UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

eXascale Infolab

University of Fribourg

Master Thesis

Improving Feature-Space Generalization Using the Typhon Framework

Author:
Jonas Fontana

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

July 21, 2023

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

Acknowledgements

I owe the success of this thesis to many people who during its realization encouraged me, helped me, listened to me, gave advice, and motivated me when I needed it most. The list of these people is very long, but some of them deserve explicit thanks.

First, I thank my family immensely for always supporting and encouraging me to do what I love, not only during this thesis and throughout my studies but always in my life, allowing me to become the person I am.

Special thanks also go to two amazing people I lived with during the master's period, Alex Guardini and Camilla Bettosini. They made this period unforgettable.

I also owe thanks to Christophe Broillet. My thesis started from a work he contributed to, and during the whole time he was always available for explanations and advices.

Finally, a huge thank is for my supervisor Dr. Giuseppe Cuccu. For the incredible amount of time he devoted in following me and teaching me. But most of all for always believing in me, often more than I did myself.

Abstract

Jonas Fontana

Improving Feature-Space Generalization Using the Typhon Framework

Feature extraction is a critical component in deep learning models, often accomplished through convolutional neural networks (CNNs) stacked at the beginning of the network. However, the fusion of feature extraction and decision-making lacks clear separation, and the training of early layers presents challenges due to limited error gradients and the requirement of extensive data. This is particularly challenging in domain-specific tasks like Computer-Aided Diagnosis in medical applications, where large datasets are often not available. In this thesis, I investigate Typhon, a meta-learning framework introduced by Cuccu et al. (2022) which leverages parallel transfer learning to improve sample efficiency and enhance the generalization of feature extraction. Building upon previous work, I adapt Typhon to the task of segmentation using a breast cancer ultrasound dataset, augmented with support datasets from other breast ultrasound and brain MRI scans. Even without extensive parameter optimization, Typhon achieves a significant performance improvement of 4% in Intersection over Union (IoU) and 9% in recall compared to state-of-the-art methods. Moreover, my results demonstrate a reduction in overfitting, enabling the model training to fully converge. These findings underscore Typhon's versatility as a meta-learning framework, empowering its application to tackle contemporary challenges in deep learning tasks.

To showcase the versatility of Typhon, I also extend its capabilities to the domain of autoencoding. Specifically, I employ Typhon to encode and decode images from diverse Atari game environments, encouraging the feature extractor to learn shared characteristics across the games. I introduce a comprehensive framework that encompasses the generation of new Atari images, training of the feature extractor using Typhon, and training of a new controller that utilizes the feature extractor's output. The iteration of these components allow to progressing through different stages of the games and obtaining training images that are not available initially.

Finally, my exploration of Typhon's applications has led to the development of a novel neural network layer called PixelPerfect. This layer enables precise identification of feature coordinates within input using a minimal number of parameters. The utilization of this layer will be demonstrated in the context of identifying features in Atari images; however, its applications possibly extend to every tasks requiring precise localization, including tumor segmentation.

Keywords: Typhon, parallel transfer, overfitting, segmentation, UDIAT, breast cancer, CAD, Atari, PixelPerfect

Contents

List of Figures

Chapter 1

Introduction

In Machine Learning models, feature extraction plays a critical role in capturing meaningful representations from raw data. It involves encoding the input into a compact set of relevant features that can effectively represent the underlying patterns and characteristics. However, training a proper feature extraction is a challenging task, since it happens at the early stages of the model, where the error gradient from backpropagation is less precise. This limited gradient signal poses difficulties in updating the feature extraction process accurately, in turn requiring a large amount of data for reliable learning. To address this challenge, I propose the use of the Typhon meta-learning framework, which allows me to integrate additional datasets into the learning process, thereby enhancing the capability of the feature extractor to generalize across different domains. This ability to generalize is highly desirable in various applications, as it enables Machine Learning models to effectively apply learned knowledge to new, unseen data or scenarios.

In the first part of the thesis, I will introduce some concepts that will later be necessary to understand the experiments conducted. Section 1.1 will provide a concise explanation of how Machine Learning is employed in the medical domain to assist professionals in the diagnostic process, while Section 1.2 and Section 1.3 will present an overview of the techniques that are concretely used. Subsequently, in Section 1.4 I will introduce some additional background that will facilitate the understanding of the application of Typhon discussed in this work, related to Reinforcement Learning.

Section 1.5 will delve into the discussion of the datasets utilize and introduce how they were collected. Additionally, Section 1.6 and Section 1.7 will present relevant works that are related with this thesis. Finally, in 1.8 I will examine the goals of this work and present the main contributions.

1.1 Computer-Aided Diagnosis (CAD)

Computer-Aided Diagnosis (CAD) methods have revolutionized the field of medical imaging by seamlessly integrating algorithms and digital tools into the diagnostic workflow. These sophisticated systems offer a wide range of applications, spanning from the detection of lesions in visualization tools such as Magnetic Resonance Images (MRIs) to radiographies (X-Ray), mammographies, ultrasounds, and various other types of medical data. By leveraging CAD methods, healthcare professionals can augment their diagnostic processes with valuable support, including second opinions and analyses that may be challenging or time-consuming for doctors to perform alone.

CAD systems have become indispensable tools in modern healthcare, playing a crucial role in assisting radiologists and other specialists in the interpretation of medical images. They have proven particularly valuable in areas where precise and accurate analysis is life-critical, such as the detection of tumors, abnormalities, and other pathological conditions. Through

advanced image processing algorithms, CAD systems can automatically identify and localize suspicious regions, providing quantitative measurements and assisting in formulating diagnostic hypotheses. This not only saves time but also enhances the accuracy and consistency of diagnoses.

CAD systems have been introduced more than 40 years ago (Oakden-Rayner, 2019) and their purpose has not been to replace doctors, but rather to enhance their capabilities. The driving goal behind the development of CAD systems is to provide support to medical professionals, offering additional insights and assisting in tasks that may be impractical or beyond human capacity. CAD systems serve as valuable second opinions, leveraging their computational power and pattern recognition algorithms to analyze medical images and provide valuable information to aid in decision-making.

Consider, for example, the benefit of integrating cancer detection models in medical machines, automatically screening for the presence of a malignant mass while conducting routine examinations. Surgeons and radiologists would not have sufficient time to meticulously examine every patient for any possibility of cancer, especially when there are no initial suspicions. However, early detection of malignant masses plays a crucial role in the successful treatment of the disease. A diagnosis by a CAD system could provide a valuable hint to medical professionals, prompting them to investigate further, potentially saving lives.

CAD systems have significantly contributed to the advancement of personalized medicine by enabling tailored treatment plans based on accurate and detailed diagnostic information. By precisely delineating the location and extent of abnormalities, CAD systems assist in treatment planning, guiding surgical interventions, and facilitating targeted therapies. This level of precision not only improves patient outcomes but also minimizes unnecessary procedures and reduces the overall healthcare costs.

Supporting medical experts with CAD systems strengthens the overall healthcare ecosystem, resulting in improved patient outcomes and a higher level of care. CAD systems have become an integral part of the diagnostic process, augmenting the expertise of healthcare professionals and providing a valuable tool for decision support. As medical imaging technologies continue to evolve, those systems are positioned to advance further, incorporating Deep Learning algorithms, multimodal image fusion, and real-time analysis capabilities. Their ongoing development and integration in clinical practice hold great promise for the future, opening new avenues for improved diagnostics, personalized treatment, and ultimately, better patient care.

1.2 Techniques for tumor identification

In this section, I will introduce how tumors are typically identified in CAD tools. Specifically, I will first provide an overview of one of the most common Machine Learning task, namely classification. I will discuss its functioning and provide numerous examples of its applications. Moving forward, I will explore segmentation as a direct evolution of classification and examine its wide range of practical uses.

In the second part of this section, I will delve into the evaluation of model performance in these tasks, a crucial step in the development of increasingly effective models. I will introduce several common evaluation metrics used in the field, discussing their mechanics, practical application, and highlighting both their strengths and weaknesses.

1.2.1 Classification

Classification is a fundamental task in the field of machine learning, encompassing various domains and applications. At its core, classification involves determining the class or category, out of a limited set, to which a given input belongs. This task finds extensive use in

diverse fields, including computer vision, fraud detection, natural language processing, and the medical domain.

In computer vision, classification enables the identification and recognition of objects, scenes, or concepts present in images. It allows machines to understand and categorize visual information, leading to a wide range of applications in various domains. Prominent datasets, such as ImageNet, have significantly contributed to the advancement of classification models by providing extensive collections of images covering thousands of different categories. By analyzing the features extracted from an image, a classification model can accurately predict the object or scene depicted in it.

The applications of classification in computer vision extend far beyond simple object recognition tasks. While distinguishing between cats and dogs may seem like a trivial example, it serves as the foundation for more sophisticated tools and technologies, such as autonomous driving systems and surveillance systems. These applications heavily rely on the pre-processig of visual information in the scene, leaging to the accurate classification of objects and scenes in real-time, enabling intelligent decision-making and ensuring safety and security.

Beyond these high-profile applications, there are relevant applications in numerous day-to-day activities. For instance, machine learning models can be trained to identify the type of a mushroom from an image, helping users determine whether it is poisonous or edible. Classification algorithms can also be utilized in image-based product searches, where users can find a specific product simply by capturing its picture. In agriculture, classification models can assist in diagnosing and treating sick plants by analyzing visual cues and providing recommendations for appropriate remedies. These examples highlight the vast potential of classification in computer vision, with applications spanning various industries and domains. Figure 1.1 shows an example of classification.

Beyond computer vision, classification plays a crucial role in a myriad of real-world applications across various domains. One such application is fraud detection, where classification models can be trained to differentiate between legitimate transactions and fraudulent ones. By analyzing patterns and identifying suspicious activities, these models contribute to the detection and prevention of financial crimes, safeguarding individuals and organizations from potential threats.

In the realm of natural language processing (NLP), classification techniques find extensive use in classifying text documents based on their content. For example, email providers leverage classification algorithms to automatically filter out spam emails, ensuring that users' inboxes are not inundated with unwanted messages.

Sentiment analysis is another area where classification techniques are used. By assigning sentiment labels to text, such as positive, negative, or neutral, models can gauge the overall sentiment expressed in a piece of writing. This has wide-ranging applications, from analyzing customer reviews and feedback to understanding public opinion on social media platforms. The ability to classify sentiment provides valuable insights for businesses, policymakers, and researchers, enabling informed decisions based on public sentiment.

The medical domain stands as a critical area where classification techniques play a vital role in improving patient care and medical decision-making. In this context, classification models are trained to analyze various medical inputs, ranging from patient records and medical images to sensor data, and classify them as indicative of a healthy or an unhealthy condition. Machine Learning models enable the development of diagnostic tools that aid medical professionals in identifying potential diseases or conditions at an early stage, facilitating timely interventions and personalized treatment plans.

One prominent example within the medical field is the utilization of Computer-Aided Diagnosis (CAD) systems (see Section 1.1) which employ algorithms to analyze medical images and classify them as originating from either a healthy patient or a patient with health issues.

Furthermore, classification models find applications in a wide range of other medical tasks, including disease prediction, risk assessment, and treatment planning. By analyzing patient data, such as genetic information, biomarkers, and clinical measurements, classification algorithms can identify patterns and risk factors associated with specific diseases. This information can then be used to predict the likelihood of developing certain conditions, enabling proactive interventions and preventive measures.

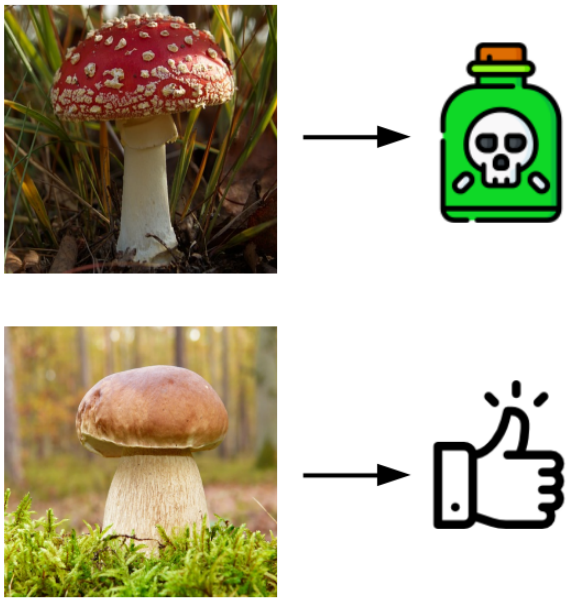


Figure 1.1: **Classification.** In a classification task, the model outputs a single label for the entire image (in this case, poisonous or edible).

1.2.2 Segmentation

Segmentation is a fundamental task in the field of computer vision which involves the partitioning of an image into coherent and meaningful regions, enabling a comprehensive understanding of visual data through the assignment of labels to individual pixels. This process can be considered an extension of classification, where the model is tasked with classifying each pixel individually, rather than the entire image as a whole. The significance of segmentation is evident in its wide range of applications across various domains, including autonomous driving, image/video editing, augmented reality, and medical image analysis.

Traditionally, segmentation methods relied on techniques such as thresholding and region growing (Liu, Deng, and Yang (2019)), which were limited in their ability to capture intricate visual patterns. However, recent advancements in the field have predominantly embraced deep convolutional neural networks (CNNs) (see Section 1.3.1) as the preferred approach for addressing segmentation challenges. By leveraging the power of Deep Learning (see section 1.3.2), these modern techniques have demonstrated exceptional performance by learning from large-scale annotated datasets. The growing availability of such datasets has allowed these models to effectively capture complex visual patterns, resulting in highly accurate segmentations.

The versatility of segmentation is evidenced by its ubiquity across diverse domains. In the context of autonomous driving, segmentation is utilized to identify and delineate various objects and regions within a scene, enabling vehicles to make informed decisions and navigate safely. In image and video editing, segmentation facilitates precise selection and manipulation of specific objects or regions, allowing for advanced editing techniques and creative effects. Augmented reality applications heavily rely on segmentation to overlay virtual objects seamlessly into real-world environments, enhancing the user experience. In the medical field, segmentation plays a critical role in analyzing medical images, aiding in the identification and delineation of anatomical structures or pathological regions for diagnosis and treatment planning.

Figure 1.2: **Segmentation.** In the left image, a model recognizes (segments) the pixels corresponding to a dog. All other pixels are classified as "not dog". In the right image, a multiclass segmentation model identifies the different objects in a kitchen (source: Carion et al. (2020)).

Evaluation Metrics

The evaluation of trained models is a critical step in assessing their performance and determining their effectiveness. While visual inspection of model outputs can provide some initial insights, it lacks the scientific rigor and scalability necessary for comprehensive evaluation. To address these challenges, the field of Supervised Learning relies on the use of metrics to quantitatively assess model performance on a large scale.

In the context of image segmentation, metrics play a pivotal role in evaluating the model's ability to accurately identify objects and regions within an image. These metrics provide objective and standardized measures of performance that can be applied consistently across different datasets and models. By withholding a portion of the dataset during training and computing metrics on this held-out set, the model's generalization and predictive capabilities can be rigorously evaluated.

In the following sections, I will delve into the specific metrics used for evaluating segmentation models, discussing their strengths, limitations, and interpretations.

Accuracy. Accuracy is a widely used metric in segmentation evaluation that measures the overall correctness of the model's predictions. It quantifies the proportion of correctly classified pixels or regions out of the total number of pixels or regions in the image. A high accuracy value indicates that the model is making correct predictions for a majority of the pixels or regions, reflecting its ability to accurately segment and classify the objects of interest. Conversely, a low accuracy value suggests that the model is struggling to correctly classify a significant portion of the image, indicating potential errors and misclassifications.

In practical terms, accuracy serves as a general indicator of the model's ability to capture the underlying patterns and structures within the image. It provides a comprehensive measure of how well the segmentation model is able to accurately delineate objects or regions

of interest. However, it is important to note that accuracy alone may not provide a complete understanding of the model's performance, especially in scenarios with imbalanced class distributions or when the cost of false positives or false negatives is not equal. In such cases, it is necessary to consider additional metrics to gain more nuanced insights into the model's behavior.

Accuracy is computed as:

$$\text{Accuracy} = \frac{\text{True Positive} + \text{True Negative}}{\text{True Positive} + \text{False Positive} + \text{True Negative} + \text{False Negative}}$$

Precision. Precision is a fundamental segmentation metric that provides insights into the model's ability to accurately identify and classify positive instances within an image. It measures the proportion of true positive predictions, which are correctly identified positive instances, out of all positive predictions made by the model. A high precision value indicates that the model has a low rate of false positives, meaning that it correctly identifies and classifies positive regions while minimizing incorrect identifications. On the other hand, a low precision value suggests a higher likelihood of false positives, indicating that the model may incorrectly identify regions as positive even when they are not.

In practical terms, precision is particularly important in applications where the cost of false positive errors is high. For instance, in the context of predicting the risk of recidivism, a false positive has the potential to devastate the life of an innocent person. While precision is a valuable metric for evaluating segmentation models, it should be used with caution and in conjunction with other metrics. Relying solely on precision can be misleading in scenarios where false negatives (missed detections) have severe consequences or when class imbalances exist, such as in medical imaging where a high precision may indicate a low false positive rate but could overlook critical cases with false negatives.

The precision is calculated as:

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

Recall. Recall, also known as sensitivity or true positive rate, assesses the ability of the model to correctly identify the presence of a target class by capturing the proportion of true positive predictions out of all actual positive instances in the data. A high recall value indicates that the model effectively detects the target class and minimizes false negatives, ensuring that fewer positive instances are overlooked.

Recall plays a crucial role in applications where the consequences of false negatives are significant. For instance, in Computer-Aided Diagnosis, missing the detection of abnormalities or diseases can have severe implications for patient care and treatment decisions. A high recall value in this context indicates that the model successfully identifies potential regions of interest, helping healthcare professionals focus their attention on areas that require further investigation or intervention. However, similarly as for the previous metrics, relying solely on recall as a performance metric can be problematic in certain scenarios. One limitation is when the cost of false positives (incorrectly classifying a negative instance as positive) is high. For instance, in security screening applications, a high recall might lead to an excessive number of false positives, causing inconvenience or delays for individuals. Therefore, a trade-off between recall and precision needs to be carefully considered to strike a balance between minimizing false negatives and false positives.

The formula for recall is given by:

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

Specificity. Specificity, also known as the true negative rate, complements recall in segmentation evaluation by measuring the model's ability to accurately identify negative instances. It quantifies the proportion of true negative predictions out of all actual negative instances in the data. A high specificity value indicates that the model effectively avoids false positive errors, ensuring that negative instances are correctly classified as such.

In various applications, specificity holds particular importance when the consequences of false positives are significant. However, it is essential to consider the limitations of relying solely on specificity as the sole performance metric. While high specificity ensures a low rate of false positives, it may come at the cost of increased false negatives. In situations where the detection of positive instances is critical, such as in medical diagnostics or anomaly detection, a high specificity value may lead to the omission of important regions or objects of interest.

We compute specificity as:

$$\text{Specificity} = \frac{\text{True Negative}}{\text{True Negative} + \text{False Positive}}$$

AUC (ROC). The Area under the Curve (AUC), intended as the Receiver Operating Characteristic (ROC) curve, is a commonly used metric for evaluating the performance of segmentation models. The ROC curve plots the true positive rate (sensitivity) against the false positive rate (1 - specificity) at various threshold values, and the AUC quantifies the overall performance of the model across all possible threshold values. AUC values range in [0.0, 1.0], with higher value indicating better performance. An AUC of 1.0 represents a perfect model that achieves a 100% true positive rate with a 0% false positive rate, while an AUC of 0.5 indicates a model that performs comparably to random guessing. Lower values correspond to even lower performance.

AUC is particularly important in applications where the balance between sensitivity and specificity is crucial, where accurately identifying positive instances while minimizing false positives is vital. By utilizing the AUC metric, for example, medical professionals can assess the overall discriminatory power of the segmentation model, determining its ability to differentiate between regions or objects of interest and background.

While being more informative than other metrics, also AUC has some limitations. In particular, it assumes that the cost of false positives and false negatives are equal. In cases where the cost of misclassification varies significantly, AUC alone may not provide a comprehensive evaluation. Therefore, it is usually recommended to interpret the AUC value in conjunction with other metrics such as sensitivity, specificity, precision, and recall to gain a more nuanced understanding of the model's applicability and make informed decisions based on the specific requirements of the application domain. Figure 1.3 provides a visual example of the AUC computation.

Dice Score. The Dice Score, also known as F1 score, is a widely used metric in segmentation tasks that combines the concepts of precision and recall into a single measure. It quantifies the similarity between the predicted and ground truth segmentations, providing an overall

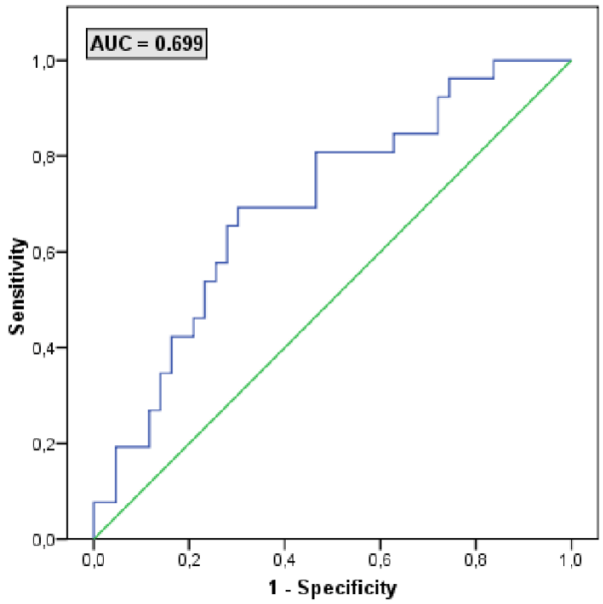


Figure 1.3: **Area Under the Curve.** The figure provides a visual intuition of how AUC is computed, by considering the area under the plot given by the sensitivity at the variation of the specificity. Source: Langhammer (2018).

assessment of segmentation accuracy. The Dice Score is calculated by taking twice the intersection between the predicted and ground truth segmentations, and dividing it by the sum of the predicted and ground truth segmentations. This formulation allows the Dice Score to highlight both false positive and false negative errors, making it a valuable metric for evaluating segmentation models.

The values of the Dice Score range in [0.0, 1.0], where a value of 1.0 indicates a perfect overlap between the predicted and ground truth segmentations, while a value of 0.0 indicates no overlap at all. It is a commonly used metric in medical image analysis, where precise segmentation of structures or regions of interest is critical for accurate diagnosis and potentially invasive procedures such as biopsies.

This metric is mathematically equivalent to the harmonic mean of precision and recall, and thus called F1, where the more generic F_β score applies additional weights, favoring one of precision or recall more than the other. The harmonic mean provides a balanced measure that considers both precision and recall, making it an ideal choice for evaluating segmentation performance.

The Dice Score can be computed as:

$$\text{Dice Score} = \frac{2 \times \text{True Positive}}{(2 \times \text{True Positive}) + \text{False Positive} + \text{False Negative}}$$

Intersection over Union. The Intersection over Union (IoU), also known as Jaccard index, is a widely used metric in segmentation tasks that measures the spatial alignment and accuracy of the predicted segmentation compared to the ground truth. The IoU is closely related to the Dice score, as both metrics assess the similarity between the predicted and ground truth segmentations. Similar to the Dice Score, the IoU calculates the overlap between the predicted

and ground truth segmentations (here called Intersection), but with a slightly different formulation, by normalizing it over the regions union. It quantifies the degree of overlap between the predicted and ground truth segmentations, providing a measure of the segmentation's quality.

The metric can assume values ranging in [0.0, 1.0], where a score of 1.0 indicates a perfect overlap between the predicted and ground truth segmentations, while a score of 0.0 represents no overlap at all. It is commonly used in various applications, such as object detection, semantic segmentation, and medical image analysis. In medical imaging, the IoU metric plays a crucial role in assessing the accuracy of anatomical structure segmentations. Accurate delineation of structures, such as tumors or organs, is essential for diagnosis, treatment planning, and disease monitoring.

The IoU is computed as the ratio of the intersection to the union of the predicted and ground truth regions:

$$\text{IoU} = \frac{\text{Intersection}}{\text{Union}} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive} + \text{False Negative}}$$

1.3 Models for segmentation tasks

In this section, I will introduce the topic of segmentation architectures, focusing on concrete machine learning models specifically designed for image segmentation tasks. The foundation of these models lies in neural networks (NN), which have become a cornerstone in the field of computer science and Machine Learning. Neural networks offer a versatile and powerful framework for addressing complex problems, making them widely adopted and highly effective in various domains.

One of the significant advancements in the field of Machine Learning has been Deep Learning (DL), an approach that harnesses the potential of large-scale neural networks using an enourmous amount of data for training. Over the past decade, Deep Learning has revolutionized the landscape of Machine Learning, unleashing remarkable progress in numerous domains. From language translation and image recognition to self-driving cars, digital assistants, and Large Language Models (LLMs), Deep Learning has played a pivotal role in most recent results, as showcased next.

After showing the importance of Deep Learning, I will shift the focus to state-of-the-art DL architectures that have emerged for medical image segmentation. These architectures have demonstrated exceptional performance in accurately segmenting medical images, providing valuable insights for diagnosis, treatment planning, and disease monitoring. Throughout this section, I will explore these architectures and discuss their applicability and relevance to my work.

1.3.1 Neural Networks

The concept of neural networks originated in 1943 with the work of (McCulloch and Pitts, 1943), who explored the functioning of neurons and devised a simple neural network using electrical circuits. Inspired by the human brain, neural networks are composed of a directed graph of densely interconnected processing nodes. In the simplest models, these nodes are typically organized into layers, forming a feed-forward structure where data flows in a unidirectional manner. For this reasons these models are called sequential models.

In a sequential model, the flow of information is processed in a sequence, where the output of one layer serves as the input to the next layer. This sequential nature allows neural networks to process and transform data through successive layers of computation, exploiting

function composition to grow their complexity fast. Each layer of nodes in the network performs a linear combination of the inputs it receives, multiplying them by the corresponding weights. This weighted sum is then passed through an activation function, which introduces non-linearity to the network. The activation function is a critical component that enables neural networks to capture complex patterns and relationships in the data.

One of the key properties of neural networks is their ability to perform function composition. Through the non-linear activation functions, neural networks are capable of combining simple functions at each layers to construct more complex functions. This process allows them to model intricate patterns and make higher-level representations of the data. By stacking multiple layers, a neural network can capture increasingly abstract and sophisticated features, enabling it to learn and generalize from complex datasets. Figure 1.4 shows an example of a simple neural network.

Furthermore, neural networks are universal function approximators. This means that given sufficient complexity in terms of the number of layers and nodes, a neural network can approximate any function. This remarkable property demonstrates the expressive power and flexibility of neural networks, although complex functions can require an extremely big network. This is the reason that brought to the establishment of Deep Learning.

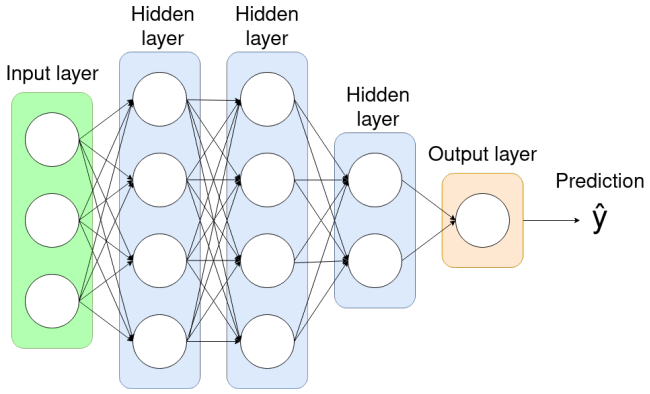


Figure 1.4: **Neural Netwok.** An example of a very simple neural network consisting on an input layer, three hidden layers and an output layer.

1.3.2 Deep Learning

The concept of Deep Learning was formally introduced by Ian Goodfellow, Yoshua Bengio, and Aaron Courville in 2016 (Goodfellow, Bengio, and Courville, 2016), although prior work had already laid the foundation in preceding years. Deep Learning sis a ML technique where numerous layers of artificial neurons are interconnected to form a large, deep neural network. Leveraging the compositional nature of neural networks, Deep Learning rapidly raises the complexity of approximated functions. As a result, remarkable achievements have been attained in diverse domains such as machine translation, autonomous driving, and image recognition (Barrault et al., 2019; Yurtsever et al., 2020; Dabre, Chu, and Kunchukuttan, 2020). Furthermore, Deep Learning has showcased impressive performance in time series forecasts, text recognition, natural language processing (NLP), game playing, and medical diagnosis (Masini, Medeiros, and Mendes, 2020; Chen et al., 2021; Wang, Babenko, and Belongie, 2011; Hirschberg and Manning, 2015; Silver et al., 2017a; Silver et al., 2017b; Bakator and Radosav, 2018).

Deep Learning models are typically trained using the Backpropagation algorithm, employing stochastic gradient descent (sgd):

$$w \leftarrow w - \eta \cdot \nabla wE(w)$$

where the network weights w are updated using the derivative of the error functin $\nabla wE(w)$, multiplied by an error rate η.

During training, the output of the model is compared with the desired output (the label), and the error is computed. This error is then utilized to adjust the weights of the last layer, aiming to align the subsequent output for the same input more closely with the label. The derivative of the layer's function guides the direction of weight adjustments to improve the output. Once the weights of the last layer are adapted, the error propagates backward through the model, sequentially adjusting each layer using the same approach. It is noteworthy that the precision and magnitude of the backpropagated error progressively diminishes as it traverses the network. Consequently, training the initial layers of a deep network becomes more challenging, underscoring the greater difficulty in training deeper networks compared to shallow ones.

Alongside the fully connected layer, several others layers have significantly contributed to the efficacy of Deep Learning. Here I present some of the most common ones:

The **convolutional layer** is widely employed, particularly in image-related tasks. This layer utilizes a small mask (or kernel) with the same dimensionality as the input (e.g. 2D on an image), which is slid across the input. At each step, the input values are multiplied by their corresponding kernel values, and the results are aggregated via summation to generate the output value. For instance, a 3x3 mask with zeros except for a 1 at the center corresponds to the identity function, preserving the input. Conversely, a 3x3 mask with uniform weights of 1/9 performs an average of the inputs, resulting in a blurred image. The difference with a standard convolution is that these weights are learned by the model, instead of being hardcoded. Figure 1.5a illustrates an example of a convolution.

In a **recurrent layer**, connections between nodes can form cycles, enabling outputs from certain nodes to be used as subsequent inputs at the next activation. This simple yet effective mechanism allows the network to process related inputs, establishing a rudimentary form of memory. Recurrent neural networks (RNNs) are particularly valuable for tasks involving temporal dynamics, such as time series analysis and speech/text recognition. In Figure 1.5b, the red arrow illustrates a recurrent connection.

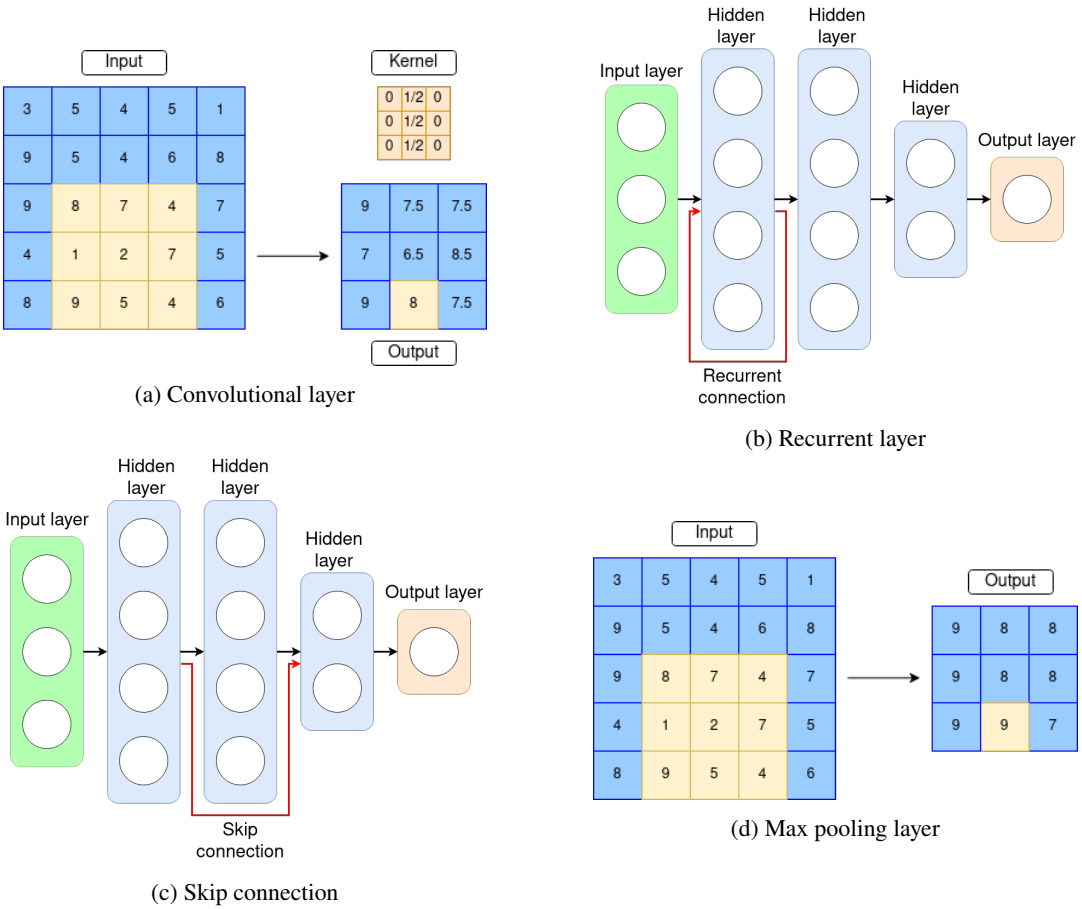
Skip connections have emerged as a prominent component in Deep Learning. Instead of forwarding the output of layer n solely to layer $n+1$, skip connections transmit it to layers further ahead, such as $n+2$ or even $n+3$, $n+4$, and so on. This approach offers two significant advantages. Firstly, during the forward pass, low-level features are propagated and can be used alongside subsequent higher-level representations. Secondly, during backpropagation, skip connections establish shorter paths from the last layer to earlier layers, usually resulting in larger and more precise error gradients. This facilitates the training of initial layers, enhancing the overall training process. In Figure 1.5c, the red arrow illustrates a recurrent connection going from the first hidden layer to the third hidden layer.

Pooling layers represent another common component often used in conjunction with convolutional layers. Pooling involves selecting a window on the input (similar to convolution) and reducing it to a single value. One popular example is max pooling, which retains only the maximum value within the selected interval. Given the convolution's ability to identify

specific patterns at various points in the input, max pooling is frequently applied after convolutions to retain the most pertinent features and discard non-maximal values, promoting the preservation of features identified with higher confidence. Figure 1.5d illustrates an example of a max pooling operation.

Batch normalization is a technique introduced by (Ioffe and Szegedy, 2015). It involves normalizing the output of the previous layer by subtracting the batch mean and dividing by the batch standard deviation. This procedure is designed to reduce the effect of covariance shift, where the distribution of the data changes during training. It has been shown to result in more efficient training and faster convergence. However, recent studies by (Santurkar et al., 2018) argue that the benefits of batch normalization are actually attributed to the smoothing effect it has on the objective function. Despite the differing interpretations, the overall outcome remains improved training performance and accelerated convergence.

Dropout is a regularization technique that was introduced by Srivastava et al. (2014) to address overfitting in neural networks. The approach involves randomly disabling (or "dropping out") certain neurons during training, encouraging the model to learn more robust and redundant representations. By reducing the reliance on individual neurons and preventing them from overfitting to specific inputs, dropout enhances the model's generalization abilities and consequently improves performance on unseen data.



(a) Convolutional layer

(b) Recurrent layer

(c) Skip connection

(d) Max pooling layer

Figure 1.5: **Deep learning layers.** Example of different elements commonly used in deep learning models. Figure (a) presents an example of a 3×3 convolution on a 5×5 input. In Figure (b), the red arrow indicates a recurrent connection. Similarly, in Figure (c) the red arrow indicates a skip connection. Finally, Figure (d) shows a 3×3 max pooling on a 5×5 input.

UNET

UNET is a highly popular Deep Learning architecture specifically designed for image segmentation tasks, originally introduced in Ronneberger, Fischer, and Brox (2015). This architecture, which gained significant attention in the Deep Learning community since its inception, is characterized by its distinctive U-shaped network structure. The U-shaped design stems from the architectural layout, where the input image is progressively encoded into higher-level features through a "contracting path" composed of convolutional and pooling layers. This encoding process allows the network encode the input into increasingly higher features. Subsequently, the symmetric decoding path employs upsampling and convolutional layers to restore the feature maps back to the original input size. A crucial aspect of UNET is the utilization of skip connections, which involve concatenating the feature maps from the contracting path with the upsampled feature maps in the decoding path. This mechanism enables the network to retain both local and global information throughout the segmentation process. UNET produces pixel-wise predictions, indicating the presence or absence of specific classes at each corresponding point in the input image. By leveraging its unique architecture and skip connections, UNET has demonstrated remarkable performance in various segmentation tasks, making it one of the principal choice for identifying objects in images. Figure 1.6 shows the architecture of UNET as it was presented in the original paper.

ResNet

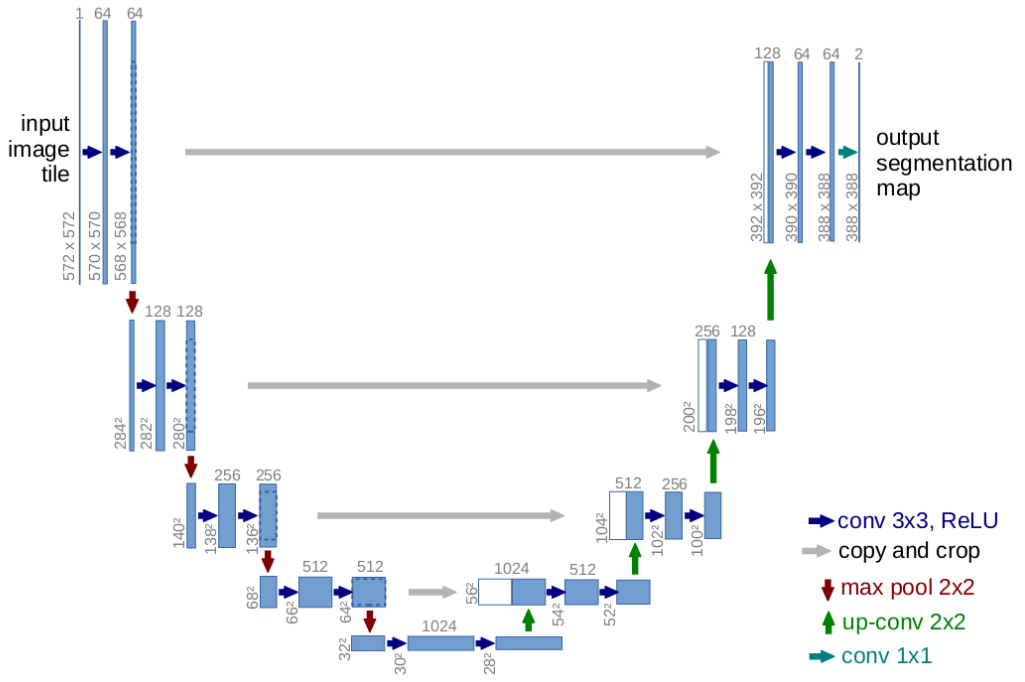


Figure 1.6: **UNET Architecture.** Original UNET architecture presented in Ronneberger, Fischer, and Brox (2015). On the left side, a sequence of encoding blocks composed of convolutional layers reduce the input shape. On the right side, the procedure is inverted through a sequence of decoding blocks until the initial shape is reconstructed. Skip connection connects the two parts, facilitating the propagation of earlier features to later decision layers.

In 2016, the paper "Deep Residual Learning for Image Recognition" (He et al., 2015) addressed the degradation problem encountered in deep neural networks. Despite the potential

of deeper architectures to capture increasingly complex features, it was observed that as network depth increased, performance would plateau and eventually degrade. This phenomenon challenged the conventional notion that deeper networks always lead to improved results.

To overcome this limitation, the authors initially reasoned that adding more layers to a network should be beneficial, as a deeper network could simply propagate the values through identity functions in the added layers, and the training error would be no greater than the one of its shallow counterpart. However, experimental results showed that as the network grew deeper, its performance deteriorated, leading to the degradation problem.

The authors proposed a novel approach based on residual learning, introducing the concept of residual blocks that facilitated the optimization of residual mappings within the network. By enabling the network to focus on learning the difference (or residual) between the input and the target output, rather than attempting to learn the entire output, they showed that deeper models could be effectively trained.

The authors therefore introduced skip connections, known as identity mappings, at regular intervals within the network architecture. These connections allowed the model to learn the difference or residual between the input and the desired output, which was found to be easier to learn. It is important to notice that these additional connections did not increase the number of parameters, nor the complexity of the training, except for the negligible element-wise addition.

The new architecture, called ResNet, revolutionized the field of Deep Learning. The versions presented in the paper demonstrated remarkable scalability, allowing for the construction and training of networks with an unprecedented depth of up to 152 layers, which surpassed by eight times the depth of the state-of-the-art VGG architecture (He et al., 2015). ResNet achieved outstanding performance on various benchmark datasets, including a remarkable 3.57% error rate on the ImageNet test set, which lead to the first place in the ILSVRC 2015 classification task (He et al., 2015). Furthermore, ResNet exhibited a relative improvement of 28% on the COCO object detection dataset, which the authors attributed solely to the employment of their extremely deep representations. The ResNet architecture also achieved top rankings in the ImageNet localization, COCO detection, and COCO segmentation challenges.

Overall, "Deep Residual Learning for Image Recognition" proposed five variants of ResNet (based on the number of layers): ResNet18, ResNet34, ResNet50, ResNet101, and ResNet152.

Figure 1.7 shows the structure of the ResNet34 architecture.

1.4 Additional background

At the end of Section 2, an additional application of the Typhon framework will be presented, demonstrating its versatility and emphasizing its role as a generic meta-learning framework that can enhance the training process of any existing model. In particular, I will focus on improving the generalization capabilities of the feature extractor component applied to Atari game images. To provide the necessary background for this work, the following sections will introduce the concept of Reinforcement Learning (RL) (Section 1.4.1), discuss a range of algorithms applicable in the RL paradigm (Section 1.4.2), and present a benchmark environment for evaluation (Section 1.4.4).

1.4.1 Reinforcement Learning

Along with Supervised and Unsupervised Learning, Reinforcement Learning is one of the three main paradigms of Machine Learnings. It deals with developing a controller in situations in which there is not a correct behaviour (or is not known), but positive or negative rewards

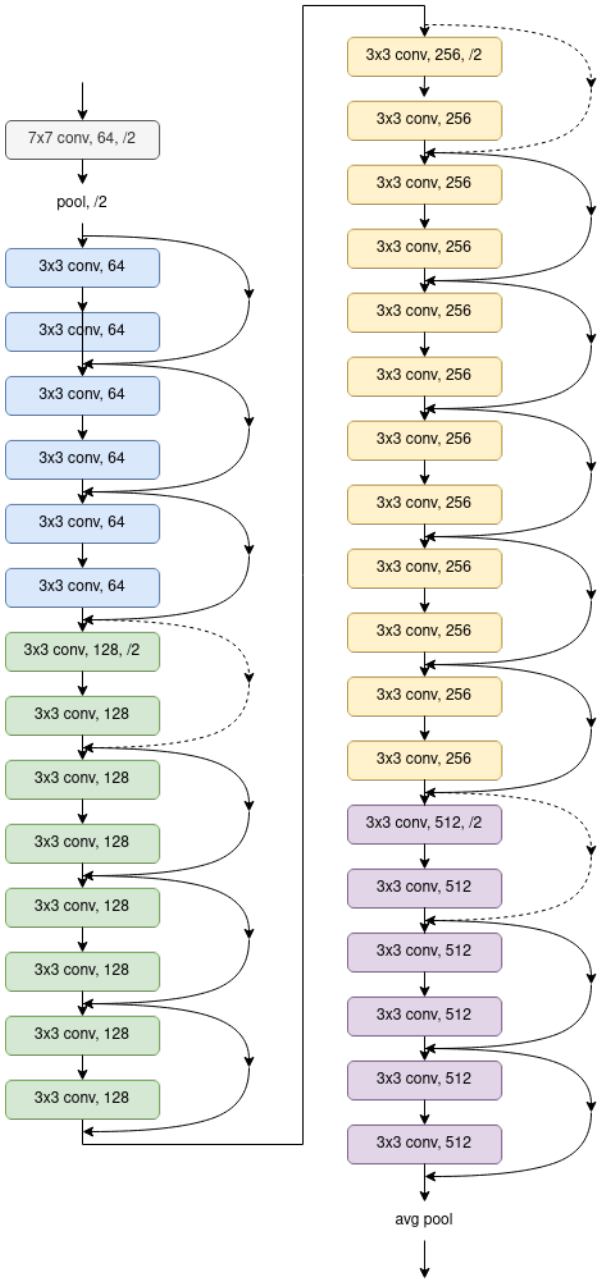


Figure 1.7: **ResNet34 Architecture.** ResNet34 architecture presented in (He et al., 2015). For each convolutional block, I indicate the size of the kernel, the number of output channels and, if different than one, the stride. The network consists mainly in four group of convolutional blocks (indicated with different colors). After each group, the shape of the input is reduced with a convolution with a stride of two, while the number of channels is doubled. There is a skip connection every two layers which propagates the input and allow the network to learn only the *residual*, i.e. the changement from the previous layer. The dotted skip connections indicates the situation in which the connection should be handled with case, since the output should have a different shape than the input.

are received after each action. RL is inspired by how humans (and animals in general) learn. We explore, try, get hurt, receive rewards, and from this decide which behaviours are better and which ones in contrasts are to avoid. Consider toddlers learning to walk: at the beginning

they fail all the time. However, they gradually start to notice that with this or that movement they are able to balance the body and still standing. After countless further trials (and falls), they start moving one feet, then another, and so on until the point they can walk.

Every time we need a model (or agent, as it is called in RL) to develop a behaviour rather than learning an existing one (like in Supervised Learning), we can apply the RL paradigm. Another classical example are videogames: we don't have an example of a perfect controller, but on the other hand is very simple to evaluate a controller's performance (with scores, distance reached in the game, coins collected, time alive, ...). We can therefore simulate multiple games, and give rewards/penalties to each agent to drive its improvement until it reaches a satisfiable performance.

The RL paradigm can be conceptualized as follows: the *agent*, typically guided by an algorithm, operates within the *environment* by selecting one of the available *actions*. The *environment* responds to the chosen *action* and undergoes an update. Following this update, the *environment* provides the *agent* with its new state (often referred to as an *observation*), along with a corresponding *reward* or *penalty*. The objective of the *agent* is to maximize the cumulative reward by determining the optimal strategy (often called *policy*) for action selection. Figure 1.8 shows this procedure.

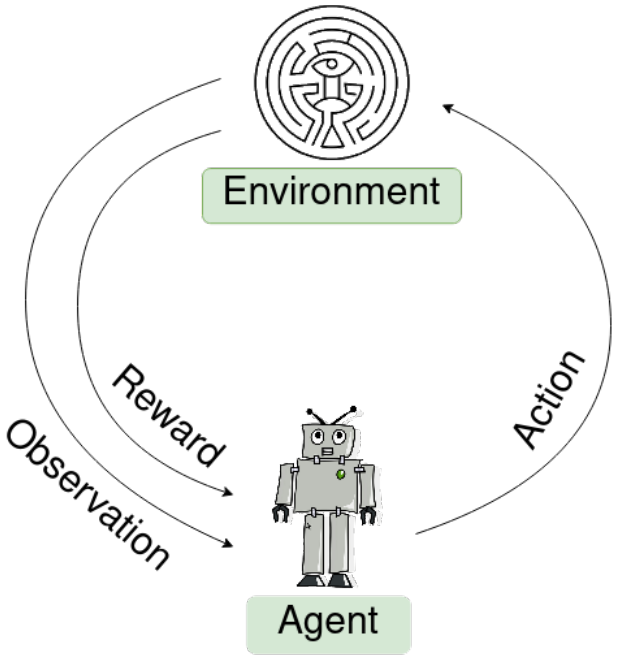


Figure 1.8: **Reinforcement Learning framework.** The agent interacts with the environment by selecting an action. The environment processes the action and provides feedback in the form of a new state (observation) and a reward or penalty.

1.4.2 Evolutionary Algorithms and CMA-ES

Evolutionary Algorithms (EA) belong to the class of Black Box Optimization (BBO) methods and draw inspiration from Darwinian evolution. The fundamental concept revolves around maintaining a population of multiple individuals. In each iteration, the population evolves through inheritance, selection, and variation, with the ultimate aim of enhancing the individuals and attaining the best possible outcomes.

Concretely, each individual within the population represents the parameters of a model. These models are evaluated, yielding a score that serves as the fitness measure for each individual. During each iteration, new individuals are generated by replicating those with the highest fitness, combining certain individuals, and/or introducing random variations.

Covariance Matrix Adaptation Evolution Strategy (CMA-ES) is a member of the Evolution Strategies family, a type of Evolutionary Algorithms. Like other algorithms within its family, CMA-ES draws inspiration from the natural selection process to optimize solutions for specific problems.

CMA-ES manages a population of individuals, represented as vectors, which serve as candidate solutions. The population is evolved based on the fitness of the individuals, where those with higher scores in the objective function have a stronger influence on the generation of offspring.

The central concept behind CMA-ES, as its name suggests, revolves around the maintenance and evolution of a covariance matrix (Hansen and Ostermeier, 1996). This matrix captures the covariances between each parameter of the candidate solutions (the values in the vector). Technically, the covariance matrix represents the second-order moments of the candidate solutions, providing insights into the relationships among different variables within the search space. By analyzing these covariances, CMA-ES can estimate the structure and shape of the fitness function, helping to determine the direction and magnitude of the next search steps.

1.4.3 Autoencoders and Variational Autoencoders

Autoencoders are a family of ML models which deals with encoding a given input into a smaller dimension (sometime called latent space or compressed feature space), and then reconstructing it. They are usually composed of a fist part called "encoder", responsible of compressing the data, and a second part called "decoder", which handles the reconstruction of the original data starting from the compresed one. Autoencoders are commonly used with images, however they can be used with any type of data. The main application of autoencoders is data compression: their ability of lowering the data size and later reconstructing it (with or without information loss) can be used to store data in a compressed form.

To reduce the size maintaining all (or most of) the information, the autoencoder must learn to identify what is relevant and what not. The compressed data will therefore contain only the information required to reconstruct the input to arbitrary precision, and the encoder thus works as a feature extractor.

In a classical autoencoders, no attention is devoted on how the latent space is constructed. In contrast, Variational Autoencoders (VAEs) ensures that the feature space is constructed in such a way that similar inputs in the original space will be close in the latent space as well, while different inputs will be distant. This enables using autoencoders for data generation. If the latent space is meaningful, we can sample points from this space and pass them to the decoder, obtaining brand new data points (for instance, images). Or we can pass two inputs to the encoder, get their representation on the latent space, compute the middle point of these representations, and pass this latter to the encoder to have an image which is an interpolation between the original two.

1.4.4 Arcade Learning Environment (ALE)

The Arcade Learning Environment (ALE) is a component of the OpenAI Gym framework, developed by OpenAI (Brockman et al., 2016). It provides as a standardized interface that facilitates interaction with a vast collection of classic Atari 2600 video games, enabling the

development and testing of reinforcement learning algorithms. Similar to the other environments in the Gym framework, ALE provides the agent with essential information, including the current state, rewards obtained, and additional relevant details. The agent can then select the next action to be executed, and subsequently receive a new set of information in response. ALE has emerged as a widely adopted benchmark for assessing the performance and capabilities of various reinforcement learning algorithms.

1.5 Datasets

In order to comprehensively evaluate the capabilities of the Typhon framework, I extended its applications to tackle the more challenging task of segmentation, specifically focusing on the recognition of malignant tumor masses, following the work of RF-Net (Wang, Liang, and Zhang, 2021). To conduct my evaluation, I utilized the UDIAT dataset, which comprises a collection of breast ultrasound images, as the primary source of data for my experiments.

However, to effectively leverage transfer learning within the Typhon framework, the incorporation of additional support datasets is crucial. These datasets encompass diverse image formats and even different body locations. While the specific features of these datasets may differ, they possess comparable visual features relevant to my segmentation task, such as the density-gradient of body tissues typical of medical imaging techniques.

In this section I will provide a detailed description of each dataset, including the specific characteristics, acquisition protocols, and past usage.

(i) The **UDIAT** dataset provides a collection of breast ultrasound images specifically focused on lesions. This dataset comprises a total of 163 breast ultrasound images, with 110 cases representing benign lesions and 53 cases corresponding to malignant lesions. The UDIAT Diagnostic Center of the Parc Taulí Corporation in Sabadell, Spain, was responsible for the acquisition and segmentation of these images.

The breast ultrasound images in the UDIAT dataset were captured in 2012 using a Siemens ACUSON Sequoia C512 system equipped with a 17L5 HD linear array transducer operating at a frequency of 8.5 MHz. All the images are presented in grayscale format, allowing for detailed analysis of the lesion structures. To ensure accurate and reliable ground-truth annotations, experienced radiologists created the corresponding segmentation masks, providing precise delineations of the lesions within the ultrasound images.

It is worth noting that the UDIAT dataset exclusively focuses on tumor images, without including any healthy patient cases. This deliberate selection enables researchers to concentrate on the crucial task of distinguishing between benign and malignant lesions, contributing to the advancement of breast cancer diagnosis and treatment.

This dataset is publicly available, but access can be requested here: http://www2.docm.mmu.ac.uk/STAFF/m.yap/dataset.php

(ii) The **BUSI** dataset, which I incorporated in my study as first support datased, also provides a collection of breast ultrasound images. The dataset was gathered in 2018 at the Baheya hospital in Egypt, encompassing a total of 780 ultrasound images obtained from 600 distinct patients aged between 25 and 75 years.

The BUSI dataset is organized into three distinct classes: normal (images without lesions), benign (images containing benign lesions), and malignant (images depicting malignant lesions). For my specific task of breast lesion segmentation, I excluded the images from the normal class to focus solely on the classification and segmentation of lesions.

Notably, the BUSI dataset presents an interesting characteristic in that some patients may exhibit multiple malignant masses within a single ultrasound image. To accurately account for this scenario, the original dataset includes multiple masks, with each mask corresponding to an individual tumor within an image. However, in my work, I opted for a simplified approach to prevent potential errors in evaluating models that successfully identify multiple masses in a single image. I merged all the masks for each patient using a logical OR operation, resulting in a single mask that represents the presence of any malignant tumor within the image.

The dataset is publicly available online at the following URL: https://www.kaggle.com/datasets/sabahesaraki/breast-ultrasound-images-dataset

(iii) The **TCGA-LGG** dataset comprises Magnetic Resonance Images (MRIs) specifically focused on the identification of low-grade glioma in brain scans. This dataset consists of MRI scans obtained from approximately 200 patients, where the FLAIR (Fluid-Attenuated Inversion Recovery) sequence serves as the input modality for my experiments.

The TCGA-LGG dataset was collected through a collaborative effort involving five prestigious institutions across the United States. These institutions include: (i) Jefferson Medical College in Philadelphia, (ii) Henry Ford Hospital in Detroit, (iii) Saint Joseph Hospital and Medical Center in Phoenix, (iv) Case Western Reserve University in Cleveland, and (v) University of North Carolina in Chapel Hill. The inclusion of data from these diverse institutions ensures a comprehensive representation of low-grade glioma cases and enhances the generalizability of findings.

By focusing on brain scans and utilizing the FLAIR sequence, the TCGA-LGG dataset offers valuable insights into the imaging characteristics and specific features associated with low-grade glioma in the brain. This dataset has been widely adopted in the scientific community for various research studies, including tumor classification, segmentation, and prognostic prediction in the context of brain imaging (Bakas et al., 2017; Ghosh, Chaki, and Santosh, 2021; Asiri et al., 2023).

The dataset is publicly available online at the following URL: https://www.kaggle.com/datasets/mateuszbuda/lgg-mri-segmentation

(iv) The **BraTS2019** dataset is a comprehensive collection of Magnetic Resonance Images (MRIs) specifically curated for the Brain Tumor Segmentation challenge conducted in 2019 (Perelman School of Medicine, 2019). This dataset focuses on brain tumors and consists of two main classes: high-grade glioma (HGG) and low-grade glioma (LGG).

Each patient in the BraTS2019 dataset is associated with four distinct MRI sequences: T1, T1 with Contrast Enhancement (T1CE), T2, and Fluid-Attenuated Inversion Recovery (FLAIR). These sequences provide complementary information and contribute to a comprehensive analysis of brain tumors.

For my research, I specifically selected the FLAIR sequence from the LGG class as it demonstrated the most similar visual features to my target dataset. By utilizing the FLAIR sequence, I aim to leverage the specific characteristics and information it offers for the accurate segmentation of low-grade glioma tumors.

When working with the BraTS2019 dataset, it is essential to consider that the provided masks are not in binary format. Instead, they represent a confidence level or "intensity" score. To simplify the segmentation task, I opted to convert the masks to binary values by applying a threshold of 0. Consequently, any pixel in the mask with a non-zero value is considered part of the malignant mass that requires segmentation.

Our selected subset of the BraTS2019 dataset comprises a total of 11,780 images from the LGG class. Among these images, 4,926 include a malignant mass, while 6,854 images do not. This carefully curated subset allows to focus specifically on segmenting malignant masses within the LGG class, enabling the development and evaluation of robust segmentation algorithms.

The dataset is publicly available online at the following URL: https://www.kaggle.com/datasets/aryashah2k/brain-tumor-segmentation-brats-2019

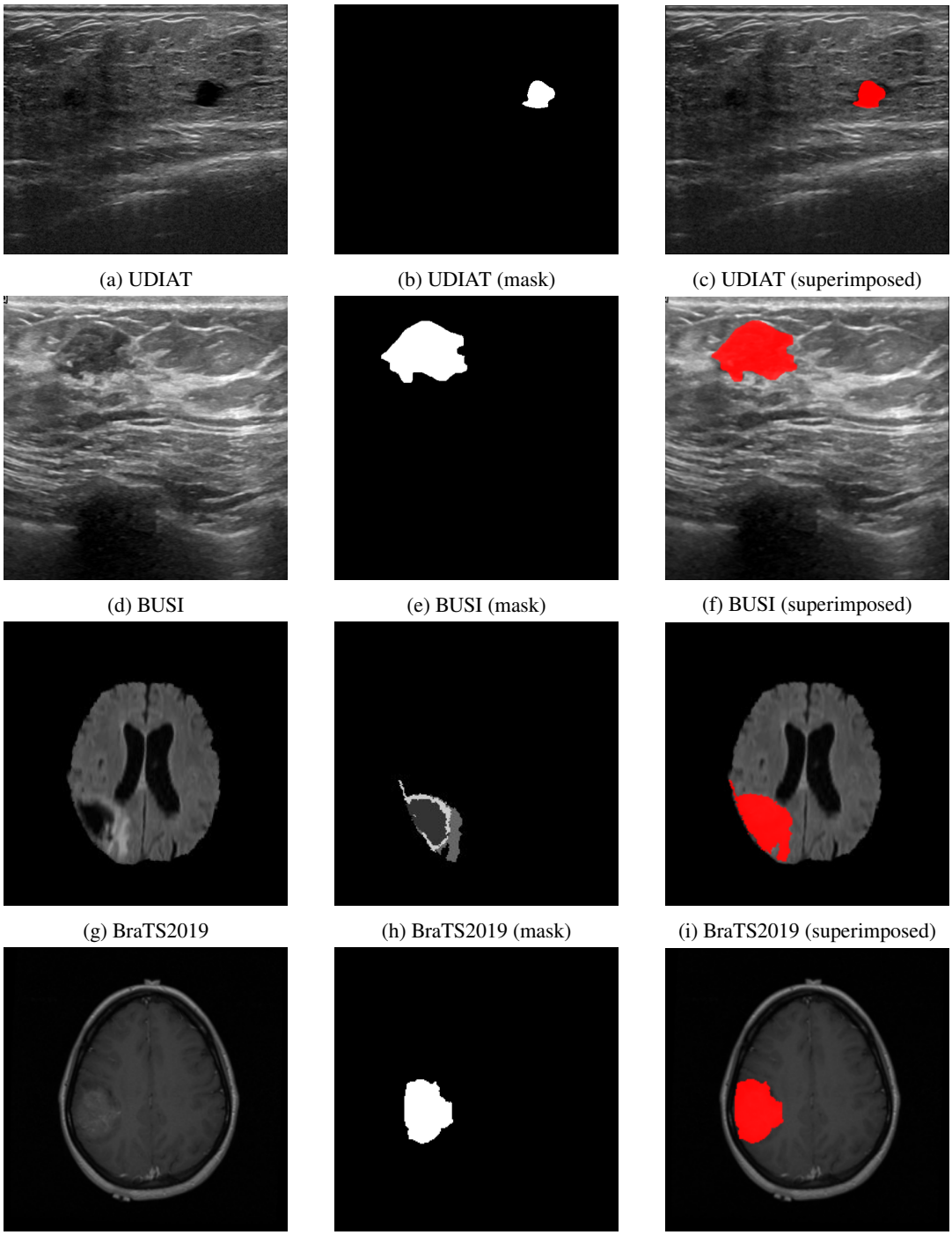
1.6 Transfer Learning

1.6.1 Classical Sequential Transfer

Classical Transfer Learning (Torrey and Shavlik, 2010; Samala et al., 2017; Samala et al., 2020; Raghu et al., 2019; Iman, Arabnia, and Rasheed, 2023; Saxena et al., 2019; Rosenstein et al., 2005; Zhang et al., 2020; Abubakar, Ajuji, and Yahya, 2020) specializes a pre-trained model on a new target task, by re-initializing and re-training the last few layers of the network. This is often beneficial as training the original network can be run once on larger datasets (and budgets) than commonly available, with each following specialization requiring but a fraction of the data and cost. This process is inherently sequential: the model is first trained on one dataset, which is then discarded for the next one, an important distinction against the parallel transfer used by Typhon. The assumption underneath sequential transfer learning is that the feature extraction mechanism learned on the first dataset is immediately applicable to the new problem. This is not granted: for example, early work attempted with limited success to port modern results on natural images (e.g. object detection) to Computer-Aided Diagnosis such as oncological diagnosis based on medical images, simply by training a model first on natural images, then applying transfer learning to the medical images (Varoquaux and Cheplygina, 2022). This creates a bias in learning, as the visual features present in natural images (straight edges, solid colors, patterned surfaces, etc.) are not found in density-based images of the inner workings of the human body such as MRIs, ultrasounds and PET scans, where everything is represented as smooth density gradients across varying tissues. For this reason, Typhon uses representation from different body parts to extract meaningful inter-related features.

1.6.2 Deep Transfer Learning

Deep Transfer Learning (DTL (Iman, Arabnia, and Rasheed, 2023)) further reduces the need for extensive labeled datasets and reduces training time, by reusing the knowledge from a source data/task when training for another target data/task. This is done by including the earlier layers of the original network in the training process, rather than keeping them "frozen" as standard. However, this leads to *catastrophic forgetting* (McCloskey and Cohen, 1989; French, 1999): to mitigate this issue, continual learning (Thrun, 1995) frameworks have been introduced. Progressive Neural Networks (Rusu et al., 2016) for example begin with just a single-column neural network, trained on the initial task, and add new columns of neurons laterally to the existing trained layers, for each new task, with initially with randomly initialized weights. As training proceeds, this leads to increased available network complexity without disrupting the learning target. Dynamically Expandable Networks (Yoon et al., 2018) use group sparse regularization instead to selectively retrain the existing network, also expands its capacity whenever necessary.



(a) UDIAT (b) UDIAT (mask) (c) UDIAT (superimposed)

(d) BUSI (e) BUSI (mask) (f) BUSI (superimposed)

(g) BraTS2019 (h) BraTS2019 (mask) (i) BraTS2019 (superimposed)

(j) TCGA-LGG (k) TCGA-LGG (mask) (l) TCGA-LGG (superimposed)

Figure 1.9: **Datasets.** Samples (and corresponding masks) from the four datasets used. The mask for BraTS2019 has not been preprocessed yet, and contains different gradations. Each row correspond to a different dataset; the images in the first column is a sample from the dataset. Second column shows a boolean mask, with white pixels corresponding to a malignant lesion. This is the target (label) for my model. Third column shows how the location of white pixels in the mask corresponds to the location of the malignant mass in the original image.

1.6.3 Multitask Learning

An architecture closer to Typhon's multi-headed implementation can be found in Multitask Learning (MTL; (Caruana, 1997)), which learns a single model to address multiple tasks in parallel. In MTL however the inputs for all tasks are concatenated and passed together as a single element, and the whole model is activated each time jointly on all tasks. This means that MTL models will have separate input connections to images from different datasets, limiting the amount of knowledge transfer across datasets. Moreover, it is not possible to activate the model on one dataset alone, as model inference requires the availability of inputs from all datasets at the same time. Typhon instead activates only a portion of the network model every time, as it uses a shared feature extractor and dedicated decision makers for each input data.

1.6.4 Heterogeneous Sequential Transfer: Hydra

Cuccu et al. (2020) presents a new approach to transfer learning by explicitly splitting a model into a feature extractor (the *body*) and a decision maker (the *head*). The authors propose to train the model on multiple and diverse datasets, utilizing a shared feature extractor and assigning a dedicated decision maker for each dataset. The name "Hydra" is derived from the Greek mythological creature, which had one body and multiple heads.

The training process in Hydra begins with an initial end-to-end training on the target dataset. To expedite the training process, the authors employ bootstrapping, where 200 different parameterizations (weights of the neural network) are generated. The models are tested on the validation set and the best-performing parameterization is selected and retained.

Following the bootstrapping phase, a loop is performed across each of the support dataset. Firstly, to avoid "moving target" problem (on which training on a different dataset pulls the model in contrasting directions), the feature extractor is frozen and the custom decision maker is added and trained independently. Subsequently, the feature extractor is unfrozen, and an end-to-end training is conducted on the support dataset, allowing the feature extractor to learn the features shared across the datasets.

This process is repeated for each support dataset Finally, the same loop is performed for the target dataset. Initially, the head is trained with the feature extractor frozen, followed by a final training to fully integrate the head with the unfrozen feature extractor.

1.6.5 Parallel transfer and the Typhon meta-learning framework

Typhon is a meta-learning framework first introduced by Cuccu et al., 2022, continuing the work of Hydra. The core idea is to have multiple models sharing an initial part, the feature extractor (or "body"), but with different decision makers specialized on different datasets (the "heads"). The name originates from Greek mythology, where Typhon was one of the parents of Hydra.

For instance, we can have a feature extractor capable of identifying fundamental visual elements such as lines, corners, or circles, and integrate multiple decision makers each one specializing in the identification of specific objects, such as cars, humans, or traffic lights. With this approach, we enable precise identification for each target object through dedicated decision makers, and at the same time the feature extractor benefits from exposure to a larger dataset. This is particularly valuable in domains where data scarcity is prevalent. Additionally, the feature extractor is trained to learn generic features that are applicable across different object categories. This characteristic is essential for achieving robust performance on unseen data, where the ability to extract generic features instead of memorizing training data becomes crucial.

As described in the original paper, this approach has not been used before mainly due to three problems:

- Catastrophic forgetting / moving target
- Unpredictable initialization
- Data imbalance

In this section I will analyze each in turn, and which solutions have been found in Typhon.

Catastrophic forgetting / moving target
The predecessor of Typhon, Hydra (Cuccu et al. (2020)), also utilized multiple heads, but these were trained sequentially. This sequential training approach posed a risk of catastrophic forgetting, where features learned during one head's training could be overwritten when training subsequent heads, without the possibility to recover. To address this issue, a solution is to train all the datasets in parallel. However, this approach faces the challenge of moving target, which hinders convergence and might render previous progress ineffective.

The first approach in Typhon to mitigate this issue is bootstrapping. By ensuring a suitable initialization for multiple heads, we can deduce that the feature extractor is heading in the right direction to learn sufficiently general features. Furthermore, the use of unusually small batches prevents the training process from favoring specific datasets exclusively, and the separation of the feature extractor and multiple decision makers enables the model to effectively learn distinctive features in the dataset-specific components while emphasizing the general features in the shared components.

Unpredictable initialization
Bootstrapping is a technique that involves initializing the model multiple times, evaluating each initialization, and then starting training with the best-performing model thus far. Although a straightforward approach, bootstrapping offers significant advantages in training, particularly in terms of time efficiency.

In Hydra, after training the first head, the feature extraction had already learned some dataset-specific features. As a result, the bootstrap approach could be applied to the following heads by evaluating different initializations and selecting the one that performed best with the existing feature extractor. However, in parallel training as employed by Typhon, all heads are trained simultaneously, necessitating the initialization at the beginning of training. To address this, Typhon introduces a requirement for initialization to perform well not only on average but also on at least two heads. This ensures that the feature extractor can extract sufficiently general information that is useful across multiple datasets. As this requirement relies on the "luck" of performing well on two specific heads and the feature extractor at the same time, multiple initializations need to be attempted. While this may require additional time, it remains a crucial aspect of the framework.

Data imbalance
One of the main challenges when training on multiple datasets is data imbalance. In real-world applications, datasets often vary in size, with some being significantly larger than others. In the case of Typhon, which requires a batch from each dataset at every epoch, smaller datasets can be exhausted much earlier. Terminating the training at this point could result in the potential loss of valuable information contained in the remaining data. On the other hand, continuing training with fewer datasets poses the risk of forgetting features specific to the excluded dataset. Data augmentation can be considered as a potential solution, but it carries the risk of diluting the quality of the training set, as it increases the dataset size without introducing new information.

To address this challenge, a specific `loop_loader` mechanism is implemented in Typhon. This mechanism reshuffles a dataset once it has been fully utilized, allowing it to be reused with new batch divisions and orders. Although this approach may theoretically result

in overfitting on the smaller dataset, the transfer learning process helps mitigate this issue by leveraging the shared feature extractor and the parallel training of decision makers.

1.7 Direct Policy Search

The classical approach in Reinforcement Learning is to explore the environment and learn the rewards received by the agent when starting from a specific state and performing a given action. The value of this state-action combination is then adapted considering future rewards made available in the newly reached state: if I rob a bank I will obtain a lot of money now (big positive reward), but if the police catch me this money will be confiscated (big negative reward) and for a long period in the future I will be in jail (a long sequence of negative reward). In my current situation, the action of robbing a bank has probably not a high value. This adaptation of the state-action value using immediate rewards and future rewards is called the Bellman equation. Since future rewards are usually less valuable than immediate ones, in the equation they are discounted. To maximize the reward, an agent will choose the action with the highest value for its current state.

In practice, memorizing all the combinations of state-action to values is usually unfeasible given the large amount of possibilities. For instance, when the state is represented by an image, a variation in a single pixel creates a new state. Additionally, if the agent reaches a previously unseen state, it has no information about the values of the possible actions. For this reason (deep) neural networks are often employed to approximate this value function, providing a smaller representation and introducing generalization capabilities.

Direct Policy Search involves approximating the function that directly maps states to actions, without explicitly estimating state-action values. Especially with high-dimensional inputs like images, the models used for Direct Policy Search handle the extraction of useful information first and then use it to make decisions. Mapping raw pixels directly to values is not realistic.

In 2019, the paper "Playing atari with six neurons" (Cuccu, Togelius, and Cudre-Mauroux (2019)) has shown that separating these two tasks (feature extraction and decision making) leads to a better generalization and improvement of the overall performance. More relevant to this work, they demonstrated the relative complexity of each part in a controller for Atari games, showing that the decision maker can be as small as a single layer with one neuron for each possible action. Not only these controllers reaches comparable results with state-of-the-art models which uses two order of magnitude more neurons, but their training is not bounded to backpropagation anymore: it is possible to use Evolution Strategies, or even Random Weight Guessing.

1.8 Research Questions and Contributions

The primary objective of my research is to investigate the advantages of the Typhon meta-learning framework in improving feature spaces via parallel transfer. To accomplish this central goal, I pursued various sub-tasks, each leading to a distinct contribution.

Obtaining and preprocessing datasets.
An essential prerequisite for the subsequent research is the acquisition of appropriate datasets. Specifically, I will obtain the primary UDIAT dataset, along with three support datasets (BUSI, TCGA-LGG and BraTS2019). Due to the varying formats and structures of these datasets, a significant contribution of this work will be their preprocessing and standardization, necessary to prepare them for utilization within the Typhon framework.

Adapting the Typhon framework for segmentation.
The Typhon framework, originally applied on classification tasks, was extended to incorporate segmentation capabilities. This adaptation involves introducing multiple key enhancements to meet the unique requirements of segmentation. These include the integration of a new loader to prepare the data, a new bootstrap configuration, the implementation of a modified error computation structure (including the integration of an appropriate loss function), and the incorporation of suitable evaluation metrics specifically tailored for segmentation tasks.

Implementation of new neural network architectures.
To enhance the segmentation capabilities of the Typhon framework, new Deep Learning architectures will be implemented. Specifically, the popular UNET and RF-Net architectures, known for their success in segmentation tasks, will be incorporated into the codebase. These architectures leverage the power of residual connections to effectively capture spatial information and improve segmentation accuracy.

Evaluation of multiple splits.
One of the crucial hyperparameters in the Typhon framework is the split point within the architecture that separates feature extraction and decision-making components. In this study, I will conduct a comprehensive analysis of multiple division points for the new architectures, aiming to identify the optimal split that yields the most favorable results. By systematically exploring various splitting configurations, I can determine the configuration that maximizes the performance and efficacy of the Typhon framework and gain deeper insights about the impact of this parameter.

Reproduction of RF-Net results.
To assess the performance on the UDIAT dataset, an analysis will be conducted by comparing my results with those reported in a previous study (Wang, Liang, and Zhang, 2021). Due to the limited accessibility of the original datasets, I will replicate their findings using the publicly available portion of the datasets. This approach ensures a fair and transparent evaluation of the framework's effectiveness while also establishing a meaningful benchmark for comparison with prior research.

Application to the UDIAT dataset.
The enhanced Typhon framework, equipped with segmentation capabilities and utilizing the newly implemented architectures, will be applied to the challenging task of breast lesion segmentation using the UDIAT dataset. This dataset, consisting of a collection of breast ultrasound images with corresponding ground-truth annotations, will serves as a benchmark for evaluating the effectiveness of the proposed framework. It will obtain new state-of-the-art results, improving reproduced results by 9% in recall and 4% in IoU.

Jupyter notebooks for result visualization.
To enable the analysis and interpretation of the segmentation results, Jupyter notebooks will be developed as an interactive platform, offering a user-friendly interface for visualizing and exploring the outcomes of the segmentation experiments. Through these notebooks, it will be possible to assess various metrics, generate informative plots, and delve into the detailed performance and characteristics of the developed models. This approach will enhance the transparency and facilitate a deeper understanding of the segmentation results obtained in my research.

Overfitting counteraction.
Addressing the common challenge of overfitting in Deep Learning, this study will explore the

potential of the Typhon framework to mitigate overfitting. Preliminary experiments will be conducted to demonstrate how the framework can effectively counteract overfitting and improve the generalization capabilities of the segmentation models, due to the parallel transfer learning process.

Development of utility tools.
To facilitate the workflow and gain valuable insights into the dataset and model performance, utility tools in the form of helper scripts will be developed to support various stages of the process. These tools will provide assistance in tasks such as visualizing and exploring numpy files that store images, examining dataset characteristics such as the proportion of samples with tumors, plotting the distribution of bootstrap values for statistical analysis, and producing visualizations of image-mask pairs for qualitative assessment. By employing these utilities, the research process will be more efficient and offer meaningful insights into the dataset and model evaluation.

Further application of Typhon.
Following the application of Typhon to the segmentation task, I will expand its codebase to address a different challenge. Specifically, I will adapt Typhon to accommodate autoencoders and variational autoencoders, which I will use with datasets consisting of Atari game images. Leveraging the parallel transfer capabilities of Typhon, I will train a model on multiple game environments, and subsequently I will assess the quality of the generated feature space by evaluating the model's performance on images from previously unseen games.

Implementation of an efficien data collection mechanism.
To gather the datasets for the autoencoder, I will develop a data collection mechanism aimed at incorporating into the dataset only images that contain novel information. This approach will enable efficient generation and expansion of the datasets as the games progress to later stages, including previously unavailable images.

Implementation of the new PixelPerfect layer.
Inspired by the work on Atari games, I will develop a new neural network layer called PixelPerfect. This layer enables exact localization of elements within the input signal, such as sprites in Atari images. Its applicability extends to any task that requires precise information about the location of elements, including object detection in images or tumor segmentation in MRI scans.

Chapter 2

Method

This chapter introduces the work done on the Typhon framework with the goal of improving feature extraction and consequently enhance decision-making processes. The focus is on the task of breast tumor segmentation, for which I developed an adapted version of Typhon specifically tailored to this purpose. Section 2.1 describes the modifications and additions made to Typhon to align it with the requirements. As Typhon is a versatile framework designed for various tasks, Section 2.2 delves into its numerous hyperparameters, providing a comprehensive explanation of each parameter to ensure proper utilization.

Following this, I delve into the preparatory work required prior to using the datasets. This includes the collection process as well as the technical aspects of preprocessing and standardization, which demanded a significant portion of time. Moreover, I introduce utility tools developed throughout this work, serving purposes ranging from data preprocessing to conversion, splitting, visualization, and more.

Having covered the groundwork, I proceed to present the concrete experiments conducted in this study. Given the extensive number of experiments (several hundreds), I opt to summarize them into groups based on their typology and objectives, rather than individually describing each one.

The latter part of this chapter showcases the final experiments that yielded the most promising results. After introducing the experimental setup, I present the outcomes and engage in a thorough discussion of the findings. The chapter concludes with an exploration of the limitations encountered during the course of this research, providing valuable insights into areas that require further improvement.

2.1 Adapting Typhon

The first implementation of the Typhon meta-learning framework was initially tested in the classification of medical images. In its original form, the framework utilized datasets that were divided into two classes: one containing images with tumors and the other with images from healthy patients. During training, the model processed batches from different datasets, and after each batch, the framework assessed the model's performance across all available samples. This parallel transfer learning approach was followed by creating individual model instances for each dataset and performing specialized training with a single dataset to further refine the model's classification capabilities.

Extending the applicability of Typhon to segmentation tasks required significant modifications. In this section, I will explore the key adaptations made to Typhon, highlighting the initial setup, the reasons behind the need for modifications, and the specific alterations implemented.

Bootstrap. In Typhon, the bootstrap plays a vital role in mitigating the moving target problem. Without it, changes made during training with one dataset could be completely disrupted by subsequent changes during training with the next dataset, resulting in the training process

being pulled in different directions. The purpose of the bootstrap is to find an initial configuration that is a viable starting point for training on multiple datasets in parallel. If, at the beginning of training, the model's performance is similar across all datasets (and reasonably good), it indicates that the shared part of the model (the feature extractor) is generating features that are general enough to be utilized by all decision makers. Conversely, if the performance of one head (dataset-specific component) of Typhon significantly surpasses the others, it suggests that the feature extractor is producing features specific to that particular dataset. In such cases, during parallel training the feature extractor will be continuously pulled in conflicting directions, hindering convergence or at least leading to the generation of non-generic features.

To determine a suitable starting point, Typhon generates multiple initializations randomly using the Xavier initializer and tests them on training and validation datasets. The initialization with the highest AUC score is retained, provided that the performance difference between the best and worst heads is not greater than 0.2.

However, when moving to segmentation tasks, this approach is no longer applicable. Firstly, the performance of a random model in a segmentation task is considerably lower compared to classification tasks, where the model statistically classifies correctly half of the samples (assuming only two classes). As a result, the predefined threshold of 0.2 becomes meaningless. To adapt the approach, a new initialization is now accepted as better if it exhibits both a higher average performance **and** a decreased difference between the best and worst model. Although not optimal, this modification significantly enhances stability in segmentation tasks.

Another change I implemented was the utilization of Dice Score (F1) as the evaluation metric instead of AUC. While AUC is a valid metric, Dice Score is generally considered more suitable for assessing segmentation models.

Additionally, due to the use of larger datasets and the increased computational time required for evaluating segmentation outputs (compared to classification outputs, which consist of only one number instead of a full image), the process of testing multiple initializations became significantly time-consuming. The likelihood of finding and identifying a good initialization is lower, as a well-performing feature extractor may be concealed by a suboptimal decision maker. To address this issue, I modified the framework to utilize only a sample of the training and validation datasets. As long as the sample size remains statistically significant and representative of the entire dataset, this adjustment does not compromise the effectiveness of the method but improves its performance.

Metrics frequency. In the original implementation of Typhon, after each training epoch (where the model was exposed to one batch from each dataset), the model's performance was evaluated by computing a set of evaluation metrics. To gain insights into the training progress, these metrics were computed not only on the validation and train datasets but also on the test dataset. The results were then saved in a CSV file, providing a useful means to visualize the training performance.

The first improvement in this regard was of a technical nature. In computer science, there is a principle known as "single responsibility", which states that every module or function in a program should be responsible for only one task. This enhances code clarity, readability, and reduces the likelihood of side effects from future changes. As a result, the first step taken was to separate the computation of metrics and encapsulate it within a dedicated function.

The second and most significant change was to introduce parameterization to control the invocation of this function, thus limiting its usage. Instead of computing the metrics after every epoch, I allowed for computation after a specific number of iterations. For instance, in the final experiments, I only required 100 data points to generate the plots. Given that the model was trained for 50'000 epochs, this change reduced the time required for metric

computation by a factor of 500, decreasing run time from over 104 days to just 5 hours. This change was crucial in enabling the execution of the large suite of the experiments presented in this thesis.

Metrics computation. For the computation of classification metrics, Typhon fed the datasets into the model, and stored the outputs and together with the labels. Only at the end performance metrics were computed. This approach was practical and efficient, as it allowed the metrics to be computed only once for each dataset. However, when I started using the framework for segmentation, I encountered some challenges.

To understand these problems, it is important to mention how I utilized the graphics card (GPU) to accelerate training. The model was stored and operated on CUDA, the GPU's memory. However, the main limitation was the size of the CUDA memory, which in my case was 32GB. This was typically sufficient to store the model and a few batches, but not for larger datasets. To address this, the data loader dynamically loaded the next batches onto the GPU and removed those that had already been processed.

During the computation of metrics, the code attempted to store the output and labels for each sample. In a classification task, these values are simply two integers. However, in a segmentation task, they are full images, resulting in storage requirements twice the size of the dataset (and in some cases exceeding the 32GB GPU memory capacity). Even with smaller datasets, this posed a challenge to Typhon's ability to work with limited memory due to the extremely small batch size it employs.

To resolve this issue, one potential solution was to transfer the outputs from the GPU memory to the main system memory, effectively removing them from CUDA. However, this approach was suboptimal as the frequent data transfers significantly impacted performance. In search of a better alternative, I realized that my metrics were invariant to the order of computation. In other words, computing the metrics per batch and aggregating the results at the end yielded the same results as computing them on the full output. Thus, I chose this alternative approach. It is worth noting that the actual batch size is not relevant for the metrics computation and does not need to be the same as during training. This flexibility allows for a trade-off between memory usage and runtime, as is often the case.

Saving samples. In the current implementation, evaluating the training progress primarily relied on metrics, which provided valuable insights into the model's performance. These metrics, accompanied by specialized scripts, facilitated the generation of informative graphs showcasing the loss evolution, performance disparities across different datasets, and various other relevant information. However, when dealing with computer vision tasks, visualizing the model's output in concrete images proves highly advantageous for gaining an immediate understanding of the model's behavior.

To address this need, I expanded the existing code to capture and save representative data samples. Alongside the regular interval for metric computation, Typhon now incorporates the functionality to randomly select an input from each dataset, pass it through the model, and retrieve the corresponding output. This data, consisting of the input, label, and output, is then saved in the designated results folder. To facilitate easy interpretation and comparison, the saved files are converted into Portable Network Graphics (PNG) images, providing a visual snapshot of the model's actions at various stages of the training process.

By incorporating this enhancement, I gain the ability to directly observe the model's predictions in a tangible and accessible format. This visual feedback complements the numerical metrics and offers immediate insights into the model's performance and its ability to accurately interpret and classify the input data. Furthermore, the inclusion of image samples in the results folder serves as a visual record of the training progression, facilitating the comparisons between different iterations of the model.

Metrics time. A seemingly minor modification that proved to be exceptionally valuable in my context was the implementation of a more precise runtime computation. Specifically, I divided the total time into two distinct components: the time devoted to actual training and the time allocated solely for metric computation. This seemingly trivial separation yielded significant improvements in performance evaluation by recognizing the bottleneck areas. Additionally, this division was crucial for accurately evaluating sample efficiency and overall performance, as the initial evaluation was overshadowed by the substantial time required to compute metrics on the extensive support datasets.

Cropping and padding. Another necessary addition pertained to the variability in image shapes within certain datasets, particularly BUSI. These datasets contained images with slightly varying lengths and widths from image to image. While this variation itself was not problematic, the library implementation used (PyTorch) did not allow for such shape differences within a batch.

Furthermore, the chosen architecture involved repeatedly halving the shape of the input (see Section 1.3.2) and then doubling it until reconstructing the original form. However, if the length or width of an image was not divisible by two during the convolutional process, the reconstructed size would be incorrect. Although this issue could be addressed by incorporating appropriate padding within the convolution itself, it would create the opposite problem with shapes that are inherently divisible. It is important to note that this problem only arose due to the utilization of multiple datasets.

To address this challenge, I introduced a parameter to determine the `"working shape"` for each dataset. If the actual shape of an image was smaller than the corresponding working shape, the image was padded with zeros (*padding*). Conversely, if the image was larger, only a portion of it was considered (*cropping*). The latter case is less desirable as it involved excluding a section of the image from training, thus I opted for a relatively large working shape in my experiments. The need for cropping only arose in edge cases, such as some images in the BUSI dataset.

Dice Loss. Another important step involved the implementation and incorporation of a specific loss function for segmentation, namely the Dice Loss. This loss function serves as an appropriate measure to evaluate the similarity between the predicted segmentation and the ground truth, as discussed in Section 1.3.2.

The Dice Loss is computed by subtracting the Dice Score from 1, providing a quantitative assessment of dissimilarity between the predicted and ground truth segmentations. While conceptually similar to the Dice Score, the implementation of the loss function requires careful consideration to ensure proper computation and enable effective error backpropagation during training.

$$\text{Dice Loss} = 1 - \text{Dice Score} = 1 - \frac{2 \times \text{Intersection}}{\text{Total Predicted} + \text{Total Ground Truth}}$$

During the calculation of the Dice Loss, it is necessary to accurately compute the loss for all elements within the batch and subsequently aggregate the values. Special attention must be given to avoid a denominator of zero, as it would result in an error in the code. Furthermore, it is important to emphasize that the loss value is equivalent to 1 minus the Dice Score. The inclusion of the negation ensures that the training process guides the model towards improvement rather than worsening its performance.

Data Loader. As mentioned in Section 1.6.5, the Typhon framework utilizes a customized loader to address class imbalance. This loader performs various tasks such as data checking, shuffling, partitioning into batches of a specified size, and subsequently returning the batches

one by one. Unlike a conventional loader, once all the batches have been utilized, the loader reshuffles them and continues to provide batches indefinitely. To ensure that the batches are loaded into memory at the appropriate moment, the loader utilizes a class from the torchvision library named `"DatasetFolder"`. This class assumes that the samples are organized into separate folders, with each folder representing a distinct class. The class name is derived from the folder name, providing a logical and structured approach to dataset organization specifically suited for classification tasks.

However, for my segmentation task, this approach was not applicable, and I had to reimplement this class to accommodate my requirements. I introduced a new class called `"Segmentationdatasetfolder"`, which extends the functionality of the torchvision DatasetFolder. By inheriting from this library class, I was able to leverage various implementation benefits, including dynamic sample loading. Nevertheless, several specific aspects needed to be re-implemented to tailor the class to my segmentation task.

Upon instantiation, the class initially traverses the dataset to gather the paths to each element. In my case, the masks for the images are stored in the same folder, and thus I utilized regular expressions to selectively collect only the paths corresponding to actual samples.

The second part of the implementation pertains to the `__getitem__()` function, which is invoked when a new sample from the dataset is requested. Within this function, the image and its corresponding mask are loaded, their shapes for consistency are verified, they are convert them into `torch.Tensor` objects, and they are transfer to the GPU. If necessary, padding or cropping operations are applied to ensure desired dimensions, and subsequently return both the input image and the mask (now serving as the label).

2.2 Typhon hyperparameters

The design philosophy behind Typhon emphasizes its versatility as a generic framework, prioritizing adaptability across various tasks, datasets, and goals rather than being tied to a specific model or algorithm. To achieve this flexibility, Typhon incorporates a comprehensive set of hyperparameters. These hyperparameters serve as tunable settings that allow users to fine-tune and customize the framework according to their specific requirements, offering a powerful toolset for tailoring the framework to the unique demands of different applications and domains. In this section, I will delve into the various hyperparameters offered by Typhon, discussing their roles, impact, and considerations in harnessing the full potential of the framework.

It should be mentioned that certain hyperparameters were already present in the original code of Typhon and are not exclusive to the thesis's work. However, gaining a thorough understanding of these hyperparameters is essential in order to fully utilize the framework's capabilities. While grasping their functionality may require some time and effort, delving into these hyperparameters is crucial for unlocking the complete potential of Typhon.

- **`paths`**: This parameter contains a dictionary of paths necessary for the code, specified as Path objects from the pathlib library. It includes the location of the dataset, architectures, path to the RAM (for faster loading), and the location to save the results.
- **`dsets_names`**: A list of dataset names. Each name should correspond to a folder within the dataset path, which should contain the subfolders `"train"`, `"validation"`, and `"test"`.
- **`architecture`**: The name of the architecture to use. Within the architecture folder, there should be two corresponding files: `{name}_fe.py` and `{name}_dm.py`, which represent the model for the feature extractor and the model for the decision maker, respectively. Each file should contain a function `get_block(dropout, in_channels=1)`, which returns an instance of a PyTorch module (the actuall model).

- `initialization`: The standard initialization of Typhon uses bootstrap. This is important to avoid the moving target problem that arises when using multiple datasets. However, the implementation offers two alternatives. It is possible to use random initialization (equivalent to bootstrap with only 1 generation), which can be useful when using Typhon with only one dataset (e.g., generating baselines). Additionally, it is possible to load a pre-existing model instead of repeating the bootstrap. The model should be placed in the output folder, specifically in the subfolder `"models"`, and should be named `"bootstrap_model.pth"`. Apart from loading an external model, this configuration is optimal when re-running an experiment with the same initialization, which will already be saved at the correct location.
  The three possible values for this parameter are `"bootstrap"`, `"random"`, and `"load"`.
- `bootstrap_size`: During bootstrap, multiple initializations are generated and tested to find a favorable starting point for multiple datasets. Since only the best model is kept, increasing the number of initializations enhances the chances of improving the starting point (although it becomes more challenging over time). This parameter controls the number of random initializations to test and should be set as high as possible (considering the trade-off with runtime).
- `bootstrap_images`: Each initialization during bootstrap is evaluated on a sample from the training and validation datasets to determine which initialization to keep. However, evaluating the entire training set at each iteration can be time-consuming, especially with large datasets. Ideally, a statistically representative sample should be used. The size of this sample is not fixed and varies from dataset to dataset. It can be estimated empirically by generating subsamples of different sizes and evaluating their deviation from the metrics obtained on the full dataset.
  This parameter controls the number of samples to use for evaluating bootstrap initializations.
- `nb_batches_per_epoch`: This parameter indicates the number of batches to use for each dataset at each iteration. Typhon should in principle keep it at 1, learning only a few pieces of information from each dataset at a time. This approach encourages the model to learn general features instead of being biased towards a specific dataset. However, in specific cases (e.g. reproducing classical training), this parameter can be adjusted to use multiple batches at a time. It also allows to easily pass the entire dataset to the model at each iteration by setting the parameter as size of the dataset/batch size
- `nb_epochs`: One epoch represents one training iteration for Typhon across all the datasets. The number of epochs determines the length of the training. This parameter takes the form of a dictionary, where the keys `"train"` and `"spec"` correspond to the number of epochs for parallel transfer training and specialization, respectively (specialization is the classical training at the end, which was determined to be unnecessary in my case).
- `lrates`: A dictionary containing the keys `"train"` and `"spec"`. Each key corresponds to a list of learning rates, one for each dataset. The learning rate controls the speed at which new information is learned during training by resizing the magnitude of the error gradient. A learning rate that is too small will slow down the training, while a learning rate that is too large may excessively change the network weights and increase the error instead of decreasing it.
- `dropouts`: Dropout is a mechanism introduced by Srivastava et al. (2014) that randomly drops connections while training a neural network. This discourages the model from relying solely on a few connections and encourages the distribution of information across all available neurons. This parameter is also a dictionary (`"train"`/`"spec"`) of dropout rates, with one rate for the feature extractor followed by one rate for each decision maker. Note that for dropout to be actually used, it must be implemented in the

architecture.

- `loss_functions`: This parameter is used to pass a list of loss functions to use, with one function for each dataset. It is not necessary to use the same loss function for each dataset, but no studies have been conducted to assess the benefit of using different loss functions.
- `optim_class`: This parameter is used to pass a list of optimizers to use, with one optimizer for each dataset. Similar to the loss functions, it is possible to use a different optimizer for each dataset, but there is no indication of whether this is beneficial or not.
- `opt_metrics`: During bootstrap and specialization, the best model at each iteration (between the current best model and the newly generated model) is saved. This parameter is a dictionary that requires the keys `"bootstrap"` and `"spec"`, specifying the metric to use for comparing the models in each of the two situations.
- `metrics_freq`: This parameter determines how frequently the metrics are computed, indicating the number of epochs between each computation. Computing metrics on the entire dataset can be time-consuming, and evaluating all samples every epoch (on 1 batch) significantly increases the running time compared to evaluating at the end of the training (as discussed in Section 2.1). However, evaluating at the end of the training provides no insight into the training evolution. Hence, this parameter is crucial for balancing training efficiency and obtaining insights into the training progress.
- `training_task`: This parameter specifies the task to perform. Originally, Typhon was only implemented for classification, and this parameter was not necessary. However, with the extension for segmentation (and for autoencoding), it has become a required parameter. Although most of the code remains the same, there are some differences, such as loader and metric computation. This parameter can take the values `"classification"`, `"segmentation"`, or `"autoencoding"`.
- `batch_size`: This parameter indicates the batch size during training and specialization, again using a dictionary with the keys `"train"` and `"spec"`. The batch size refers to the number of training samples passed together to the model during training. While classical training tends to use larger batch sizes to increase generalization and counteract overfitting, Typhon takes the opposite approach. Cuccu et al. (2022) has shown that very small batches, with sizes of 8 or even 4, offer better results. The key idea is for the model to learn only a very small amount from each dataset at a time, to avoid being pulled too much in one direction (the moving target problem) and to focus on learning generic features.
- `cuda_device`: This parameter specifies the name of the device to which the model is sent. Ideally, it should be a graphics card (GPU), but it can also be the CPU. However, training performance will be severely penalized if the CPU is used instead of a GPU.
- `resume`: During training, Typhon constantly saves checkpoints, a mechanism which allows to resume training after an interruption if necessary. This parameter takes the form of a boolean value, either *true* or *false*.
- `img_dims`: As discussed in Section 2.1, the size of the input is relevant for some models (e.g., UNET). This parameter, which represents what I previously called "working_size", is a list of tuples indicating the desired width and height of the images for each dataset. If the actual sizes do not match, the loader will take care of padding (with zeros) and cropping as necessary.
- `mu_var_loss`: This parameter is only relevant for the autoencoding task. It allows for the use of not only classical autoencoders but also variational autoencoders (Kingma and Welling, 2019), which have demonstrated to be very effective. When a variational autoencoder is split between encoder and decoder, we not only have the output but also two additional parameters (`mu` and `logvar`). This boolean parameter (*true* or *false*) specifies whether or not this is the case, as it has an impact on how the output of the

feature extractor is handled.

- `remove_mode`: Another boolean parameter specific to autoencoding. When set to true, an image representing the mode of the dataset is subtracted from the inputs, allowing the model to focus on encoding the features that vary from image to image and ignore the common parts.

2.3 Dataset preprocessing

This section presents the preprocessing steps made to prepare the dataset. Almost each dataset collected was stored in a different file format, with a different stucture, different range of the values. The uniformization is necessary to use them together, and understanding the different structures and files required a significant amount of time and effort.

Next I will outline the preprocessing steps undertaken to prepare the dataset. Each collected dataset was stored in a distinct file format, featuring varying structures and value ranges. The standardization of these datasets was imperative to enable their combined utilization with Typhon. Although it may appear as a brief task, understanding the diverse structures and formats necessitated a considerable amount of time and effort. In this I was aided by Broillet (2022), which provided valuable scripts that served as a useful starting point.

2.3.1 UDIAT

The full datasets used in Wang, Liang, and Zhang (2021) are no longer accessible at the links provided in the paper. However, I was able to obtain some parts of it. I submitted a request to the authors of Yap et al. (2018), following the procedure explained in the website of the Department of Computing and Mathematics of the Manchester Metropolitan University[1].

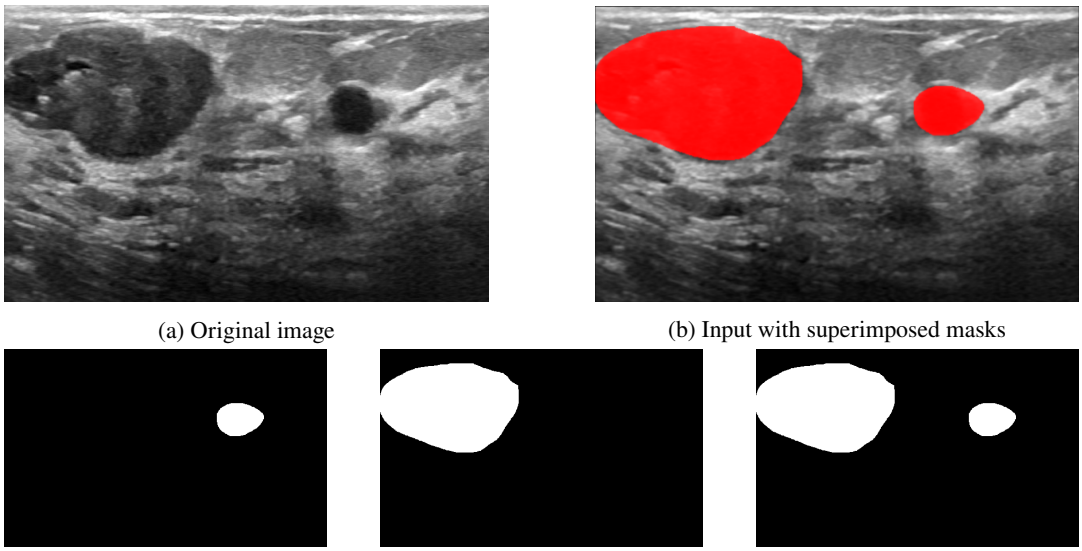
Once I obtained the dataset in the form of Portable Network Graphics (PNG) images, I converted them into numpy arrays and performed normalization by rescaling the values to fall within the range of 0 to 1. This normalization process is commonly employed to improve the training of neural networks.

2.3.2 BUSI

I obtained the BUSI support dataset from Kaggle. The dataset was organized into three main folders, each containing images of different categories: normal ultrasounds (without a tumor), benign tumor scans, and malignant tumor scans. The images, along with their corresponding masks, were in the PNG format, which can be easily converted to numpy (npy) matrices using Python libraries such as Pillow or OpenCV-Python. After conversion, I normalized the pixel values, which were originally integers ranging from 0 to 255, to floating-point values within the range of 0 to 1.

Additionally, I addressed the issue of multiple tumors within the same image. Some images in the dataset depicted more than one tumor, and initially, there were separate masks for each one of them (see Figure 1.9e). However, this approach would have been suboptimal for my purposes, as a model correctly identifying all the tumors would be penalized when evaluated against the individual masks. This is because many pixels would be considered false positives. To overcome this, I merged all the masks corresponding to the same input image. In the merged mask, any pixel marked as a tumor in *at least one* of the individual masks is considered as part of a tumor, while all other pixels are considered non-tumor.

[1] http://www2.docm.mmu.ac.uk/STAFF/m.yap/dataset.php



(a) Original image

(b) Input with superimposed masks

(c) First mask

(d) Second mask

(e) Resulting mask

Figure 2.1: **Merging masks in the BUSI dataset.** The input image (a) reveals two tumors, which in the original dataset are identified with two distinct masks ((c) and (d)). During the preprocessing phase, I merge them into a single mask, shown (e).

2.3.3 TCGA-LGG

I obtained the TCGA-LGG support dataset from Kaggle. The dataset is structured with one folder per patient, and each patient's subfolder contains multiple brain scans captured at different heights, along with their corresponding masks. The image files are stored in the Tag Image File Format (TIFF), which is commonly used for storing high-quality images using lossless compression. Although less common than PNG, TIFF is still widely supported by libraries such as Pillow and OpenCV-Python.

Similar to the BUSI dataset, I converted the TIFF images to numpy matrices and rescaled the pixel values to the range [0.0, 1.0]. However, due to the presence of multiple brain scans at different heights, some images contained very few pixels representing a tumor. These images had the potential to negatively affect the model training. To improve the quality of the support dataset, I decided to discard images whose masks contained less than 25 tumor pixels (but at least one). This filtering step aimed to ensure that the dataset primarily consisted of images with sufficient tumor representation (considering that each image comprises 65,536 pixels), or without any tumor at all.

2.3.4 BRATS2019

Same as for the other support datasets, I obtained BraTS2019 from Kaggle. The dataset is divided into High Grade Glioma (HGG) and Low Grade Glioma (LGG), with each patient having a dedicated folder. Within each patient's folder, the dataset includes a mask and four sequences: fluid attenuated inversion recovery (FLAIR), T1, T1 with contrast enhanced (T1CE), and T2. All the files are in the Neuroimaging Informatics Technology Initiative (NIFTI) format, and I utilized the `nibabel`[2] library in Python to work with them.

Each file consists of 155 brain scans captured at different heights. The difference with the TCGA-LGG dataset is that those scans are merged into a single document. However, since the models I use are not designed to process 3D images, I extracted individual slices from

[2]https://pypi.org/project/nibabel/

the volume to create separate images. For each slice, I included a copy of the corresponding mask, and converted both images into a numpy array. No additional rescaling was required.

Similarly to the previous dataset, I removed images containing only a small section of a tumor. I applied the same filtering procedure, considering only slices with no tumors or with a tumor covering 25 pixels or more.

2.4 Utility functions and tools

In this section, I will introduce a collection of tools and utility scripts/functions that have been developed to streamline and automate various aspects of this work. Throughout the thesis, numerous repetitive tasks, such as result analysis and graph plotting, have been carried out repeatedly. While these operations may not be inherently complex or time-consuming, standardizing them with utility scripts has greatly expedited the process and ensured consistency, reducing errors deriving from the human component. Additionally, certain tasks, such as dataset analysis and creating combined plots merging Typhon training and subsequent specialization, are less frequent but more intricate. By creating dedicated tools for these specific tasks, I was able to approach them with meticulous attention and facilitate potential reproduction if required. In the following section, I will present the key tools and scripts that were developed during this thesis. For each tool, I will briefly discuss its necessity (or usefulness) and provide an overview of its functionality.

Count tumors. The first group of helper tools has been developed to facilitate dataset analysis and inspection. The first tool in this group aims to inspect a dataset (or any input folder along with its subfolders), and identify all masks that contain a tumor (i.e. non-empty masks). This functionality enables the generation of a summary distinguishing samples with tumors from those without tumors. Additionally, it serves as a means to verify the integrity of masks and validate claims that certain datasets exclusively consist of tumor images (see Section 1.5). The script achieves this by recursively exploring all subfolders within the specified input path. For each file encountered, it examines whether it represents a mask (identified by the `"mask.npy"` suffix). If the maximum value in the mask is 1, it indicates the presence of a tumor; otherwise, it signifies its absence. The tool provides a count of the total number of masks, as well as separate counts for tumors and non-tumor samples, serving as a valuable reference for dataset analysis.

Visualize images with specific file extension. Following the preprocessing stage, all samples in the datasets are converted into numpy arrays, resulting in files with the `.npy` extension. This transformation offers significant benefits, such as reduced file size, improved usability, and enhanced consistency. However, a drawback of using numpy files is that they are not directly human-readable, making it inconvenient to inspect individual images, especially in cases where the dataset is extensive. To address this issue, I have developed a script capable of opening various file formats. The script supports `npy` files, but also "Digital Imaging and Communications in Medicine" (DICOM) files stored in the `.dcm` format, "Neuroimaging Information Technology Initiative" (NIFTI) files stored in the `.nii` format, Scalable Vector Graphics (SVG) images, and other formats. After decoding the format type, the script utilizes the Pillow library to display the output to the user. Additionally, when encountering `dcm` documents (which may consist of multiple scans), the script enables convenient navigation through the different images. By configuring this script as an action in Linux, users can open and inspect most images simply by right-clicking on them, significantly enhancing working efficiency.

Plot bootstrap values. Currently, there is no definitive method for determining the optimal length of the bootstrap process. The likelihood of obtaining a favorable initialization varies

depending on multiple factors, including the model's architecture and the dataset's similarity. Instead of attempting to precisely estimate this length, I opted for an empirical approach. To accomplish this, I developed a script that analyzes the experiment's log, which is saved using the `brutelogger`[3] library. By utilizing regular expressions, the script identifies the iteration at which better models were discovered during the bootstrap process and quantifies the degree of improvement. The resulting plot typically exhibits a decline in the frequency of finding superior models. Based on this empirical observation, it is possible to establish a bootstrap length that proves satisfactory for the specific settings.

Plot masks. Helper tools have also played an inportant role in the initial analysis of the results. One such script has been particularly useful, as it allows to feed a given input to a (trained) model and obtain a visualizations of the corresponding output, along with the original input itself and the associated label (in this case, the mask). Although this tool does not provide any numerical result, its visual analysis proves highly practical. It offers an intuitive understanding of the model's behavior, highlighting its current limitations and identifying any issues encountered. Such visual insights can be particular convenient for gaining a deeper understanding of the model's performance.

Analyze results. This script serves as the primary tool for analyzing the results of my experiments. It is a Jupyter notebook that encompasses most of the necessary analysis steps following each experiment. The script is based on a similar tool developed by Broillet (2022). The notebook offers a user-friendly graphical interface with radio buttons, enabling the selection of a desired experiment. Upon selection, the script loads the corresponding CSV file generated by Typhon, containing all the relevant metrics.

The user can further customize the analysis by selecting specific datasets and metrics of interest. For each combination of dataset and metric, the script generates plots depicting the metric's evolution over the epochs. Additional parameters, such as starting and finishing epochs, allow for further customization, such as focusing on specific intervals. Various options for titles, labels, and other visual elements are also available. The resulting plots are saved in the experiment's output folder. Figure 2.2 shows an example of the plots that can be generated with this tool.

In the second part of the notebook, a histogram plot showcases the final values of each metric for each dataset. By including a column with target values, the model's performance becomes immediately evident, allowing for easy comparison against the baseline. This tool proves invaluable for gaining an overview of the current state of the models, and having all those analysis automatically performed in a single notebook is crucial when performing tens or hundreds of experiments.

Plot with confidence interval. The tools highlighted above all focus on one experiment at a time. However, I also needed to evaluate the performance across multiple runs to ensure that a favorable outcome is not simply due to chance. To address this, I developed a new script that is similar to the previous one, but this time it generates plots by merging data from multiple experiments. Again, it consist in a jupyter notebook.

To accomplish this, it loads the individual CSV files into separate Pandas dataframes. These dataframes are then merged, allowing to combine the results from multiple experiments into a single dataset. In the resulting merged dataframe, it calculates the average for each epoch, which serves as the primary line in the plot. Additionally, it incorporates slightly transparent lines to represent the standard deviation, providing insight into the stability of the results. This visualization approach is commonly employed when showcasing the outcomes of multiple experiments, as it helps to demonstrate the consistency and reliability of the observed improvements, ensuring that they are not solely the product of fortuitous occurrences.

[3] https://pypi.org/project/brutelogger/

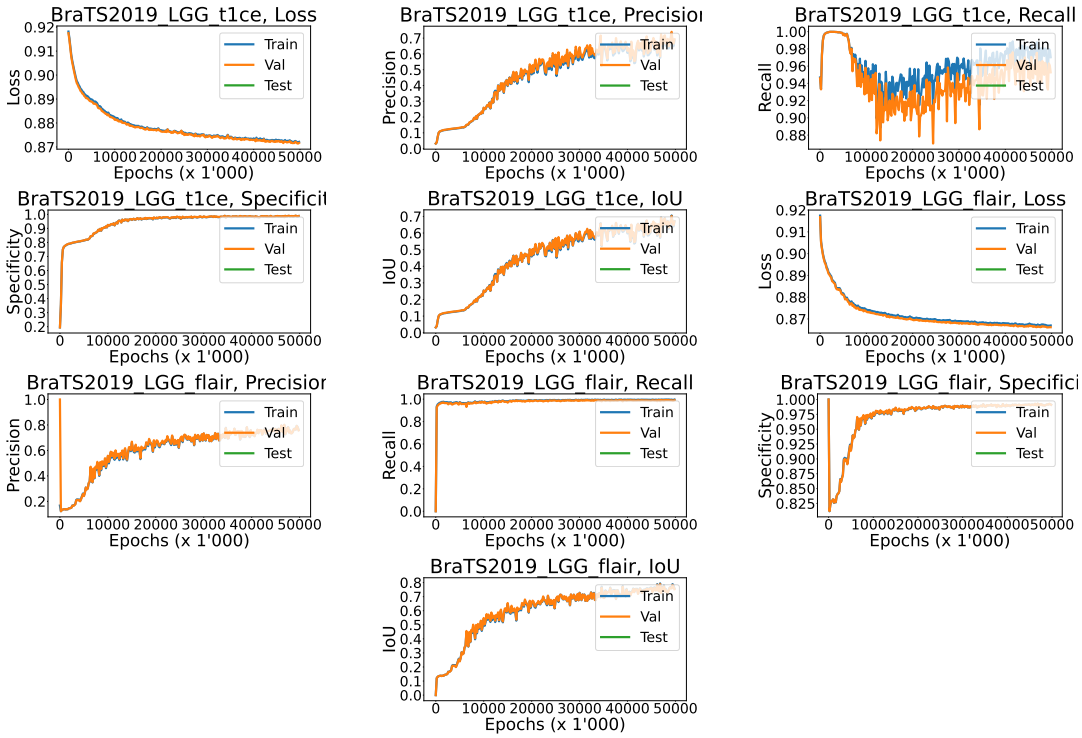


Figure 2.2: **Example of results visualization.** For each dataset and each metric selected, a graph with the evolution of the metric is plotted. Various customizations are possible, such as moving or changing the legend, hiding the test plot, and others.

Compute metrics on multiple runs. Similarly, I required a tool to merge multiple sets of results. This task is more complex than simply loading multiple files, as the tables containing the metrics have multiple dimensions. Each table encompasses dimensions for the dataset, split (train, validation, and test), and metric. Using a procedure similar to the previous tool, this notebook effectively merges these tables, calculates the average and standard deviation, and enables the creation of tables such as the one presented in Table 2.1.

Plot training and specialization. Another requirement that could not be easily fulfilled using my primary results notebook was the generation of a plot illustrating a training with Typhon followed by a classical training (specialization). I developed another Jupyter notebook that enables the user to select two experiments. It then loads the files containing their respective metrics and concatenates them. Subsequently, it plots the combined evolution of the metrics for each combination of metric and dataset. This tool is essential for comprehending the impact (or lack thereof) of an additional classical training following a Typhon-based training. However, it could potentially be used to concatenate any two experiments.

2.5 Preliminary experiments

It is important to clarify that the results presented in the following section (2.7) are the culmination of an extensive series of experiments and tests, which served as the building blocks that led to the final outcomes. They were crucial in establishing the foundational aspects of my study and testing the hypotheses I formulated. Various factors were explored, such as different datasets and subsets (e.g. comparing T1 series versus T2 series of the BraTS2019 dataset to determine the best features as support data), the selection of metrics for the bootstrap process, the configuration of the bootstrap itself, and the determination of the optimal point to

split the model into the feature extractor and decision maker components. Additionally, I also explored some values for the hyperparameters, although no extensive study has been done in this direction.

Given the impracticality of exhaustively testing all possible combinations of factors, I proceeded empirically by leveraging the insights gained from each experiment to guide the design of subsequent ones. In this section, I aim to organize the experiments into meaningful clusters, sharing common objectives from which valuable insights were derived. It is important to note that the intention of this section is not to provide an exhaustive account of every individual experiment conducted, but rather to provide a high-level overview of the journey that led to the final results. Undoubtedly, there were numerous other experiments that did not yield favorable outcomes, were interrupted, or were inadvertently omitted. Acknowledging every single experiment would render this section long and disjointed. Nonetheless, it is important to recognize that many of these "failed" experiments played a vital role by indicating paths that should be avoided.

Furthermore, a significant number of experiments were conducted to implement and verify the additions discussed in Section 2.1. While these experiments could also be considered "preliminary", delving into their detailed descriptions here would not yield any substantial benefits. It is sufficient to acknowledge that for each addition or modification in the code, corresponding experiments were executed to aid in their development and subsequently ensure their proper functioning. These intermediate experiments were foundamental in the overall progress of this thesis.

2.5.1 Batch 1

Design. The primary goal of the initial step was to gain a comprehensive understanding of the available datasets and determine their compatibility with one another. Typhon leverages the advantages of training with diverse datasets, as their utilization enhances the feature extractor's generalization. This capability allows the feature extractor to identify features that are generic and can be beneficial for multiple decision makers. However, it is crucial to ensure that the datasets share at least some common features. For instance, attempting to employ a dataset of skyscrapers as a support dataset for a cancer detection task would likely prove inefficient. This is due to the significant dissimilarity between the visual features of the images, with skyscraper images primarily featuring straight lines and corners, while cancer images are characterized by variations in density gradients.

Outcome. I began my experiments by testing different combinations of datasets. Initially, I explored using multiple sequences from the BraTS2019 dataset in combination with the TCGA-LGG dataset, as both containes MRI images of brains. Concurrently, I also evaluated the combination of UDIAT and BUSI datasets, both consisting of breast ultrasounds. Other datasets such as Head-Neck-PET-CT (Vallières et al., 2017) and LIDC-IDRI (Armato III et al., 2011) where also considered and tested. In these initial tests, I observed that although I occasionally achieved better results than the baseline, these results were not consistent. This can be attributed, in part, to the absence of bootstrap initialization; however the high similarity between the datasets used also plays a crucial role, limiting the benefits offered by Typhon. Indeed, when I incorporated samples from all four datasets, I started to observe more promising outcomes (albeit still only occasionally). This further highlights the importance of a proper initialization, particularly when the datasets exhibit less similarity.

Another noteworthy observation from these experiments was that there was no significant difference between using the HGG or LGG sequences in the BraTS2019 dataset, with latter having a smaller size (though still in the order of thousands) and being therefore more efficient to use. Subsequent experiments also indicated that the "flair" sequence yielded the best results.

2.5.2 Batch 2

Design. Although the previous set of experiments provided some valuable insights, the results exhibited significant fluctuations. Even when conducting two nearly identical experiments or duplicating a single experiment, the outcomes could vary considerably. I hypothesized that these discrepancies were to attribute to the initialization process. Gradient descent algorithms aim to converge to the nearest local minimum, and as a result, different initializations can yield divergent results, see Figure 2.3 for an example illustrating two different gradient descent initializations and their convergence paths. This phenomenon explains why in Deep Learning the same model is often trained multiple times to obtain better results. Consequently, I considered necessary to conduct a thorough examination of the bootstrap approach to assess its actual effectiveness and determine the optimal duration required for reliable results.

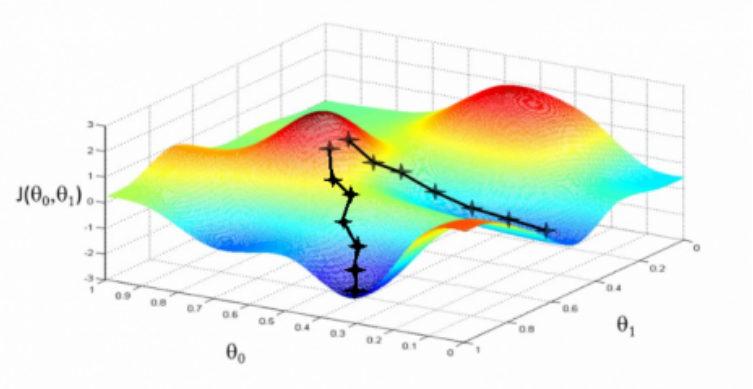


Figure 2.3: **Two gradient descent paths with different initializations.** The path of gradient descent in optimizing a function with two variables (which could be two weights of a neural network, the parameters of a linear regression, or any other variable). Although starting from a similar configuration, the non-convex shape of the objective function leads to very different results.
Source: Stanford's Andrew Ng's MOOC Machine Learning course.
(https://www.coursera.org/learn/machine-learning)

Outcome. I conducted multiple experiments with and without the bootstrap initialization. For the experiments involving the bootstrap, I tested various lengths, specifically 500 epochs, 2000 epochs, and 5000 epochs. It was immediately evident that the inclusion of bootstrap initialization led to result stabilization and overall improvements. While this outcome was expected and had been previously mentioned in Cuccu et al. (2022), it was crucial to validate its impact in my specific segmentation scenario.

Simultaneously, I observed that increasing the length of the bootstrap did not proportionally enhance the model's performance. Transitioning from 500 to 2000 epochs yielded only a marginal average improvement, and extending from 2000 to 5000 epochs had a negligible impact. This behavior had also been hypothesized beforehand.

It is important to note that the precise values of these results are highly specific to the current experimental settings, particularly the datasets and the model used. Consequently, they cannot be directly applied to other experiments. Nevertheless, these findings underscore the significance of the bootstrap initialization and the importance of determining an appropriate size to avoid wasting excessive time with negligible returns.

2.5.3 Batch 3

Design. The natural next step in my investigation of the bootstrap process was to explore the evaluation of different initializations for a segmentation task and how to compare their performance. While the code already incorporated the requirement that an initialization should perform well on multiple datasets to ensure the generalizability of the extracted features, the definition of "performing well" for a dataset remained to be determined. The original implementation utilized the AUC score as a metric; however, I made the decision to experiment with this parameter and explore alternative metrics to assess the quality of the bootstrap initializations.

Outcome. I opted to utilize the metrics presented in Section 1.2.2. While more sophisticated systems could be employed, such as combining multiple metrics, I deemed it unnecessary, especially considering that some metrics already encompass a combination of others and provide an overall performance score. Consequently, I conducted tests using recall, IoU, and Dice Score as metrics to assess the quality of the bootstrap, one at a time. Recall is arguably one of the most critical metrics in cancer detection, as failing to detect a tumor is worse than erroneously detecting one that does not exist. The other two metrics, IoU and Dice Score, try to find a balance between false positives and false negatives, taking into account the overall performance.

As expected, employing recall did not yield the best results. This is probably a consequence of its blindness to false positives. The outcomes of using Dice Score and IoU were instead comparable and better, with Dice Score exhibiting a slight advantage. I attribute this to the loss function employed during training (Dice Loss), which directly aligns with the Dice Score metric. Consequently, I established the use of Dice Score as the preferred metric for the evaluations of the bootstraps.

2.5.4 Batch 4

Design. Although the main research objective did not revolve around achieving optimal results on a specific dataset, but rather on evaluating the impact of integrating the Typhon meta-learning framework into an existing procedure, I conducted experiments to examine certain hyperparameters. The focus was primarily on identifying an optimal learning rate for specific datasets, or at the very least, determining the impact of this parameter. Furthermore, I investigated the potential advantages of employing distinct learning rates for distinct dataset (trained concurrently with Typhon), in order to determine if such an approach could lead to significant improvements.

Outcome. I conducted experiments to test multiple learning rates, ranging from $1e-8$ to $1e8$, while training with the four datasets mentioned in Section 1.5. As expected, models trained with very small learning rates (smaller than $1e-5$) required a substantial number of epochs to converge. Conversely, using a learning rate greater than $1e-1$ resulted in unstable training, caused by the drastic changes introduced at every training step with one batch of a dataset, which are immediately reverted by the subsequent batch. This issue is further exacerbated by Typhon's utilization of small batches, where the individual information learned at each step is less generic.

However, I also observed that varying the learning rate within the range of $1e-5$ to $1e-1$ did not have a significant impact on the final results. While these findings are not definitive (as there are other factors that could influence the results, such as the model split, initialization method, and batch size), they suggested that Typhon is robust to variations in this parameter.

2.5.5 Batch 5

Design. Optimal positioning of the split is crucial for leveraging the advantages of Typhon effectively. Placing the split too early in the model can undermine the benefits offered by Typhon, while positioning it too late may not provide sufficient computational complexity for the decision makers to adapt to their specific datasets. Determining the precise location of the optimal split point is a complex task that depends on various factors, including the model structure and the level of similarity among the datasets. Unfortunately, there is no definitive guideline for determining the ideal split point. I therefore conducted multiple experiments to identify the most suitable split point for my specific case.

Outcome. At the current stage, I were working with both UNET and RF-Net architectures. I aimed to find the optimal split for each architecture that would best adapt to my specific situation and datasets.

For UNET, I created four versions with different splits. Considering my expectation that the feature extractor would be more complex than the decision maker, similar to the findings in the classification task in Cuccu et al. (2022), I performed all the splits in the "decoder" part, which comprises four decoding blocks. Specifically, I created splits between the first and second block, second and third block, third and fourth block, and after the last block, in this case leaving only one convolution in the decision maker. Figure 2.4 illustrates the tested splits. After conducting multiple experiments, I determined that the most effective separation point is after the first decoding block.

Similarly, for RF-Net, I designed three versions with different splits, taking into account the insights gained from the UNET experiments. I decided to split the model before the first block in the decoding part (in the middle of the model), just after it (similar to the best split found for UNET), and after the second block. Once again, I observed that having a feature extractor consisting of the entire encoder and one block of the decoder yielded the best results in my situation. Figure 2.5 shows these different split possibilities.

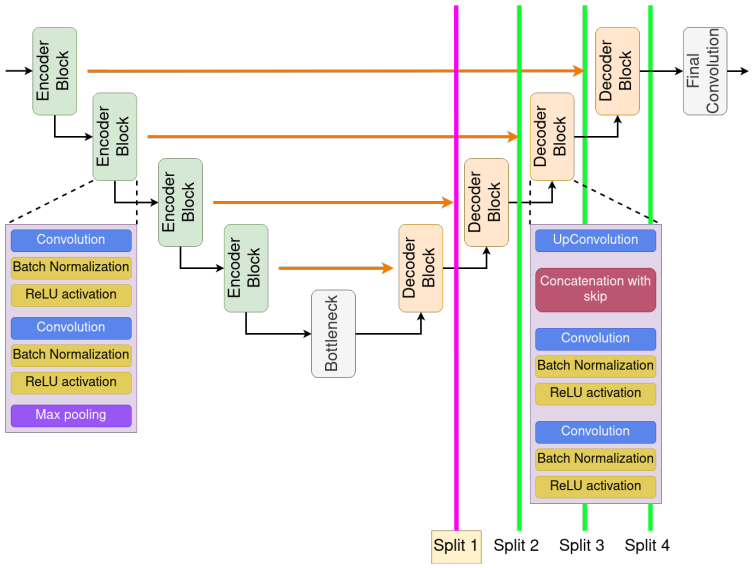


Figure 2.4: **Tested splits for UNET.** The green vertical lines represents the tested splits into feature extractor and decision maker. The pink line shows where the best split was found, i.e. between the first and the second decoder blocks. Orange lines represent skip connections.

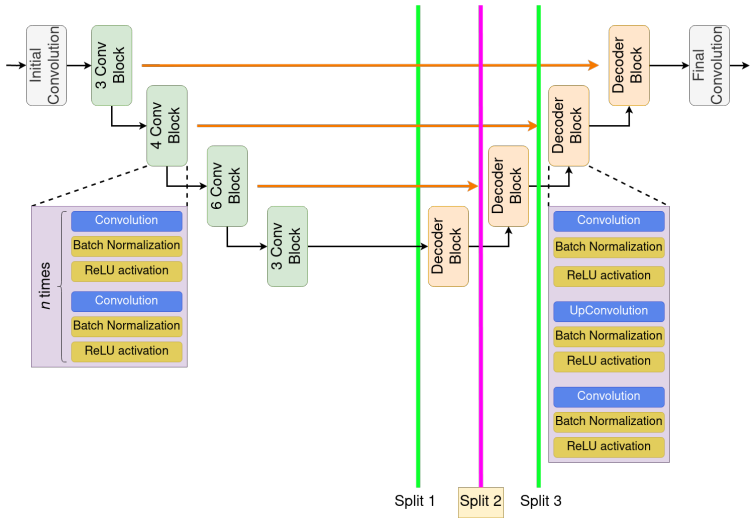


Figure 2.5: **Tested splits for RF-Net.** The green vertical lines represents the tested splits into feature extractor and decision maker. The pink line shows where the best split was found, i.e. between the first and the second decoder blocks (similar as UNET). Orange lines represent skip connections.

2.5.6 Batch 6

Design. Despite achieving increased stability in the results by determining the appropriate metric and length for the bootstrap, I still observed variations in the outcomes. This problem was expected, considering that the bootstrap parameters are highly dependent on the datasets employed, the model architecture, and, notably, the position of the model's partitioning into the feature extractor and decision maker components (the model split). Consequently, I deemed necessary to reevaluate the bootstrap process. Using this opportunity, I also decided to explore a new approach to the bootstrap, aiming to eliminate the reliance on an absolute value that determines the maximum discrepancy between the best and worst-performing models. Ideed, the previous hardcoded approach quickly becomes obsolete when the employed metric changes.

Outcome. I conducted further tests on the bootstrap length using the new architectures. These experiments were considerably longer compared to the previous ones due to the use of datasets that were two orders of magnitude larger. To accommodate this, I modified the code to randomly sample from the training and validation datasets when evaluating a particular initialization (as described in 2.1). Since I did not conduct an exhaustive study to determine the statistically representative sample size for each dataset, I opted for a larger sample size (at least 1/3 of the full dataset for the larger ones and the entire datasets for the smaller ones) to be safe and avoid introducing biases. Still, this adjustment already reduced the bootstrap time.

The second change I implemented was the requirement for the difference between the best and worst decision makers to be smaller than the previous best initialization, in order to accept a new initialization as better. Previously, the requirement was that the difference had to be smaller than 0.2. While this criterion was acceptable when using AUC for classification tasks (where the random guessing, if the classes are balanced, would score 0.5), it is meaningless in the context of segmentation. My modification, although not perfect, is agnostic to the metric used and eliminates the need for manually selecting a threshold.

2.5.7 Batch 7

Design. As highlighted in the introduction of this section, the experiments conducted encompassed numerous varying parameters. I explored different support datasets, bootstrap procedures and lengths, hyperparameters, and even variations in model architecture. However, optimizing these variables in isolation is insufficient. The optimal value for a parameter when considered independently may not hold true when other variables are altered. This challenge is further compounded by the fact that while the bootstrap partially stabilizes the initialization process, chance can still influence outcomes, leading to potentially misleading insights regarding specific parameter sets.

Consequently, I reached a point where it became necessary to examine the combination of multiple parameters. I conducted experiments involving alterations of the support datasets, bootstrap size, and training duration. I used the values determined in earlier experiments as starting point, hoping that the optimal values were not too distant from the previous findings.

Outcome. Once I had established a satisfactory initialization mechanism, a suitable metric, an optimal model split, and an overall understanding of the hyperparameter values, I conducted comprehensive tests by combining all these elements. I performed new experiments by varying the bootstrap size, training multiple models with the same initialization, and allowing the models to train for longer periods. I also tested the hyperparameters used in the comparison paper (Wang, Liang, and Zhang, 2021) and refined all the variables in preparation for the experiment presented in the next section. While this phase may appear superfluous, it was crucial to integrate and validate all the components, and it required a significant amount of time and effort.

2.6 Final experiment setup

In order to assest the benefits of using the Typhon meta-leaning framework, I compare its performance again RF-Net on the UDIAT dataset. Since I could not get access to part of the data used in the original training, I reproduced the experiment with the data I had, leading to some discrepancies between the published results and the reproduced ones. This is however not important, since the goal was to study the benefit of integrating the Typhon architecture. I decided to use the standard Dice Loss and Adam optimizer (as in RF-Net), and to keep a learning rate of $2e-4$ with a batch size of 8 (16 for the run reproducing the RF-Net results, as in the original paper).

2.6.1 Hardware

All experiments were conducted on a machine equipped with an Intel(R) Xeon(R) 6142 CPU featuring 64 cores running at 2.60GHz. The machine had 6GB of RAM per core, and it was equipped with 8 NVidia Tesla V100-SXM2 GPUs operating at 1.3GHz, each with 32GB of HBM2 VRAM. Although the hardware used in my setup is not state-of-the-art and most components are over 5 years old, the experiments were still able to run effectively due to the utilization of small batch sizes and the high sample efficiency that are characteristic of Typhon applications.

2.6.2 DevOps

To facilitate the experiment procedures, I employed several additional tools. Firstly, I utilized rsync for efficient data transfer across the servers. Rsync, which is based on SSH, considers the last modification timestamp of a file to determine whether it needs to be transferred again.

This feature enables to interrupt the transfer of large datasets and resume it at a later time, providing flexibility and efficiency.

The second essential tool I utilized is tmux. Tmux allows to continue running a command even after disconnecting from the server, which is crucial for long-running tasks like the ones in this study. Additionally, tmux enables the convenient management of multiple bash sessions within a single SSH connection. I leveraged this capability to simultaneously handle multiple experiments, taking advantage of the availability of multiple GPUs.

2.6.3 Model

In a segmentation task, the objective of the model is to generate a mask of the same size as the input, indicating, for for each pixel, whether it belongs to a malignant mass or not. This precise localization and segmentation of masses provide crucial support to radiologists. Segmentation is considerably more challenging than classification, as it requires the model to maintain accurate location information throughout the process. State-of-the-art models such as UNET and its derivatives often rely on *skip connections* (see Section 1.3.2) to propagate information from the original input. In order to compare with RF-Net (Wang, Liang, and Zhang, 2021), I also utilize a derivative of UNET, featuring a ResNET34 as the encoder and an additional convolution block at the beginning. Adapting this architecture to Typhon requires careful handling of the split connections between the feature extractor and the decision makers. However, once this is properly addressed, no further modifications are necessary.

An important decision to make is the point at which to split the model, separating the portion dedicated to the feature extraction and the part utilized by the decision makers. If the split is too early, the benefits of Typhon may not be fully realized. In the extreme case, the split occurs before the model, resulting in completely separate models for each dataset. On the other hand, if the split is too late within the model, it limits the capacity of the decision makers. Although the feature extraction part may have more flexibility, the decision makers could be significantly constrained and potentially not complex enough to handle the task, leading to a sub-optimal model.

Figure 2.6 shows a version of RF-Net adapted for Typhon.

2.6.4 Specialization

In the predecessor of Typhon, Hydra (Cuccu et al. (2020)), a method called “specialization” was introduced. This approach involves performing an additional round of classical training on top of the best model achieved during the transfer learning phase. The purpose of this specialization step is to further refine the model’s performance. However, Typhon’s remarkable capability to mitigate overfitting raises the question of whether this additional specialization is truly necessary. It is plausible that Typhon’s inherent capacity to learn all the salient features of the data during the initial phase renders the supplementary specialization step redundant.

Despite preliminary indications supporting the hypothesis of additional specialization’s ineffectiveness, I chose to proceed with testing. I therefore conducted an extensive classical training phase atop the best model achieved through Typhon training. Given the capabilities of Typhon to contrast overfitting, the best model typically emerged from the later stages of training.

2.7 Results and Discussion

In this section, I present the results obtained and provide a comprehensive analysis of their significance. The focus will primarily be on the outcomes of the final experiment, representing

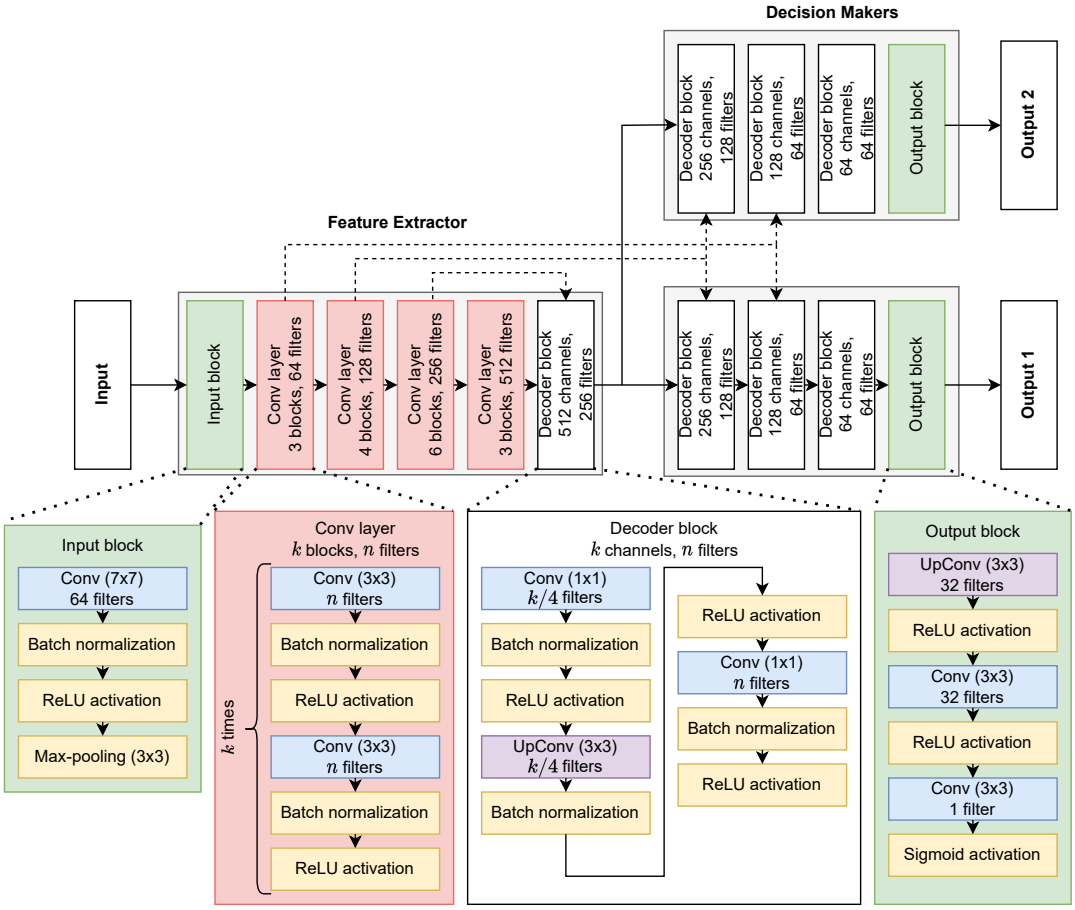


Figure 2.6: **Network architecture adapted to the Typhon framework.** The original RF-Net model (Wang, Liang, and Zhang, 2021) is split into a *feature extractor* and a set of *decision makers* (two two in the example). Particular care is given to maintain residual connections (dashed lines) between the two parts. The training for each one dataset only sees one end-to-end, monolithic model constituted by its specific decision maker coupled with the shared feature extractor: output 1 is computed based on an image from dataset 1, and in that case the decision maker for output 2 is not activated.

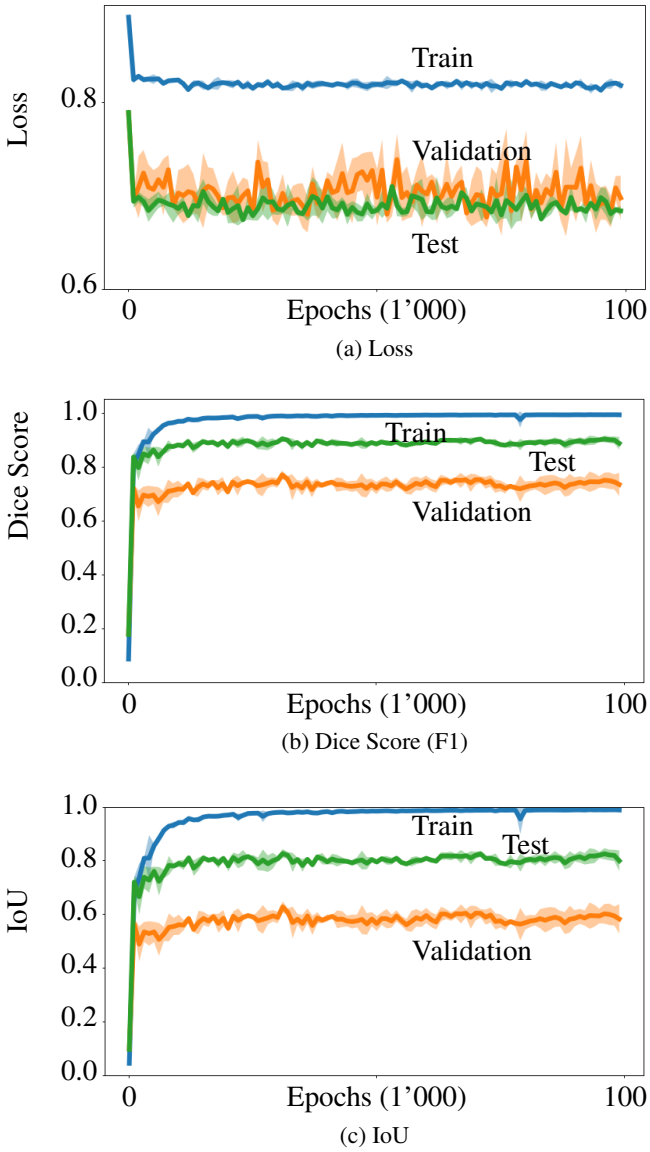
the culmination of my research efforts. By thoroughly examining various aspects, I aim to understand the implications and impact of these results.

2.7.1 UDIAT segmentation

In this part, I will analyze the results obtained from the final experiments. I will demonstrate how Typhon effectively enhances the feature extraction capabilities of a model, leading to improvements in the overall performance. Furthermore, I will highlight other notable advantages of Typhon, including its ability to mitigate overfitting. I will discuss the enhanced sample efficiency achieved through the utilization of Typhon and showcase how it effectively learns all available information, rendering an additional specification unnecessary.

Figure 2.7 provides a comprehensive overview of the training performance specifically on the UDIAT dataset, which serves as target dataset.

Runtime analysis. To ensure a consistent and fair comparison of runtimes, each run was restricted to using only one GPU. The reproduction of the RF-Net baseline took approximately 4 hours, from which 15 minutes were dedicated to computing the metric values. On the other hand, for Typhon, I initiated the training process with a bootstrap of 22 hours. The number of



(a) Loss

(b) Dice Score (F1)

(c) IoU

Figure 2.7: **Training performance.** Loss, Dice (F1) and IoU values collected during training on the train, validation and test sets. Test and validation follows similar patterns as training, the main difference in performance can be expected based on variance in the tiny dataset of just 163 samples. Performance on test and validation datasets does not deteriorate after arbitrarily long training, indicating that **overfitting does not set in**. Plots are averages over five runs, with (tight) error bands.

epochs to evaluate during the bootstrap, which determines its runtime, is left to the discretion of the user, allowing for the exploration of longer durations to potentially enhance the initial performance. I observed that the peak performance of the bootstrap was attained after the first 10 hours.

Moving to the parallel training phase, Typhon necessitated approximately 16 hours to complete, of which 5 hours to compute the metrics, effectively reducing the actual training time to 11 hours. The variance in metric computation arises primarily due to the contrasting sizes of the support datasets, with two of them being **two orders of magnitude larger** than the UDIAT dataset.

It is crucial to emphasize that the reported results already incorporate the performance

improvements achieved through the optimized metric computation. Without these optimizations, the total runtime would have been significantly longer.

Sample efficiency. Sample efficiency refers to the capacity to extract maximum information from individual samples. For instance, humans can grasp the rules of tic-tac-toe after observing just a few matches, exemplifying remarkable sample efficiency. This ability holds significant importance for Deep Learning models operating in scenarios where the samples are limited or costly to obtain. For instance, a reinforcement model designed for autonomous driving cannot afford to crash thousands of vehicles before learning to brake effectively. Similarly, in the medical domain, models typically encounter limitations in accessing extensive datasets due to privacy regulations. In these and numerous other situations, the ability to derive valuable insights from each individual training instance becomes crucial, establishing sample efficiency as a fundamental characteristic.

To verify that Typhon training is able to counter overfitting, I let training run for much longer than strictly necessary. The length of the training was choosen to let Typhon use the same number of training points as the baseline, i.e. the reproduction of RF-Net results on the UDIAT dataset, which reached top performance after using roughly 800'000 examples. To better understand the sample efficiency, it is important to remember that the UDIAT dataset only have 163 images. Typhon reached peak performance after roughly 248'000 examples from the target dataset, with additionally 744'000 examples from the support datasets, summing to 992'000 images seen after 31'000 epochs. At that point, Typhon had learned 4 datasets in approximately 5 hours, proving a sample efficiency of one to two order of magniture higher than classical learning on this task, depending on whether we considered it only for the target dataset or for the support datasets as well.

Segmentation results. Table 2.1 shows the results on the UDIAT dataset. The values presented are the average over 5 runs, $\pm$ standard deviation. Since I did not have access to the full dataset used to train the model on the original RF-Net paper, I had to split the UDIAT dataset itself (which was the test dataset in the paper) and use a part of it for training and a part for testing. This is the reason why the table presents the rows *RF-Net (published)* and *RF-Net (reproduced)*. While this means I do not have the exact same conditions, the goal is to show the benefit of Typhon on any starting setting. For this reason my analysis focuses on the comparison between the *reproduced* results and the results obtained with Typhon.

It is immediately evident that the utilization of the Typhon framework enhances performance across most metrics compared to the baseline. The only exception appears to be Precision. However, it is crucial to note that this metric exhibits the highest volatility, with a standard deviation of 0.86%. Consequently, precision varied across experiments, being better in some instances while worse in others. Thus, additional metrics must be considered to obtain a more comprehensive understanding.

Particularly important is the recall. While misidentifying a malignant mass where there is none is problematic, failing to identify one when it exists can be catastrophic for the patient. Recall measures the proportion of pixels correctly identified as tumors out of the total pixels representing tumors. My experiments yielded a substantial increase in recall, surpassing the reproduced baseline by more than 7%. This finding demonstrates Typhon's remarkable capability to generalize from diverse datasets, leading to a significant improvement in malignant mass detection.

The benefits of Typhon are further validated by the results obtained for the last two metrics. I observed an improvement of nearly 3% in Dice Score (also known as F1 score). The Dice Score combines recall and precision, considering both false positives and false negatives. It is commonly used to evaluate segmentation tasks, and the improvement in this metric underscores the success of my experiments.

Lastly, I achieved an improvement of over 4% in Intersection over Union (IoU), also known as the Jaccard index. This result is not particularly surprising since IoU is closely related to the F1 score, utilizing a different measure to compute the same parameterthe overall similarity between the prediction and the ground truth.

The improvement in this final metric further emphasizes the advantages of Typhon across the overall score, not limited to any specific aspect. I argue that these results are largely attributable to Typhon's ability to generalize, derived from its optimal utilization of multiple and diverse datasets. A more generalized and effective feature extractor can accommodate various inputs, exhibiting higher generalization capabilities and the ability to extract meaningful information from previously unseen samples.

There is however another aspect that impact the results of Typhon, which is its ability to counteract overfitting and therefore continue to learn until full convergence.

Overfitting results. Overfitting poses a significant challenge in modern Deep Learning. As the complexity of a model only sets the lower bound for the function it can approximate, there is a tendency to construct increasingly larger models, often exceeding what is necessary for the given problem. Consequently, during training, the model may reach a point where it begins to memorize the training samples rather than extracting meaningful information to facilitate generalization. This outcome manifests as a model with impressive performance on the training set but performing poorly when faced with unseen data, which is the true objective.

The conventional method to address overfitting involves setting aside a subset, known as the validation set, which remains untouched during the training process. This set is periodically evaluated, and when the model's performance begins to decline (despite still improving on the training set), it indicates the emergence of overfitting, where the model starts to memorize the training samples. At this point the training process is suspended (*early stopping*) (Prechelt (2002)). However, and this is critical, this approach does not ensure that all available information has been fully learned. Rather than actively preventing overfitting, the focus is on acknowledging its insurgence and terminating the training before it compromises the training so far.

From the performance graph (Figure 2.7) one can see that the model trained with Typhon does not suffer from overfitting. I let the training run much longer than required, and even though top performance was already reached after 31'000 epochs, I let it run for 100'000 epochs. Additionally, the last row in Table 2.1 shows that not the single data points have been memorized, leading to all metrics being maximized. However, this did not lead to a degradation on the test set, showing that the generalization's abilities of the model where not affected.

These results warrant further investigation as they provide strong indications that the parallel transfer within Typhon can effectively alleviate or even completely counteract overfitting. Previous research did not extensively explore this approach due to inherent challenges with parallel transfer (as introduced in Section 1.6.5). However, with the solutions introduced in Typhon, this approach holds significant potential to impact modern Deep Learning. The benefits of addressing the overfitting problem are major, emphasizing the need for further investigations into these results.

Specialization. In the implementation from which Typhon derived, Hydra (Cuccu et al., 2020), after the training on all the datasets there was phase of dedicated training, for which each dataset (or at least the target) was trained individually using classical training. This feature has been inherited from Typhon, which offers the possibility to train a copy of the final model (where also the feature extractor is dedicated to one single dataset and not shared anymore). The fact that Typhon counteracts overfitting and allow the training to learn all the useful informations however strongly hinted that this part was not necessary.

I therefore decided to run a classical training on top of the best model trained with Typhon. From this, I obtained two main results, depicted in Figure 2.8: (i) no additional information is learned during this specialization training, making it superfluous; and (ii) also this additional training do not seem to lead to overfitting. To my understanding, this is provoked by the strong generalization provided by the feature extractor, and by the fact that the model has at the same time a perfect performance on the training set. This is particularly interesting because it shows that the training points have been fully memorized, however there is no loss in the model's ability to generalize.

	IoU	**F1/Dice**	**Specificity**	**Precision**	**Recall**
RF-Net (published)	73.09 $\pm$ 0.64	81.79 $\pm$ 0.76	n/a	78.61 $\pm$ 0.97	90.07 $\pm$ 1.00
RF-Net (reproduced)	79.73 $\pm$ 0.97	88.72 $\pm$ 0.60	99.36 $\pm$ 0.48	95.82 $\pm$ 0.07	78.64 $\pm$ 3.32
Typhon	**84.05** $\pm$ 0.63	**91.33** $\pm$ 0.37	**99.61** $\pm$ 0.07	95.08 $\pm$ 0.86	**87.87** $\pm$ 0.72
Typhon (on train set)	98.92 $\pm$ 0.11	99.46 $\pm$ 0.06	99.98 $\pm$ 0.0	99.48 $\pm$ 0.04	99.44 $\pm$ 0.08

Table 2.1: **Performance on the UDIAT dataset.** Adapting RF-Net into the Typhon framework improves performance across all metrics without further tuning. The first three rows present the results on the unseen data of the test set; the last row, added for perspective, presents the results on the *training set*, highlighting the ability of parallel transfer learning to fully learn the dataset until convergence, thanks to the mitigation of overfitting.

2.7.2 Typhon framework

The initial implementation of Typhon was already highly customizable, offering a wide range of hyperparameters and the ability to easily adapt to various datasets, loss functions, optimizers, and more. However, its implementation was limited to classification tasks, which significantly restricted its applicability.

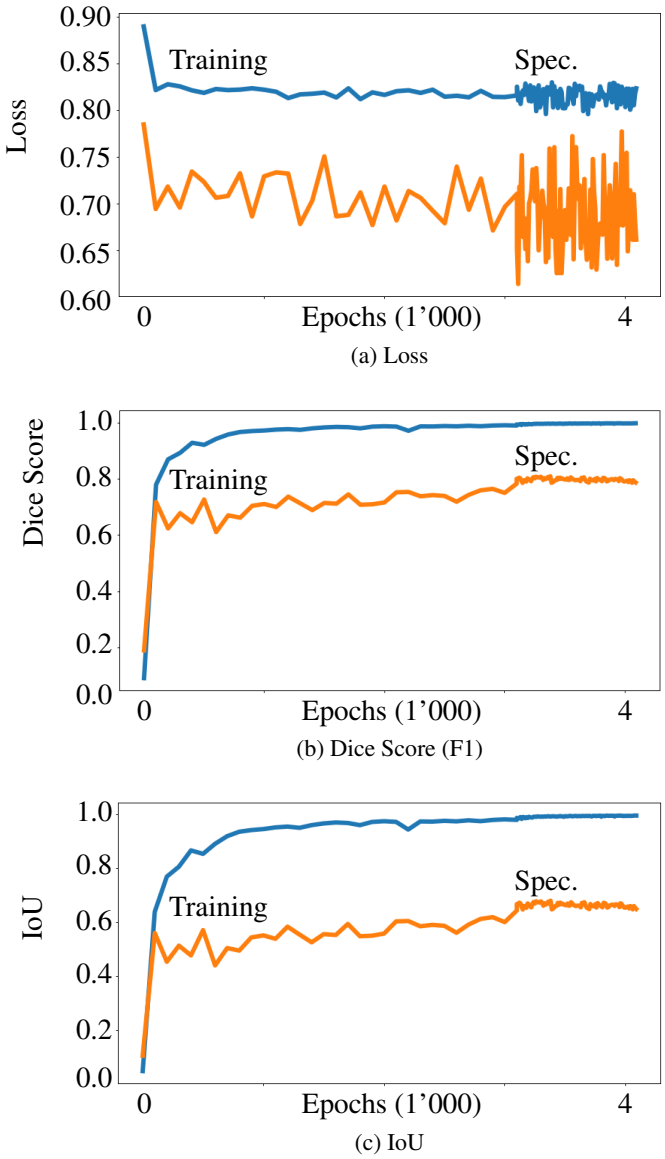
Throughout this thesis, I expanded the implementation of Typhon to work with segmentation by incorporating several modifications and additions. Firstly, the code now includes a brand new loader that directly extends a PyTorch class and inherits its main functionalities, but have been adapted to suit my specific needs.

The initialization process, known as the bootstrap, has not only been adjusted for segmentation but also strengthened to enhance stability. It no longer relies on fixed (hardcoded) thresholds but instead dynamically adapts to the situation, resulting in an improved initialization. This is crucial, as initializing the feature extractor to effectively capture general features is fundamental in overcoming the moving target problem.

Furthermore, the general error computation loop has been tailored to accommodate segmentation. This adaptation was indispensable since a segmentation model produces an output that represents each pixel of the input, rather than a single value for classifying the entire image. Alongside the implementation of a dedicated loss function for segmentation, these modifications have enabled effective training of the model for this new task.

In addition, I have integrated different versions of UNET and RF-Net (presented in Section 1.3.2) into Typhon, ready for use. For both architectures, multiple splits in feature extractor and decision maker are now available. I have also conducted tests to determine which versions yield superior performance, although these results may vary when applied to different datasets, especially those that exhibit significant differences in similarity.

During this thesis, I also made improvements to various aspects of the Typhon codebase, which were not essential for my segmentation task but significantly enhanced the framework



(a) Loss

(b) Dice Score (F1)

(c) IoU

Figure 2.8: **Specialization.** At the end of the training, I extract the monolithic model for the target dataset and resume classical training. Curves represent performance on the training (top, blue) and validation (bottom, orange) sets. No additional information is learned, supporting the thesis that parallel transfer learning can be prolonged until full convergence, thanks to its counteracting of overfitting.

in terms of performance, customizability, and result analysis. These modifications and additions have further improved the usability of Typhon, making it an even more powerful meta-learning framework.

One of the key modifications I made relates to the frequency of metric computation. Previously, computing the performance on the entire datasets every single epoch was acceptable for classification tasks with limited data. However, in modern Deep Learning, this approach is often infeasible due to runtime constraints. My modification opens up new possibilities and applications for Typhon, enabling it to be used in scenarios that were previously excluded due to datasets sizes or number of epochs needed.

Another modification I made concerns the way metrics are computed. Previously, all labels and outputs were stored on the GPU, and the metrics were computed at the end. While this approach worked well for classification, it poses challenges when working with segmentation tasks where labels and outputs are much larger. Storing them on the GPU often becomes impractical and memory-intensive, if not unfeasable. This also undermines Typhon's ability to work with limited memory, despite its small batch size. With my modification, this issue no longer exists, allowing Typhon to run on large datasets even with older hardware and limited memory.

To enhance Typhon's customizability and adaptability, I introduced additional functionalities. One notable addition is the ability to specify the shapes of the input images, which are then handled through padding and cropping. This feature is particularly useful when working with inputs of varying shapes, such as breast ultrasound images obtained manually. Instead of performing scaling, padding, or cropping during the preprocessing phase, Typhon now handles this preparation directly.

Lastly, I included an important addition to facilitate result analysis. Every time metrics are computed, Typhon now also loads a random input from each dataset and saves it along with the corresponding label and output in a human-readable format. This seemingly minor modification allows for instant visualization of the training progress without the need for additional tools or numerical metrics. It enables immediate assessment of whether the model is learning correctly and allows to adopt corrective measures if needed.

2.8 Limitations

It is worth noting that the model used for the breast tumor segmentation task may be considered overly complex for the specific requirements of the task. This raises the question of whether a more simplified architecture could achieve comparable results while reducing computational resources.

Regarding Typhon, it is a versatile meta-learning framework applicable to various Deep Learning applications. It is however important to notice that there is no formal guarantee that its implementation will consistently improve results across all domains. Currently, my understanding is based on empirical evidence rather than formal theoretical guarantees. Another aspect that requires further exploration is the application of Typhon to large-scale datasets. I hypothesize that Typhon's sample efficiency and ability to mitigate overfitting could yield improved performance on such datasets. However, this hypothesis remains to be validated through dedicated investigations. Finally, while Typhon exhibits robustness to small variations in hyperparameters, an additional aspect that is particular significative is that my experiments required limited hyperparameter optimization. Further research efforts in this direction, focusing on fine-tuning the hyperparameters of the Typhon framework, could uncover additional insights and enhance overall performance.

2.9 Further applications

Given the results achieved in the segmentation task, the potential to extend the adaptability of the Typhon codebase to various types of Machine Learning models became evident. In this section I will describe the application of Typhon to use an autoencoder to extract sophisticated visual features from Atari game images, then connect a policy controller on top of it to learn to play the correponding game. The primary objective is to leverage Typhon's capability to simultaneously learn from multiple environments, which in this case corresponds to learning to encode the reconstruct images from multiple Atari games with a single, multi-headed Typhon model. This approach facilitates the training of a feature extractor that can recognize

generic features which are not bounded to a single environment, and potentially useful also in unseen environments. This ability becomes particularly important given the frequent changes in colors and images encountered across different levels in Atari environments. Obtaining images for subsequent levels is challenging, as it necessitates a controller capable of progressing enough in the game. A feature extractor robust to this changement could be able to already operate only with the dataset available at the beginning.

To train the feature extractor I decided to use an autoencoder model (see Section 1.4.3). The encoder part corresponds to the *head* of Typhon (the feature extractor), while the decoder is the *body* (acting as a "decision maker"). The first part of this work is therefore to adapt the Typhon codebase to work with autoencoders, and variational autoencoders, which requires additional parameters. Subsequently, I will present the experiments conducted, dividing them into three classes. The first part, described in Section 2.9.2, will present the data collection process. Section 2.9.3 describes the different architectures tested, and present the final version. Section 2.9.4 presents a new framework I created, which merges these two parts (data collection and autoencoder training) and f a controller which will use the feature space generated by the feature extractor. Repeting these three steps allows to generate more and more advanced controllers.

During the training of the autoencoders, I realized that the main difficulty was to precisely encode the location of sprites (sub-immages that represent objects and that can be found in different location, but always in the same form. For instance, the group of pixels representing the avatar is a sprite). To the best of my knowledge, existing neural network models do not directly address this issue. I therefore implemented a new neural network layer called PixelPerfect. Although still in its early stages, I decided to conduct a feasibility study to assess its applicability, specifically by incorporating it into the autoencoder model for Atari images. The technical implementation of this new module is then presented in Section 2.9.5.

2.9.1 Typhon adaptations

Adapting Typhon to autoencoding required changes similar to those presented in regards to segmentation. The main change involved adding a custom data loader specifically designed for autoencoders. This loader is tasked with loading individual samples (without masks) and returning both the input sample and a copy of it, which is used as the label. For convenince, I also implemented the possibility to load PNG images and not only numpy arrays.

Some small additions were required in the code to ensure accurate computation of the metrics. For normal autoencoders, I used the Binary Cross Entropy (BCE) loss, which is readily available in the PyTorch library. However, for variational autoencoders, a custom loss had to be implemented to account for the additional parameters returned by the network. The inclusion of these parameters in the model's output also required adjustments in the computation of the metrics, and the corresponding code was modified to accommodate these changes.

Initially, I explored a research direction that involved removing the mode image from the dataset inputs. The idea was to focus on learning the parts of the images that change, while disregarding the static background. Consequently, I incorporated the option to perform this preprocessing step in the code, although I soon abandoned this approach in favor of more promising directions for my experiments.

2.9.2 Data collection

To train the model, it was necessary to initially assemble an appropriate dataset of images generated by different Atari games. Unlike the segmentation task previously presented in this thesis, a readily available dataset of Atari images was not available. Consequently, I had to collect such images, which required particular attention to some aspects. These images are

returned by the game environment for each action sent to it by the agent, as they are supposed to be the observation based on which to select the next action. This means that an agent is required to send actions over a span of time to collect these images. The initial selection is for a completely random agent, which uniformly selects among all possible actions. Once the images are obtained, the next step is to select only images which brings new information should be included in the dataset. It is more advantageous to have 100 unique samples rather than 10'000 duplicates, as the essence of successful training lay in learning the underlying "information" within the dataset, rather than memorizing individual elements. Secondly, if images are directly captured during gameplay, there is a potential bias in sample distribution. Early game phases would be overrepresented while later stages would be underrepresented, as the progression to the latter phases necessitated the agent to remain in the game (without losing) for a longer duration.

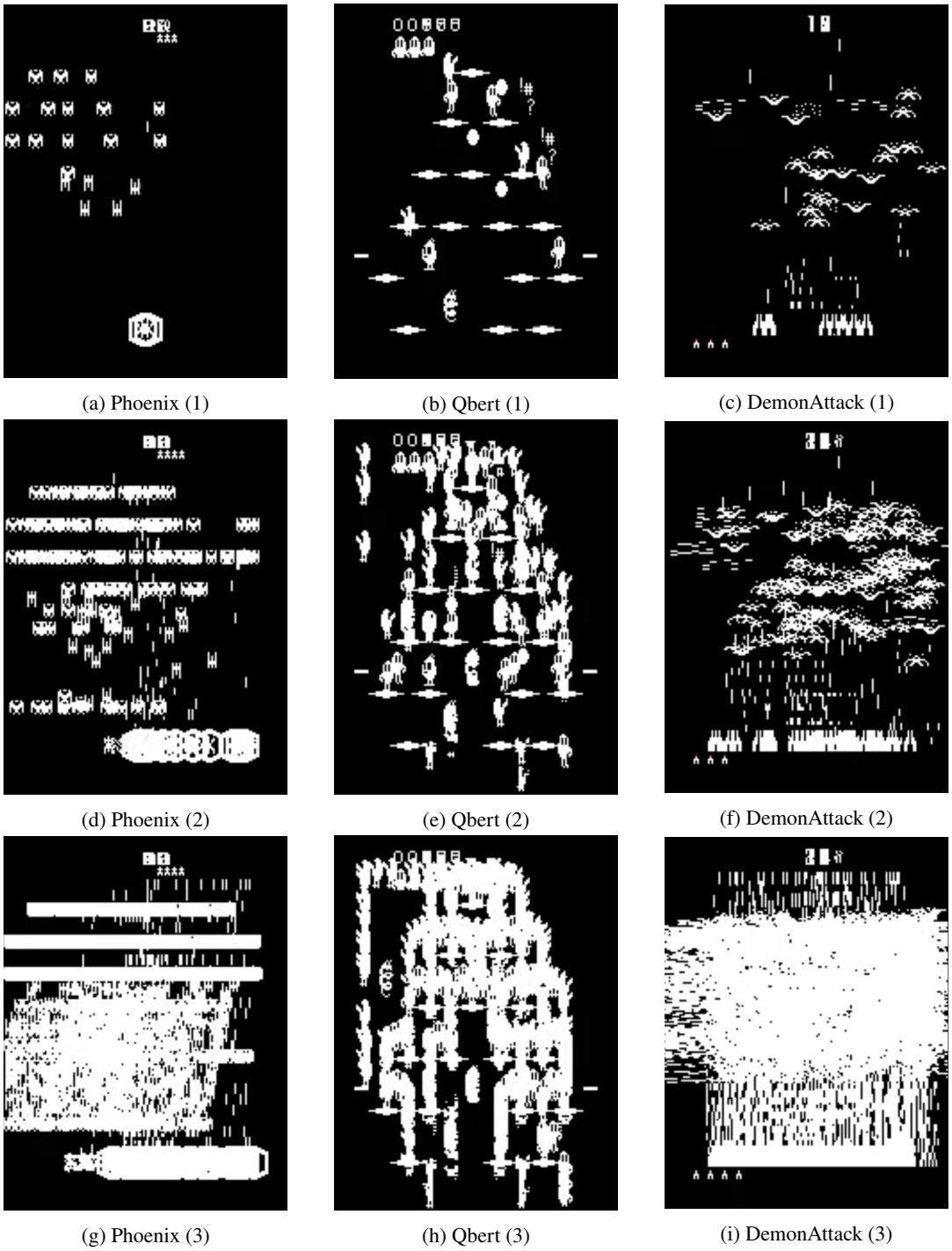
The initial approach involved playing Atari games using a random policy and saving any frame that differed significantly from the previously saved data. To determine if a frame was different, I initially computed the pixel-wise difference. I developed a script that ran for a fixed duration, calculating the difference between each pair of frames. Based on these differences, I derived a threshold, denoted as T, which represented a difference greater than 90% of all the computed differences. Subsequently, I initiated a new run, saving only frames that had a difference of at least T compared to all previously saved frames. Before addition, frames are shuffled to avoid introducing biases due to the precedence of observations. The code executed iteratively, adding newly generated images to the dataset as long as they exhibited sufficient dissimilarity from the existing dataset. As the dataset grew in size, meeting this condition becomes increasingly challenging, reaching a point where no new images can be added and the generation process is terminated.

Despite running multiple instances of the game in parallel, this approach was highly inefficient. As anticipated, the rate of new image additions gradually decreased over time. However, the computational time required to compare each new image with the entire existing dataset increased exponentially. Consequently, only a small number of images were being added to the dataset every hour, and it would have taken several days, if not weeks, to complete the dataset generation as intended. An alternative approach was therefore necessary.

In order to overcome these limitations, I opted for an approach consisting in two phases. In a first phase, I recorded a first group of frames from the game, without any limitation. From these, I computed the most frequent value for every pixel, resultin in what is technically the mode image. Since Atari images consist mostly in fixed backgrounds with a few moving sprites (enemies, bullets, avatar), this results in the background only. In the second phase, I generated again new frames by playing multiple games in parallel with a random policy, as before. For each new frame, I then compared the pixel values with those of the mode image. If a sufficient number of pixels were different, and these different pixels were located in positions that had not yet been considered, I saved the frame. To keep track of the pixels that had already been considered, I maintained a mask of boolean values initialized as false for each pixel. Every time a new frame was saved, pixels that differed from the mode image were marked as true in the mask. The code execution terminated after a specified number of iterations. Figure 2.9 shows an example of the binary mask generated for different environment after a some images have been added to the dataset, indicating which pixels have already been covered.

This system offers an empirically effective method to efficiently add images that are likely to provide new information. The trade-off between the number of images and their difference from one another can be easily adjusted by setting the threshold for the number of new pixels required to save a frame. This allows for flexibility in capturing frames that contribute with

unique content while avoiding excessive redundancy.



(a) Phoenix (1) (b) Qbert (1) (c) DemonAttack (1)

(d) Phoenix (2) (e) Qbert (2) (f) DemonAttack (2)

(g) Phoenix (3) (h) Qbert (3) (i) DemonAttack (3)

Figure 2.9: **Binary masks.** The white pixels indicate the positions for which at least one of the images in the dataset differs from the mode image. I use it too estimate which new images carries information that are not yet in the dataset.

2.9.3 Autoencoder training

The next step was to design a model capable of encoding an image from an atari game, which has a shape of 160×210 pixels, into a smaller 1D vector. This vector could then be used as

input for a controller: if the model is able to reconstruct the original image, then all the important information are preserved across all the layers, including the compressed representation between the encoder and the decoder.

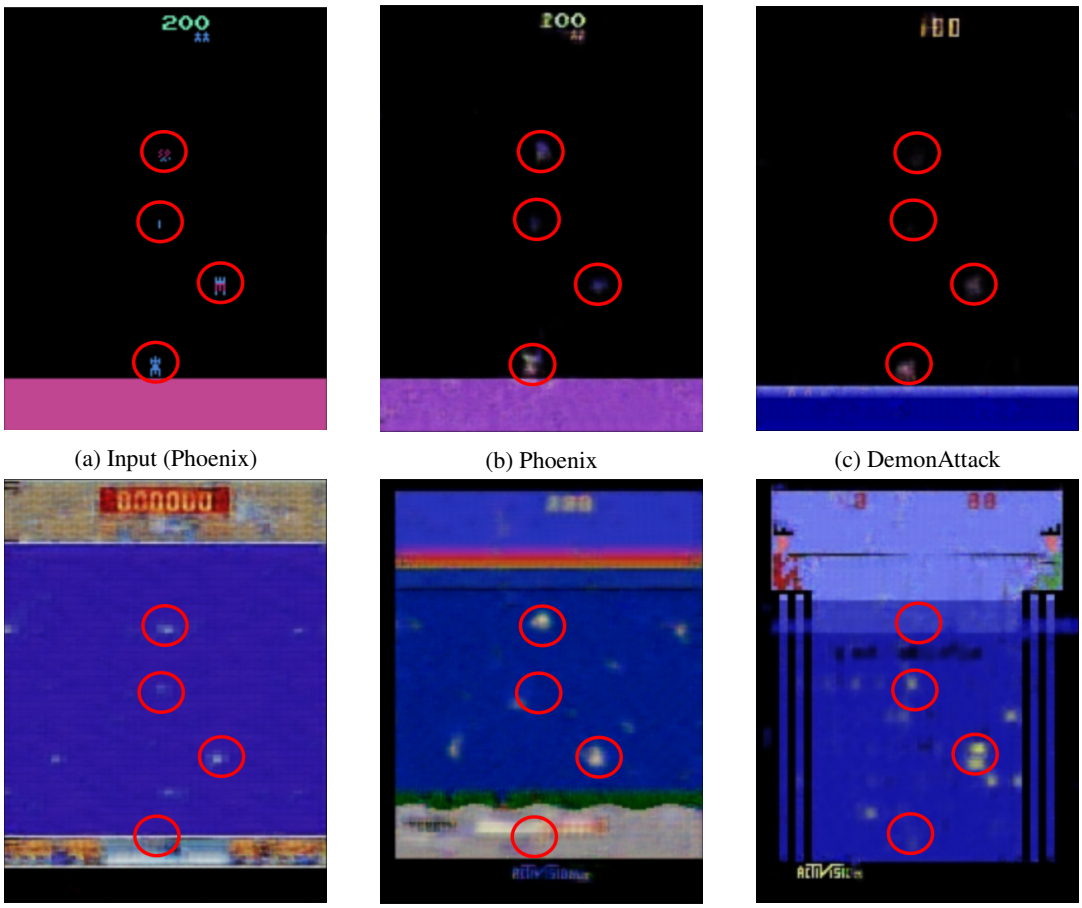
Initially, I experimented with models composed of sequences of convolutional blocks, mainly using the Rectified Linear Unit (ReLU) activation function, employing a stride of two in the convolutions to reduce the spatial dimensions. However, this approach did not yield optimal results in terms of reconstruction quality. Consequently, I decided to add max_pooling layers after the convolution, which is the classical way to reduce the shape of the input. Although these models showed an improvement compared to the previous structure, the reconstruction performance remained suboptimal.

At this point, I explored the use of variational autoencoders (VAEs) as an alternative. VAEs are known to produce a more structured and meaningful latent space with a well-defined distribution of samples. For this reason, I aimed at obtaining more stable results and better generalization to unseen data. However, this approach did not achieve the desired results, as evidenced by a significantly lower reconstruction quality compared to traditional autoencoders.

The first model that showing promising results for my task was a standard autoencoder consisting of three convolutional blocks to preprocess the input. Specifically, these blocks are changes the input in the following order: 3 to 64 channels using a 5x5 kernel, 64 to 64 channels with a 5x5 kernel, and 64 to 1 channel with a 1x1 kernel. Subsequently, the size was reduced through two convolutional layers with a 2x2 kernel and a stride of 2. Although the classical way to reduce the size is using max pooling, I found that convolutional layers outperformed it in this case. The resulting matrix was then flattened and passed through two fully-connected layers. The first layer reduced the dimensionality from 2080 to 128, followed by an ELU activation function and a dropout rate of 0.1. The second fully-connected layer transformed the 128-dimensional representation back to 128 dimensions, further elaborating the information. The decoder mirrored the encoder architecture, with the exception that it omitted the 128-to-128 fully-connected layer and used transpose convolutions to double the shape.

Although the reconstructed images from this model were visibly different than the input, they did show an approximate positioning of important elements within the scene. This was demonstrated by inputting an image from one environment into the decision maker trained on a different, yet similar, environment. I observed that the reconstructed image correctly placed the sprites at their respective locations. The quality of the reconstructed sprites were not major concerns for my objective, as the goal was to extract the position and type of elements, with the ultimate aim of constructing a controller working with this information. Figure 2.10 shows the reconstruction of an image from the Phoenix environment with a decoder (the "decision maker") trained with other datasets, which were trained together with Typhon.

The sprites in the images generated by Atari games are in fact simple and small, which led me to simplify the model and include only one convolutional layer with 256 output channels at the beginning. I also combined the convolution with one output channel and a kernel size of 1x1 with the two convolutions that reduced the shape, resulting in an equivalent convolution with a kernel size of 8x8 and a stride of 4. The decoder remained the same as before. Figure 2.11 summarizes the architecture of the model just described. This updated model showed improved performance despite having a smaller architecture. Importantly, it demonstrated that Atari images are inherently simple and can be effectively learned with a compact feature extractor. However, this simplicity also presented challenges in extracting low-level features that are generic across all environments, suggesting the difficulty of outperforming the baselines (consisting on training on only one environment) using Typhon.



(a) Input (Phoenix) (b) Phoenix (c) DemonAttack

(d) Time Pilot (e) Seaquest (f) FishingDerby

Figure 2.10: **Reconstruction with other decision makers.** The red circles have been added manually and show the position of the interesting features in the input image. The subsequent images display the reconstructions obtained using decision makers trained on other environments. It is interesting to notice that most of the decision makers reconstruct a sprite from their respective games in positions that correspond to the sprites in the input image. This observation suggests that the encoding of feature locations is achieved, at least partially, in a generic manner, aligning with the objective of my experiments.

Based on my observations and the need for more precise encoding of feature locations, I developed a novel neural network layer called "PixelPerfect". The details of this layer are presented in Section 2.9.5. This layer has shown impressive results in image encoding and subsequent reconstruction. However, it is still in its early stages of development and, in the context of Atari games, it works better when applied to single environments rather than using Typhon's parallel transfer learning. Nevertheless, I conducted an experiment where I fed an image of one environment to a model which used the PixelPerfect layer and that had never seen that particular environment before. As shown in Figure 2.12, and the distinctive sprites in the image were correctly located, demonstrating the potential of the PixelPerfect layer and motivating me to focus my efforts on further developing it.

2.9.4 Development af a new framework

Having established the means to acquire new data and a procedure for training models with this data, the subsequent step is to integrate them into a larger framework that incorporates the

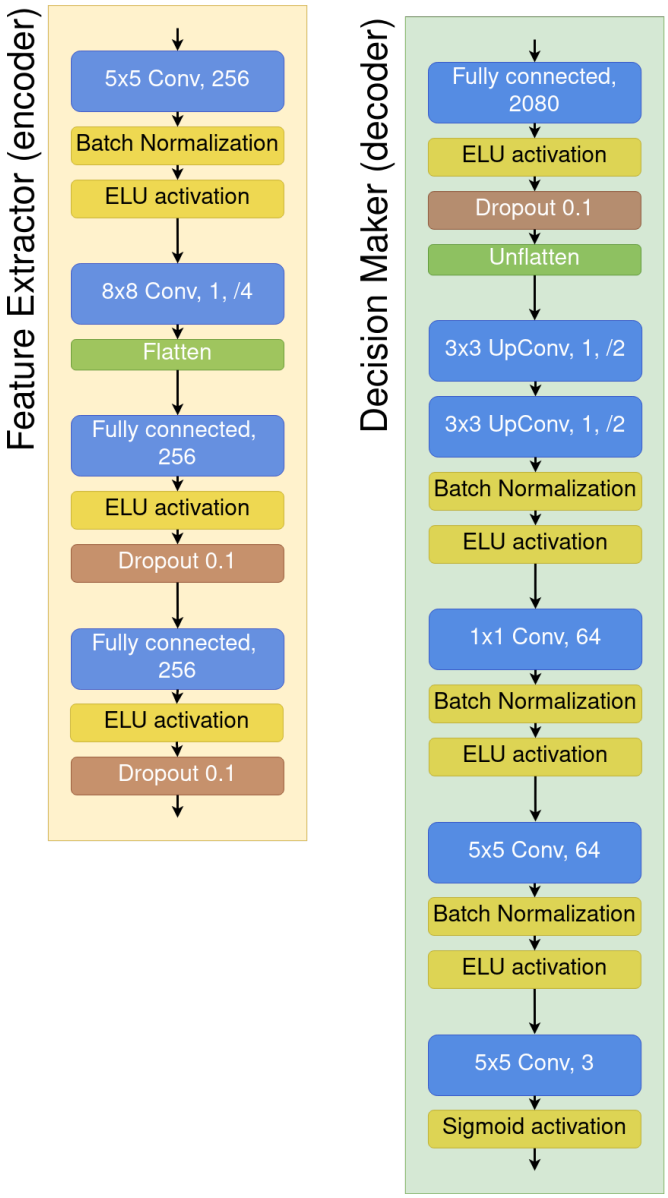
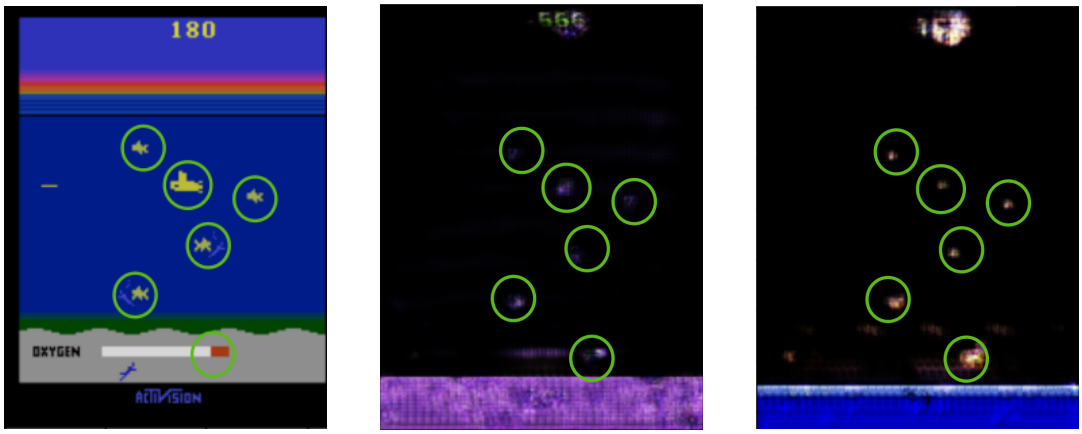


Figure 2.11: **Typhon autoencoder.** The autoencoder consists only of four blocks, two convolutional blocks and two fully connected, which is enough to encode the simple Atari sprites.

training of better and better controllers. Thus, I developed a new framework called AtariTyphon, building upon the relevant components of Typhon. Additionally to the data collection part outlined in Section 2.9.2 and the autoencoder tryining presented in Section 2.9.3, I incorporated the training of a new controller operating on the features extracted. A better controller can replace the random action selection and facilitate progress in the environment, reaching game states previoulsy unattainable and thus collecting new observations. This images can be utilized to further train the autoencoder.

The framework combines the three component data collection/autoencoder training/controller training and iterates over them. For each game, it begins by generating new images using the technique described in Section 2.9.2. Once it has a dataset for each environment, it initializes the model, consisting of a feature extractor and multiple decision makers, using bootstrap (or another chosen technique). Next, it proceeds with the parallel training of Typhon to train the



(a) Input (Seaquest) (b) Phoenix (c) DemonAttack

Figure 2.12: **PixelPerfect on unseen environments.** After training the model on the Phoenix, DemonAttack, and Qbert environments, I tested it with an input from the unseen environment Seaquest. To facilitate analysis, I manually added green circles to indicate the positions of the sprites in the input image within the different reconstructions. It is evident that despite the unfamiliarity of this new environment to the model, it is capable of encoding the positions of the majority of important features. This demonstrates the model's ability to generalize and extract meaningful features even in previously unseen environments.

autoencoder, taking advantage of the shared features across environments to extract generic information in the feature extractor. After the training is completed, the decision makers are discarded, and a controller is attached to the end of the feature extractor. This controller is trained using the chosen algorithm, typically CMA-ES (see Section 1.4.2). The loop is then repeated, with the improved controller capable of reaching further stages in the games and acquiring new images. This enables a better and more generalized training of the feature extractor model, with a subsequent better training of the controller, and so on. The loop continues for a predetermined number of iterations, determined by the user.

Each component of the framework is designed to operate independently, offering the flexibility for individual customization. For example, the data collection phase can be replaced with a pre-existing dataset. Likewise, alternative algorithms and architectures can be used to train the autoencoder, and the same flexibility applies to the controller component. This clear separation is essential for evaluating AtariTyphon as a framework, rather than a specific implementation, enabling its application to various scenarios and tasks.

2.9.5 PixelPerfect layer

I will now introduce a novel neural network layer called PixelPerfect, which enables the exact localization of features found in input signals. The development of this layer was inspired by my work with Atari games, where precise localization of game sprites within an observation image was crucial. However, the applications of the PixelPerfect layer extend beyond this, with application on any task that require precise information about the location of elements, such as object detection in images or tumor segmentation in MRI scans. Incorporating the PixelPerfect layer into a neural network architectures can significantly improve the accuracy and effectiveness of various computer vision and more generically signal processing tasks.

The next section presents a description of the current state of the implementation of the PixelPerfect layer. I will discuss the various components of the layer and provide insights into the reasons behind their design choices. Following that, Section 2.9.5 will address the current boundaries of this approach, highlighting areas where further development should focus.

Implementation

The model I employ begins with two convolutional blocks, to detect relevant features in the input images. In my case, both blocks were empirically designed to have 128 channels, but this value can be adjusted based on the diversity of features expected in the input dataset. If it contains multiple types of features, a larger number of kernels may be required, as each kernel identifies a specific feature type. Both blocks are followed by batch normalization and exponential linear unit (ELU) activation functions. These additional layers contribute to the model's ability to extract meaningful features and improve its overall performance (see Section 1.3.2).

The initial step of the actual PixelPerfect layer involves identifying the features that have been most confidently recognized by the previous convolutions for each pixel. To accomplish this, I leverage the capabilities of the PyTorch `topk` module. This module enables to extract the top k maximum values from an input tensor and returns both the values and its corresponding positions. For each pixel, I obtain the indices of the k kernels that have recognized the feature with the highest confidence. Further analysis and preliminary results indicate that in this case, it is enough to set k to 1. This is because for each pixel in an image, only one element should be visible. Nonetheless, the option to an arbitrary value for k is still present in the code.

Once this is done, the channel dimension is moved back to its original position (it has now a size of k). Next, the width and height dimensions of the images are flattened into a single vector. The `topk` function can then be used again to collect the coordinates of the n features that need to be extracted, along with their respective indices which, when converted back to 2D, correspond to the locations in the image. The resulting output will include the coordinates of the features, the values of those features (which indicate the confidence in their recognition), and the indices of the kernels that recognized the strongest features for each pixel during the initial topk operation (this should indicate the type of the feature). Consequently, the size of this encoded feature representation is three times the number of desired features. Figure 2.13 shows this procedure.

For the decoding part, I begin by creating an empty matrix filled with 0s, matching the shape of the input. The PyTorch `scatter` function is then employed to insert the values of the identified kernels at their respective positions, along with the values representing the confidence of the recognition. These matrices (two if $k = 1$, but possibly more) are concatenated and processed using three transposed convolutions, with the purpose to reconstruct the sprites from the input. Finally, the output is merged with the background, which was generated independently. In my case, this process is relatively straightforward since each decision maker is responsible for only one environment, and can thus employ a separated network to memorize the corresponding background. The combined output and background are passed through a final sequence of convolutions to merge the information. Figure 2.14 shows this procedure.

By utilizing only PyTorch modules, my implementation is fully differentiable, allowing for training using backpropagation. The experiments demonstrate the **remarkable capability of encoding sprites in a compressed space**. Considering that an input with a shape of $160\times210\times3$ is compressed into a feature space of shape 3×128, it achieves a compression factor of over $250\times$!

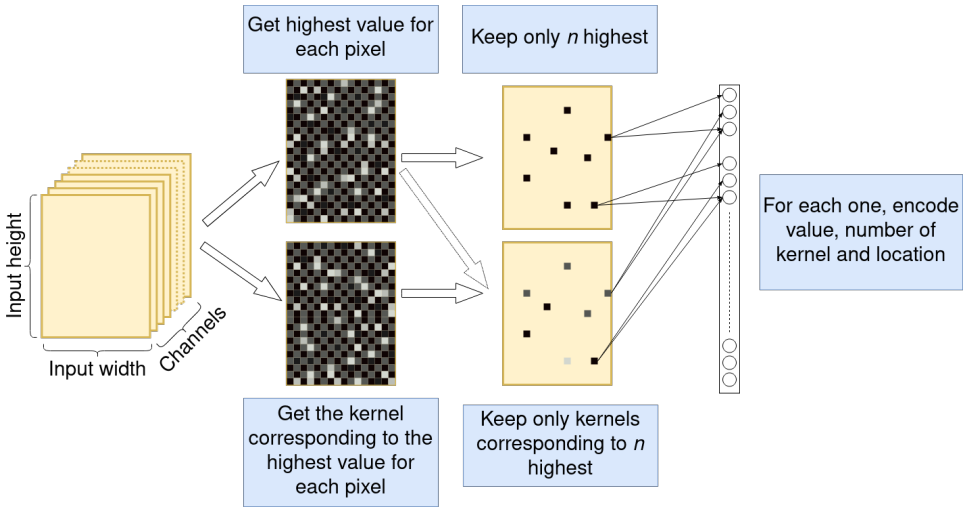


Figure 2.13: **PixelPerfect architecture.** One or more convolutions are used to extract features from the input, generating multiple channels. Next, only the highest value for each pixel is retained, while also keeping track of the kernel in which this value was found. The top k values are selected, and the remaining features are encoded into a compact vector representation. For each feature, the vector includes the corresponding value, the kernel number responsible for its recognition, and a single value representing the pixel location.

Current limitations

The Pixel Perfect layer proved to be highly effective and efficient to train. However, during my experimentation, I observed a significant disparity in the quality of the reconstructed images across different datasets. To gain insight into this phenomenon, I developed a tool that visualizes the model's output alongside the extracted feature locations. This tool provided a deeper understanding of the model's behavior.

I discovered that in environments where the model performed well, it utilized the memorized features to encode the positions of sprites that varied from one image to another. However, in other environments, the model encoded details in positions that were shared by every element in the dataset, such as the text displaying the name of the environment (see Figure 2.15). This finding surprised us, as I expected elements present in every image to be easily memorized by the decision maker in the layers responsible for reconstructing the background.

I identified that the issue arises from the complexity of certain backgrounds and the limitations of the existing layers responsible for their reconstruction. In cases where there are intricate details to render, the current sequence of convolutional blocks may struggle to generate them accurately. Consequently, the extracted features are "stolen" to improve the level of detail in areas where the error can be significantly reduced. This behavior aligns with the goal of training algorithms, which is to minimize overall reconstruction error. In this process, sacrificing the reconstruction of a sprite may be considered an acceptable trade-off to achieve a smaller error in reconstructing text, particularly if the color difference between the sprite and the background is not significant.

Another significant limitation of the current implementation of the Pixel Perfect layer is the utilization of the information regarding which kernel recognized a particular feature. Currently, I simply provide the kernel number as input to a network, which is responsible for processing this information and producing a useful output. This approach is not efficient,

as a part of the computational power of the network needs to be devolved in converting this value to a value actually related to the corresponding sprite. Additionally, the same feature can appear in different positions within the encoded feature vector. For example, if feature A is absent in one image and feature B is recognized with high confidence, feature B will occupy an early position in the feature vector (sorted by confidence). However, in a subsequent image, feature A may appear and be recognized with even stronger confidence, resulting in feature B being encoded in a later position of the feature vector. This complicates the task for any component that utilizes the output of the Pixel Perfect layer, including the controllers mentioned in Section 2.9. To complicate the problem, imposing a specific order is not easy because the same feature may be recognized multiple times within an image (e.g., multiple enemies in an Atari game).

2.9.6 Results and Discussions

In the first phase of this second application of Typhon, I successfully developed a rapid method to collect images from various stages of Atari games. Although it is possible to accellerate it even further by removing the shuffling step discussed in Section 2.9.2, I have observed that doing so results in a bias toward frames from the initial stages of the games, which appear more often. Still, even with this necessary preprocessing step, I developed a system to efficiently and automatically generate and extend a dataset, easily adaptable with the employment of a custom controller. This characteristic is crucial, since the random action selection (which is the usually the only choice as the beginning) becomes quickly insufficient to reach further phases of the games and collect new information.

Regarding the training of the autoencoder with Typhon, I have successfully demonstrated the framework's capability to extract meaningful features, and to generalize to unseen environments. This is particularly evident from Figure 2.10 and Figure 2.11. This highlights the generalization ability of the feature extractor, showcasing Typhon's potential as a meta-learning framework that enhances the feature extraction component of a model.

However, there are two main limitations to consider. Firstly, Atari images are inherently simple and uniform, as well as small in size. They can be learned almost perfectly using a small model, and when Typhon is tasked with extracting general features, it disrupts this inherent simplicity. As a result, the baseline training on a single environment tends to have a significant advantage and often yields better results compared to Typhon.

The second limitation arises from the precise localization of features during the reconstruction process. To my knowledge, there is no straightforward method to encode the coordinates of features in a compact yet trainable way that can perfectly reconstruct the original input and be compatible with backpropagation. In response, I have developed my own solution called PixelPerfect layer. This tool is still in its early stages and faces the same limitations mentioned earlier. It is easier to encode the entire sprite as a whole rather than splitting the feature extraction into lower-level features. Consequently, Typhon is significantly disadvantaged since a fully formed sprite is not shared among different environments, and memorizing a full element is easier.

The new framework that incorporates all the elements has been implemented to work seamlessly with minimal parameterization. Simply specifying the desired environments is sufficient, without the need to provide the datasets themselves, as the program takes care of generating them and initiating the loop. All components function as expected and, similar to Typhon, diverse information are saved and made available to the user for subsequent analysis.

This includes examples of images during autoencoder training, videos showcasing the distribution of information in the added images, demonstrations of controller performance, and more.

The implementation of the framework offers a practical mean to incorporate Typhon with a controller training, and its modularity allows for its use not only with Atari but also with other applications. Each part of the loop is designed to be as independent as possible, making it easy to substitute and customize specific components as needed. For example, one can easily change the training algorithm for the controller from CMA-ES to other evolutionary algorithms, random weight guessing, or any other training algorithm. The same flexibility applies to the architecture of the model or the method used to generate new datasets.

Nevertheless, there are still some limitations specific to my use of the code. The main limitation is the inherent simplicity of Atari images, which restricts the potential performance gains with Typhon. However, there is another limitation that has not been addressed yet. When using the PixelPerfect layer, we can obtain the exact coordinates of elements in the images. Yet, what I truly need are the coordinates relative to the avatar. It is not guaranteed that a linear controller can effectively handle absolute coordinates, and in my experiments I did not achieve satisfactory results in this regard.

In addition, I have introduced the PixelPerfect layer, a novel neural network layer that enables precise localization of objects within input data. This fully differentiable layer can be seamlessly incorporated into various models requiring object localization, enhancing their performance. Although the PixelPerfect layer is still in its early stage of implementation, a feasibility study has demonstrated its effectiveness and indicated its potential. I have currently showcased its application in the autoencoding of Atari game images, and with the improvements discussed in Section 2.9.5 its performance can be extended to numerous other tasks, including tumor segmentation presented in the first part of this thesis.

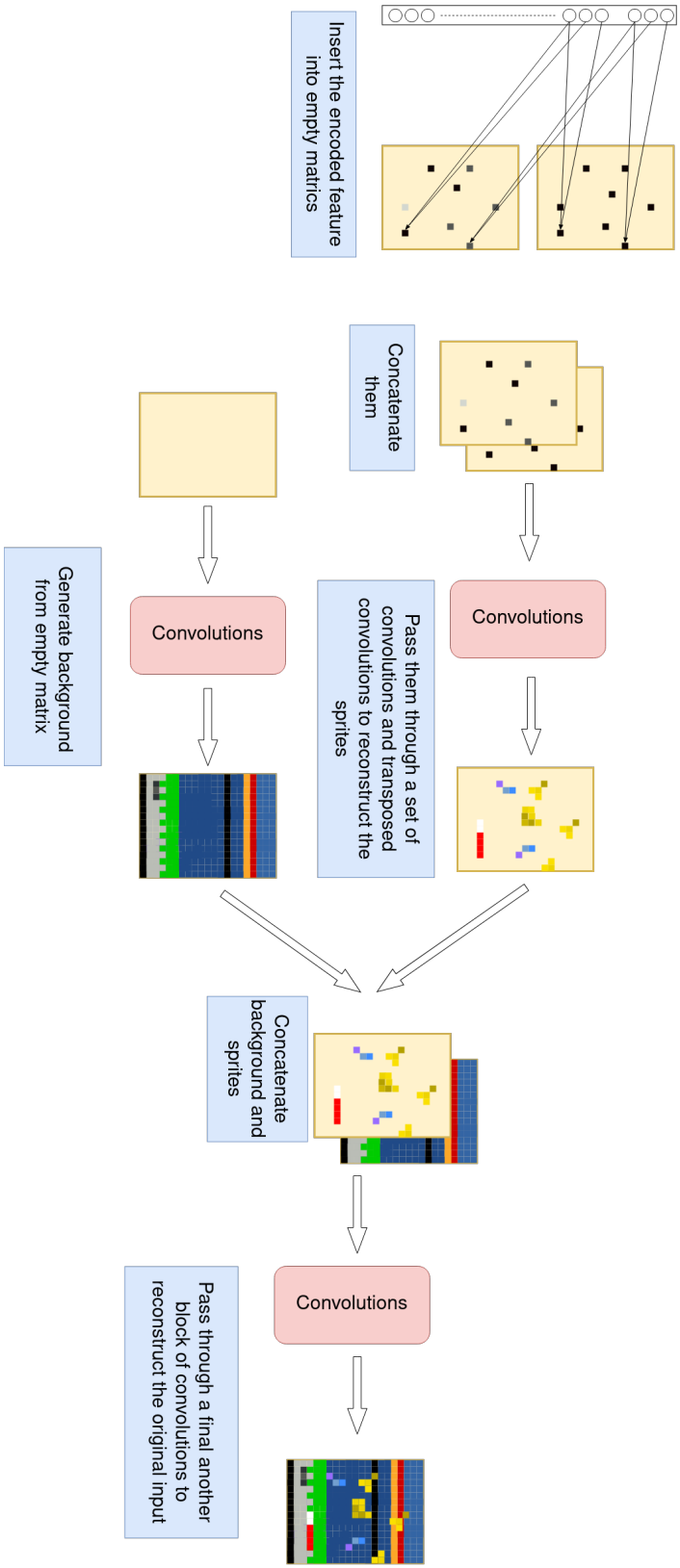
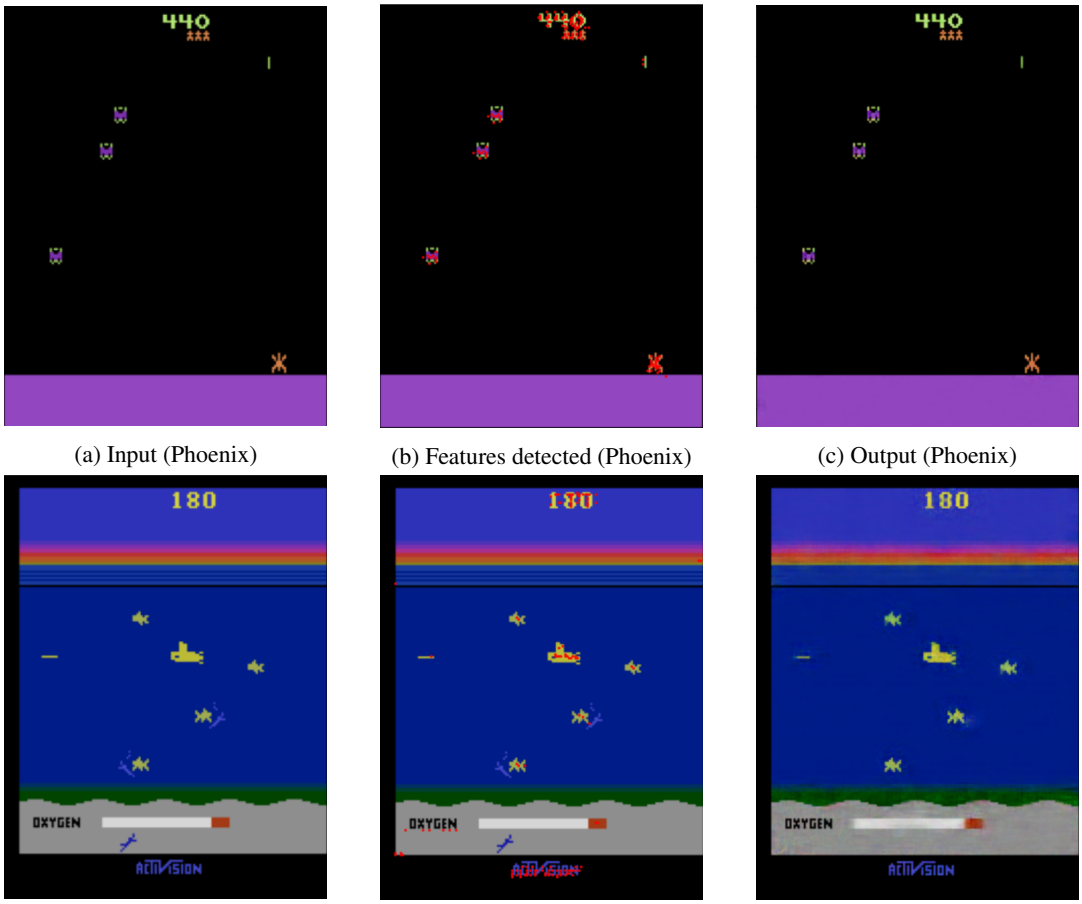


Figure 2.14: **Decoder for PixelPerfect architecture.** The encoded values are replaced at their location into empty matrices, and a sequence of convoutions and transposed convolutins are used to reconstruct the sprites. At the same time, the background is reconstructed independently. Background and sprites are then concatenated, and a final sequence of convolutions is used to generate the output image.



(a) Input (Phoenix) (b) Features detected (Phoenix) (c) Output (Phoenix)

(d) Input (Seaquest) (e) Features detected (Seaquest) (f) Output (Seaquest)

Figure 2.15: **Features detected.** The red dots in (b) and (e) represent the location of the encoded features. In the Phoenix environment, which has a very simple background, all the detected features are related to the moving sprites in the game. Seaquest however has a much more complex background, including the "ActiVision" logo. Despite the logo being is present in all images of the dataset, and could potentially be memorized by the decision makers, certain features are detected in its location, to facilitate its impressively accurate reconstruction. However, this also means that some game sprites, particularly those with a color similar to the background, can sometime not be not properly encoded, as demonstrated with the scuba divers in the scene.

Chapter 3

Conclusion

My research had several key objectives. Firstly, I analyzed the effectiveness of Typhon in separating the feature extraction process from the decision-making component. With this, I aimed to enhance the generalization capabilities of the feature extractor and consequently the overall performance. Additionally, I also wanted to test the sample efficiency of the model, recognizing the challenges associated with training the feature extraction part, which often receives a smaller error gradient. This objective is particularly crucial in domain-specific applications where limited datasets are available.

To achieve these goals, I adapted the Typhon meta-framework for a segmentation task and conducted experiments using an ultrasound dataset (UDIAT) of limited size. I integrated additional support datasets comprising different types of medical images and new body parts.

Notably, training a shared feature space with multiple datasets resulted in a significant improvements in various key segmentation metrics, particularly Dice Score, Recall and Intersection over Union. Although the incorporation of the large support dataset increased the overall running time, I experienced an enhancement in sample efficiency. Furthermore, I verified that the model successfully learned the distinctive features present in the dataset, and subsequent training attempts using traditional methods did not yield any further improvements.

One discovery during my research was the absence of overfitting in the validation and test sets, despite the model fully memorizing the training set. This finding can be attributed to the advantageous effects of parallel transfer, which mitigated the risk of overfitting and contributed to the robustness and generalization capabilities of the model.

In addition to conducting experiments on the UDIAT dataset and expanding the Typhon framework for segmentation tasks, my work has yielded other results. Firstly, I have provided a standardized version of the four datasets mentioned in 1.5. Through an accurate preprocessing phase, I have transformed these datasets in a uniform format with consistent data types, mask types, and value ranges, enabling direct utilization with classical machine learning models.

Furthermore, I have implemented the UNET and RF-Net architectures within Typhon. Through experimentation, I have explored various splitting points within these architectures and identified the parts better suited for feature extraction and decision making. While the optimal splitting points may slightly vary depending on the dataset used, I have provided a general overview of the recommended configurations.

Alongside the main experiments, I have developed a set of utility tools for dataset analysis and result interpretation. These tools enable tasks such as determining the rate of tumors in a dataset, studying the impact of bootstrap initialization, visualizing metric progression during training, generating performance histograms, and merging data from multiple models. With minimal adaptations, these tools can be applied to future experiments, significantly enhancing workflow efficiency by automating repetitive and time-consuming tasks.

Concerning Typhon, I have implemented various improvements and modifications to its codebase. Key enhancements include adjustments to the frequency and methodology of metric computation, resulting in substantial runtime performance improvements (up to 500x faster for metric computation in my case) and enabling Typhon to run efficiently on GPUs with limited memory.

At the end of the main experiment, I have conducted a further study on the applicaiton of Typhon to a completely different task, namely autoencoding of Atari game images. Through this investigation, I have showcased Typhon's capability to extract shared features from diverse datasets and train a feature extractor with the capacity to generalize to unseen data. This characteristic is particularly significant in scenarios where part of the data is unavailable for training in the earlier stages, as common in online learning. Building upon the Atari experiment, I have then developed an effective and fast method for integrating new images into an existing dataset, and I have provided a generic framework that encompasses the entire procedure necessary for the development of a controller.

The exploration of Atari images during this research endeavor resulted in the creation of a novel neural network layer designed to achieve precise feature localization within input data, called "PixelPerfect". A preliminary study showcases the layer's effectiveness in the context of Atari images, and identified potential avenues for further enhancement. This new layer has applicabilities across various domains where precise localization is required, offering an approach to improve results in crucial applications such as tumor detection and autonomous driving.

3.1 Future Work

Following the accomplishments and insights gained from my research, several directions for future exploration and improvement emerge. First, a more comprehensive investigation is warranted to fully understand the potential of Typhon in preventing overfitting, particularly in tasks where overfitting is known to pose significant challenges. This exploration could involve applying Typhon to other complex tasks with limited datasets.

Another intriguing research direction in the domain of overfitting would involve splitting a single dataset into multiple sub-datasets. Intermediate experiments using only UDIAT and BUSI datasets, which exhibit significant similarity, have shown promising results in this regard. If the dataset used is sufficiently large to accommodate such splitting, this approach could enable the introduction of parallel transfer. By employing different initializations for the decision makers, this method could potentially yield effects similar to those observed in this thesis, particularly with regard to overfitting mitigation.

Regarding the CAD task analyzed in this work, a future improvement can be achieved by redesigning the architecture used, in particular developing a more compact and streamlined model. By doing this we can potentially achieve enhanced performance in terms of both accuracy and computational efficiency.

An additional area of focus for enhancing tumor segmentation with Typhon regards the bootstrap initialization. Conducting a comprehensive investigation into various bootstrap methods specifically tailored for segmentation could certainly be beneficial. This is particularly important as the bootstrap component plays a vital role in mitigating the challenges posed by the moving target problem in parallel transfer. While the current bootstrap implementation has

demonstrated satisfactory performance in most of the cases, there exists a significant potential for improvement. Refining the initialization process could lead to faster convergence and ultimately enhance the quality of the segmentation results. In addition, good initializations of the feature extractor are often discarded due to poor decision makers, while the latter should not impact the process. This could be improved by adding an inner iteration testing multiple decision makers for each initialization of the feature extractor.

Expanding the applicability of the Typhon framework to other tasks is another promising area for future research. The versatility of Typhon's feature extraction capabilities makes it a promising candidate for a wide range of machine learning applications. Investigating and adapting Typhon to tasks beyond segmentation, such as object detection, natural language processing, or time series analysis, holds the potential to unlock its benefits in diverse domains and further validate its effectiveness.

Another valuable implementation that Typhon could potentially benefit from is the incorporation of an adaptive learning rate strategy, which is commonly employed in training state-of-the-art models. Currently, the learning rate remains constant throughout the entire training process. However, researches have demonstrated the advantages of gradually reducing the learning rate during training to enhance precision and accelerate convergence (You et al., 2019; Ding, 2021). Therefore, a valuable addition to the framework would involve implementing this principle by introducing a decay factor that adjusts the learning rate after each epoch. This adaptive learning rate mechanism would contribute to the overall effectiveness and efficiency of the training process within Typhon.

Regarding the further application of Typhon with Atari games presented in Section 2.9, multiple research directions for future exploration opens up. Firstly, further efforts should be dedicated to finding an effective method for extracting general features from Atari images. One potential approach is to split Typhon before the bottleneck of the autoencoder, allowing the last part of the encoding to be specific to each environment. However, this would increase the complexity of the subsequent controller built on top of it, requiring careful consideration and design.

Additional research directions concern the controller. As the features represent the absolute positions of the elements, while the optimal decision depends on their relative position to the player, a linear controller may not be adequate. Therefore, alternative controllers should be investigated, in particular exploring the use of more advanced models. It is important to note that "more advanced" does not necessarily imply larger architectures, but rather introducing additional complexity to address the challenge of absolute coordinates. The controller should be able to incorporate a component that identifies the extracted feature representing the avatar and subsequently translates the absolute positions of other elements into relative positions with respect to the controllable sprite. This would enable the controller to better understand and operate in the game environment.

Given that the feature extraction component, including the PixelPerfect layer, can be trained independently (as discussed in Section 2.9.4), and considering that the controller can probably be very simple (as proven by Cuccu, Togelius, and Cudre-Mauroux (2019)), the choice is however not restricted to differentiable functions like neural networks. This derives from the fact that we can use other training algorithms, such as Evolution Strategies (ES) or even a simple Random Weight Guessing (RWG).

In this direction, utilizing algorithms that automatically evolve the architecture could provide

valuable assistance. One such algorithm that has shown effectiveness is "NeuroEvolution of Augmenting Topologies" (NEAT, Stanley and Miikkulainen (2002)). NEAT evolves the architecture by adding or removing new nodes when the performance plateaus and can no longer be improved. This type of algorithm, which starts with small architectures and progressively enhances them, can be particularly valuable in cases where the optimal architecture is not known in advance. Other algorithms of similar nature include Stanley, D'Ambrosio, and Gauci (2009), Zoph et al. (2018), and Papavasileiou, Cornelis, and Jansen (2021). By leveraging these evolutionary algorithms, we can explore and discover more sophisticated architectures for the controller, potentially leading to a better handling of absolute coordinates and consequently improved overall performance in Atari games.

The PixelPerfect layer, presented in the Section 2.9.5, is still in its early stages. The ability to extract a compressed set of features along with their precise locations in a differentiable manner, all the while still allowing for backpropagation, has countless applications. To begin with, it could be integrated into Typhon to enhance segmentation precision. Another intuitive use is with autoencoders, as proposed in Section 2.9.3. Nonetheless, significant work needs to be done in this direction. Firstly, a robust method for reconstructing these features must be developed. In cases where the identified features are highly precise (such as in Atari games), this requires only a small network with a few layers. However, it remains to be proven how to achieve this with real-world inputs.

Another issue with the current stage of the PixelPerfect layer is the lack of determined order in which features are identified; it depends on the confidence of identification. Consequently, in one image, a certain feature may be outputted first, while in another image where other features have been identified with greater confidence, the same feature may appear as the third or fourth value. Currently, this problem is addressed by including the kernel number responsible for identifying the feature along with its coordinates. However, this solution is suboptimal. One potential trade-off could be made if the number of different features is known in advance. In this case, each feature could be assigned a fixed range of positions within the feature vector. For example, one type of feature is always encoded in the first 10 positions, the second type in positions 11 to 20, and so on. Although determining which feature corresponds to each position may be challenging, it would still significantly improve the consistency of the feature vector. It would also be necessary to set a limit on the number of features that can be detected for each type to ensure that the feature vector remains within the defined range.

Another research direction that arises from the current state of this work is the development of a better model for generating background images. Although this is not directly related to the Pixel Perfect layer itself, improving the reconstruction of backgrounds can significantly enhance its performance by enabling it to focus on the elements that actually require encoding. One possibility is to directly load the mode image of the dataset, but there are other more advanced and interesting alternatives available. Since the background is generated in isolation, any architecture that has demonstrated the ability to generate images can be utilized. For example, a generator from a Generative Adversarial Network (GAN) architecture (Goodfellow et al., 2020) could be employed.

Bibliography

Abubakar, Aliyu, Mohammed Ajuji, and Ibrahim Yahya (Apr. 2020). "Comparison of Deep Transfer Learning Techniques in Human Skin Burns Discrimination". In: *Applied System Innovation* 3, p. 20. doi: 10.3390/asi3020020.

Armato III, Samuel G et al. (2011). "The lung image database consortium (LIDC) and image database resource initiative (IDRI): a completed reference database of lung nodules on CT scans". In: *Medical physics* 38.2, pp. 915–931.

Asiri, AA et al. (2023). "Brain Tumor Detection and Classification Using Fine-Tuned CNN with ResNet50 and U-Net Model: A Study on TCGA-LGG and TCIA Dataset for MRI Applications". In: *Life* 13, p. 1449.

Bakas, Spyridon et al. (2017). "Advancing the cancer genome atlas glioma MRI collections with expert segmentation labels and radiomic features". In: *Scientific data* 4.1, pp. 1–13.

Bakator, Mihalj and Dragica Radosav (2018). "Deep learning and medical diagnosis: A review of literature". In: *Multimodal Technologies and Interaction* 2.3, p. 47.

Barrault, Loïc et al. (2019). "Findings of the 2019 conference on machine translation (WMT19)". In: ACL.

Brockman, Greg et al. (2016). "Openai gym". In: *arXiv preprint arXiv:1606.01540*.

Broillet, Christophe (2022). "Pre-Processing Segmentation Datasets for the Hydra Framework". en. MA thesis. University of Fribourg, eXascale Infolab.

Carion, Nicolas et al. (May 2020). *End-to-End Object Detection with Transformers*. en. arXiv:2005.12872 [cs]. url: http://arxiv.org/abs/2005.12872 (visited on 07/05/2023).

Caruana, Rich (1997). "Multitask learning". In: *Machine learning* 28, pp. 41–75.

Chen, Xiaoxue et al. (2021). "Text recognition in the wild: A survey". In: *ACM Computing Surveys (CSUR)* 54.2, pp. 1–35.

Cuccu, Giuseppe, Julian Togelius, and Philippe Cudre-Mauroux (Mar. 3, 2019). *Playing Atari with Six Neurons*. arXiv: 1806.01363[cs,stat]. url: http://arxiv.org/abs/1806.01363 (visited on 04/12/2023).

Cuccu, Giuseppe et al. (Dec. 10, 2020). "Hydra: Cancer Detection Leveraging Multiple Heads and Heterogeneous Datasets". In: *2020 IEEE International Conference on Big Data (Big Data)*. 2020 IEEE International Conference on Big Data (Big Data). Atlanta, GA, USA: IEEE, pp. 4842–4849. isbn: 978-1-72816-251-5. doi: 10.1109/BigData50022.2020.9378042. url: https://ieeexplore.ieee.org/document/9378042/ (visited on 04/06/2023).

Cuccu, Giuseppe et al. (Dec. 2022). "Typhon: Parallel Transfer on Heterogeneous Datasets for Cancer Detection in Computer-Aided Diagnosis". In: *2022 IEEE International Conference on Big Data (Big Data)*. 2022 IEEE International Conference on Big Data (Big Data), pp. 5223–5232. doi: 10.1109/BigData55660.2022.10020446.

Dabre, Raj, Chenhui Chu, and Anoop Kunchukuttan (2020). "A survey of multilingual neural machine translation". In: *ACM Computing Surveys (CSUR)* 53.5, pp. 1–38.

Ding, Yimin (Jan. 2021). "The Impact of Learning Rate Decay and Periodical Learning Rate Restart on Artificial Neural Network". en. In: *2021 2nd International Conference on Artificial Intelligence in Electronics Engineering*. Phuket Thailand: ACM, pp. 6–14. isbn: 978-1-4503-8927-3. doi: 10.1145/3460268.3460270. url: https://dl.acm.org/doi/10.1145/3460268.3460270 (visited on 07/06/2023).

French, Robert M. (Apr. 1999). "Catastrophic forgetting in connectionist networks". en. In: *Trends in Cognitive Sciences* 3.4, pp. 128–135. issn: 1364-6613. doi: 10.1016/S1364-6613(99)01294-2.

Ghosh, Sourodip, Aunkit Chaki, and KC Santosh (2021). "Improved U-Net architecture with VGG-16 for brain tumor segmentation". In: *Physical and Engineering Sciences in Medicine* 44.3, pp. 703–712.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep learning*. MIT press.

Goodfellow, Ian et al. (2020). "Generative adversarial networks". In: *Communications of the ACM* 63.11, pp. 139–144.

Hansen, N. and A. Ostermeier (May 1996). "Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation". In: *Proceedings of IEEE International Conference on Evolutionary Computation*. Proceedings of IEEE International Conference on Evolutionary Computation, pp. 312–317. doi: 10.1109/ICEC.1996.542381.

He, Kaiming et al. (Dec. 2015). *Deep Residual Learning for Image Recognition*. en. arXiv:1512.03385 [cs]. url: http://arxiv.org/abs/1512.03385 (visited on 07/05/2023).

Hirschberg, Julia and Christopher D Manning (2015). "Advances in natural language processing". In: *Science* 349.6245, pp. 261–266.

Iman, Mohammadreza, Hamid Reza Arabnia, and Khaled Rasheed (2023). "A review of deep transfer learning and recent advancements". In: *Technologies* 11.2, p. 40.

Ioffe, Sergey and Christian Szegedy (2015). "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. pmlr, pp. 448–456.

Kingma, Diederik P and Max Welling (2019). "An Introduction to Variational Autoencoders". In: *arXiv preprint arXiv:1906.02691*.

Langhammer, Romy (Jan. 2018). "Metabolomic Imaging for Human Prostate Cancer Detection using MR Spectroscopy at 7T". PhD thesis.

Liu, Xiaolong, Zhidong Deng, and Yuhan Yang (2019). "Recent progress in semantic image segmentation". In: *Artificial Intelligence Review* 52, pp. 1089–1106.

Masini, Ricardo P, Marcelo C Medeiros, and Eduardo F Mendes (2020). "Machine Learning Advances for Time Series Forecasting". In: *arXiv preprint arXiv:2012.12802*.

McCloskey, Michael and Neal J. Cohen (Jan. 1989). "Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem". en. In: *Psychology of Learning and Motivation*. Ed. by Gordon H. Bower. Vol. 24. Academic Press, pp. 109–165. doi: 10.1016/S0079-7421(08)60536-8.

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

Oakden-Rayner, Luke (May 2019). "The Rebirth of CAD: How Is Modern AI Different from the CAD We Know?" en. In: *Radiology: Artificial Intelligence* 1.3, e180089. issn: 2638-6100. doi: 10.1148/ryai.2019180089. url: http://pubs.rsna.org/doi/10.1148/ryai.2019180089 (visited on 06/22/2023).

Papavasileiou, Evgenia, Jan Cornelis, and Bart Jansen (2021). "A systematic literature review of the successors of neuroevolution of augmenting topologies". In: *Evolutionary Computation* 29.1, pp. 1–73.

Perelman School of Medicine (2019). *Multimodal Brain Tumor Segmentation Challenge 2019*. Accessed July 6, 2023. https://www.med.upenn.edu/cbica/brats-2019/.

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

Raghu, Maithra et al. (2019). "Transfusion: Understanding transfer learning for medical imaging". In: *Advances in neural information processing systems* 32.

Ronneberger, Olaf, Philipp Fischer, and Thomas Brox (May 18, 2015). *U-Net: Convolutional Networks for Biomedical Image Segmentation*. arXiv: 1505.04597[cs]. url: http://arxiv.org/abs/1505.04597 (visited on 04/12/2023).

Rosenstein, Michael T et al. (2005). "To transfer or not to transfer". In: *NIPS 2005 workshop on transfer learning*. Vol. 898, pp. 1–4.

Rusu, Andrei A et al. (2016). "Progressive neural networks". In: *arXiv preprint arXiv:1606.04671*.

Samala, Ravi K et al. (Nov. 2017). "Multi-task transfer learning deep convolutional neural network: application to computer-aided diagnosis of breast cancer on mammograms". In: *Physics in Medicine & Biology* 62.23, pp. 8894–8908. doi: 10.1088/1361-6560/aa93d4.

Samala, Ravi K et al. (2020). "Generalization error analysis for deep convolutional neural network with transfer learning in breast cancer diagnosis". In: *Physics in Medicine & Biology* 65.10, p. 105002.

Santurkar, Shibani et al. (2018). "How does batch normalization help optimization?" In: *Advances in neural information processing systems* 31.

Saxena, Priyansh et al. (Dec. 2019). "Predictive modeling of brain tumor: A Deep learning approach". en. In: *arXiv:1911.02265 [cs, eess]*.

Silver, David et al. (2017a). "Mastering chess and shogi by self-play with a general reinforcement learning algorithm". In: *arXiv preprint arXiv:1712.01815*.

Silver, David et al. (2017b). "Mastering the game of go without human knowledge". In: *nature* 550.7676, pp. 354–359.

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

Stanley, Kenneth O, David B D'Ambrosio, and Jason Gauci (2009). "A hypercube-based encoding for evolving large-scale neural networks". In: *Artificial life* 15.2, pp. 185–212.

Stanley, Kenneth O and Risto Miikkulainen (2002). "Evolving neural networks through augmenting topologies". In: *Evolutionary computation* 10.2, pp. 99–127.

Thrun, Sebastian (1995). "A lifelong learning perspective for mobile robot control". In: *Intelligent robots and systems*. Elsevier, pp. 201–214.

Torrey, Lisa and Jude Shavlik (2010). "Transfer Learning". en. In: *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*. doi: 10.4018/978-1-60566-766-9.ch011.

Vallières, Martin et al. (2017). *Data from Head-Neck-PET-CT*. doi: 10.7937/K9/TCIA.2017.8OJE5Q00. url: https://wiki.cancerimagingarchive.net/x/24pyAQ.

Varoquaux, Gaël and Veronika Cheplygina (2022). "Machine learning for medical imaging: methodological failures and recommendations for the future". In: *NPJ digital medicine* 5.1, p. 48.

Wang, Kai, Boris Babenko, and Serge Belongie (2011). "End-to-end scene text recognition". In: *2011 International conference on computer vision*. IEEE, pp. 1457–1464.

Wang, Ke, Shujun Liang, and Yu Zhang (2021). "Residual Feedback Network for Breast Lesion Segmentation in Ultrasound Image". In: *Medical Image Computing and Computer Assisted InterventionMICCAI 2021: 24th International Conference, Strasbourg, France, September 27October 1, 2021, Proceedings, Part I*, pp. 471–481.

Yap, Moi Hoon et al. (2018). "Automated Breast Ultrasound Lesions Detection Using Convolutional Neural Networks". In: *IEEE Journal of Biomedical and Health Informatics* 22.4, pp. 1218–1226. doi: 10.1109/JBHI.2017.2731873.

Yoon, Jaehong et al. (2018). "Lifelong Learning with Dynamically Expandable Networks". In: *International Conference on Learning Representations*. url: https://openreview.net/forum?id=Sk7KsfW0-.

You, Kaichao et al. (Sept. 2019). *How Does Learning Rate Decay Help Modern Neural Networks?* en. arXiv:1908.01878 [cs, stat]. url: http://arxiv.org/abs/1908.01878 (visited on 07/06/2023).

Yurtsever, Ekim et al. (2020). "A survey of autonomous driving: Common practices and emerging technologies". In: *IEEE access* 8, pp. 58443–58469.

Zhang, W. et al. (2020). "Deep Model Based Transfer and Multi-Task Learning for Biological Image Analysis". In: *IEEE Transactions on Big Data* 6.2, pp. 322–333.

Zoph, Barret et al. (2018). "Learning transferable architectures for scalable image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710.