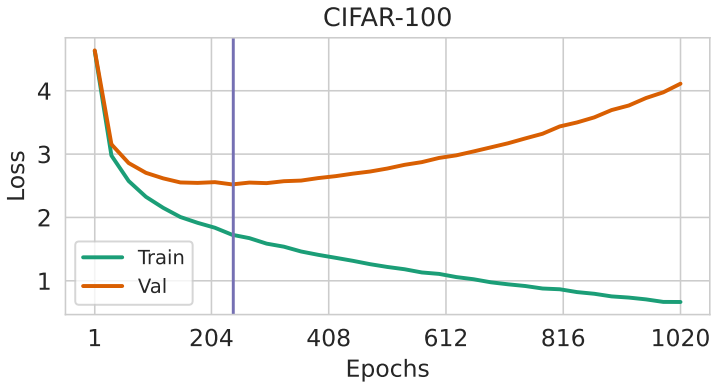


FIGURE 1.4: **Example of a model overfitting.** The local minimum of the loss function on the validation set (in orange) is shown by the blue line. After this line, the model is overfitting, as the validation loss increases while the training loss is decreasing. This example comes from a real experiment done in this work, using a classical learning algorithm on the CIFAR-100 dataset [29] (see Section 4.1.1).

1.8 Available methods against overfitting

Current methods to mitigate overfitting include early stopping [30], dropout [31] and regularization [32]. The principle of **early stopping** is to monitor the performance of the model on the validation set and stop the training process *at the right time*, that is when the loss on the validation set is at its local minimum. Overfitting starts when the loss increases on the validation set, while continuing to decrease on the training set (see Figure 1.4). This indicates when the model starts to learn patterns that are relevant only to the training set, i.e. patterns that are not useful to achieve a better generalization anymore. **Dropout** is used to avoid that neurons are depending or relying too much on each other when taking the decision. To mitigate this problem, the dropout methods simply zeros the activation, output, of one neuron with a probability p. Thus, some activations become null, `zero`, and they will not be propagated further in the network. Note that dropout only zeros activations during the training phase, but not during the inference phase, where all activations are taken into account for the final decision. **Regularization** could also be used. The goal is to add some penalty term to the loss function, so that the optimization problem becomes a bit different, as the objective function has been modified. Different penalty terms can be applied such as the L1 or L2 norm of the parameters [33, 34]. This ensures to reduce the impact of less useful features that have a too high impact during the computation of the predictions of the network. In most cases however early stopping is considered sufficient.

1.9 Transfer learning

One possible way to mitigate the data scarcity problem (see Section 1.6) is by using **transfer learning** (TL) [35, 36]. TL requires having one *large* dataset representing a source task, and one *small* dataset representing a similar target task, on which it is not possible to train a deep neural network due to data scarcity. The principle of transfer learning is to *transfer* the knowledge of the dataset of the source task to the dataset of the target task. Transfer learning first trains a model on the source task, and then specializes this trained model on the target task. This is done by re-initializing the last layers of the network when specializing on the target task. This is possible as the vanishing gradients problem is less impactful on the last layers. However, the model can be exposed to **catastrophic forgetting**. Indeed, when dealing with multiple source tasks, the model can at some point forget what it has learned from the first source tasks. As a consequence, some source task's knowledge could not be transferred at all to the target task, if it is stored in the last layers that have been re-initialized multiple times.

1.10 Multi-task learning

Another way besides transfer learning to mitigate the data scarcity problem is to use **multi-task learning** (MTL) [37]. The goal is to train a network that is able to perform on different tasks *at the same time*. The tasks have a common feature representation inside the network, i.e. a shared feature space. This is achieved by concatenating the input of each tasks, activating the network on this concatenated input, and finally by splitting the output at the end, to get a separated output for each task. This means that each task is learned in conjunction, rather than in sequence as in transfer learning. Learning tasks in sequence can lead to the **moving target** problem, in which there are too many different, diverging, objectives making the model not being able to learn a clear and general objective. MTL avoids the moving target problem by learning the tasks in conjunction, as a single task. However, MTL could suffer from data imbalance between the tasks, making it harder to train on some smaller specific task. Additionally, to be able to solve multiple tasks at the same time, the model needs to be significantly more complex. But as the model is getting more and more complex, overfitting can arise quicker than expected, thus becoming bad for generalization on the other tasks or on unseen data (see Section 1.7).

1.11 Motivation

With the onset of deep learning, neural networks model are getting bigger and bigger, and therefore the amount of data required to train them becomes considerable. Nonetheless, training such large models can lead to vanishing gradients, where the first layers receive little feedback error (see Section 1.6). Mitigating the vanishing gradients problem is crucial since the very first layers are extremely important, as they take the role of extracting the features from the original input. It is thus difficult to train large models in some fields where data is not abundant. However datasets coming from the same field have sometimes common features, but they can be too heterogeneous between each other, making it impossible to simply merge them into bigger datasets. For example, to train a model that can predict if there is a cancerous tumor on a prostate MRI, a great amount of data is needed. Yet other body parts containing cancerous tumors as well cannot be used since they do not represent the prostate, making them heterogeneous and thus cannot be merged.

This problem is tackled by the use of a newly designed meta supervised learning algorithm, called **Typhon** [38]. Typhon is able to train a single model on different, heterogeneous datasets, yet containing common features to increase the performance, on possibly all datasets compared to classical learning. It bridges transfer learning and multi-task learning. Typhon differs from transfer learning in the sense that it learns the tasks in parallel rather than sequentially, while being distinct from multi-task learning as Typhon learns each task as a separated entity rather than viewing them as a single task. Additionally, Typhon diverges from mixture of experts [39] as the latter trains multiple, *expert* models independently with the help of a *gating network* that decides which model has to be activated depending on the input, while Typhon trains a single model on all the data and does not require such gating network. Typhon is described thoroughly in Chapter 2. However, this algorithm requires a set of heterogeneous datasets having similar features, and as mentioned this is not always possible. This work thus aims at extending the original Typhon algorithm into a modified algorithm that can use the power of Typhon on a single dataset, taking off the multiple datasets requirement.

1.12 Contributions

This work proposes to extend the applicability of the Typhon learning algorithm to other datasets, other tasks, and finally to two new versions of Typhon that require only a single dataset. More specifically, this work:

1. Proposes a refactored, merged version of the original Typhon algorithm, such that it can not only be used in classification tasks, but also in segmentation and auto-encoding tasks. The new version of Typhon is a combination of different contributions of various works, including [40, 41, 38, 42]. The code is available on GitHub[1].

2. Creates a new algorithm of the Typhon family, a single-dataset version, containing *super-classes* by combining common classes together. Typhon treats each super-class as a complete dataset, that contains multiple classes.

3. Creates another novel single-dataset Typhon algorithm, such that it is possible to use Typhon with *any* single dataset. This new algorithm performs one-class classification on each class independently, inspired by one-class algorithms such as one-class support vector machines [43].

4. Proposes a new metric that measures the overfitting of a model during its training phase. This enables the possibility to compare the resilience of various algorithms against overfitting.

5. Validates the performance of the new Typhon algorithms in mitigating overfitting and increasing sample efficiency.

The next Chapter will present comprehensive details about the Typhon framework in particular, before getting into the new algorithms in Chapter 3. Chapter 4 details the experimental setup used and the results obtained. Finally, Chapter 5 presents the conclusions and hints at some promising avenues for future work.

[1] https://github.com/ChristopheBroillet/typhon-single-dataset

Chapter 2

The Typhon framework

This chapter introduces the Typhon framework, from its origin in Section 2.1 to its current form in Section 2.3, passing by explaining the steps taken to arrive there in Section 2.2. This chapter finally ends by showing the use of Typhon in a proven successfully application in Section 2.4.

2.1 Heterogeneous sequential transfer: Hydra

The story of Typhon started with the Hydra framework [40] in 2019. Hydra was a new learning algorithm that was created as a new attempt to mitigate the data scarcity problem common in many real-world applications. It was illustrated specifically in the application to detect cancerous tumors on medical images. This specific medical application was chosen because in this field, images are very hard to obtain due to various reasons: the privacy information of the patients, the cost of the work to label the images, or the heterogeneity of the devices or hardware to record and capture the images. Additionally, as explained in Section 1.6, deep learning algorithms such as deep neural networks are very data-hungry, i.e. they need a tremendous amount of data to mitigate the problem of vanishing gradients, and to actually learn the underlying function that generated the data.

The goal of Hydra was to apply transfer learning across different heterogeneous dataset, in the specific case of medical images. For example, datasets of different modalities such as magnetic resonance images, computed tomography scans or even ultrasound images, but also coming from different body parts such as the brain, the prostate or the lungs. By utilizing Hydra on different datasets, the trained model is able to aggregate information it learns from one organ and transfer it into another one. This is the so-called *sequential* transfer learning, as it learns one dataset after another. At the end, it finally re-learns on a target dataset, so that it can be specialized on this particular dataset.

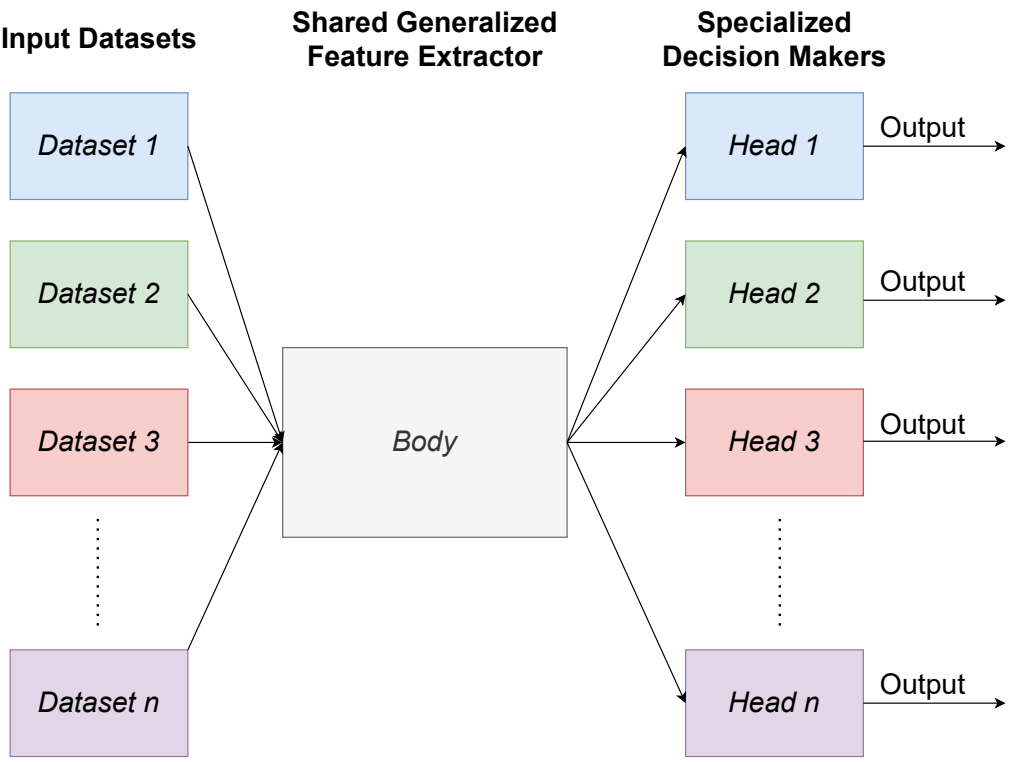


FIGURE 2.1: **Generic Hydra/Typhon architecture.** Different input datasets are required to train a model with Hydra/Typhon. These input datasets firstly pass through a feature extractor, the body, that is shared across all datasets. Then, the output of the feature extractor is sent to the corresponding decision maker, the head that is specific to the dataset, which is the only one activated, the others remain inactive. These heads output the required result depending on the task the model is trained on.

Hydra is very similar to transfer learning but still a bit different in the sense that the parts of the network trained are not always the same during the training process. By taking a network architecture, Hydra splits it into two distinct parts: (i) the **feature extractor** (FE) and (ii) the **decision maker** (DM). The location of the split is arbitrary, thus becoming an hyperparameter. The core concept of Hydra is that the FE is shared across all datasets while there is one single DM per each dataset, resulting in the Hydra name coming from the mythical creature that has one body, the FE, and several heads, the DMs. This architecture makes the FE very generalized across all dataset and able to learn abstract features, while one DM is specialized to one specific dataset. Figure 2.1 shows the generic, abstract architecture of a model trained with Hydra. However, Hydra requires to have one dataset selected as target. This target dataset is the main objective and is improved using different other datasets as support datasets. The learning process using Hydra goes as shown in Algorithm 1. For more information, the original work is referred [40].

Algorithm 1 Hydra

Requires: Complete Hydra model $\widehat{f}_\theta\langle FE, \{DM_i\}\rangle$, target dataset D_0, list of support datasets $\{D_i\}$
for each epoch **do** ▷ Train the target dataset
  $batch \leftarrow getBatch(D_0)$
  $\widehat{f}_\theta\langle FE, DM_0\rangle \leftarrow train(\widehat{f}_\theta\langle FE, DM_0\rangle, batch)$
for each support dataset D_i **do**
  $Freeze(FE)$
  for each epoch **do** ▷ Train on a support dataset with frozen FE
    $batch \leftarrow getBatch(D_i)$
    $\widehat{f}_\theta\langle FE, DM_i\rangle \leftarrow train(\widehat{f}_\theta\langle FE, DM_i\rangle, batch)$
  $Unfreeze(FE)$
  for each epoch **do** ▷ Train on the same support dataset
    $batch \leftarrow getBatch(D_i)$
    $\widehat{f}_\theta\langle FE, DM_i\rangle \leftarrow train(\widehat{f}_\theta\langle FE, DM_i\rangle, batch)$
$Freeze(FE)$
for each epoch **do** ▷ Train the target dataset with frozen FE
  $batch \leftarrow getBatch(D_0)$
  $\widehat{f}_\theta\langle FE, DM_0\rangle \leftarrow train(\widehat{f}_\theta\langle FE, DM_0\rangle, batch)$
$Unfreeze(FE)$
for each epoch **do** ▷ Train the target dataset
  $batch \leftarrow getBatch(D_0)$
  $\widehat{f}_\theta\langle FE, DM_0\rangle \leftarrow train(\widehat{f}_\theta\langle FE, DM_0\rangle, batch)$
return $\widehat{f}_\theta$

FIGURE 2.2: When starting to train a support dataset, the decision maker (DM) for this particular dataset is initialized randomly. To avoid error propagation due to the newly initialized DM, the feature extractor (FE) is *frozen*, that is the parameters of the FE are not updated. In that way, only the new DM is being trained on the dataset in a first phase. When the DM is enough trained, the FE is unfrozen and starts to learn patterns from this support dataset.

2.2 Trying to switch from sequential to parallel transfer

As described in Section 2.1, Hydra uses sequential transfer learning to improve the performance on a target dataset by using multiple, heterogeneous support datasets sequentially. The knowledge of the support datasets can be transferred further to the next support datasets and hopefully to the target dataset. However, as seen in Section 1.9, using multiple source datasets can lead to the catastrophic forgetting problem, where the knowledge of the very first datasets can be forgotten during training. Hence the useful knowledge from a support dataset could possibly not be transferred to the target dataset.

While trying to mitigate the catastrophic forgetting problem occurring in Hydra, the framework has evolved to become **Typhon**. Instead of being based on *sequential* transfer learning, now Typhon uses *parallel* transfer learning, bridging transfer learning and multitask learning. It means it is able to learn different heterogeneous

datasets *at the same time*, i.e in parallel, simulating multi-task learning. Typhon still differs from multi-task learning in the sense that it treats each tasks or datasets independently, rather than as a single entity. However, another problem arising now is the moving target, as mentioned in Section 1.10. Indeed, the algorithm seems to learn different objectives at the same time, because of the difference in the datasets, such as its format or its contained information. But instead of being a problem, it is in fact the strength of the Typhon way of learning: using moving target to learn different objectives and aggregate common information or knowledge useful across all datasets.

2.3 Addressing the parallel transfer problem: Typhon

Typhon is a meta-learning algorithm that is capable of learning different and heterogeneous datasets at the same time, using parallel transfer learning, that bridges multi-task learning and transfer learning. Precisely, Typhon is (i) a multi-task learning algorithm, where the goal is to train a model that is able to solve different tasks without aggregating them while (ii) re-using knowledge from other tasks or datasets to further improve all other tasks in a parallel transfer learning way. To this end, Typhon splits the network architecture akin to Hydra. Given a neural network architecture, Typhon splits the network arbitrarily in two main parts: (i) one common, shared **feature extractor** (FE) and (ii) several **decision makers** (DMs), one per each dataset. The name *Typhon* has also a meaning as Typhon is the father of Hydra in the mythology, while being a generalization with higher performance than Hydra in the algorithmic world. Figure 2.1 shows the generic architecture of Typhon.

The training with Typhon goes as follows. In one epoch, Typhon sees one batch of each dataset. For one specific batch, Typhon trains the FE and the corresponding DM, and repeat this step for all datasets. Thus there is one rotation over all datasets at each epoch. Due to this repetition, catastrophic forgetting is mitigated (see Section 1.9), as Typhon re-learns the patterns specific to each dataset it has already learned, but possibly forgotten in the meantime. This can be illustrated by a student that revises different subjects, such as literature, geography and computer science. Every day, the student starts by learning literature, then switch to geography and finally computer science. At the end of the day, it is possible that some topics in literature are already forgotten, but as the student will learn them again the day after, it will be less prone to forget them again as the revision continues. Algorithm 2 shows how a model is trained using Typhon.

Algorithm 2 Classic Typhon

Requires: Complete Typhon model $\widehat{f}_\theta\langle FE, \{DM_i\}\rangle$, list of datasets $\{D_i\}$
for each epoch **do**
  for each dataset D_i **do**
    $batch \leftarrow getBatch(D_i)$
    $\widehat{f}_\theta\langle FE, DM_i\rangle \leftarrow train(\widehat{f}_\theta\langle FE, DM_i\rangle, batch)$
return $\widehat{f}_\theta$

FIGURE 2.3: The Typhon algorithm is much simpler and more elegant than the Hydra algorithm shown in Algorithm 1.

Finally as Typhon learns different datasets in parallel, it does actually not output only one single trained model as in classical learning, in transfer learning or even with Hydra, but n trained models, one per each dataset. This may make the cumulative training time faster than classical learning algorithms (see Section 4.4).

To make the training actually possible from Algorithm 2, Typhon needs a continuous stream of batches coming from the datasets. This is achieved in the framework with a *loop loader*: it wraps a dataset, and gives batches, with constant size, until the dataset is exhausted. After that, the loop loader reloads and shuffles the dataset, and then continues to give batches ad eternam. Each dataset is wrapped up in a loop loader, so that Typhon can pick a batch from any dataset whenever it wants.

Typhon can be applied to any machine learning tasks or neural network architectures, making it a meta-learning algorithm. Indeed, the only thing to do is to arbitrarily set the separation between the feature extractor part and the decision maker part within the architecture. However the location of the split is considered as a new hyperparameter to be optimized, as the learning rate or the batch size for example.

2.4 Typhon in computer-aided diagnosis

As Typhon is a meta-learning algorithm, it can be applied in principle to *any* applications using supervised learning. Starting with classification for example, Typhon can learn similar datasets that classify some objects such as images or other supports, and is able to learn the abstract common patterns in order to improve the performance of the classification on the datasets. This Section presents an applied example of Typhon used to train a neural network that can detect cancerous tumors on medical images, i.e. classify whether a medical image contains a cancerous tumor or not. This task belongs to the topic of computer-aided diagnosis.

Cancer leads to millions of deaths every year. To prevent and possibly cure it, the diagnoses need to be done by medical doctors as soon as possible. Due to tiredness or a considerable amount of side work, these professionals may not see correctly the very first stages of the cancer, i.e. very small early cancerous tumors. Machines could help radiologist to detect cancerous tumor on medical images, becoming computer-aided diagnosis machines.

The problem of cancerous tumor detection can be turned into a classification problem, and solved via supervised learning. Different methods have already been shown [44, 45]. The performance of these algorithms are limited because the true underlying function is very hard to approximate as it is very complex, thus a deep neural network is required. Unfortunately as seen before, data is very limited in this field making the training procedure of such large neural network difficult.

In order to mitigate the problem of data scarcity, complex models can be trained using the Typhon learning algorithm. As explained in Section 2.3, Typhon can train a model with different heterogeneous datasets, coming from different body parts for example, but also from different modalities, i.e. types of images such as magnetic resonance images or ultrasound images. Typhon is able to aggregate all useful features from all body parts and can apply what it has learned to other organs. This implies that classifying a specific cancerous tumor, for example a prostate cancer, is easier. In classical learning, a tremendous amount of data containing images of the prostate is required, whereas in Typhon the data can also contain images from other modalities, or coming from other body parts. More information can be found in the original work [38].

Chapter 3

Methods

This work proposes two new algorithms of the Typhon family focused on single-dataset applications with the potential to mitigate overfitting and improve sample efficiency. At each epoch, Typhon sees data from different datasets, thus moving the objective target, and retains only information or features that are useful across all datasets. As mentioned in Section 2.3, this is done by training the model on one batch per each dataset per epoch, which mitigates catastrophic forgetting. This will thus remove noise or features that are not useful between all the datasets. This Chapter describes two different methods to use Typhon with a single dataset. In Section 3.1 the dataset is separated into *super-classes*, containing real classes that have common features, and assigning one super-class to one decision maker. Furthermore, in Section 3.2 the idea is developed more and assigns each single class to one decision maker. Finally in Section 3.3, a new metric to measure overfitting is proposed and described.

3.1 Datasets with super-classes: Two Levels Typhon

In order to leverage the generalization capability of Typhon, the datasets should be similar enough to have common features to share, but also be distinct enough so that specific information can be learned about each super-class. So far, only a set of different, heterogeneous datasets has been used with Typhon. The first idea to extend Typhon to a single-dataset version could be to make various sub-datasets from a single one, and train Typhon on them as previously described in Section 2.3 and Algorithm 2. This can be achieved if the original dataset has clear distinct separations between the classes, enabling to aggregate them into *super-classes*. Indeed, in some classification datasets, some classes have common features shared between them. For example, in a dataset containing images of animals, one super-class could be `sea-animal`, or `bird`. The true effective classes of the dataset are dispatched into those super-classes, depending on what type of animal they describe. Thus, inside those super-classes there will be the real classes of the datasets, simply aggregated.

Once this structure is built, Typhon can be used to train a model having one decision maker, one head, per each super-class. Each decision maker classifies the real classes coming from its super-class. Thus, each head will have one output neuron per each sub-class contained in the super-class. Each neuron outputs then the probability of belonging to the sub-class (see Section 1.5.2). For example, a head assigned to the `sea-animal` super-class will classify `shark`, `cat-fish` and `whale`, whereas the `bird` head will classify `pigeon`, `crow` and `eagle`. The original multi-class classification problem is split into multiple sub-classification problems. The hypothesis is that this way Typhon can learn specific class-features useful across the whole dataset, which would have not been retained by using classical learning algorithms, thus improving the performance of the classification. Figure 3.1 shows example classes

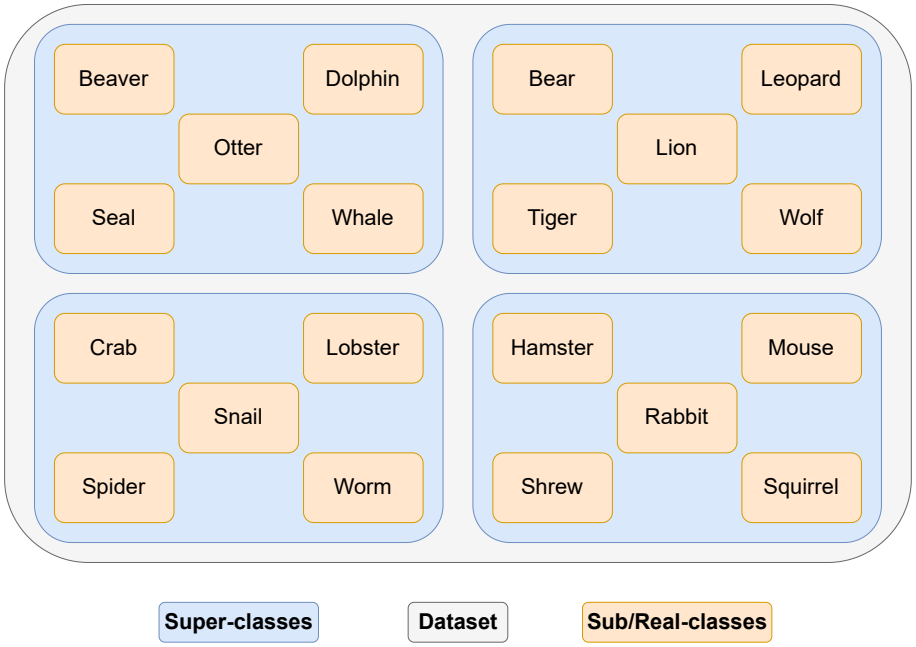


FIGURE 3.1: **Splitting data into super-classes.** Example super-classes from the CIFAR-100 dataset [29], which can be learned using Two Levels Typhon. The full dataset has first a separation between super-classes, which then contain the real, original classes of the dataset.

from the CIFAR-100 dataset [29] that are dispatched into super-classes, which are assigned to Typhon's heads.

To use Typhon with this method, the dataset has to be prepared and correctly formatted beforehand, as shown in Figure 3.1. Firstly, the super-classes structure needs to be designed: the model should know which sub-class belongs to which super-class, by encoding the sub-classes among the super-classes. Secondly, the model still needs to remember what are all the sub-classes, because the original task's goal is to classify the inputs among those sub-classes. An implementation solution for this two requirements is to use *one-hot* labels (see Section 1.5.2). This first new algorithm is called **Two Levels Typhon**. The name Two Levels Typhon is coming from this dataset preparation, where the model needs to know (i) the (sub-)class level labels and (ii) the super-class level labels.

3.1.1 Inference and evaluation

The model needs to be able to output a prediction, in order to be evaluated. The trick here is, for a given input, to activate all heads, corresponding to super-classes, aggregate the activations, which are obtained by applying the standard logisitic activation function to each individual neuron, and take the highest one as the inferred, predicted class. Indeed, each head is able to distinguish if the input belongs to its own super-class, where in average all its activations will be higher than the other heads, but also to which particular sub-class, corresponding to the real predicted class. This process makes however the inference part potentially slower than in classical models, since all heads need to be activated in turn. To mitigate this problem, the inference is split into two parts. First the inputs pass through the feature extractor only once, and its output is then distributed across all decision makers. In that

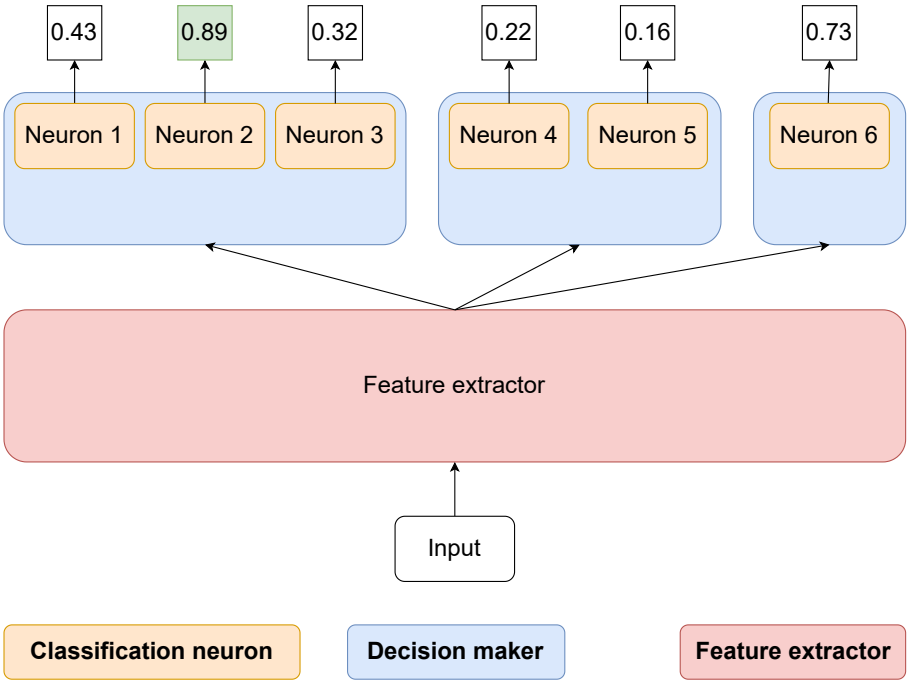


FIGURE 3.2: **Two Levels Typhon model inference.** The architecture is composed of one shared feature extractor, in red, and of multiple decision makers, in blue, one per each super-class. Each decision maker performs multi-class classification, thus contains in its last layer one neuron per each sub-class, in orange, that outputs the classification score. The input firstly goes through the feature extractor, and is then distributed across all decision makers. The highest score or activation among all output neurons, in green, is interpreted as the prediction of the network, same as with a classic monolithic architecture.

way, the feature extractor is activated only once, thus saving computation time during the inference part. Figure 3.2 shows how the inference works within the model.

This process is repeated for all observations of the dataset that is inferred. The advantage of the combination of this evaluation and the shape of the model is that the full model can be evaluated to compare the performance against classical models, but additionally each decision maker can be evaluated separately as well. Indeed, it is possible to construct metrics for each specific decision maker, enabling to detect which one performs better than the others. That is, for a given observation, the decision maker should be able to tell if the input is belonging to one of its sub-classes but also if it is not. This can give a hint on how to optimize the hyperparameters further for each decision maker, improving the overall performance of the model.

3.1.2 Training

In order to be able to train a model with Typhon and this super-classes dataset architecture, the fundamental original Typhon algorithm needs to be modified. This altered algorithm with super-classes requires the binary cross entropy as loss function (see Section 1.3). This is firstly because the outputs or the predictions of the model contain classification information on all classes and secondly because the labels are encoded in a *one-hot* fashion (see Section 1.5.2). More precisely, this can be interpreted as multiple binary classification problems.

In theory, the original Typhon algorithm (see Algorithm 2) could be used as is to train a dataset with super-classes, by rotating over each super-class, and train accordingly the specific head of the corresponding super-class. However in practice, that means that each head only sees observations coming from its own super-class. This implies that the heads are not trained to detect samples that are not coming from their own super-class. To solve this problem, a modified Typhon algorithm is proposed, and is called Two Levels Typhon. Given a batch of one super-class, the algorithm makes a two steps training, inspired by one-class algorithms (outlier detection) such as one-class support vector machines [43]. These steps are the following:

1. Two Levels Typhon trains the batch on its corresponding head. This is called **positive training**, since the label contains the positive sub-class, i.e. the `one`.
2. Two Levels Typhon trains the same batch on other, different head(s), where the label becomes all `zeros`. This part is called **negative training**.

For the negative training part, all heads need eventually to see samples for all other super-classes, implying that at each new negative training, another head is selected to be trained on the batch. This ensures that all heads are seeing observations for all (negative) super-classes. The Two Levels Typhon algorithm enables to train on a specific, chosen number of negative heads, thus introducing a new hyperparameter. Using this two-steps training, each head learns to distinguish positive observations coming from its own super-class, and is able to tell what is the sub-class, but it can also tells when an input is not coming from one of its sub-class, where all its activations should be near `zero`. This makes the output activation of each neuron stronger, and thus the highest activation can be trusted more as being the real class. This new algorithm is described in Algorithm 3.

Algorithm 3 Two Levels Typhon

Requires: Complete Typhon model $\widehat{f_\theta}\langle FE, \{DM_i\}\rangle$,
single dataset with super-classes $\{D_i\}$, number of negative heads to train m
for each epoch **do**
  for each super-class D_i **do**
    $batch \leftarrow getBatch(D_i)$
    $\widehat{f_\theta}\langle FE, DM_i\rangle \leftarrow train(\widehat{f_\theta}\langle FE, DM_i\rangle, batch)$ ▷ Positive training
    for $1, \ldots, m$ **do**
      $j \leftarrow nextHeadToTrain(i)$ ▷ Select a head for negative training
      $\widehat{f_\theta}\langle FE, DM_j\rangle \leftarrow train(\widehat{f_\theta}\langle FE, DM_j\rangle, batch)$ ▷ Negative training
return $\widehat{f_\theta}$

3.2 Single-dataset version: Ultra Typhon

Two Levels Typhon requires a dataset that has a clear partitioning between classes, such that they can be dispatched into super-classes. Two Levels Typhon, and standard Typhon, can in principle not be used with datasets that do not have this clear separation. However, this work proposes a second new algorithm of the Typhon family, which is a truly single-dataset version of Typhon, that can be applied to *any* single dataset. This new algorithm is called **Ultra Typhon**. Ultra Typhon is in fact a special case of the Two Levels Typhon algorithm. Instead of having one decision maker per each super-class that performs multi-class classification, Ultra Typhon uses one decision maker per each proper class, doing one-class classification. This implies that each head has only one single output neuron, that tells the probability of belonging to the class.

The dataset preparation for Ultra Typhon is very simple. The labels need to be encoded using the *one-hot* encoding (see Section 1.5.2). Otherwise no other preparation is needed, as Ultra Typhon uses the original shape of the dataset.

3.2.1 Inference and evaluation

The inference and the evaluation part are also very similar to Two Levels Typhon. One observation goes into the network as input, passes through the feature extractor. The activation of the feature extractor is distributed across all decision makers in order to save time, and all heads are activated. Those activations are aggregated and taken as a whole. There is one activation per each class, and the highest one is chosen to be the prediction of the network. This also enables the opportunity to evaluate (i) the full model as a whole and compare it with other model's performance, e.g. with classical learning, but also (ii) each head separately, to verify which class seems harder to learn for the model, which additionally makes it possible to tune the hyperparameters for each head **independently**. Figure 3.3 shows the inference of a model trained with Ultra Typhon.

3.2.2 Training

To train a dataset with Ultra Typhon, the same concept of positive and negative training is used, as explained in Section 3.1.2. The difference is that the label is no more a collection of `zeros` and a single `one`, but the label for each head turns out to be a single number, either a `one` (the input belongs to the class) or a `zero` (the input does not belong to the class) because each DM has only one output neuron. This again requires the binary cross entropy as loss function (see Section 1.5.2), as the probability of each single class is computed. The rotation over the datasets, simulated by the classes themselves, is again used: given a batch from one class, Ultra Typhon trains the model on (i) the corresponding head (positive training, the label is a `one`) and (ii) other head(s) (negative training, the label is a `zero`). Algorithm 4 shows the training process used by Ultra Typhon.

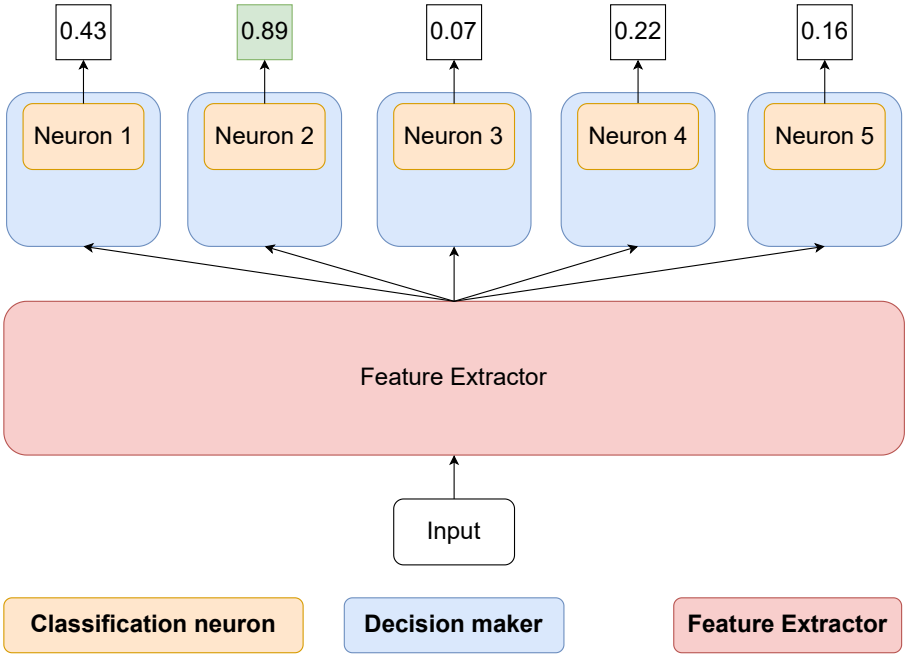


FIGURE 3.3: **Ultra Typhon model inference.** The inference is similar to Two Levels Typhon (see Figure 3.2). The output neurons, in orange, are put in the last layer of each decision maker and give the classification scores. The highest activation among all those neurons, in green, is interpreted as the prediction of the network, same as with a classic monolithic architecture.

Algorithm 4 Ultra Typhon

Requires: Complete Typhon model $\widehat{f}_\theta\langle FE, \{DM_i\}\rangle$,
single dataset with classes $\{D_i\}$, number of negative heads to train m
for each epoch **do**
  for each class D_i **do**
    $batch \leftarrow getBatch(D_i)$
    $\widehat{f}_\theta\langle FE, DM_i\rangle \leftarrow train(\widehat{f}_\theta\langle FE, DM_i\rangle, batch)$ ▷ Positive training
    for $1, \ldots, m$ **do**
      $j \leftarrow nextHeadToTrain(i)$ ▷ Select a head for negative training
      $\widehat{f}_\theta\langle FE, DM_j\rangle \leftarrow train(\widehat{f}_\theta\langle FE, DM_j\rangle, batch)$ ▷ Negative training
return $\widehat{f}_\theta$

3.3 Overfitting score

Both new Two Levels Typhon and Ultra Typhon algorithms leverage the moving target problem as a mitigation tool for overfitting. To actually see *how much* are those algorithms with a given model prompt to overfitting, a metric is required. The metric should be fair across all type of experiments: it should be invariant to (i) the length of the experiments and also to (ii) the type of loss function. To fulfill those requirements, a new metric is proposed in this work, and further used in the experiments in Chapter 4.

Algorithm 5 Overfitting score

Requires: Training losses $T = [x_T(t_0), \ldots, x_T(t_n)]$,
validation losses $V = [x_V(t_0), \ldots, x_V(t_n)]$, timestamps $t_0, \ldots, t_n$
$T \leftarrow smooth(T)$ ▷ Smooth with moving average
$V \leftarrow smooth(V)$
$S = []$ ▷ Empty list to store point-wise scores
for each timestamp t_i, $i = 1, \ldots, n-1$ **do**
  Compute $\frac{d}{dt}x_T(t_i)$ ▷ According to Equation 3.1
  Compute $\frac{d}{dt}x_V(t_i)$
  if $\frac{d}{dt}x_V(t_i) \leq 0$ or $\left(\frac{d}{dt}x_V(t_i) > 0 \text{ and } \frac{d}{dt}x_T(t_i) > 0\right)$ **then** ▷ No overfitting
    $s = 0$
  else ▷ Overfitting
    $s = \frac{d}{dt}x_V(t_i) - \frac{d}{dt}x_T(t_i)$
  Add s to the list S
score $= average(S)$
return score

The core idea is to compare the trends between the values of the loss function on the training set and the ones on the validation set. As suggested in Section 1.7, this is an indicator when overfitting is occurring: at the point where the validation loss reaches its local minimum, it will increase afterwards thus having worse predictions, while the training loss is still decreasing, and this is where the model starts to overfit. The proposed metric will therefore aggregate those trends. To represent the trend of a curve, the derivative is often used. It describes the instantaneous variation of a point on a curve. However the training loss and validation loss curves are not continuous thus the derivative cannot be used as is. Instead, a discrete version of the derivative is used.

Let T and V be the set of measurements on the training and validation sets respectively. At each timestamp $t_0, \ldots, t_n$, the value of the loss function $x(t_i)$ is measured on both sets, leading to $T = [x_T(t_0), \ldots, x_T(t_n)]$ and $V = [x_V(t_0), \ldots, x_V(t_n)]$ being two time-series. Then the (discrete) derivative at the point $x(t_i)$ is defined by Equation 3.1, which is the slope of the secant between the two neighbor points. Note that both the timestamps $\hat{t}_i$ in the denominator of Equation 3.1 and the loss values $\hat{x}(t_i)$ are normalized in the range $[0, 1]$. This is to ensure that the overfitting metric is invariant to (i) the values between the timestamps but also to (ii) the values of the loss function. Thus, this metric only measures the overfitting within a given *time window*, and consequently should be applied to experiments having the same number of epochs or number of samples trained on for a fair comparison. The computation of the overfitting score is described in Algorithm 5.

$$\frac{d}{dt}x(t_i) = \frac{\hat{x}(t_{i+1}) - \hat{x}(t_{i-1})}{\hat{t}_{i+1} - \hat{t}_{i-1}} \tag{3.1}$$

This new proposed metric will be used later in Chapter 4 to compare different algorithms, such as classical learning, Typhon, Two Levels Typhon and Ultra Typhon, on their ability to mitigate overfitting.

Chapter 4

Experiments

This Chapter presents several experiments using the two new Typhon algorithms, Two Levels Typhon described in Section 4.5 and Ultra Typhon explained in Section 4.6. Those experiments are at the core of this work, and show practically how the assumptions and ideas developed and designed in Chapter 3 are applying to experiments with real datasets. The setup such as the datasets and the model architectures used are all described in Section 4.1 and Section 4.2 respectively. To have a better comparison baseline, the original Typhon algorithm [38] is also tested in Section 4.4.

4.1 Datasets

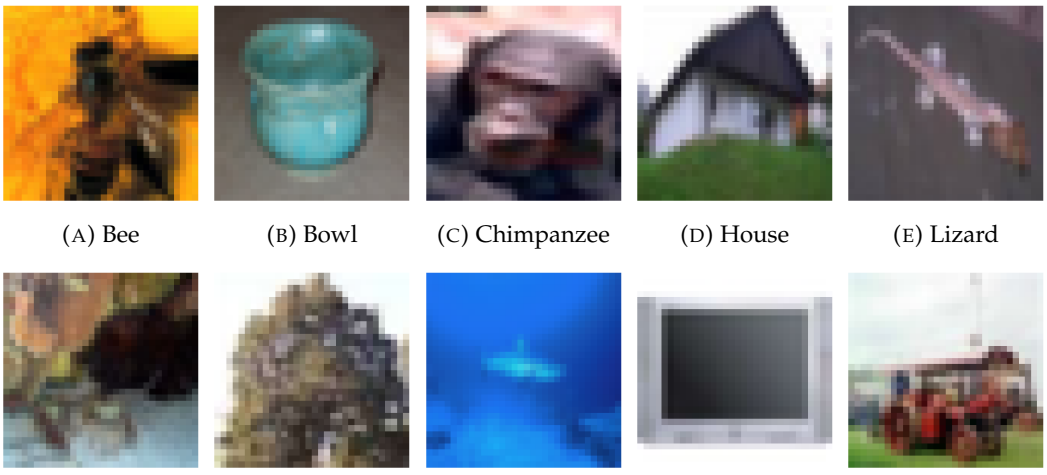
This first Section details the two datasets used in the experiments, namely the CIFAR-100 and CIFAR-10 [29] datasets, which contain images coming from different classes. These two datasets have been widely used already [46, 47, 48].

4.1.1 CIFAR-100

This first dataset is used for the Two Levels Typhon algorithm experiments. The Two Levels Typhon algorithm requires a dataset having a clear separation between the classes into super-classes (see Section 3.1). This can be manually done for some datasets, if they contain some common features between classes, but in this work a dataset already meeting those requirements is used. This is the CIFAR-100 dataset [29]. This dataset regroups small 32x32 pixels color images. The training set consists of 50'000 images while the test set contains 10'000 images. The CIFAR-100 dataset is composed of a total of 100 classes, i.e. 500 training images and 100 test images per class, divided into 20 super-classes. Each super-class is composed of 5 (sub-)classes. The detailed classes in this dataset are shown in Table 4.1, while some examples images are shown in Figure 4.1. This dataset is freely available[1].

No specific pre-processing is made for this dataset. All three color channels are kept, and each of them has pixel values in the range $[0, 255]$. The original test set remains untouched and is directly used as the test set in the experiments. However, as there is no official validation set, the latter is created from the training set. Out of the 50'000 images in the training set, 10'000 are taken randomly to form the validation set, while the rest forms the training set. In the end, the training set is composed of 40'000 images (400 images per class in average), the validation set of 10'000 images (100 images per class in average) and the test set of 10'000 as well (100 images per class).

[1] https://www.cs.toronto.edu/~kriz/cifar.html



(A) Bee (B) Bowl (C) Chimpanzee (D) House (E) Lizard

(F) Lobster (G) Oak tree (H) Shark (I) Television (J) Tractor

FIGURE 4.1: **CIFAR-100 examples images.** 10 random images, 32x32 pixels, coming from different super-classes. The sub-class name is written in caption. There are in total 20 super-classes, which are listed with their sub-classes in Table 4.1.

Super-class	**Classes**
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
food containers	bottles, bowls, cans, cups, plates
fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
household electrical devices	clock, computer keyboard, lamp, telephone, television
household furniture	bed, chair, couch, table, wardrobe
insects	bee, beetle, butterfly, caterpillar, cockroach
large carnivores	bear, leopard, lion, tiger, wolf
large man-made outdoor things	bridge, castle, house, road, skyscraper
large natural outdoor scenes	cloud, forest, mountain, plain, sea
large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates	crab, lobster, snail, spider, worm
people	baby, boy, girl, man, woman
reptiles	crocodile, dinosaur, lizard, snake, turtle
small mammals	hamster, mouse, rabbit, shrew, squirrel
trees	maple, oak, palm, pine, willow
vehicles 1	bicycle, bus, motorcycle, pickup truck, train
vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

TABLE 4.1: **Classes and super-classes of the CIFAR-100 dataset.** A total of 100 classes are separated into 20 super-classes. Each super-class is assigned to one decision maker in Two Levels Typhon.

4.1.2 CIFAR-10

The second dataset, CIFAR-10, is used for the experiments with Ultra Typhon. The power of Ultra Typhon is that it can train a model on any single dataset. In this work, in order to make a parallel with the Two Levels Typhon, the CIFAR-10 dataset [29] is used. This dataset is, as its sibling CIFAR-100, composed of color images of size 32x32, divided into 10 different classes, but here with no super-classes. The classes are: `airplane`, `automobile`, `bird`, `cat`, `deer`, `dog`, `frog`, `horse`, `ship` and `truck`. The training set is also containing 50'000 images and the test set 10'000. Each class is thus composed of 5'000 training images and 1'000 test images. Figure 4.2 shows some example images from this dataset, which is freely available[2].

(A) Airplane (B) Automobile (C) Bird (D) Cat (E) Deer

(F) Dog (G) Frog (H) Horse (I) Ship (J) Truck

FIGURE 4.2: **CIFAR-10 examples images.** These are 10 random images, 32x32 pixels, from the CIFAR-10 dataset. There is one image per each available class, with the class label as caption.

Similarly as for the CIFAR-100 dataset, each image has three color channels, and each of those channels has pixel values in the range $[0, 255]$. The original test set is kept as the test set in the experiments. In the CIFAR-10 dataset, there is again no official validation set, so it is created out of the training set: 10'000 images are taken randomly to form the validation set. Ultimately, the training set is composed of 40'000 images (4'000 images per class in average), the validation set of 10'000 images (1'000 images per class in average) and the test set of 10'000 as well (1'000 images per class).

4.2 Model architectures

As the two datasets contain images, the model trained is a convolutional neural network. The chosen architecture is taken from the CIFAR-10 tutorial[3] of the *Caffe* deep learning framework [49]. In the rest of this work, the model architecture will thus be referred as **Caffe**. The architecture of the model is composed of three convolutional blocks, that is a convolutional layer followed by a rectified linear unit (ReLU) [50] activation function and a pooling layer. The convolutional blocks are finally followed by two linear (fully connected) layers. This model has 116'321 trainable parameters.

[2] `https://www.cs.toronto.edu/~kriz/cifar.html`

[3] `http://caffe.berkeleyvision.org/gathered/examples/cifar10.html`

In the experiments, the split between the feature extractor and the decision makers is made at different place, so it can be analyzed and decided which split is better. Again, this split location turns out to be a new hyperparameter introduced in the original Typhon algorithm. Four different splits are used, leading to four *different* models to compare. Figure 4.3 shows the convolutional neural network architecture used for the experiments, as well as the location of the splits. Table 4.2 shows the size of the feature extractor and of each decision maker depending on the location of the split.

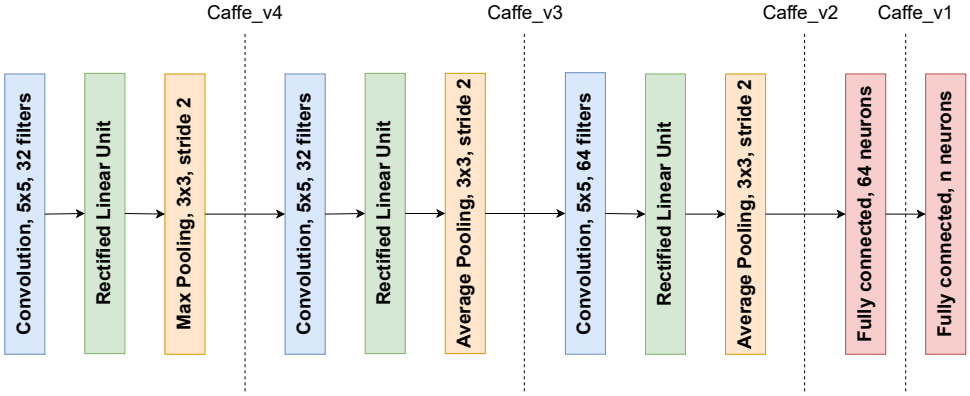


FIGURE 4.3: **Convolutional neural network architecture and splits.** Neural network architecture used for the experiments. It has three convolutional blocks (i.e. convolution, ReLU activation, and pooling), followed by two linear, fully-connected layers. This architecture has a total of 116'321 trainable parameters. Optional dropout is used between the ReLU and the pooling layers, and between the two linear layers. The dashed lines represent the different splits between the feature extractor (left part) and each decision maker (right part).

	Parameters in the FE	Parameters in each DM
Caffe_v1	116'256	65
Caffe_v2	79'328	36'993
Caffe_v3	28'064	88'257
Caffe_v4	2'432	113'889

TABLE 4.2: **Size of the different models.** Depending on the location of the split in the architecture, the feature extractor (FE) and each decision maker (DM) have a different number of trainable parameters.

4.3 Hardware and performance measurement

All the experiments have been run on a server with a 64-core Intel(R) Xeon(R) 6142 CPU at 2.60GHz, 6GB of RAM per core, and eight NVidia GeForce RTX 2080 Ti GPUs at 2.1GHz with 10GB of GDDR6 vRAM each. However, to get fair results and performance, each experiment only utilized a single CPU core and a single GPU.

To compare the multiple experiments done in this work, accuracy and F1-score on the test set (see Section 1.4), as well as the overfitting score (see Section 3.3) are used. Additionally, the accuracy on the training set is provided to assess the state

of the learning process by the end of the experiment. Specific hyperparameters for the two new algorithms, such as the architecture splitting (see Section 4.2) and the number of heads, decision makers activated during the negative training part (see Section 3.1.2 and Section 3.2.2) are also mentioned. Finally, the runtime for each experiment is given.

4.4 Standard Typhon

This Section presents the performance of the original, standard Typhon algorithm, running on both CIFAR-10 and CIFAR-100 datasets in a parallel way. This is firstly to show the performance of the original Typhon algorithm and serves also as a comparison baseline for the two new single-dataset algorithms that are at the core of this work, Two Levels Typhon and Ultra Typhon.

4.4.1 Results

In the first set of experiments, the original Typhon algorithm is used with the datasets CIFAR-10 and CIFAR-100. Table 4.3 shows the performances of the original algorithm with the different architecture splits. Classical learning experiments, with and without the addition of dropout, are also put for comparison.

	Split	Accuracy	F1-score	Overfit	Train acc.	Runtime [h]	# samples
Classic	-	0.67 | **0.33**	0.67 | **0.33**	1.16 | 0.48	**0.99** | **0.74**	14 | 20	$6 \cdot 10^7$ | $6 \cdot 10^7$
Dropout	-	**0.71** | 0.23	**0.70** | 0.23	**0.00** | **0.00**	0.76 | 0.25	15 | 20	$6 \cdot 10^7$ | $6 \cdot 10^7$
Typhon	v1	0.64 | 0.27	0.64 | 0.26	**0.00** | **0.00**	0.72 | 0.35	**17**	$6 \cdot 10^7$
Typhon	v2	0.65 | 0.29	0.65 | 0.28	**0.00** | **0.00**	0.72 | 0.37	**17**	$6 \cdot 10^7$
Typhon	v3	0.67 | 0.29	0.67 | 0.28	**0.00** | **0.00**	0.76 | 0.40	**17**	$6 \cdot 10^7$
Typhon	v4	0.65 | 0.29	0.65 | 0.28	**0.00** | **0.00**	0.74 | 0.40	**17**	$6 \cdot 10^7$

TABLE 4.3: **Results of the original Typhon experiments.** In the metrics, the first number stands for the CIFAR-10 test set while the second one is for the CIFAR-100 test set. The best model to keep is selected according the best accuracy obtained on the validation set over the run. Dropout has a value of 0.5. Classical learning requires separated experiments for each dataset, thus having two runtimes and number of samples while Typhon learns both datasets in parallel. All experiments used a learning rate of $5 \cdot 10^{-6}$ and a batch size of 256.

4.4.2 Discussion and analysis

These experiments demonstrate a couple of facts:

- **Typhon is able to mitigate overfitting.** Looking at the overfitting score in Table 4.3, classical learning is overfitting while Typhon shows still no sign of overfitting, with its `zero` score. This means that Typhon is able to generalize better than classical learning. Moreover, by looking at the accuracy on the training set, Typhon could still learn more useful and general patterns by running the experiments longer.

- **Typhon is close to dropout**. Both methods, Typhon and classical learning with a dropout value of 0.5, have a `zero` overfitting score, meaning both algorithms can still learn more useful and general patterns. This is also confirmed by the fact that there is still room for improvement in the accuracy on the training set. However, note that the comparison with dropout is hard, as the network is virtually smaller, less complex, during the training process since dropout deactivates some neurons within the network, which leads *naturally* to less overfitting.

- **Typhon has a faster runtime compared to classical algorithms.** The classical learning algorithms require to have separated experiments for each dataset, while Typhon is able to learn the two datasets in parallel. This has an impact on the total, combined runtime. Indeed, in Table 4.3 the total runtime for both datasets in classical learning is around 35 hours, while Typhon only needs 17 hours. This shows that Typhon reduces significantly the runtime in comparison with classical algorithms.

- **Typhon improves sample efficiency**. Compared to classical algorithms, Typhon is able to learn similar patterns with less observations seen. Indeed, while the accuracy values are similar, the number of samples seen by Typhon per dataset is **divided by two**. In this example experiment, using a classical algorithm, the number of samples is set to $6 \cdot 10^7$ for one dataset, while in Typhon this number is the same but for the two datasets in parallel, making the effective number of samples seen per dataset equals to $3 \cdot 10^7$. This means that Typhon is capable of reaching similar performance by seeing two times less input observations, which shows that it has a better sample efficiency than classical algorithms. This additionally strengthens the hypothesis that learning similar datasets in parallel as in Typhon is efficient.

4.5 Two Levels Typhon

This Section describes and showcases the results of the experiments using the Two Levels Typhon algorithm. It firstly gives the results of the experiments by showing the training curves and the metrics, and it ends with a discussion and further analysis of those results. The different hyperparameters are also given so that the experiments can be reproduced.

4.5.1 Results

Two Levels Typhon is tested on the different splits (see Figure 4.3) in order to check the behavior of the algorithm. Figure 4.4 shows then the training curves of those experiments, while Table 4.4 shows the actual hyperparameters used for each experiment and the metrics. Moreover, the influence of the number of activated heads during the negative training (see Section 3.1.2 and Section 3.2.2) is studied.

	Split	Neg. heads	Acc.	F1	Overfit	Train acc.	Runtime [h]	# samples
Classic	-	-	0.20	0.19	**0.00**	0.23	**2**	$3 \cdot 10^6$
Dropout	-	-	0.14	0.12	**0.00**	0.15	**2**	$3 \cdot 10^6$
2L Typhon	v1	19	0.21	0.21	0.01	0.31	20	$3 \cdot 10^6$
2L Typhon	v2	19	0.30	0.29	0.18	0.58	20	$3 \cdot 10^6$
2L Typhon	v3	19	0.30	0.30	0.74	0.89	20	$3 \cdot 10^6$
2L Typhon	v4	19	**0.34**	**0.34**	0.32	**0.99**	20	$3 \cdot 10^6$
2L Typhon	v2	1	0.22	0.21	**0.00**	0.28	**13**	$3 \cdot 10^6$
2L Typhon	v2	5	0.29	0.29	0.01	0.52	14	$3 \cdot 10^6$
2L Typhon	v2	10	0.29	0.29	0.10	0.56	16	$3 \cdot 10^6$
2L Typhon	v2	15	**0.30**	**0.30**	0.20	**0.62**	18	$3 \cdot 10^6$
2L Typhon	v2	19	**0.30**	0.29	0.18	0.58	20	$3 \cdot 10^6$

TABLE 4.4: **Results of the Two Levels Typhon experiments.** This Table shows the results of the experiments with Two Levels Typhon (2L Typhon), on the test set of CIFAR-100. The model to keep is selected according to the best accuracy obtained on the validation set. Note that the architecture in the classical training has no split, and dropout has a value of 0.2, lower than before as a dropout of 0.5 was not able to learn much with this number of samples in this experiment. All experiments used a learning rate of $5 \cdot 10^{-6}$ and a batch size of 256.

4.5.2 Discussion and analysis

With the help of Figure 4.4 and Table 4.4, some observations can be extrapolated:

- **A small feature extractor leads to higher accuracy.** Over the four experiments of Two Levels Typhon shown in Table 4.4, reducing the size of the feature extractor (see Table 4.2) results in an increased accuracy on the test set. This is also true for the accuracy on the training set, where the smallest feature extractor, Caffe_v4, seems to have almost learned everything it could as demonstrated by the 99% accuracy score. This could mean that the given architecture is not able to perform better on this particular dataset, and thus it requires to make the architecture more complex to get better results.

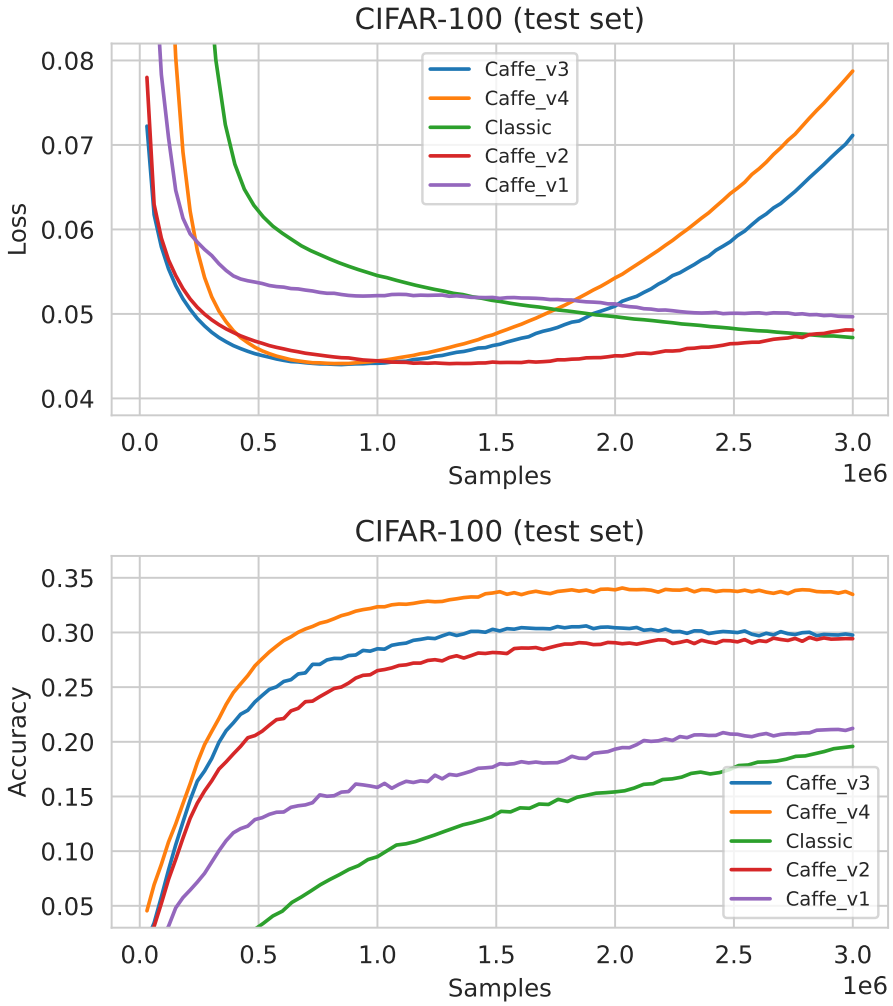


FIGURE 4.4: **Two Levels Typhon training curves.** The graph on the top shows the binary cross entropy loss (see Section 1.3) on the test set of CIFAR-100 against the number of samples on which the model has been trained, and the graph on the bottom shows its accuracy. Classic refers to classical training while the other curves represent the different architectures used, with a different split between the feature extractor and the decision maker.

- **A small feature extractor has a higher sample efficiency.** A direct consequence from the above mentioned point is that a larger feature extractor (FE) requires more samples, observations to reach similar performance compared to a smaller one. This is expected, since the features outputted by a larger FE are more complex than the ones outputted by a smaller FE. Additionally, reducing the complexity of the FE inherently increases the complexity of the decision maker (DM), which makes the classification part more robust. Indeed, the FE and the DM have their own complexity, as the original network's complexity is split between the two parts. Outside this preliminary study, the standard practice should be for the FE and each DM to have their own architecture.

- **The location of the split between the feature extractor and the decision maker is very impactful.** By looking at the metrics of the various experiments in Table 4.4, the size of the feature extractor and the decision maker modify consequently the results of the outcome. The overfitting score is almost `zero` with the Caffe_v1 architecture, then increases with Caffe_v2 and Caffe_v3 splits and finally decreases with the Caffe_v4 architecture. As mentioned in the above points, this has also a direct impact on the performance, such as the accuracy and the F1-score. Consequently, the location of the split, and inherently the complexity of both the feature extractor and the decision maker, should be treated as a new **hyperparameter** to fine-tune.

- **The number of heads activated during the negative training should be considered as a new hyperparameter.** In the bottom of Table 4.4, results from different number of heads activated are shown. This has, as the location of the split, a direct impact on the performance of the algorithm. Looking at the results, it seems that a higher number of activated heads leads to a higher accuracy. However, it also increases the overfitting score, and the running time since more heads are activated. Thus, the number of heads activated during the negative training part should also be fine-tuned for each experiment and each dataset, as the location of the split.

4.6 Ultra Typhon

This new Section has a similar format than the previous one. It first presents the overall results of the experiments, and then study them in order to reach some conclusion that are presented and discussed further.

4.6.1 Results

The results in this section are first presented, this time using the Ultra Typhon algorithm. The training curves are shown in Figure 4.5 while the metrics and the hyperparameters of the experiments are shown in Table 4.5. Furthermore, the impact of the number of heads activated during the negative training part of the algorithm is investigated.

	Split	Neg. heads	Acc.	F1	Overfit	Train acc.	Runtime [h]	# samples
Classic	-	-	0.62	0.62	**0.00**	0.70	**2**	$6 \cdot 10^6$
Dropout	-	-	0.49	0.47	**0.00**	0.49	**2**	$6 \cdot 10^6$
Ultra Typhon	v1	9	0.66	0.65	0.04	0.82	13	$6 \cdot 10^6$
Ultra Typhon	v2	9	**0.67**	0.66	0.04	0.83	13	$6 \cdot 10^6$
Ultra Typhon	v3	9	**0.67**	**0.67**	0.06	**0.89**	13	$6 \cdot 10^6$
Ultra Typhon	v4	9	0.66	0.66	0.01	0.86	13	$6 \cdot 10^6$
Ultra Typhon	v2	1	0.58	0.58	**0.00**	0.65	**10**	$6 \cdot 10^6$
Ultra Typhon	v2	5	0.64	0.64	**0.00**	0.76	11	$6 \cdot 10^6$
Ultra Typhon	v2	9	**0.67**	**0.66**	0.04	**0.83**	13	$6 \cdot 10^6$

TABLE 4.5: **Results of the Ultra Typhon experiments.** This Table shows the results of the experiments with Ultra Typhon, on the test set of CIFAR-10. The model to keep is selected according to the best accuracy obtained on the validation set. Note that the architecture in the classical training has no split, and dropout has a value of 0.5. All experiments used a learning rate of $5 \cdot 10^{-6}$ and a batch size of 256.

4.6.2 Discussion and analysis

By analyzing the Ultra Typhon algorithm with Figure 4.5 and Table 4.5, some consequences are:

- **Ultra Typhon has a higher sample efficiency than classical learning.** With the same number of samples seen, Ultra Typhon has both a higher accuracy and F1-score than classical learning algorithms. Moreover, the accuracy on the training set at the end of the experiment with Ultra Typhon is higher from 12% up to 19% compared to when using a classical algorithm. This means that Ultra Typhon is able to learn the useful, general patterns faster than a classical learning algorithm, resulting in a quicker convergence for this algorithm.

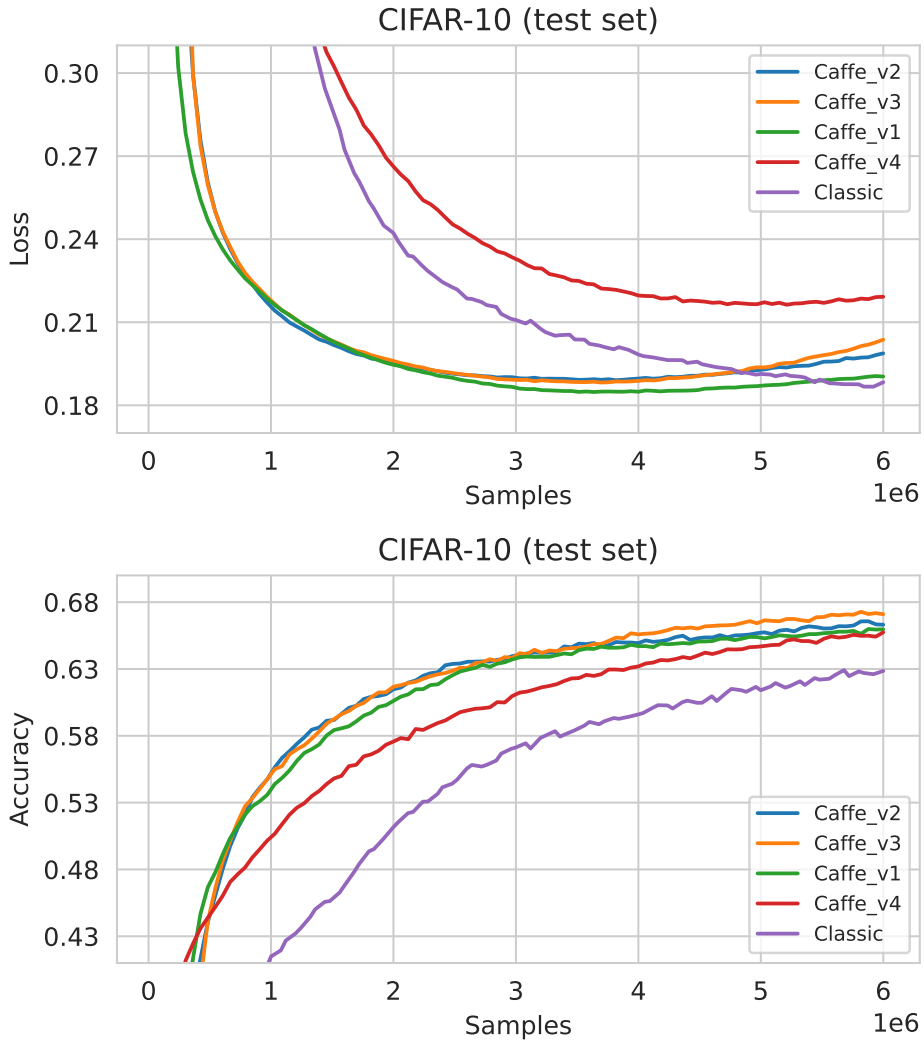


FIGURE 4.5: **Ultra Typhon training curves.** The graph on the top shows the binary cross entropy loss (see Section 1.3) on the test set of CIFAR-10 against the number of samples on which the model has been trained, and the graph on the bottom shows its accuracy. Classic refers to classical training while the other curves represent the different architectures used, with a different split between the feature extractor and the decision maker.

- **Ultra Typhon is resilient against overfitting.** In addition to have a higher sample efficiency compared to classical algorithms, Ultra Typhon is also robust against overfitting. Indeed, the algorithm is capable of reaching 82% up to 89% accuracy on the training set with almost no overfitting, as shown by the overfitting scores in Table 4.5 but also by the curves in Figure 4.5. Furthermore, this means that Ultra Typhon could potentially still learn more features from the training set that are useful for generalization, thus possibly increasing further the accuracy on the test set. This showcases that parallel transfer can be used as a tool to mitigate overfitting.

- **The position of the split between the feature extractor and the decision maker is important**, but not as crucial as in Two Level Typhon (see Table 4.4). Indeed, taking a look at Table 4.5 reveals that for the four different architectures, no one is particularly more robust, or leads to better results. All experiments using Ultra Typhon obtained an accuracy around 66% on the test set of CIFAR-10. However, this may not be the case for other architecture nor datasets. As such, the position of the split should always be treated as an hyperparameter to optimize.
- **The number of activated heads during the negative training should be considered as a new hyperparameter.** The performance and resilience against overfitting is changing depending on the number of activated heads. As shown in Table 4.5, the accuracy and the F1-score can be increased by 10% by activating more heads. However, this fact could be different for other architectures or other datasets, and could additionally lead to more overfitting. As a consequence, the number of activated heads needs to be handled as a new hyperparameter.

Chapter 5

Conclusion

This work extends the Typhon meta-learning algorithm [38] used in classification tasks, capable of training a neural network on multiple heterogeneous datasets, into to two different new single-dataset versions. The algorithms of the Typhon family are based on the parallel transfer paradigm, which bridges transfer learning [35, 36] and multi-task learning [37]. They however have the advantage to learn each task as a separate, independent target.

The first version, Two Levels Typhon, is applicable when different classes in the dataset can be grouped or aggregated into super-classes, that have common features, and train one decision maker per each super-class. Each decision maker is thus solving a multi-class classification problem. As shown in the experiments using the CIFAR-100 dataset [29], Two Levels Typhon has a higher sample efficiency compared to classical learning algorithms, implying a higher accuracy on the datasets. This is because the algorithm take advantage of parallel transfer, which can effectively transmit the knowledge learned from one decision maker to the other ones. However, Two Levels Typhon requires more hyperparameter optimization, since the position of the split between the feature extractor and the decision maker, as well as the number of heads activated during the negative training part, are new hyperparameters to fine-tune.

If super-classes are not available, Ultra Typhon splits the dataset differently, by assigning one decision maker per each class, thus each decision maker solves a one-class classification problem. Ultra Typhon has also a higher sample efficiency in comparison with classical learning algorithms. This means that Ultra Typhon performs better than classical algorithms while still being resilient against overfitting. Ultra Typhon is thus robust with datasets containing little data, as it is capable of learning features or patterns useful for generalization, where classical learning algorithm would fail.

In conclusion, the content of this work has several implications. Firstly, the original Typhon algorithm no longer requires to have multiple datasets, but can be applied on a single dataset as well, although in some applications the use of multiple heterogeneous can be still valuable. Moreover, Typhon can be used as a mitigation tool against overfitting, achieving similar results in terms of metrics and performances compared to classical learning. Typhon however has overall still a higher overfitting score than dropout, but this should be investigated further since the models with dropout are virtually smaller, thus naturally leading to less overfitting. Finally, the whole Typhon family algorithms has a higher sample efficiency compared to classical learning algorithms. This makes them promising candidates to use when having datasets with sparse or little data available, such as in the medical field.

5.1 Future work

Although this work shows some auspicious results and behaviors of the algorithms of the Typhon family in classification tasks, more work has to be pursue. This includes firstly to apply Typhon on other datasets, but also on other neural network architectures, such as transformers [27]. Indeed, the field of natural language processing is a topic largely studied, in which Typhon could potentially improve the efficiency of the learning process of large language models, such as GPT [28].

Moreover, the code-base of Typhon should be optimized further. Refactoring needs to be applied in order to improve the runtime and the memory management of Two Levels Typhon and Ultra Typhon, especially during the training phase. This can be solved by trying new libraries or more optimized functions, or proposing other, equivalent formulations of the algorithms.

Finally, other learning tasks should be explored with Typhon. For example, segmentation tasks could also suffer from data scarcity, in which Typhon is a good candidate algorithm to use. Furthermore, in auto-encoding tasks where the bottleneck is hard to train, Typhon could also potentially improve the learning process, due to the higher sample efficiency of the whole Typhon family over the classical learning algorithms.

Bibliography

[1] Brook Taylor. *Methodus incrementorum directa & inversa*. Inny, 1717.

[2] Georgi P Tolstov. *Fourier series*. Courier Corporation, 2012.

[3] Irving John Good. "Rational decisions". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 14.1 (1952), pp. 107–114.

[4] Timothy O Hodson. "Root mean square error (RMSE) or mean absolute error (MAE): When to use them or not". In: *Geoscientific Model Development Discussions* 2022 (2022), pp. 1–10.

[5] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine learning* 20 (1995), pp. 273–297.

[6] Mervyn Stone. "Cross-validatory choice and assessment of statistical predictions". In: *Journal of the royal statistical society: Series B (Methodological)* 36.2 (1974), pp. 111–133.

[7] Mervyn Stone. "An asymptotic equivalence of choice of model by cross-validation and Akaike's criterion". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 39.1 (1977), pp. 44–47.

[8] David MW Powers. "Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation". In: *arXiv preprint arXiv:2010.16061* (2020).

[9] Karimollah Hajian-Tilaki. "Receiver operating characteristic (ROC) curve analysis for medical diagnostic test evaluation". In: *Caspian journal of internal medicine* 4.2 (2013), p. 627.

[10] Lee R Dice. "Measures of the amount of ecologic association between species". In: *Ecology* 26.3 (1945), pp. 297–302.

[11] Thorvald Sorensen. "A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on Danish commons". In: *Biologiske skrifter* 5 (1948), pp. 1–34.

[12] Paul Jaccard. "Étude comparative de la distribution florale dans une portion des Alpes et des Jura". In: *Bull Soc Vaudoise Sci Nat* 37 (1901), pp. 547–579.

[13] Paul Jaccard. "The distribution of the flora in the alpine zone. 1". In: *New phytologist* 11.2 (1912), pp. 37–50.

[14] Michael S Lewis-Beck and Andrew Skalaban. "The R-squared: Some straight talk". In: *Political Analysis* 2 (1990), pp. 153–171.

[15] Davide Chicco, Matthijs J Warrens, and Giuseppe Jurman. "The coefficient of determination R-squared is more informative than SMAPE, MAE, MAPE, MSE and RMSE in regression analysis evaluation". In: *Peerj computer science* 7 (2021), e623.

[16] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[18] Yann LeCun et al. "A theoretical framework for back-propagation". In: *Proceedings of the 1988 connectionist models summer school*. Vol. 1. 1988, pp. 21–28.

[19] Yann LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (Dec. 1989). Conference Name: Neural Computation, pp. 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541.

[20] Henry J. Kelley. "Gradient Theory of Optimal Flight Paths". In: *ARS Journal* 30.10 (1960). Publisher: American Institute of Aeronautics and Astronautics _eprint: https://doi.org/10.2514/8.5282, pp. 947–954. DOI: 10.2514/8.5282. URL: https://doi.org/10.2514/8.5282 (visited on 12/30/2023).

[21] Augustin Cauchy et al. "Méthode générale pour la résolution des systemes d'équations simultanées". In: *Comp. Rend. Sci. Paris* 25.1847 (1847), pp. 536–538.

[22] Herbert Robbins and Sutton Monro. "A stochastic approximation method". In: *The annals of mathematical statistics* (1951), pp. 400–407.

[23] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. Jan. 29, 2017. DOI: 10.48550/arXiv.1412.6980. arXiv: 1412.6980[cs].

[24] R. Fletcher. *Practical Methods of Optimization*. Google-Books-ID: z3m_EAAAQBAJ. John Wiley & Sons, July 26, 2000. 470 pp. ISBN: 978-0-471-49463-8.

[25] Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 1999.

[26] Kunihiko Fukushima, Sei Miyake, and Takayuki Ito. "Neocognitron: A neural network model for a mechanism of visual pattern recognition". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.5 (Sept. 1983). Conference Name: IEEE Transactions on Systems, Man, and Cybernetics, pp. 826–834. ISSN: 2168-2909. DOI: 10.1109/TSMC.1983.6313076.

[27] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017.

[28] Tom B. Brown et al. *Language Models are Few-Shot Learners*. July 22, 2020. DOI: 10.48550/arXiv.2005.14165. arXiv: 2005.14165[cs].

[29] Alex Krizhevsky and Geoffrey Hinton. "Learning multiple layers of features from tiny images". In: (2009). Publisher: Toronto, ON, Canada.

[30] Lutz Prechelt. "Early stopping-but when?" In: *Neural Networks: Tricks of the trade*. Springer, 2002, pp. 55–69.

[31] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014). Publisher: JMLR. org, pp. 1929–1958.

[32] Federico Girosi, Michael Jones, and Tomaso Poggio. "Regularization theory and neural networks architectures". In: *Neural computation* 7.2 (1995), pp. 219–269.

[33] Mark Schmidt. "Least squares optimization with L1-norm regularization". In: *CS542B Project Report* 504.2005 (2005), pp. 195–221.

[34] Xiong Luo, Xiaohui Chang, and Xiaojuan Ban. "Regression and classification using extreme learning machine based on L1-norm and L2-norm". In: *Neurocomputing* 174 (2016), pp. 179–186.

[35] Lisa Torrey and Jude Shavlik. "Transfer Learning". In: *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*. IGI Global, 2010, pp. 242–264. ISBN: 978-1-60566-766-9. DOI: 10.4018/978-1-60566-766-9.ch011.

[36] Sinno Jialin Pan and Qiang Yang. "A Survey on Transfer Learning". In: *IEEE Transactions on Knowledge and Data Engineering* 22.10 (Oct. 2010). Conference Name: IEEE Transactions on Knowledge and Data Engineering, pp. 1345–1359. ISSN: 1558-2191. DOI: 10.1109/TKDE.2009.191.

[37] Rich Caruana. "Multitask Learning". In: *Machine Learning* 28.1 (July 1, 1997), pp. 41–75. ISSN: 1573-0565. DOI: 10.1023/A:1007379606734.

[38] Giuseppe Cuccu et al. "Typhon: Parallel Transfer on Heterogeneous Datasets for Cancer Detection in Computer-Aided Diagnosis". In: *2022 IEEE International Conference on Big Data (Big Data)*. IEEE, 2022, pp. 5223–5232.

[39] Robert A Jacobs et al. "Adaptive mixtures of local experts". In: *Neural computation* 3.1 (1991), pp. 79–87.

[40] Giuseppe Cuccu et al. "Hydra: Cancer detection leveraging multiple heads and heterogeneous datasets". In: *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2020, pp. 4842–4849.

[41] Jiyoung Lee and Giuseppe Cuccu. "P-Hydra: Bridging Transfer Learning And Multitask Learning". In: (2020).

[42] Jonas Fontana, Giuseppe Cuccu, and Philippe Cudré-Mauroux. "Improving Feature-Space Generalization Using the Typhon Framework". In: (2023).

[43] Larry M Manevitz and Malik Yousef. "One-class SVMs for document classification". In: *Journal of machine Learning research* 2.Dec (2001), pp. 139–154.

[44] Jia Ding et al. "Accurate Pulmonary Nodule Detection in Computed Tomography Images Using Deep Convolutional Neural Networks". In: *Medical Image Computing and Computer Assisted Intervention – MICCAI 2017*. Ed. by Maxime Descoteaux et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 559–567. ISBN: 978-3-319-66179-7. DOI: 10.1007/978-3-319-66179-7_64.

[45] Yutong Xie et al. "Transferable Multi-model Ensemble for Benign-Malignant Lung Nodule Classification on Chest CT". In: *Medical Image Computing and Computer Assisted Intervention – MICCAI 2017*. Ed. by Maxime Descoteaux et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 656–664. ISBN: 978-3-319-66179-7. DOI: 10.1007/978-3-319-66179-7_75.

[46] Alex Krizhevsky, Geoff Hinton, et al. "Convolutional deep belief networks on cifar-10". In: *Unpublished manuscript* 40.7 (2010), pp. 1–9.

[47] Ke Zhang et al. "Multiple feature reweight densenet for image classification". In: *IEEE access* 7 (2019), pp. 9872–9880.

[48] Yulin Wang et al. "Not all images are worth 16x16 words: Dynamic transformers for efficient image recognition". In: *Advances in neural information processing systems* 34 (2021), pp. 11960–11973.

[49] Yangqing Jia et al. "Caffe: Convolutional architecture for fast feature embedding". In: *Proceedings of the 22nd ACM international conference on Multimedia*. 2014, pp. 675–678.

[50] Kunihiko Fukushima. "Cognitron: A self-organizing multilayered neural network". In: *Biological Cybernetics* 20.3 (Sept. 1, 1975), pp. 121–136. ISSN: 1432-0770. DOI: 10.1007/BF00342633.