HOCHSCHULE HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Extending ”PathoLearn” with an End-To-End Artificial Intelligence Platform

Jannes Neemann

Master-Thesis in Computer Science

October 12, 2023

Author Jannes Neemann
166172
jannes@neemann.net

First examiner: Prof. Dr. Frauke Sprengel
Abteilung Informatik, Fakultät IV
Hochschule Hannover
frauke.sprengel@hs-hannover.de

Second examiner: Dr. Nadine S. Schaadt
Institut für Pathologie
Medizinische Hochschule Hannover
Schaadt.Nadine@mh-hannover.de

Declaration of authorship

I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Hannover, October 12, 2023 Signature

Abstract

Pathologists need to identify abnormal changes in tissue. With the developing digitalization, the used tissue slides are stored digitally. This enables pathologists to annotate the region of interest with the support of software tools. PathoLearn is a web-based learning platform explicitly developed for the teacher-student scenario, where the goal is that students learn to identify potential abnormal changes. Artificial intelligence (AI) and machine learning (ML) have become very important in medicine. Many health sectors already utilize AI and ML. This will only increase in the future, also in the field of pathology. Therefore, it is important to teach students the fundamentals and concepts of AI and ML early in their studies. Additionally, creating and training AI generally requires knowledge of programming and technical details. This thesis evaluates how this boundary can be overcome by comparing existing end-to-end AI platforms and teaching tools for AI. It was shown that a visual programming editor offers a fitting abstraction for creating neural networks without programming. This was extended with real-time collaboration to enable students to work in groups. Additionally, an automatic training feature was implemented, removing the necessity to know technical details about training neural networks.

Table of Contents

1 Introduction **1**

2 Machine Learning **4**
2.1 Overview . . . 4
2.2 Supervised Learning . . . 6
2.3 Feedforward Neural Networks . . . 8
2.3.1 Artificial Neurons . . . 8
2.3.2 Multilayer Perceptron . . . 9
2.3.3 Nonlinear Activation Functions . . . 10
2.4 Backpropagation . . . 12
2.4.1 Overfitting . . . 15
2.4.2 Batch Normalization . . . 16
2.5 Datasets . . . 17
2.5.1 Dataset Splitting . . . 17
2.5.2 Digital Pathology Challenges . . . 18
2.5.3 Data Augmentation . . . 18
2.6 Convolutional Neural Network . . . 19
2.6.1 Convolutional Layer . . . 20
2.6.2 Pooling Layers . . . 23

3 Building Blocks of Neural Networks **24**
3.1 Classification . . . 24
3.2 Object Detection . . . 28
3.3 Image Segmentation . . . 30
3.4 Pre-Trained Models . . . 34

4 The Artificial Intelligence Lifecycle and Software Tools **36**
4.1 The Artificial Intelligence Lifecycle . . . 36
4.2 Choosing a Deep Learning Framework . . . 37
4.2.1 Popularity of Different Deep Learning Frameworks . . . 38
4.2.2 PyTorch vs. TensorFlow . . . 39
4.3 Comparison of Existing End-To-End Artificial Intelligence Platforms . . . 44
4.3.1 ClearML . . . 50

4.4 Software Tools for Teaching Artificial Intelligence 53

5 Requirements **56**
5.1 Stakeholders and Target Groups 56
5.2 User Stories 57
5.3 Functional Requirements 58
5.4 Non-Functional Requirements 59

6 Implementation **61**
6.1 General Software Architecture 61
6.2 Centralized Authentication 62
6.3 Creating Neural Network Architectures 65
6.3.1 Visual Programming Editor 65
6.3.2 Predefined Neural Network Architectures 68
6.3.3 Collaboration 71
6.4 Training Neural Network Models 75
6.4.1 Creating Datasets 75
6.4.2 Parsing Visual Programming Editor Nodes 78
6.4.3 Training workflow 81
6.5 Serving Neural Network Models 84

7 User Test **88**
7.1 Execution 88
7.2 Surveys 89
7.3 Results 89

8 Requirements Fulfillment **91**

9 Conclusion and Future Work **93**

A Appendix **96**
A.1 Comparison of Pre-Trained and Not Pre-Trained CNNs 96
A.1.1 Dataset 96
A.1.2 Model Configuration 96
A.1.3 Results 97
A.2 The Project and Experiment Page 99
A.3 Dataset Metadata 101
A.4 Dataset Template Code 102
A.5 The Dataset Page 103
A.6 Lightning Model for Classification Tasks 104
A.7 PyTorch Pooling Layers with Same Padding 105
A.8 PyTorch Layers for Addition and Concatenation 106
A.9 The Inception Module Realized in PathoLearn 107

A.10 The Residual Block Realized in PathoLearn 108
A.11 Evaluating the Best Neural Network Model Serving Format 109
A.11.1 Training Environment 109
A.11.2 Datasets 109
A.11.3 Neural Network Architecture and Training Configuration 110
A.11.4 Metric Gathering 111
A.11.5 Results 113
A.12 Examplatory Pre- and Postprocessing script 115
A.13 Surveys 116
A.13.1 Survey before Using Patholearn 116
A.13.2 Survey after Using Patholearn 118
A.13.3 Survey Answers before Using PathoLearn 121
A.13.4 Survey Answers after Using PathoLearn 123
A.14 Evaluating the Visual Programming Editor Real-Time Performance . . . 127
A.14.1 Environment 127
A.14.2 Procedure 127
A.14.3 Results 129

1. Introduction

Artificial intelligence (AI) and machine learning (ML) are becoming increasingly important in medicine. Through the emerging digitalization of medical data and the increasing performance of hardware, new AI technologies arise. Many fields of the health sector e.g., diagnostics, patient monitoring, and robotics, already utilize AI, with a tendency to increase even more in the future [Ado21].

Pathology is a subfield of medicine that is about the study of diseases [Fun18]. Essentially, it examines the structural and functional changes resulting from a disease. Primary, this happens on a microscopic level, where individual cells or regions of a tissue sample are investigated under a microscope. Pathologists can identify abnormal tissue changes e.g. which cells are cancerous. In recent years, with the development of digital microscopes and specialized scanners, called whole slide scanners (WSS), the work of pathologists can be supported with software tools. A WSS moves a digital microscope over a tissue sample, takes multiple images, and stitches them together to form a high-resolution digital representation of the tissue, called a whole slide image (WSI) [HPS20]. Through specialized software, WSIs can be viewed on a monitor, and pathologists can annotate them to mark regions of interest. WSIs and corresponding annotations build the foundation for today's diagnostics and enable to share the results with other pathologists.

WSIs also introduce a new way of teaching pathology. Instead of preparing multiple microscopes with tissue samples for students, software can be used to display WSIs. Every student can access the same digital tissue sample, and the teacher can more easily show and explain relevant regions on the slide. This also comes with new possibilities for student exercises. PathoLearn is an open source teaching platform for pathology that utilizes the ability to annotate WSIs with different types of annotations (e.g. point, line, or polygon) [Nee21]. It uses a course system where teachers can create courses and tasks. Each task uses a WSI, where the student is required to draw the correct annotations on the WSI to solve it. Different from other available software systems, PathoLearn generates automatic feedback that supports the student in improving their solution. The teacher is only required to create a sample solution. Figure 1.1 shows an extract of the PathoLearn frontend for teachers. Each annotation created by a teacher has an outer and an inner threshold. The resulting area ring defines where the annotation drawn by the student would still be considered correct. As students must

also be capable of classifying the annotation, the teacher can define multiple annotation classes (see Figure 1.1 top right corner) and assign one to each annotation (light and dark blue colored annotations). Teachers can also add informative annotations or regions of interest to guide students to the correct area on the WSI.

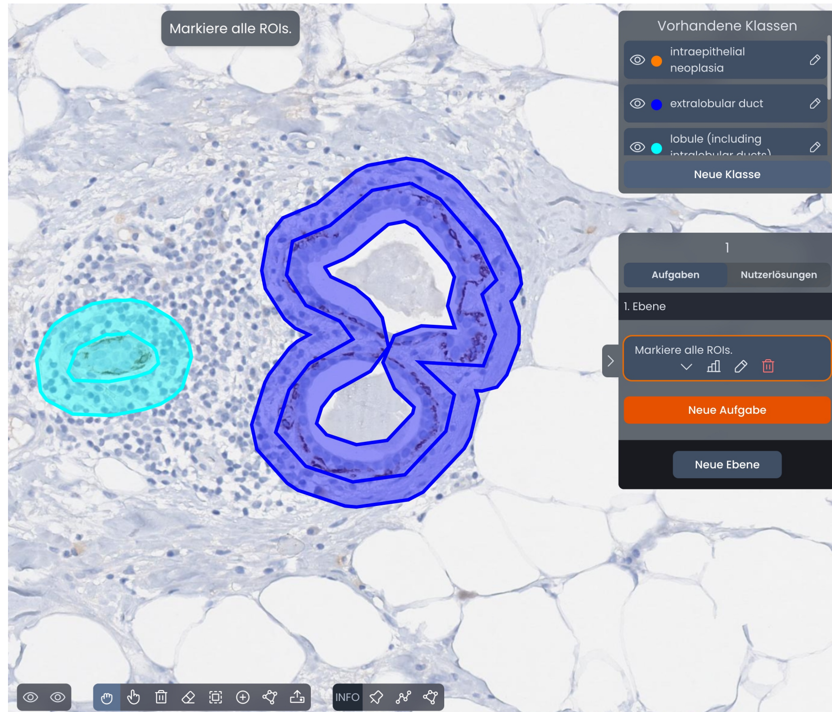


Figure 1.1.: Extract of the PathoLearn frontend for teachers to create WSI tasks. A teacher can draw annotations and assign an annotation class (annotation color) from a list of classes (top right) to them.

Students are offered a different view (see Figure 1.2). They initially only see the informative annotations and regions of interest. Through the given work order (top of the image), students have to identify where the requested annotations are located. Figure 1.2 illustrates how the feedback is presented to a student. Each annotation receives feedback through color highlighting. Depending on the task difficulty chosen by the teacher, more detailed feedback is displayed. This can be seen with the correct (green) annotation, where parts of the polygon line are highlighted yellow to indicate that these parts are outside of the defined threshold. The student can also select an annotation class for each annotation. If the selected is wrong, the annotation receives a violet highlight. Additionally, summed-up feedback is given for the entire task, informing the student of the number of missing annotations and listing how many annotations are correct, wrong, or have the wrong class. Through these feedback ways, students can incrementally improve their solutions and learn to identify pathological relevant structures on tissues.

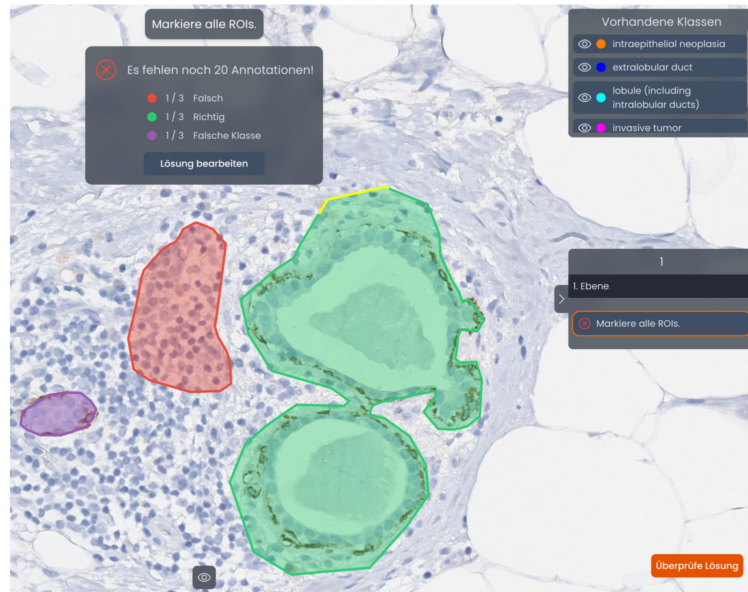


Figure 1.2.: Extract of the PathoLearn frontend for students to solve a task. The automatic feedback is displayed. The students get annotation-specific feedback and a summed-up version of the entire task.

In recent years, AI and ML technologies have supported or even replaced the task of a pathologist examining tissue slides. It was shown that AI and computer vision algorithms could detect diseases on tissue samples [CJW+18, RRS+22]. The judgment of a tissue region, whether it is pathologically relevant or not, can vary among pathologists [RRS+22]. This problem is resolved through algorithmic solutions [RRS+22]. The integration of AI and ML into the daily work routine of pathologists will only increase in the upcoming years, so it is crucial to raise awareness and teach the basics of AI and ML early in education. Therefore, this thesis aims to evaluate and implement an end-to-end AI platform into PathoLearn that enables students to create AI without requiring knowledge about programming.

To realize this, different areas of AI and ML will be highlighted. In Chapter 2, the fundamentals of ML are presented. Pathologists primarily work on WSIs, so state-of-the-art image algorithms and building blocks are given in Chapter 3. Besides creating an algorithm, different steps are required to train and evaluate it. These steps form a lifecycle, which is explained in Chapter 4, including different software solutions that realize these steps. Afterward, a detailed requirements analysis determines the features required in the platform (see Chapter 5). Based on these and the previous results, the design and implementation of the platform are presented in Chapter 6. In Chapter 7, the platform was tested by users. Finally, in Chapter 8, it is checked whether all defined requirements are fulfilled. Chapter 9 finishes this thesis with a conclusion and an outlook on future work.

2. Machine Learning

This chapter briefly overviews ML and the core concepts needed for creating AI based on images.

2.1. Overview

For incorporating intelligence into a machine or computer program, many different methods and fields have emerged [WLLT21]. Figure 2.1 displays the different subfields of AI.

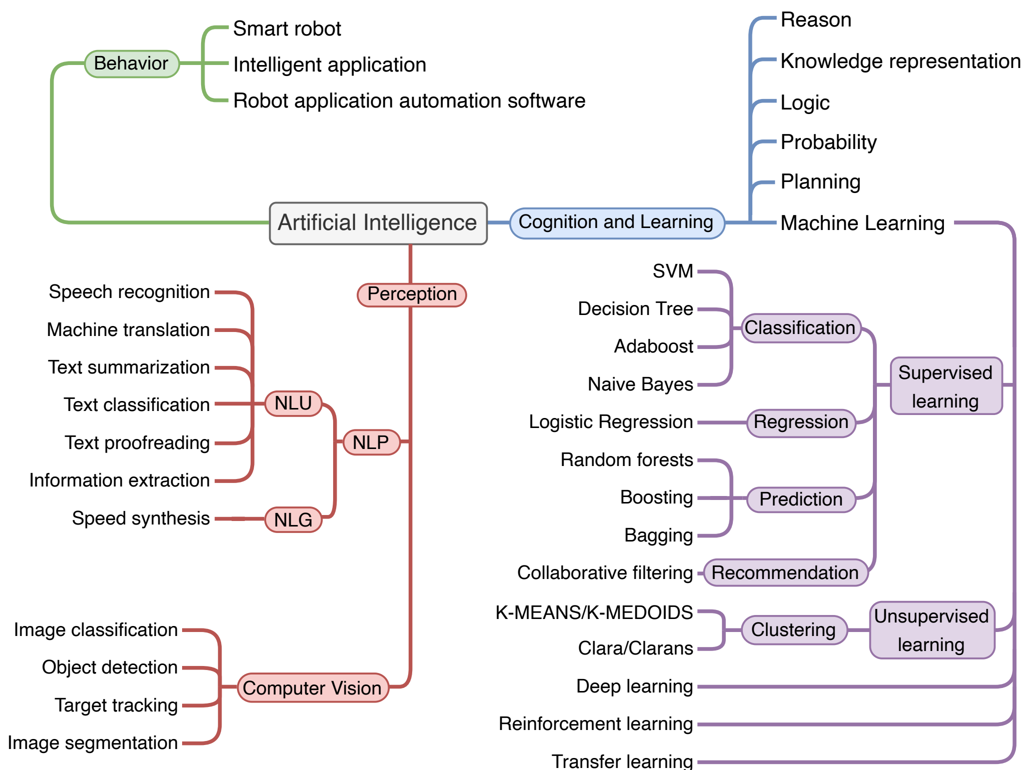


Figure 2.1.: A brief classification of artificial intelligence fields and methods. (Adopted from [WLLT21].)

Natural Language Processing (NLP) is about letting programs understand the language spoken and written by people (natural language) [CPS13]. This can be further divided into Natural Language Understanding (NLU) and Natural Language Generation (NLG). The task of NLU is to process and understand human language [CPS13]. This is done through speech recognition and semantic understanding [WLLT21]. NLG is generally about text generation using predefined semantics and grammatical rules [CPS13, WLLT21].

Computer vision tries to extract information from digital visual data, like images or videos, to understand, interpret or manipulate their content [Gol19, WLLT21]. Its primary goal is to extract target features from the data. This includes, for example, edge detection, shape detection, corner detection, and color-based segmentation [WLLT21]. Many different areas of computer vision have risen in the past years.

Figure 2.2 reviews typical computer vision areas/tasks. Image classification involves identifying and categorizing an entire image into a specific image from a predefined set of images [WS19]. The first image shows that image classification mainly works on images containing only one object with one of the given classes. Object detection extends classical image classification with a localization task. It tries to detect the objects on the image and draws an appropriate bounding box around them. This method allows for multiple instances to be present in the image. A more advanced task is instance segmentation. It intends to achieve a more fine-graded object localization. Bounding boxes contain many pixels that do not belong to the classified object. Instance segmentation accompanies that and makes pixel-wise decisions about whether it belongs to the object. This allows for distinguishing between different instances of objects with the same class. Semantic Segmentation does not identify objects. Instead, it decides for every pixel which class each pixel has. The resulting mask covers the entire image [WS19].

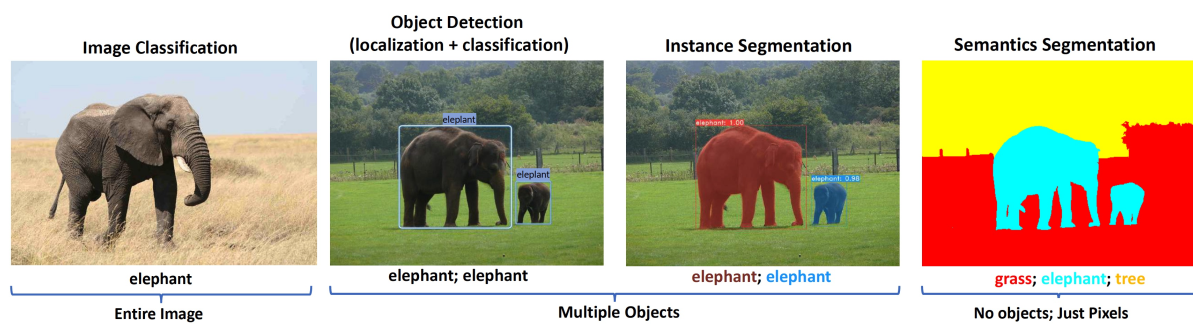


Figure 2.2.: Typical image analysis task in computer vision [WS19].

2.2. Supervised Learning

ML algorithms aim to find (learn) patterns, mappings, and relationships in the data to be able to make predictions on new data (see Figure 2.3). The learning happens through incrementally improving the performance by iteratively changing parameters in the algorithm. This is known as *training*. Supervised, unsupervised, and reinforcement

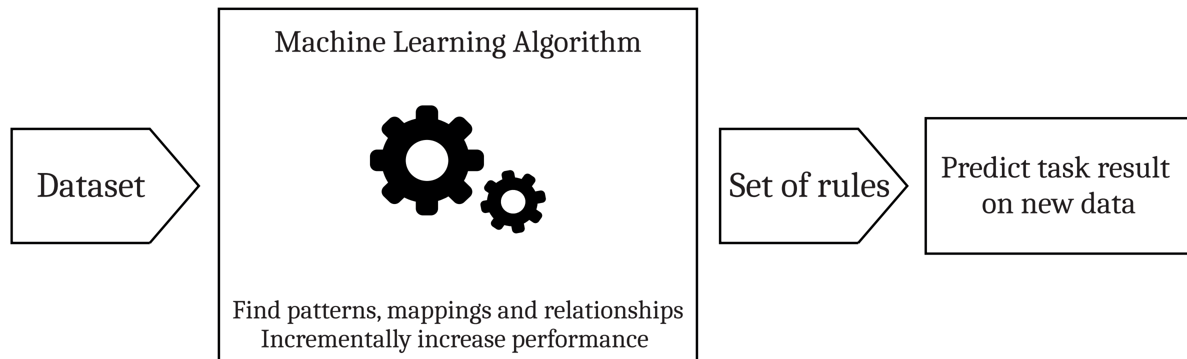


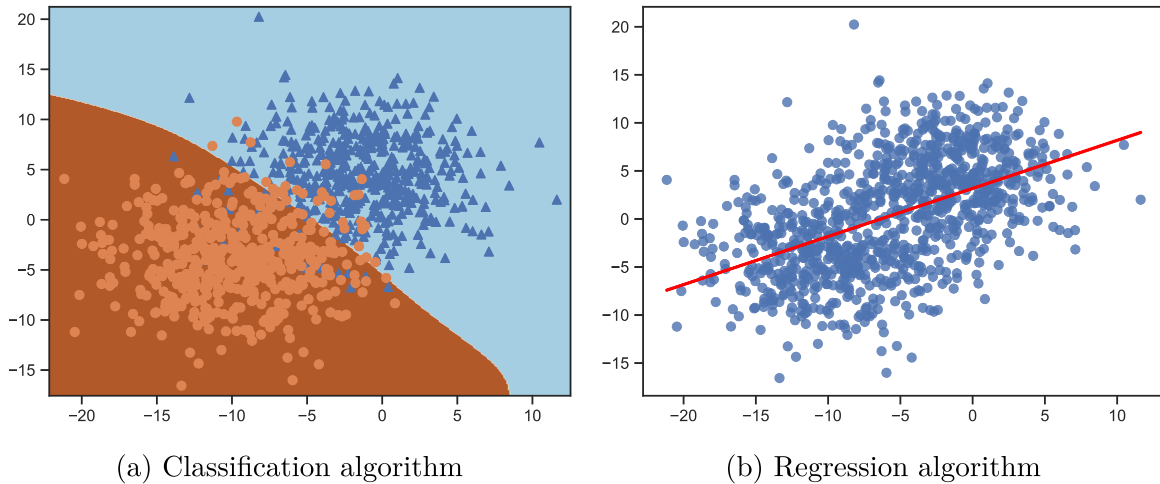
Figure 2.3.: Example workflow of a machine learning algorithm. It finds patterns in the provided dataset. The algorithm tries to improve its performance incrementally. Finally, a set of rules is found, which can be used to get the task result on new data.

learning are ML subcategories and describe different approaches to how ML algorithms learn (see Figure 2.1). As supervised learning is primarily used for computer vision tasks [GBC16], this approach will be explained in more detail. Interested readers can refer to these publications: [CHP21, Sah20, FLHI+18], for an introduction to the other learning methods.

Supervised learning is based on data where corresponding labels are available. In computer vision tasks, labels can be classes, bounding boxes, or masks. Two types of problems can be solved with supervised learning: classification and regression. As explained in the previous section, classification is about giving the input image a label. Object detection tasks use regression to approximate the position of the bounding boxes of objects. Therefore, a classification problem is about mapping the input data to discrete or categorical output variables. A *regression* problem maps the input data to continuous or numerical output [CHP21].

Figure 2.4 shows the graphical difference between classification and regression. Classification attempts to estimate an optimal border between data points so that data points of the same label are on the same side of the border (see Figure 2.4a). Regression tries to find relationships between the data points and the continuous value. This results in a function that outputs a continuous value for the input data (see Figure 2.4b). Formally,

we can define a supervised learning problem as follows: A labeled dataset D is given with $D = \{(x_1, y_1), \ldots, (x_N, y_N)\}$, where N is the number of samples, e.g., the number of images, x_i is a concrete instance of D, e.g., an image (also called feature vector), and y_i is the corresponding label. Additionally, $f(\cdot)$ describes the mapping the algorithm should learn. The label of the new data j is therefore calculated as $y_j = f(x_j)$. Classification and regression algorithms aim to minimize the error on the given samples x_i in D and generalize to unseen data samples [CHP21].



(a) Classification algorithm

(b) Regression algorithm

Figure 2.4.: Figures showing the visual difference between a classification algorithm (a) and a regression algorithm (b). The classification algorithm created a border separating data points to one of the two labels. The regression algorithm learned a linear function for approximating the continuous output value (linear regression).

Often, the dataset does not have a lot of data, which can decrease the algorithm's performance. This is where transfer learning (TL) can be helpful. TL is about improving an ML algorithm of one domain by transferring information from a related domain [WKW16]. The target domain tries to learn a mapping function $f^T : \boldsymbol{x}_T \rightarrow \boldsymbol{y}_T$, where $\boldsymbol{x}_T$ is the input data and $\boldsymbol{y}_T$ the corresponding labels. The source domain, on the other hand, has an already learned mapping $f^S : \boldsymbol{x}_S \rightarrow \boldsymbol{y}_S$. The information stored in the learned mapping f^S is used to improve the performance of f^T [ZQD+21]. Generally, the source domain has lots of input data with a larger range (e.g., many different labels), and the target domain has less input data with a smaller range (e.g., fewer labels). In this scenario, TL uses the already learned information of the source domain to improve the performance of the decision function of the target domain [ZQD+21]. How this method is often used will be explained in Section 3.4.

2.3. Feedforward Neural Networks

Together with the supervised learning approach, feedforward neural networks (FNN) are the most used method for realizing computer vision tasks [GBC16]. These networks contain a variety of different nodes that are connected to form a network. Each node has several parameters that are updated (learned) during the training. Given the dataset D with $\boldsymbol{x}$ being the input data, $\boldsymbol{y}$ the corresponding labels, the goal is to find (learn) a $\boldsymbol{\theta}$, such that

$$f(\boldsymbol{x}, \boldsymbol{\theta}) = \hat{\boldsymbol{y}}, \tag{2.1}$$

where the mapping function f is defined by the network and $\hat{\boldsymbol{y}}$ is the output. Its purpose is to accurately approximate the desired output $\boldsymbol{y}$. A successfully trained network can predict an accurate output estimate $\hat{\boldsymbol{y}}$ for a new unknown input $\boldsymbol{x'}$. Feedforward is used because the input is not passed back through the network.

2.3.1. Artificial Neurons

An Artificial neural network (ANN) often uses artificial neurons as nodes. The idea of combining multiple nodes into a network is inspired by the inner workings of the human brain [Bis94]. Neurons in the brain form a dense network with billions of neurons with up to 10,000 connections per neuron [ACG+09, Zha19]. Neurons communicate through electrical impulses. Each neuron requires a minimum amount of impulse voltage from its neighbors to fire an impulse (activation threshold). The network can change dynamically depending on the experience, e.g., learning in a classroom or a stressful event [CM08]. Fundamentally, the brain learns by continuously changing the connection strength between two neurons based on the activity of the neurons [Heb49]. These changes in the network result in a modification in the subsequent behavior, thoughts, and feelings [CM08].

Researchers tried to replicate this functionality numerically instead of electrical impulses. The *percetron*, created by Rosenblatt in 1957, is, until today, the basis of the artificial neuron [Ros58]. Figure 2.5 illustrates such an artificial neuron. It accepts n input values $\boldsymbol{x} = (x_1, ..., x_n)$. Each input value x_i is multiplied with a corresponding weight w_i of the weight vector $\boldsymbol{w}$ (connection strength). All values are summed through the function g. Further, a parameter b, called bias, is added, which is independent of $\boldsymbol{x}$. $\boldsymbol{w}$ and b are elements of $\boldsymbol{\theta}$ and therefore can be learned.

The result is the intermediate value z, which can be written as

$$z = \boldsymbol{w}^T \boldsymbol{x} + b. \tag{2.2}$$

After that, z is passed through an activation function f. It introduces nonlinearity to the otherwise purely linear computations (see Section 2.3.3). Applying f results in

$$\hat{y} = f(\boldsymbol{w}^T \boldsymbol{x} + b), \tag{2.3}$$

completing the computation with the output value $\hat{y}$, also called the neuron's activation value or prediction.

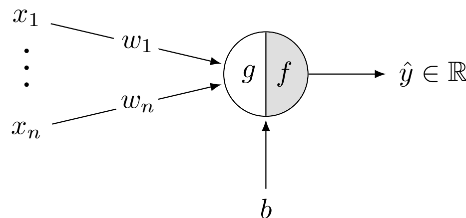


Figure 2.5.: An artificial neuron. It takes n values as input, multiplied with a weight w_i, and summed by the function g. Afterwards, bias b is added. The result is passed through the activation function f to get the prediction $\hat{y}$.

2.3.2. Multilayer Perceptron

A multilayer perceptron (MLP) is a special form of ANN that only contains artificial neurons [Kan03]. Figure 2.6 displays the structure of a MLP, generally known as a neural network (NN). It consists of multiple layers. The first layer is always the input layer, which accepts the input data $\boldsymbol{x}$ such as pixel values. This is followed by multiple layers of artificial neurons, called hidden layers. Each element of $\boldsymbol{x}$ is passed to each neuron in the first hidden layer. The calculations are the same as presented in Section 2.3.1. Each neuron of the next layer uses the computed activation values of every neuron in the previous layer as input. Therefore, this type of layer is also called *fully connected* layer [AMAZ17]. Finally, the output layer calculates the class probabilities to form the final prediction $\hat{\boldsymbol{y}}$ [Kan03, GBC16, Gé19].

The structure of a MLP allows for storing parameters and activation values in matrices. The activation value of a hidden layer is stored in a matrix $\boldsymbol{a}$, where $a_i^{[l]}$ stores the value for the i-th neuron in the l-th layer. As each neuron stores a weight for each incoming connection and a bias, a matrix $\boldsymbol{w}$ and $\boldsymbol{b}$ can be created per layer. This allows for efficient matrix calculations, and multiple elements of $\boldsymbol{x}$ can be processed simultaneously (mini-batch processing) [Gé19].

Depending on the number of layers and the number of neurons in each layer, MLPs can approximate various functions. This is referred to as the network capacity [GBC16]. Increasing the number of layers (network depth) or neurons should increase the network

capacity. Conversely, the training speed is reduced as more computations must be made, and more parameters must be updated.

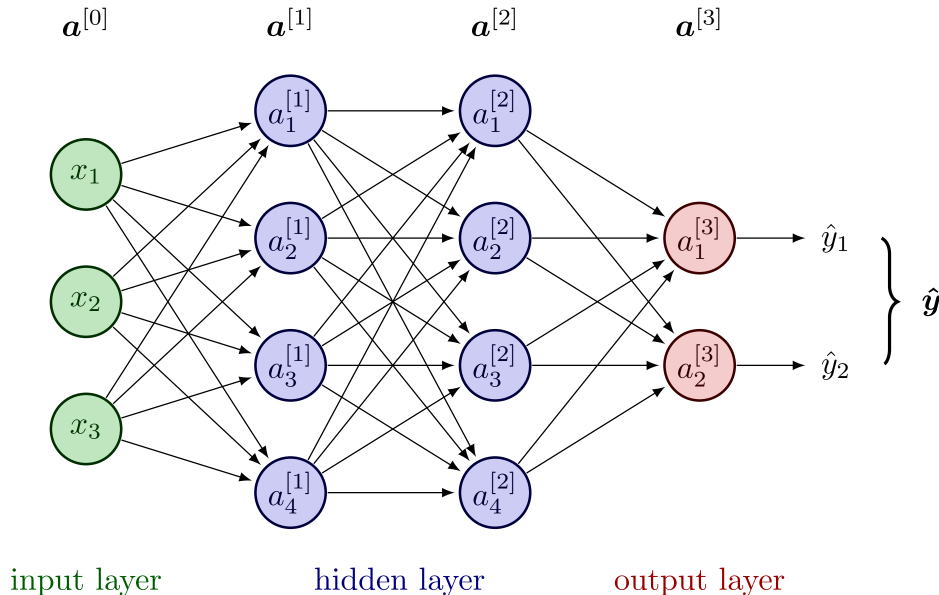


Figure 2.6.: Multilayer perceptron. The input layer represents the input $\boldsymbol{x}$. Followed by the hidden layer consisting of artificial neurons. They receive the activation values $a_i^{[l]}$ of previous layers, where l indicates the layer and i is the specific neuron. The neurons in the output layer calculate the network output (prediction) $\hat{\boldsymbol{y}}$.

2.3.3. Nonlinear Activation Functions

Artificial neurons only calculate linear functions. This results in the problem that a deeper network would behave like a single-layer network, because the sum of all layers is also a linear function. This is prevented by using nonlinear activation functions in each neuron. These functions are inspired by the thresholding functionality in the human brain. The neuron only fires (activates) if the gathered impulses are above a certain threshold (see Section 2.3.1). This thresholding in a NN is realized through the chosen activation function, where the size of the impulse is given by z.

Figure 2.7 illustrates three different activation functions. The rectified linear unit (ReLU) is one of the most common activation functions, especially in hidden layers [GBC16]:

$$f_{ReLU}(z) = max(0, z), \tag{2.4}$$

plottet on the left. Historically, was the sigmoid function (right) often used:

$$f_{sigmoid}(z) = \frac{1}{1 + e^{-z}}. \tag{2.5}$$

As the function quickly saturates for smaller and larger z values, the ability to learn with gradient-based methods is problematic (see Section 2.4). The hyperbolic tangent (tanh) improved this and is still used in some NNs:

$$f_{tanh}(z) = \frac{sinh(z)}{cosh(z)} = \frac{1 - e^{-2z}}{1 + e^{-2z}}. \tag{2.6}$$

ReLU is linear for positive z and constant for negative z values. Goodfellow *et al.* states that NNs can learn faster if the activation function behaves more closely to a linear function [GBC16].

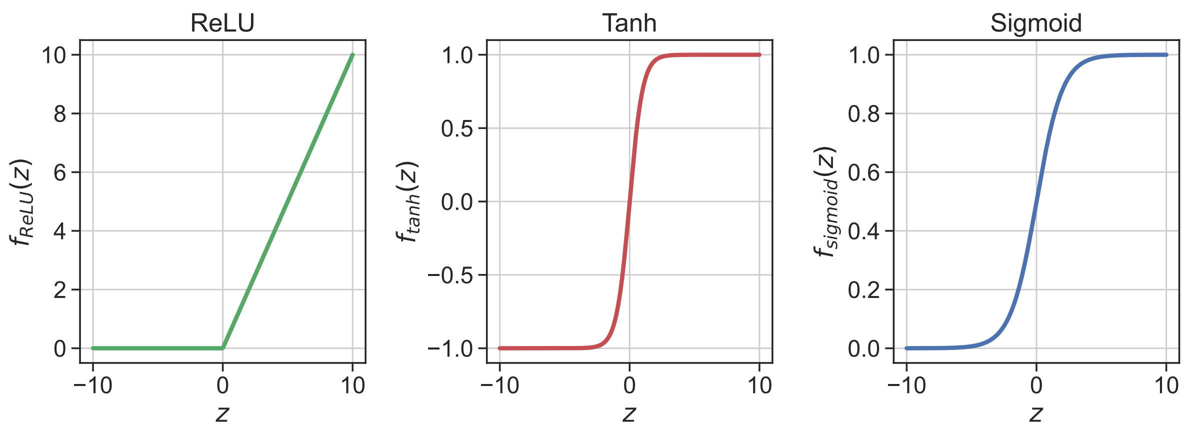


Figure 2.7.: Plots of the rectified linear unit (left), tanh (middle), and sigmoid (right) nonlinear activation functions.

The output layer of FNNs needs to be handled differently depending on the tasks at hand. For classification tasks, the activation values of the output neurons have to be mapped to probabilities in the range of 0 to 1. Given n classes, the output layer has n neurons without an activation function. For each neuron's intermediate value z_i, the softmax function is applied:

$$f_{softmax}(z_i) = p_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}. \tag{2.7}$$

It maps each intermediate value z_i to a probability p_i. The sum of all intermediate value probabilities adds up to 1 to form the final prediction probability vector $\hat{\boldsymbol{y}}$ of size n. Regression or segmentation tasks normally do not need a final activation function, as no probabilities need to be calculated.

2.4. Backpropagation

The backpropagation algorithm enables training for multilayer ANNs. The goal of the algorithm is to decrease the distance (error) between the prediction $\hat{\boldsymbol{y}}$ and the expected output $\boldsymbol{y}$ by iteratively updating $\boldsymbol{\theta}$ [RHW86].

Generally, the backpropagation algorithm is a gradient-based learning method [LBOM12]. It uses the gradient of a function to enable NN learning. Additionally, the gradient descent algorithm is used to update the parameter of the NN. Gradient descent is an iterative optimization algorithm for finding the local minimum of a differentiable function [LSJR16]. In the case of backpropagation, this function is called the *cost function*, which calculates the average error between the predictions $\hat{\boldsymbol{y}}$ of the ANN and the expected output $\boldsymbol{y}$ of the labeled dataset [LBOM12]. The error or loss between a single prediction $\hat{y}_i$ and a labeled data point y_i is calculated by the loss function $\mathcal{L}(y_i, \hat{y}_i)$. This can be an arbitrary (piecewise) differentiable function returning a scalar value. For classification tasks with n classes the cross-entropy loss is often used:

$$L_{CE} = -\sum_{j=1}^{n} y_j log(\hat{y}_j). \tag{2.8}$$

The higher the difference between the probability $\hat{y}_i$ and y_i, the bigger the loss value. The following function describes the formal definition of the cost [LBOM12]:

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(y_i, \hat{y}_i), \tag{2.9}$$

with m being the number of training elements. Thereby, the objective of the algorithm is to find the local minimum of the cost function, as this means that the ANN makes few false predictions

$$\underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, \mathcal{J}(\boldsymbol{\theta}). \tag{2.10}$$

The gradient of $\mathcal{J}$ is used to find the local minimum. Specifically, it uses the negative gradient, as following the positive gradient would lead to a local maximum. Therefore, the update of $\boldsymbol{\theta}$ can be defined as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{\varepsilon}\nabla\mathcal{J}(\boldsymbol{\theta}_t), \tag{2.11}$$

where t is the current step of the algorithm and $\boldsymbol{\varepsilon}$ is the learning rate. The update size is determined by the size of the gradient weighted by the learning rate. The learning rate is a parameter that must be chosen with care. As displayed in Figure 2.8, the gradient descent algorithm was applied for 30 steps with different learning rates. If the learning rate is too low, the algorithm needs many steps before converging. With a high learning rate, updates can overshoot or even diverge, which could result in the failure to find the

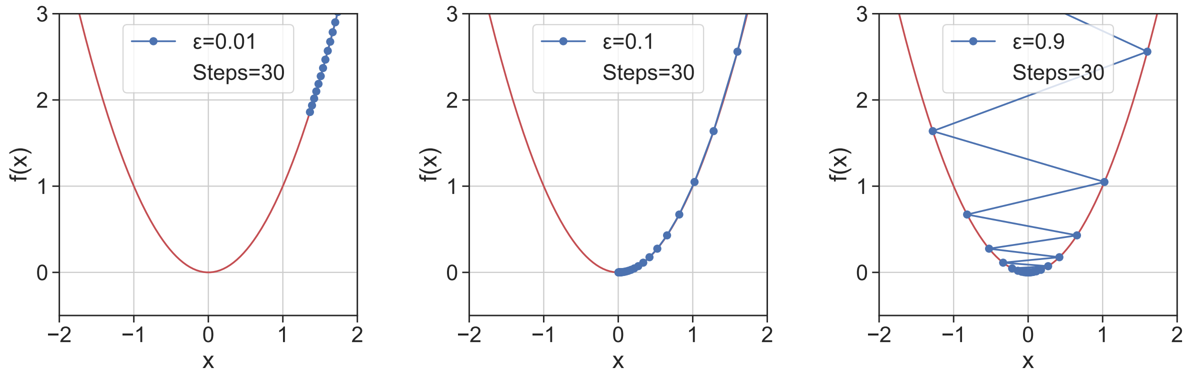


Figure 2.8.: Displays the effect of different learning rates ε for finding the local minimum with Gradient Descent. If the learning rate is too low, the algorithm needs many steps before reaching the local minimum (left). A well-chosen learning rate can find the local minimum in adequate steps (middle). If the learning rate is too high, updates can overshoot or even diverge from the actual function (right).

local minimum. A well-chosen learning rate is important for successful ANN learning in an acceptable amount of time.

In a multilayer NN, the forward pass describes the input data flow and processing through the neuron layers. As already explained in Section 2.3.2, each layer uses the previous layer's output to calculate its output value. This results in a composite function, as each neuron uses an activation function f to calculate its output. To calculate the gradient, the backpropagation starts at the output layer and flows through each layer back to the input layer. This is called a *backward pass*, hence the name backpropagation [LSM+20]. Due to the composite function, the algorithm uses the chain rule from calculus to calculate the gradient of the cost function. Given by Equation 2.9, the cost function gradient results from the average of the individual gradients of the lost function:

$$\nabla \mathcal{J}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial \mathcal{J}(\boldsymbol{\theta})}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{J}(\boldsymbol{\theta})}{\partial w_n} \\ \frac{\partial \mathcal{J}(\boldsymbol{\theta})}{\partial b} \end{pmatrix} = \begin{pmatrix} \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial w_1} \mathcal{L}(y_i, \hat{y}_i) \\ \vdots \\ \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial w_n} \mathcal{L}(y_i, \hat{y}_i) \\ \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial b} \mathcal{L}(y_i, \hat{y}_i) \end{pmatrix}. \tag{2.12}$$

This equation only describes a single neuron of one layer. If the layer has multiple neurons, $\boldsymbol{\theta}$ would be a matrix where the number of columns equals the number of neurons in that layer.

Considering a FNN with a single hidden layer and one neuron in each layer, the gradient

for the input layer (w_1) for one training example i is given by

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial a_i}\frac{\partial a_i}{\partial w_1}, \tag{2.13}$$

with a_i being the output of the neuron in the hidden layer. The gradient is computed by repeated application of the chain rule. With each layer going back, another chain rule is applied. Given the loss function $\mathcal{L} = \frac{1}{2}(y_i - \hat{y}_i)^2$, with $\mathcal{L}' = -(y_i - \hat{y}_i)$, Equation 2.13 can be written as

$$\frac{\partial L}{\partial w_1} = -(y_i - \hat{y}_i)w_2 f_2'(w_2 a_i) x f_1'(w_1 x_i), \tag{2.14}$$

where f_1 is the activation function of the hidden layer and f_2 is the activation function of the output layer. This is repeated for the remaining training examples.

Due to the repeated calculation of the chain rule, many terms are used multiple times. Additionally, the activation values of the forward pass can be reused in the backward pass (see a_i in Equation 2.14). This makes backpropagation an efficient algorithm for calculating the gradient [LBOM12].

The presented backpropagation algorithm, also known as batch gradient descent [TZZ23], uses the entire training data before updating the parameters. This can make the calculations very slow. Additionally, datasets that do not fit into the memory can not be trained with batch gradient descent [TZZ23]. Therefore, new methods have been developed that do not use the entire dataset.

The most straightforward optimization is doing a parameter update for every training example. This is known as Stochastic Gradient Descent (SGD). This algorithm is much faster. Due to the frequent updates with high variance, the objective functions fluctuate heavily. Batch Gradient Descent finds a local minimum based on the range of the parameter. SGD can potentially jump to a new or better local minimum. This causes the risk that the update overshoots (see Figure 2.8 (right)), which hinders the algorithm's convergence [TZZ23].

Mini-batch gradient descent is the most often used method when training a NN [TZZ23]. Instead of the entire training data, it uses smaller batches, commonly between 16 and 256. This allows to choose the batch size according to the dataset and the memory size. Nowadays, the term SGD is used for mini-batch gradient descent [TZZ23]. Using mini-batches reduces the fluctuations, resulting in a more stable convergence while keeping nearly the same performance as the original SGD due to highly optimized matrix operations [TZZ23].

Additional methods have been developed on top of mini-batch gradient descent to overcome specific challenges. The *Momentum* method [RM87] accelerates SGD in the relevant direction and reduces fluctuations [Qia99]. This is done by adding a fraction γ of

the previous update step u_{t-1} to the current update step:

$$\begin{aligned} u_t &= \gamma u_{t-1} + \boldsymbol{\varepsilon} \nabla \mathcal{J}(\boldsymbol{\theta}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - u_t. \end{aligned} \tag{2.15}$$

As mentioned, choosing the correct learning rate is important for fast and successful convergence. A learning rate scheduler [RM51] adjusts the learning rate if the change in the cost function result is below a threshold. The scheduler and threshold must be defined before the training and do not include specific dataset characteristics. Many different methods have been developed that enable learning rate adjustments based on the importance of the parameters in the NN. Adagrad [DHS11], Adadelta [Zei12], RMSprop [TH12], Adam [KB17], and AdamW [LH19] being the one widely known and used. How they work is beyond the scope of this thesis, but interested readers can refer to the original papers referenced above or [TZZ23].

2.4.1. Overfitting

The training data is used to minimize the cost function $\mathcal{J}$. This can bring the risk that the trained NN does not generalize well for unseen data, with possible new patterns. This is called overfitting [LG00]. The goal of a learning algorithm should be to achieve a low training error (cost) and low test cost, also called generalization error. Additionally, the gap between the training and generalization errors should be small [GBC16]. A model with low capacity cannot minimize the cost function, as it cannot capture all patterns in the training data. If the model has a high capacity, it can capture training data patterns that are not present in unseen data (test data), resulting in a larger generalization gap (see Figure 2.9).

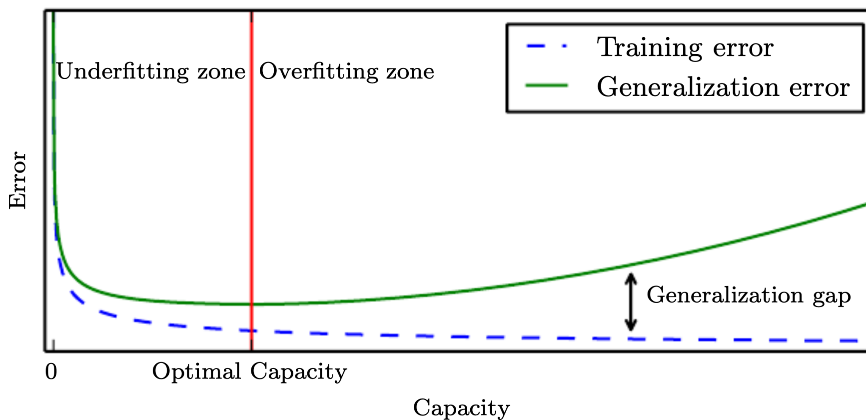
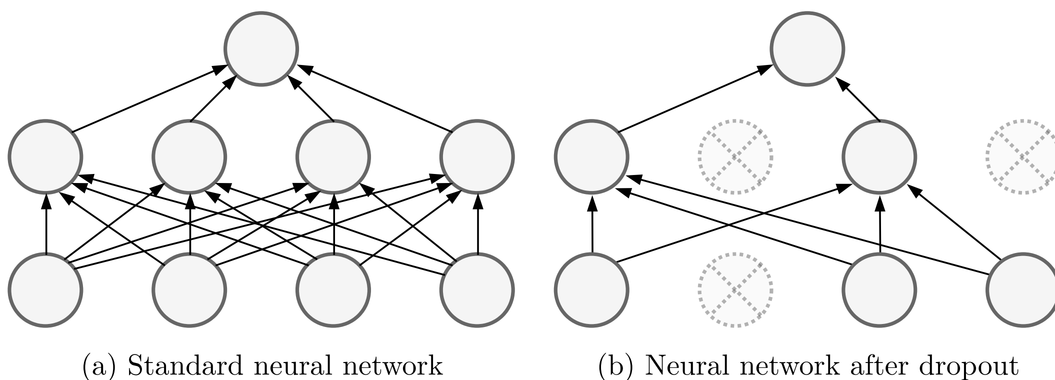


Figure 2.9.: With low network capacity, the training and generalization error is high (underfitting). If the capacity gets too large, overfitting occurs [GBC16].

One method of reducing the risk of overfitting is *dropout* [SHK+14]. Training multiple different networks and combining their prediction should improve the overall performance. Each network has a different architecture or was trained on different data to generalize better. This is computationally very expensive. Dropout mitigates this by randomly dropping out neurons in the NN, creating thinned new sub-networks (see Figure 2.10). Dropping a neuron also removes all connections to it. A neuron is dropped with a probability of p, which is often set to 0.5 and recalculated for each training batch [SHK+14]. This dropout of neurons forces the active neurons to learn features independently without depending on the input of other neurons [GZM20]. When testing the NN, the entire network without dropout is used. To average the weights of every trained sub-network, the weights of a neuron are scaled down by their probability p.



(a) Standard neural network (b) Neural network after dropout

Figure 2.10.: Affect of applying dropout. **Left:** A standard NN with a hidden layer. **Right:** The NN after applying dropout. The crossed neurons are dropped. (Adopted from [SHK+14].)

2.4.2. Batch Normalization

Another method for speeding up and improving the reliability of models is batch normalization [IS15a]. It normalizes every layer in the NN, and the normalization is computed for each mini-batch [IS15a]. Normalization speeds up the convergence, and higher learning rates can be used [IS15a, LBOM12, GZM20]. As Figure 2.11 displays, the input data is normalized to have a mean of zero and a variance of one. Additionally, batch normalization has its own learnable parameters γ and β. These parameters shift and scale the normalized data [GZM20]. As the network learns them, each batch normalization layer can find optimal parameters for the best prediction result.

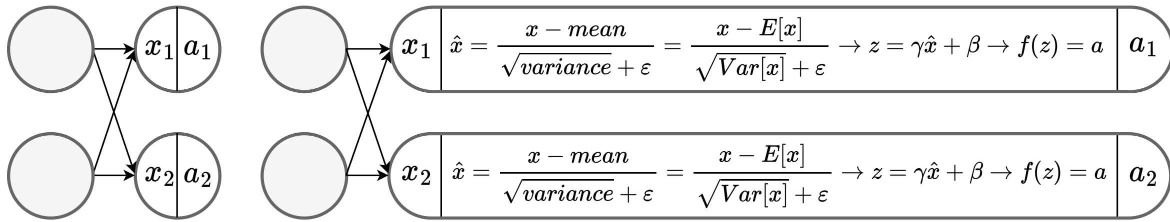


Figure 2.11.: Difference between neurons without and with batch normalization. The input data is normalized to have a mean of zero and a variance of one. The parameters γ and β are learnable. (Adopted from [GZM20].)

2.5. Datasets

Datasets are the foundation of ML algorithms and ANN. The data should represent the task that should be solved. Preprocessing is important, as the data is essential for learning success. Structuring the data and handling empty (NULL) values are examples of preprocessing steps [CHP21]. Especially for computer vision tasks, resizing the images to the same size ensures that the network learns the features correctly. This can be extended with image normalization, where the channel-specific mean and standard deviation normalize the image channel:

$$c_{norm}[j][i] = \frac{c[j][i] - mean[j]}{std[j]}, \tag{2.16}$$

where j is the current channel, e.g., red, green, or blue, $c[j][i]$ is the current pixel value of channel j, and $mean[j]$ and $std[j]$ are the mean and standard deviation values for channel j respectively [Nor23].

2.5.1. Dataset Splitting

Besides preprocessing the entire dataset, splitting it into a *training set*, *validation set*, and *test set* is common [Gé19]. The training set is used for training the NN. The training aims to learn general features for the task defined. Therefore, it is important to evaluate with the test set how the network would perform on data that it has not seen before. The validation set is used as an evaluation set during training. After a defined number of training iterations, the network is evaluated with the validation set. It identifies if the chosen NN with its hyperparameters (e.g., number of layers and neurons, learning rate, see Section 2.4) is correctly learning without running all training iterations.

2.5.2. Digital Pathology Challenges

The already introduced WSIs come with additional dataset challenges. The high resolution of WSI ($100,000 \times 100,000$ pixel is not uncommon) makes the processing more complicated [RRS+22]. Different methods must be applied to fit the images onto the hardware, e.g., scale the entire slide to a much smaller resolution or split it into multiple images. The first method is rarely used, as much detail and information is lost. Splitting the images into small patches keeps all the information, but contextual information is lost as, for example, tumor regions are split up. As explained in Chapter 2, most ANNs need labels for training. In pathology, the creation of labels is an expert-driven process. Only experts are capable of annotating a tissue slide with the correct labels. Considering the number of images and the size of each slide, this is a time-consuming and laborious task. Therefore, creating a dataset with suitable labels is very expensive and takes a lot of time. This is considered a major bottleneck in training NNs for pathology-specific tasks [CAB+21].

2.5.3. Data Augmentation

To overcome the issue of limited dataset size, data augmentation techniques are a common way to increase the size of the training set. Fundamentally, this is achieved through inserting random variations into the existing training images while preserving their labels [OSP+22]. The possible augmentation algorithms range from common image manipulation to NN, which can generate new arbitrary images [XYFP23].

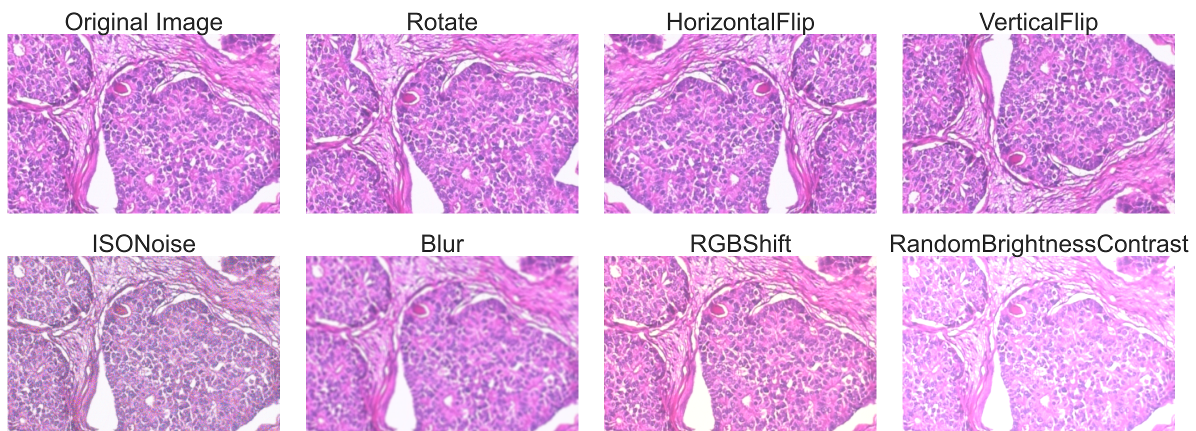


Figure 2.12.: Different augmentation methods applied to an image. (Image used from [SOPH16].)

Image manipulation algorithms are often used, as they can be easily applied during training. Methods like *translation*, *rotating*, and *flipping* are examples of geometric trans-

formations on images [XYFP23]. Additionally, noise, blur, or changes in the brightness of the images can simulate the variations occurring in image capturing with a WSS. As displayed in Figure 2.12 these methods can change the original image drastically. Therefore, evaluating which methods can be applied without changing or removing the relevant features that should be learned from the image is necessary. For example, an image of a city skyline can be horizontally flipped but not vertically.

2.6. Convolutional Neural Network

A *Convolutional Neural Network* (CNN) is a particular case of FNN [TM20]. They have shown to be very successful in the fields related to pattern recognition, especially in image processing and voice recognition [AMAZ17]. CNNs implicitly assume image-like input data [TM20]. This allows encoding specific properties directly into the NN architecture and increases computation efficiency [TM20, AMAZ17].

Like artificial neurons, CNNs have emerged from findings in medicine. Experiments performed on cats by David H. Hubel [Hub59] and T. N. Wiesel [HW59] revealed significant insights into the structure of the visual cortex. Neurons in the visual cortex only react to a small region of the visual field (small *local receptive field*). The combined receptive fields of all neurons cover the entire visual field. They also showed that some neurons only react to specific features in their local receptive field, e.g., horizontal or vertical lines. Additionally, neurons could have the same receptive field but react on different features. In addition, some neurons use the output of the neighboring neurons to react to more complex patterns. This creates a network of neuron layers, which can recognize any simple or complex feature in the entire visual field (see Figure 2.13) [GM04, Gé19].

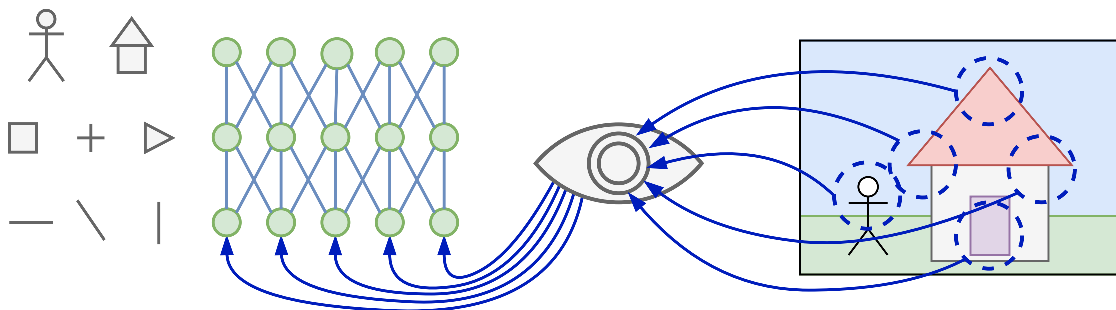


Figure 2.13.: Neurons in the visual cortex only react to a small visual field region (small local receptive field), displayed by the blue circles on the house. Some neurons react to specific features like corners or edges. Others use the output of the neighboring neurons to create a greater receptive field for detecting more complex features like shapes or objects. (Adopted from [Gé19].)

CNNs use similar feature detection with interconnected neurons to detect simple and complex features. Besides the already introduced linear layers and activation functions, they use additional layers called *convolutional layers* and *pooling layers*.

2.6.1. Convolutional Layer

The convolutional layer uses the same concept as the visual cortex. Every neuron has a local receptive field, meaning not every neuron connects to every input value (e.g., every pixel), where the output and input values are fully connected (see Section 2.3.2). This is commonly implemented with a convolutional kernel [TM20]. In the case of an image, this kernel uses a local region of the image and applies a filter. The filter is also called *feature detector*, which aims to extract specific features from the input image [AAS20]. Therefore, different types of filters can be used. The kernel slides over the image until every pixel is covered. Figure 2.14 visualizes a $3 \times 3 \times 3$ filter applied to an image. For each image channel, a 3×3 kernel is used. Afterward, all calculated values are added by g to form a combined value passed through an activation function f, creating an activation value. This creates a new "image" called *activation map*. This activation map contains the extracted features of the image. Therefore, in the case of a convolutional layer, the map is known as a *feature map* [AAS20]. Besides having multiple kernels per filter, a convolutional layer can use multiple filters to create a stack of feature maps.

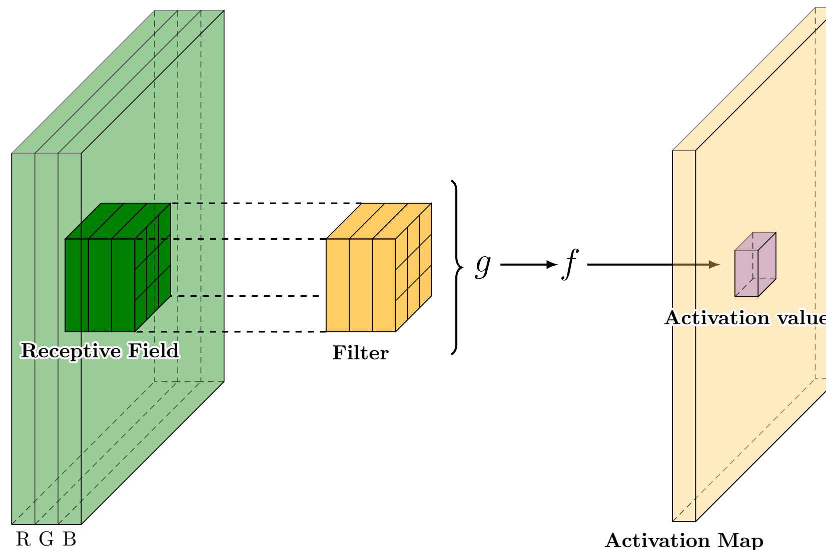


Figure 2.14.: Visualization of a convolutional layer. A 3×3 kernel is applied to each input channel. Resulting in a $3 \times 3 \times 3$ filter. The values for each kernel are combined by g and passed through an activation function f, resulting in a value of the activation map.

Convolution

The sliding filter mechanism of a convolutional layer can be implemented using a mathematical *convolution* (this is why the layer is called a convolutional layer). Given the functions x and w, the convolution $(x * w)(a)$ is defined in all dimensions as:

$$(x * w)(a) = \int x(t)w(a - t)\, dt, \tag{2.17}$$

where a is in $\mathbb{R}^n$ for any $n \geq 1$. The visual effect of a convolution with the Gaussian function $w(a) = exp(-x^2)$ is displayed in Figure 2.15.

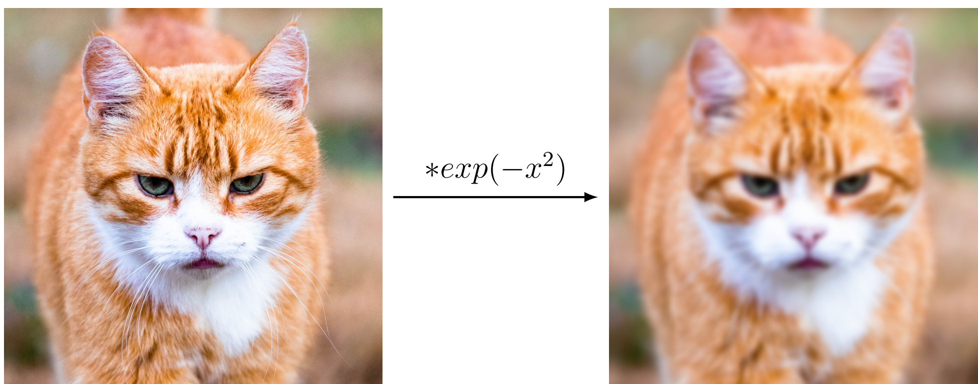


Figure 2.15.: The convolution result of an input image with the Gaussian function $w(a) = exp(-x^2)$. The Gaussian function has a blur effect on the input image.

In the context of CNN, the image-like input data does not allow for continuous functions due to the discrete nature of image sensors [TM20]. The input image $\boldsymbol{x}$ and the kernel $\boldsymbol{w}$ can be expressed as matrices. The input image is of size $\boldsymbol{x}_{n_1 \times n_2}$ and the kernel of size $\boldsymbol{w}_{m_1 \times m_2}$, with $m_1 \leq n_1$ and $m_2 \leq n_2$. With those two matrices, the discrete convolution is defined as:

$$(\boldsymbol{x} * \boldsymbol{w})(i, j) = \sum_{k_1=0}^{m_1-1} \sum_{k_2=0}^{m_2-1} x_{i+m_1-k_1, j+m_2-k_2} w_{k_1,k_2}. \tag{2.18}$$

Figure 2.16 illustrates the application of a 3×3 kernel on a 6×6 input image. The resulting feature map has a size of four 4×4. This shrinking of the output map is a common effect when applying a sliding filter to images. As the picture shows, no activation values can be calculated for the border pixels, as the kernel cannot slide further. This results in a dimension reduction of the output image. If another convolution layer uses

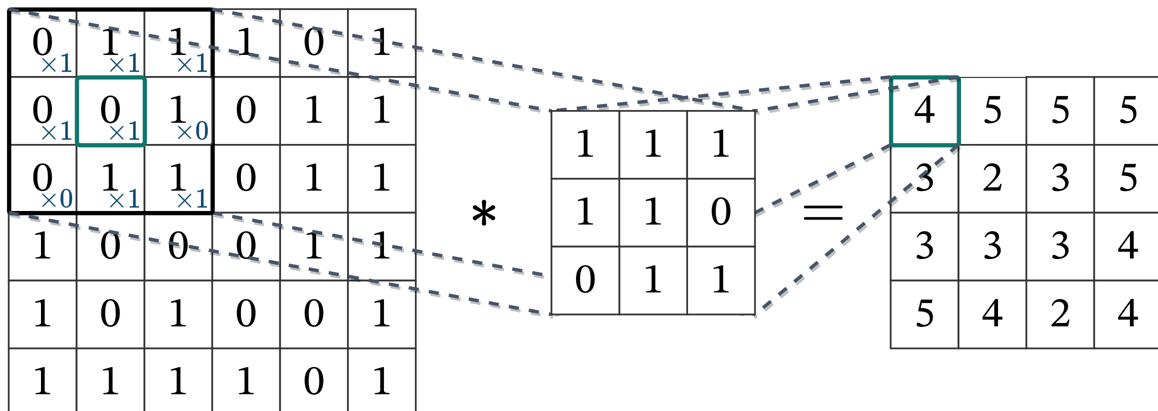


Figure 2.16.: Visualizes the convolution operation on a 6×6 input image with a 3×3 kernel. The resulting feature map is of size 4×4, as the sliding kernel cannot process the border pixel.

the feature map as input, the output will be again reduced in its dimensions. Additionally, the dimensions could be further reduced using a different kernel size or stride. The *stride* determines the size of the sliding step. After calculating the activation for one pixel, the kernel moves *stride*-many pixels further on the input image. In Figure 2.16, a stride of size 1 was used. If a stride of 2 had been used, the resulting feature map would only be of size 3×3.

Padding

This dimension reduction results in two problems. First, information is lost on the border of the image. Therefore, no features can be learned from them. Second, smaller feature maps reduce the prediction quality of the CNN [AAS20]. To overcome these problems, padding is added to the input image and feature maps.

This method is called zero-padding, as zeros are added around the border of the input. The amount of padding p determines the size of the output. Three different modes can be defined: *full*, *same*, and *valid*-padding. Valid-padding uses $p = (0, 0)$ (no padding), as displayed in Figure 2.16. In the case of full-padding, $p = (m_1 - 1, m_2 - 1)$ increases the output's dimension. Same-padding keeps the output dimension equal to the input dimension. The size of the padding depends on the filter size and stride value.

Sometimes, the reduction of the input size is necessary. Smaller input reduces the computational effort and therefore speeds up the CNN [AAS20, TM20]. To effectively reduce the size without losing feature information requires different algorithms. In CNNs, these are implemented in so-called *pooling layers* [AAS20, TM20].

2.6.2. Pooling Layers

The purpose of pooling layers is to reduce the dimension of the feature maps while keeping the most important features. Pooling layers work with the same mechanism as the convolutional layer. A sliding kernel moves over the input image or feature map and extracts new values depending on the chosen kernel. Instead of a convolution operation, the kernel uses only the values of the input. Commonly Max-Pooling- or Mean-Pooling-Layers are used [TM20, AAS20]. As displayed in Figure 2.17, pooling layers mostly use a 2×2 kernel with a stride of two [TM20, AAS20]. This creates non-overlapping rectangular neighborhoods. In each neighborhood, the maximum value is selected. In the case of a Mean-Pooling-Layer, the average of all values in the neighborhood would be calculated. In this Max-Pooling-Layer configuration, the output would be an activation map of size 3×3.

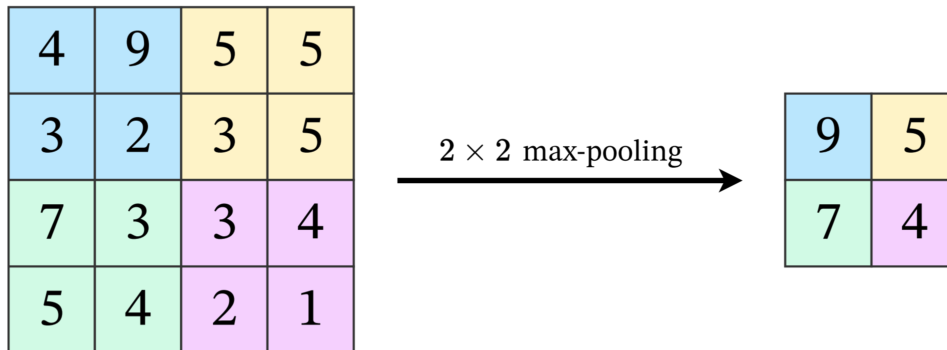


Figure 2.17.: Application of a Max-Pooling-Layer on 4×4 input data. The layer uses a 2×2 kernel and returns the maximum value of every rectangular neighborhood.

3. Building Blocks of Neural Networks

After introducing the general inner workings of NNs and CNNs, this section presents different building blocks and state-of-the-art NNs to realize computer vision tasks. These NNs are comprised of many of the previously presented layers. Combined with the backpropagation algorithm, they can learn intricate features from the input data. Such NNs are often grouped under the category *deep learning* (DL) [LBH15].

3.1. Classification

As explained in Chapter 2, classification tasks in computer vision are about giving an entire image a class label. The labels can either be binary, e.g., whether the image is tumors or not, or use multiple labels, e.g., what type of cancer is on the image. Many algorithms and ANN architectures have been developed to solve classification problems [LLR+22]. Often, ANN architectures based on CNNs are used [LLR+22, DCM+21]. State-of-the-art CNN architectures use specific combinations of convolutional, pooling, and fully connected (FC) layers (see Section 2.3.2) [DCM+21]. The primary goal of convolutional and pooling layers is to learn features from the training images to distinguish the defined classes. The FC layers are behind the feature learning layers and map the two-dimensional feature maps to the actual class probabilities.

LeNet-5 is known as the first CNN architecture, which was used for recognizing handwritten characters [LBBH98]. As displayed in Figure 3.1, it uses a simple combination of convolutional, pooling, and (FC) layers. Later architectures use wider (larger kernels) and deeper (more layers) models (e.g., AlexNet [KSH12] and VGG [SZ15]). Using larger kernels and more layers allows for learning more complex features. On the other hand, this results in many learnable parameters. For example, the VGG-16 architecture uses 16 convolutional layers (excluding the pooling layers) and has 138 million learnable parameters. As every parameter is stored on the GPU's memory, the memory profile is very high, and the learning is slow, as complex gradient calculations are needed to update every parameter [AŠ20].

Increasing the number of used layers should theoretically increase the model's performance [AŠ20]. When using backpropagation as a learning algorithm, degradation can appear on deeper networks. As the derivates of each layer influence the previous layer,

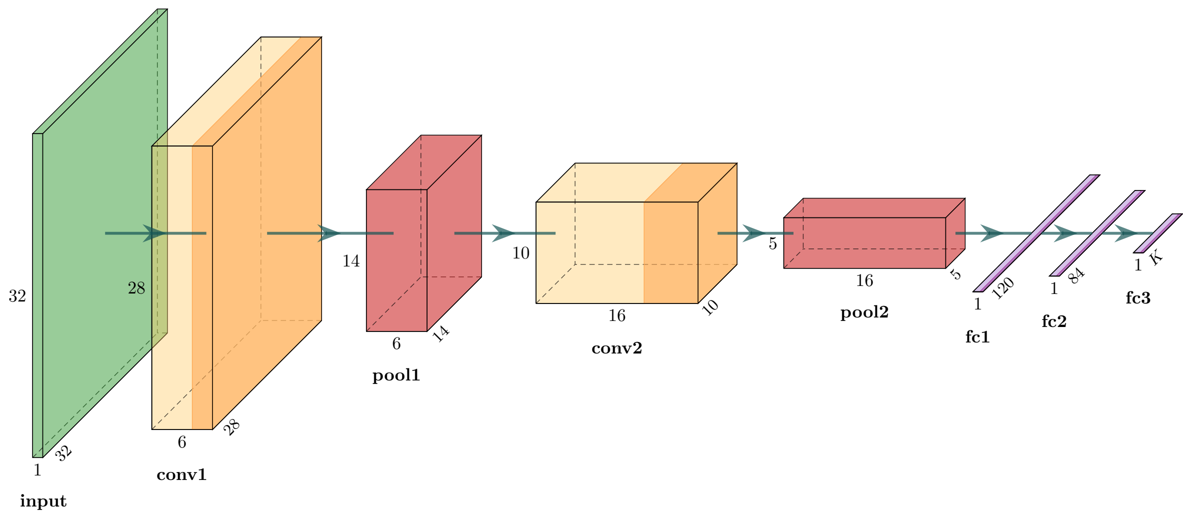


Figure 3.1.: Diagram of the LeNet-5 architecture. It uses two convolutions (yellow), two pooling (red), and three fully connected layers (purple). The last layer has K output neurons. This can be adjusted to the number of classes/labels in the chosen dataset. The model requires 32×32 images with one channel as input (green) [LBBH98].

small partial derivatives are exponentially passed through the network, resulting in the *vanishing gradient* problem. Too small gradients hinder the learning process, as the algorithm possibly converges too early, resulting in degradation and a high training and test error [HZRS15]. He *et al.* [HZRS15] proposed residual learning to overcome degrading. As explained in Section 2.6, a convolutional layer tries to extract specific features from the input data. The layer learns a mapping of the input data x to the features. The presented architectures rely on finding specific mappings $\mathcal{H}(x)$ in the convolutional layer stacks to reach good performance. In residual networks, instead of expecting that the layers successfully learn the $\mathcal{H}(x)$ mapping, an explicit residual function $\mathcal{F}(x) := \mathcal{H}(x) - x$ can be approximated. The original mapping, therefore, becomes $\mathcal{F}(x) + x$ [HZRS15]. In a FNN *skip connections* are used to realize this function. As displayed in Figure 3.2, the skip connection adds the input x to the output of the last layer in the stack before it is passed through the activation function. The skip connection uses a simple identity function and adds no additional parameters or computational complexity to the network [HZRS15]. This architecture is also called a *residual block*.

As mentioned in Section 2.6, convolutional and pooling layers can change the dimension of the input data x. The skip connections perform a projection with 1×1 convolutional layers to match the dimension of the output of $\mathcal{F}(x)$. The skip connections enable passing information from previous layers to layers much further in the network. This allows much deeper networks, as information is preserved and the degradation process is reduced.

The residual networks (ResNets) use different residual blocks depending on the depth of the network (see Figure 3.3). It uses two 3×3 kernels for shallower networks or two 1×1 kernels and a 3×3 kernel for deeper networks. The 1×1 kernels are for reducing and restoring the input data's dimension. The 3×3 kernel operates on smaller input/output dimensions, decreasing the computational effort and the overall training time [HZRS15].

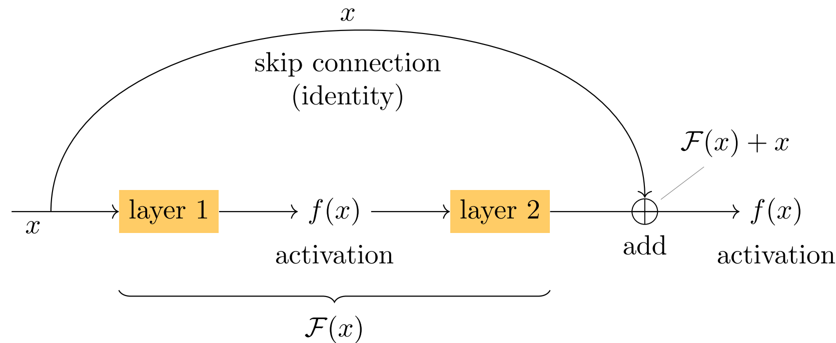


Figure 3.2.: A residual block uses a skip connection to add the input x present before the first layer to the output of the last layer. The stack of layers learns the residual function $\mathcal{F}$, and x is added elementwise.

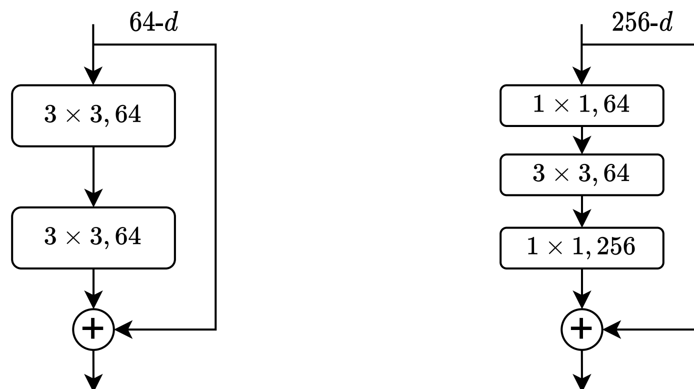


Figure 3.3.: Different residual blocks. Left uses two 3×3 kernels. The right diagram displays a "bottleneck" residual block. It uses two smaller 1×1 kernels and one 3×3 kernel [HZRS15].

Convolutional layers are always stacked sequentially in the presented architectures and building blocks. Therefore, the selected kernel size in each layer is essential for the feature extraction. The inception block enables parallel use of different kernel sizes

[SLJ+14]. Therefore, simple features with smaller kernel sizes and complex features with larger kernels can be learned simultaneously. Figure 3.4 displays the architecture of the inception block. It uses a variety of 1×1, 3×3 and 5×5 kernel sizes. It uses a similar concept of dimension reduction as the "bottleneck" residual block by first applying a 1×1 convolution kernel (yellow) to reduce dimensions and then applying the larger kernels (blue) to increase the performance. In the last layer, all paths are concatenated to create a combined feature map. This inception block was first introduced and integrated into

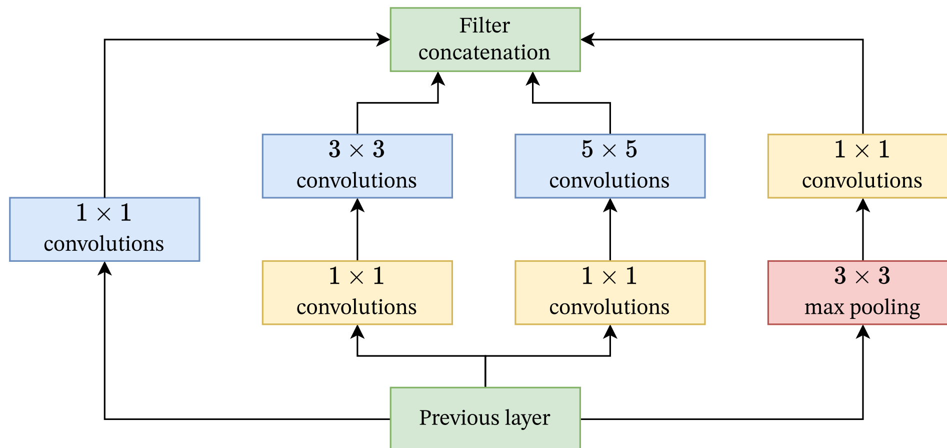


Figure 3.4.: The inception block. (Adopted from [SLJ+14].)

the GoogleNet architecture in 2014 [SLJ+14]. Since then, several versions of architectures have been created that utilize inception blocks [IS15b, SVI+15, SIVA16]. Some also extended the inception block with residual blocks by adding skip connections [SIVA16].

In recent years, *Transformer* networks have shown promising results in computer vision tasks, outperforming state-of-the-art CNN [BCG+21]. They were first introduced in NLP, but their concepts could be translated to images [VSP+17]. Fundamentally, they do not use convolutional operations [BCG+21]. Therefore, they are not explained in more detail in this thesis.

3.2. Object Detection

Object detection tasks extend the typical classification task. As Chapter 2 mentions, object detection involves localizing a specific class/object on an image. This makes the task complex, as the object's spatial position and size are relevant for the detection (see Figure 3.5). Two main approaches have successfully detected objects on images: object detection based on regions and object detection based on regression.

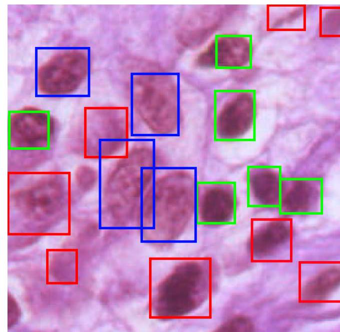


Figure 3.5.: Bounding boxes drawn around the nucleus of cells. The color indicates a specific label. These bounding boxes can be used to train object detection NN. (Image and bounding boxes obtained from [AAH$^+$22].)

Object Detection Based on Regions

As illustrated in Figure 3.5, the objects to be detected may vary in size. Solving this task with state-of-the-art CNN architectures (see Section 3.1) would require using different-size sliding kernels to ensure no objects are missed during feature extraction. Additionally, multiple instances of the same object could be present in the image. Hence, a thorough and precise search of the entire image is necessary to avoid overlooking any probable object positions. The training of such a CNN is very complex and time-intensive. To reduce the number of regions to analyze, the method of *region proposal* was developed [UvGS13, ZJ17]. Region proposal figures out the possible regions of interest (ROI) on the image where an object could be located in advance, so fewer regions must be investigated for localization and classification. Different algorithms can be used for finding those regions, e.g., selective search, edge Boxes, etc. [ZJ17].

The Regions with CNN features (R-CNN) [GDDM14] architecture, developed by Girshick *et al.* in 2014, is one of the first methods that combined region-based proposals with CNNs. It uses the selective search algorithm to propose approximately 2,000 ROIs. For each ROI, a CNN is trained separately. The ROIs are cropped and warped because FC layers only work on fixed-size input and output [ZJ17]. The learned features are

combined with some classifier algorithm or NN (e.g., Scalable Vector Machine or FC layers in combination with the softmax activation function) to classify which object is present in the image. An additional NN comprised of FC layers predicts the bounding boxes of the found object [GDDM14].

As training a CNN for every ROI is time-consuming, improvements were proposed. The Fast R-CNN trains a CNN on the entire input image and makes region proposals on the learned feature maps [Gir15]. As the classifier and regression model only accept fixed-size input vectors, the ROIs are resized through "ROI pooling" [ZJ17]. ROI pooling is based on the Spatial Pyramid Pooling layer proposed by He *et al.* [HZRS14]. As Figure 3.6 displays, each ROI is divided into a grid that matches the desired output size. Due to the different aspect ratios of the ROI, the grid cells can be of different sizes. In each grid cell, max pooling is applied, resulting in the required output data size.

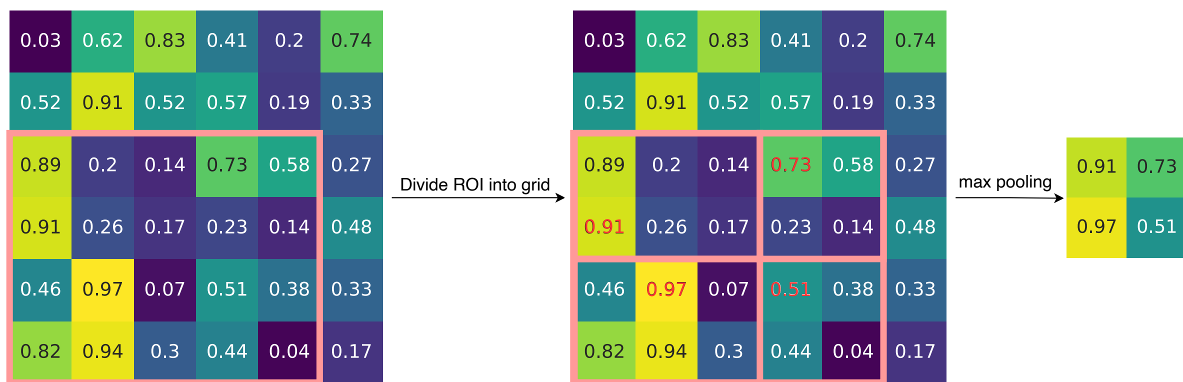


Figure 3.6.: Application of ROI pooling onto an example feature map.

The selective search algorithm reduces the overall performance, as it runs on the CPU and is generally compute-intensive. The Faster R-CNN architecture uses a NN called a Region Proposal Network (RPN) [RHGS16]. This NN is trained to propose the ROIs. The overall speed of the R-CNN network is increased as the training and proposing can be done on the GPU.

Object Detection Based On Regression

The presented region-based methods use two main stages: generating ROI and making predictions on the ROIs (bounding boxes and class probabilities). You Only Look Once (YOLO) methods skip the ROI generation process and use a regression approach. As Figure 3.7 displays, the regression idea is based on dividing the image into a grid of size $S \times S$, and each grid cell predicts bounding boxes with a confidence score and class probabilities, removing the necessity of the region proposal algorithm [RDGF16].

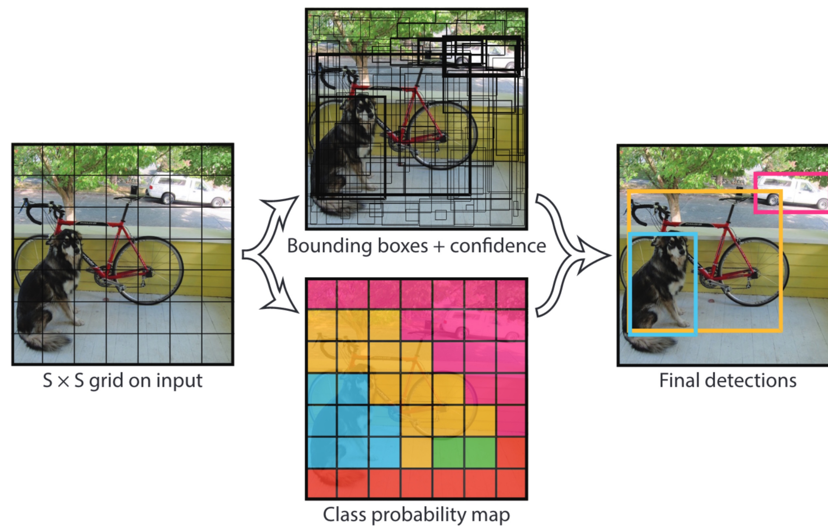


Figure 3.7.: The general method of You Only Look Once. The image is divided into a $S \times S$ grid. Bounding boxes with a confidence score and class probabilities are predicted for each grid cell. In the final step, these are combined to create the final object detections [RDGF16].

The main advantage of the regression method is faster prediction. This is especially important for object detection on videos. The model prediction must be as fast or faster as the video's frames per second (FPS). If it can reach 30 FPS, the model is capable of real-time object prediction. Neither of the presented region-based object detection models reaches this threshold [RDGF16]. The YOLO models, on the other hand, can reach hundreds of FPS. [RDGF16, TC23]. Higher prediction speeds often come with a tradeoff in accuracy [RDGF16, TC23].

3.3. Image Segmentation

As briefly introduced in Chapter 2, image segmentation is an extension of object detection. Image detection is about finding the concrete shape of an object on an image, not just the bounding box. As displayed in Figure 3.8, concrete annotations or masks of the objects in the image are needed to train segmentation NN successfully.

A state-of-the-art instance segmentation NN is the Mask R-CNN [HGDG18] architecture. It extends the Faster R-CNN with an additional step after the ROI pooling for generating segmentation masks. For each ROI, a CNN is trained that outputs a binary mask. The used CNN is known as a fully convolutional network (FCN) [HGDG18]. FCNs only consist of convolutional, pooling, and upsampling layers [LSD15]. Therefore, they allow arbitrary input sizes, as no FC layers are present. Upsampling layers

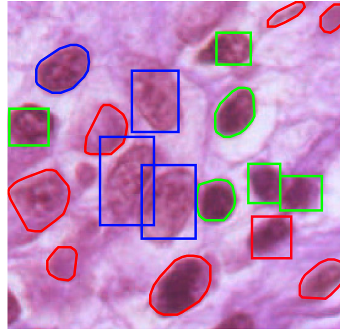


Figure 3.8.: Annotations of the nucleus of cells. These offer a precise annotation of the nucleus compared to bounding boxes. The color indicates a specific label. Those polygons can be used to train segmentation NN. (Image and annotations obtained from [AAH+22].)

can be interpreted as an inverse convolution operation with a fractional stride of $\frac{1}{s}$ and is implemented using interpolation [LSD15]. Figure 3.9 displays the architecture of a FCN. It uses the same structure as presented in the previous architectures but does not output a one-dimensional vector. Instead, it outputs K maps with the same size I as the input image. This allows for pixel-wise predictions, which can be used for semantic segmentation [LSD15]. The upsampling layer, also called *deconvolution layer*, ensures that the input and output sizes are equal. The primary difference between the output of

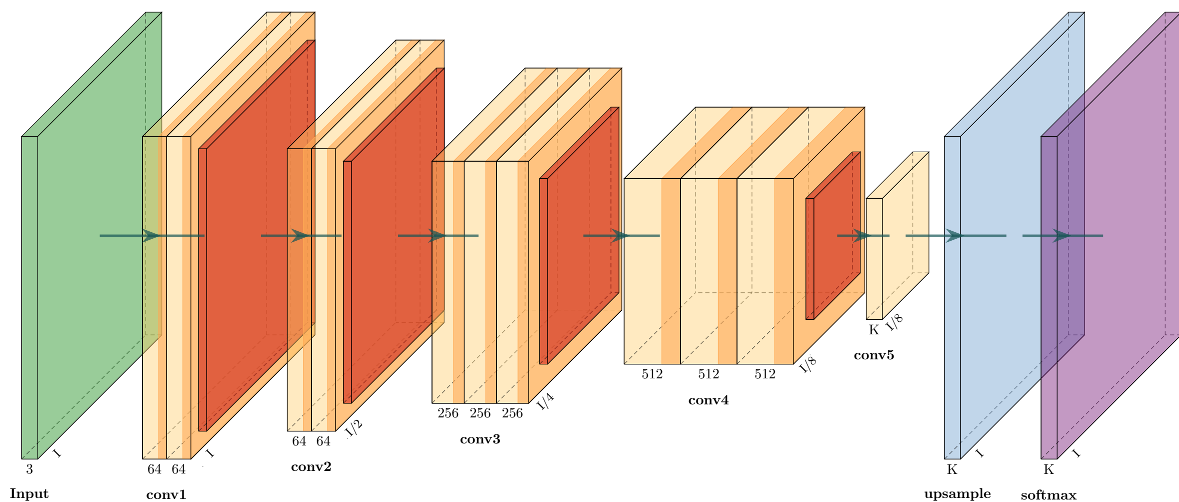


Figure 3.9.: A fully convolutional network. It can process an arbitrary input size K and output a size $K \times I$ map as it only uses convolutional operations. Pooling layers are used to downsample the data (orange). The Upsampling layer (blue) increases the map dimension to the desired output size. A final softmax layer is used to get pixel-wise probabilities.

a classification CNN presented in Section 3.1 and a FCN is shown Figure 3.10. Instead of outputting probabilities for each provided label through FC layers, a FCN outputs probabilities for each label for every pixel in the input image through deconvolution layers. Additionally, FCN can use skip connections from lower layers to higher layers to

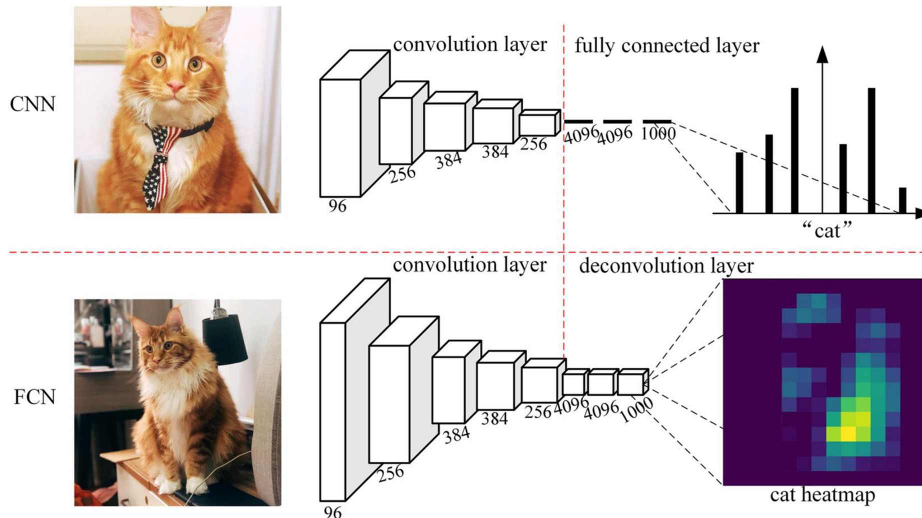


Figure 3.10.: A classic CNN outputs probabilities whether a cat is on the image or not. The FCN outputs a heatmap of the probabilities for each pixel if a cat is visible [YMW+20].

combine information and improve prediction performance, as shown with the ResNet architecture (see Section 3.1) [LSD15]. Therefore, the Mask-RCNN trains a FCN for each ROI. In combination with the predicted class, each mask can be labeled individually to enable instance segmentation.

Another prominent architecture, especially for medical image segmentation, is the U-Net architecture [RFB15]. The main difference between FCN is that it uses transposed convolutional layers instead of interpolation for upsampling. Those layers can be learned to help recover spatial information [RFB15]. Figure 3.11 displays the building blocks of the architecture. It can be split into two parts: the encoder and decoder path. The encoder path uses several convolutional and pooling layers to downsample the input image and learn the feature maps. The decoder pass uses the feature maps and reduces the number of channels. The segmentation map produced has the same width and height as the input image, and the channels match the number of labels defined. The skip connections concatenate the feature maps learned in the decoder path with the corresponding layer in the decoder path. These feature maps contain high-resolution features about the image and support the decoder path to construct a new image.

Like the R-CNN architecture family, the U-Net uses state-of-the-art CNN architectures to learn the input image's features in the decode path. These CNNs are essential for

successful object detection and segmentation, so the selected architecture is often called a *backbone*. A common technique for those backbones is to use already trained models on large datasets to improve the general performance and reduce the necessity of extensive training data.

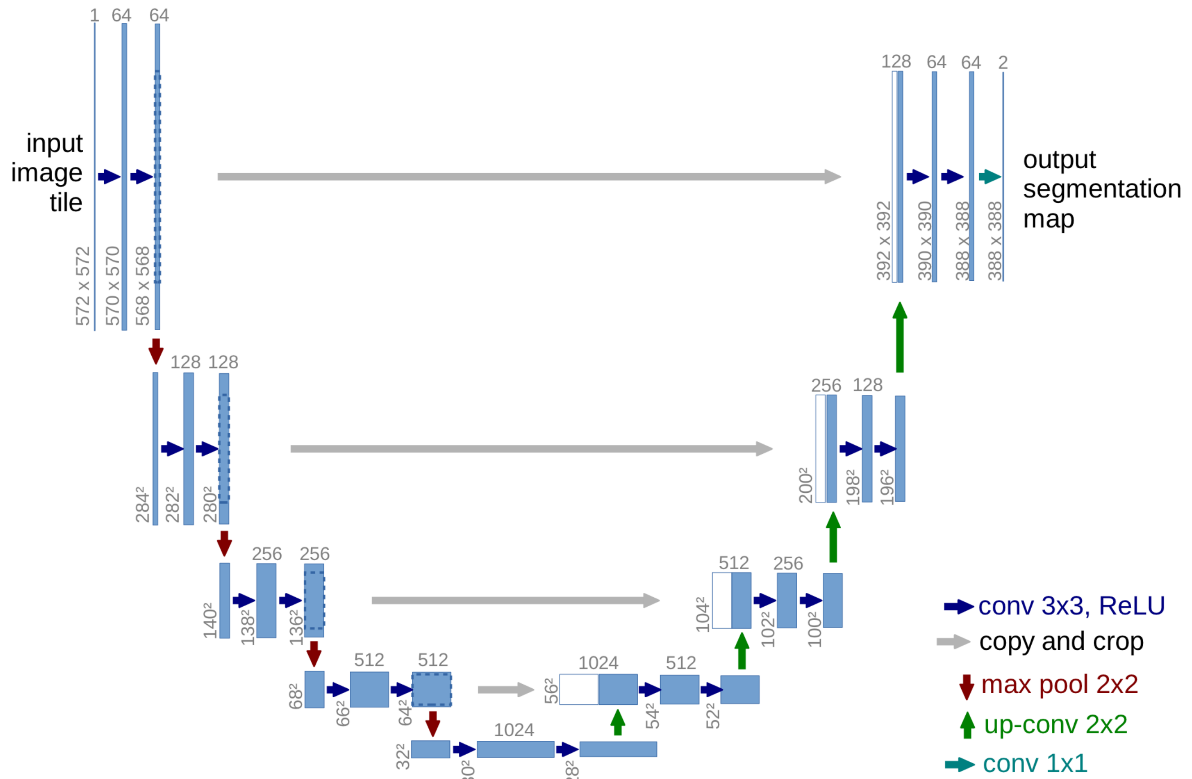


Figure 3.11.: The U-net architecture [RFB15].

3.4. Pre-Trained Models

As already mentioned, having large amounts of training data is essential for the performance of deep CNNs. Often, this data is not available for specialized tasks. As mentioned in Section 2.2, TL is used to transfer information from a trained model to improve the performance of another model. As displayed in Figure 3.12, the pre-trained model was first trained on a larger dataset, e.g., the ImageNet dataset [RDS+15], that captures a variety of different images. The features (weights) learned are then transferred to the model of the specialized task.

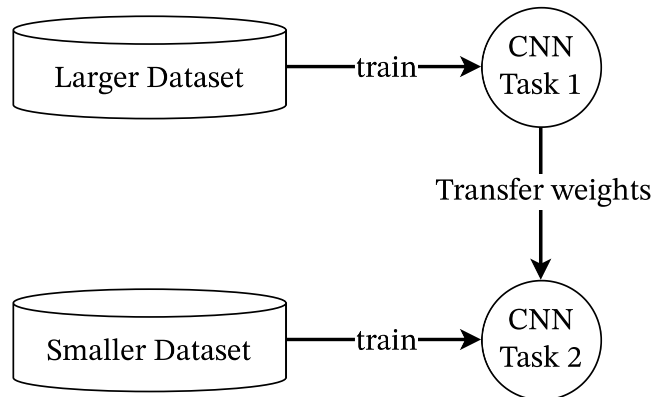


Figure 3.12.: Transfer Learning in the case of CNN training. The weights of a CNN trained on a larger dataset can be used to train the CNN on smaller datasets to increase its performance and reduce training time.

As explained in Section 2.6, the first layers in a CNN capture simpler features, like lines or shapes. Only the last layers capture the task-specific features. Thereby, those layers are replaced. In the presented architectures, these are the FC layers. Their weights are learned during the training (see Figure 3.13).

A comparison in Section A.1 showed that pre-trained models have an approximately 20% higher accuracy on the test dataset. Additionally, they converge faster. It was also tested how the model performs if a pre-trained model was trained on a completely different domain (ImageNet) or a comparable medical domain (histology images). The ImageNet dataset used for the pre-trained model contains 1.281.167 training images and one thousand different classes. The histology dataset contains 159.267 training images and only two different classes. The test accuracy on the ImageNet dataset showed an approximate 0.5% higher accuracy than the medical domain (99.25% and 98.74%). Therefore, it made nearly no difference which pre-trained model was used. Whether it is generally true that a comparable performance can be accomplished with pre-trained models from the same domain with potentially less data must be evaluated in a more complex test.

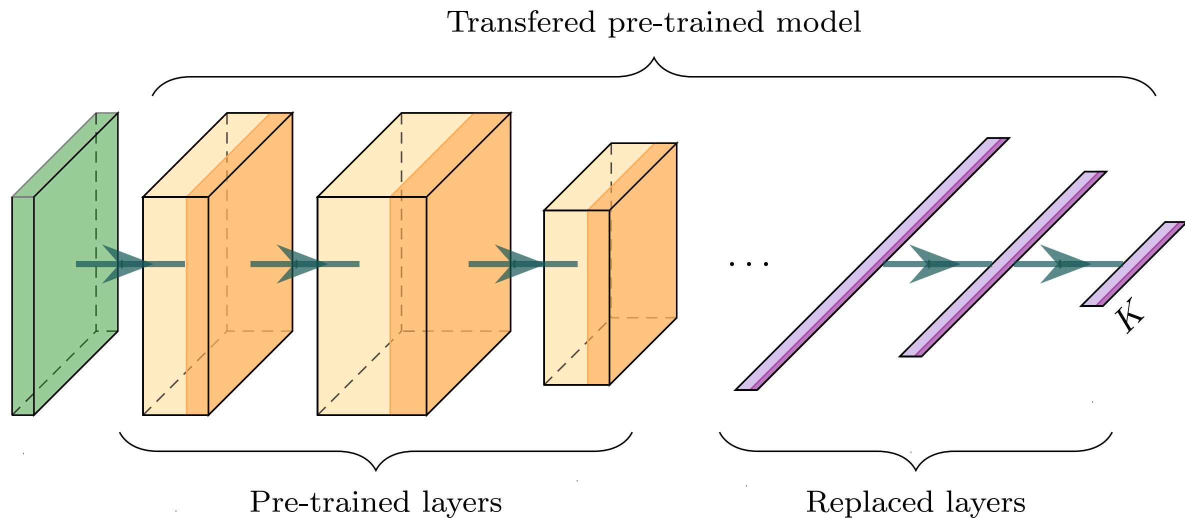


Figure 3.13.: Diagram of a model that uses transfer learning to train. It uses pre-trained layers in the first layers and untrained layers in the last layers. The last layers are then trained in the training process to fit the required specifications of the task.

Other literature also confirms that pre-trained models show better performance and save time, as no laborious feature engineering and fine-tuning of the NN is needed to receive good results on the dataset [SRG+16, KBKT17, JNS22].

4. The Artificial Intelligence Lifecycle and Software Tools

After presenting building blocks and state-of-the-art neural networks for computer vision tasks, this section will present the lifecycle of AI. Additionally, programming frameworks for creating and training NNs will be evaluated. Lastly, existing End-To-End Artificial Intelligence platforms that cover the AI lifecycle are compared.

4.1. The Artificial Intelligence Lifecycle

Creating AI involves different steps. Figure 4.1 displays a high-level ML/AI lifecycle. As explained in Section 2.5, data is essential for the performance of AI models. Thereby, the lifecycle begins with data collection and preprocessing. This results in a dataset that can be used for model training. Model training is about choosing and training an AI architecture, which is then evaluated to determine whether the model meets the desired results. In the final steps, the trained model is served for users to make predictions or inferences on data [SFH21].

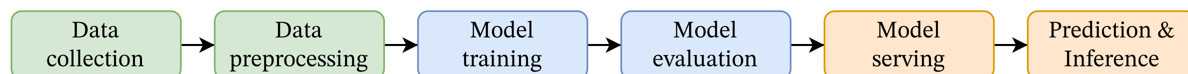


Figure 4.1.: A high-level Machine Learning/Artificial Intelligence lifecycle. It starts with the collection and preprocessing of data. Afterward, a fitting AI model has to be trained and evaluated. If a good performing model is found, it is served on which prediction and inference can be done. (Adopted from [SFH21].)

These steps require specific hardware and software tools. Datasets need to be easily usable and modifiable. Besides much storage, software is needed to manage the datasets with an easy-to-use interface. AI models can be created by writing code in a programming language that describes the model architecture, training, and evaluation workflow. This requires libraries that ease this process. Training a model needs specialized hardware and software to allow efficient training and orchestration between multiple servers. A general introduction to the specialized hardware and software is given in Section 4.2.

Finally, model serving makes the trained model available to users for predictions and inference on data. As different users can interact with these models, features like privacy, access control, auditability, logging, and monitoring are key aspects that need to be considered [SFH21].

In software development, "DevOps" is a common term for describing the continuous process of developing and operating software [EGHS16, AAK17]. It includes all stages of the general software development cycle: planning, requirement analysis, design, implementation, testing and integration, and maintenance. DevOps tries to maximize the automation of this lifecycle with software tools to improve the transitions between the different phases and ease communication between teams and customers. Methods like Continuous Integration (CI) and Continuous Delivery (CD) are commonly used [EGHS16, AAK17]. These concepts can also be transferred to the ML lifecycle. Machine Learning Operations (MLOps) is about automating the processes of dataset creation, model training, and model serving, providing methods and software tools to ensure continuous development and integration of ML models [SFH21]. The following sections present popular DL frameworks that allow the creation and training of AI models. Then, different software tools are compared that enable MLOps to form an end-to-end AI platform.

4.2. Choosing a Deep Learning Framework

With the emerging adoption of AI in industries, the development of frameworks to create and train AI models has also increased rapidly. Training deep learning models is very compute intensive, considering their use of many different layers and the vast amount of data to process. As explained in Section 2.6, the core of computations are matrix operations. CPUs are not optimized for these operations and can not process this much data with high throughput. Over the years, the training process shifted to GPUs. GPUs are fundamentally used for graphics processing and are highly optimized for integer and floating-point operations. They have to perform fewer instructions in a very short time for a lot of data, e.g., a shader program needs to be applied to every pixel on the screen thirty times a second. These processing tasks can be heavily parallelized. Thereby, GPUs have a lot more cores compared to CPUs.

In 2007, NVIDIA released the CUDA programming model to write and run highly parallelized programs directly on the GPU on so-called CUDA cores [GLGN+08, IZG18, HZJM22]. Modern GPUs can have thousands of those cores. In recent years, Tensor cores were added, which are highly optimized for general matrix operations [HZJM22]. CUDA and Tensor cores increased the training speed of AI models immensely [ASP17, BD18, HZJM22].

4.2.1. Popularity of Different Deep Learning Frameworks

Many deep learning frameworks are developed to utilize GPUs' parallelization capabilities. Those frameworks ease the process of building and training AI models by using high-level abstraction. To determine the most relevant frameworks, multiple sources were analyzed. Nguyen *et al.* [NDB+19] found TensorFlow [AAB+15, Dev23b], Keras [C+15], Microsoft CNTK [Mic23], Caffe [JSD+14], Caffe2 [Caf23], Torch [CKF11], PyTorch [PGM+19], MXNet [CLL+15], Theano [TAA+16], and Chainer [TOA+19] being the most popular deep learning frameworks in 2019. As of 2023, Caffe2 was merged into PyTorch [Tea18]. The development of Microsoft CNTK was stopped in August 2022 [ale22]. Torch and Theano are not under active development anymore [Tor23a, MIL17]. The Chainer project's development was stopped as the cooperation behind the framework migrated to PyTorch [Pre19].

"Papers with Code" is a website where statistics about ML papers are collected [Pap23a]. One statistic is the percentage of frameworks used in papers that open sourced their code. Figure 4.2 displays the share of frameworks used relative to the number of papers published in that time over the last five years. PaddlePaddle [Pad23], JAX [BFH+18], and MindSpore [Min23] are frameworks that were not mentioned by Nguyen *et al.* . PaddlePaddle and JAX were rarely used during the entire timespan. MindSpore is a relatively new DL framework that went open source in 2020 and seems to gather interest in the research community at the beginning of 2022. PyTorch steadily increased its share over the years, with over 50% of the papers using this framework in the last two years. TensorFlow was used a lot in 2018. Since then, it has been on a constant downward path and has been used in less than 5% of papers in 2022 and 2023. Keras is not explicitly mentioned. This could be because TensorFlow uses Keras as a high-level API [Ker23]. Thereby, open source code that utilizes Keras is classified as TensorFlow.

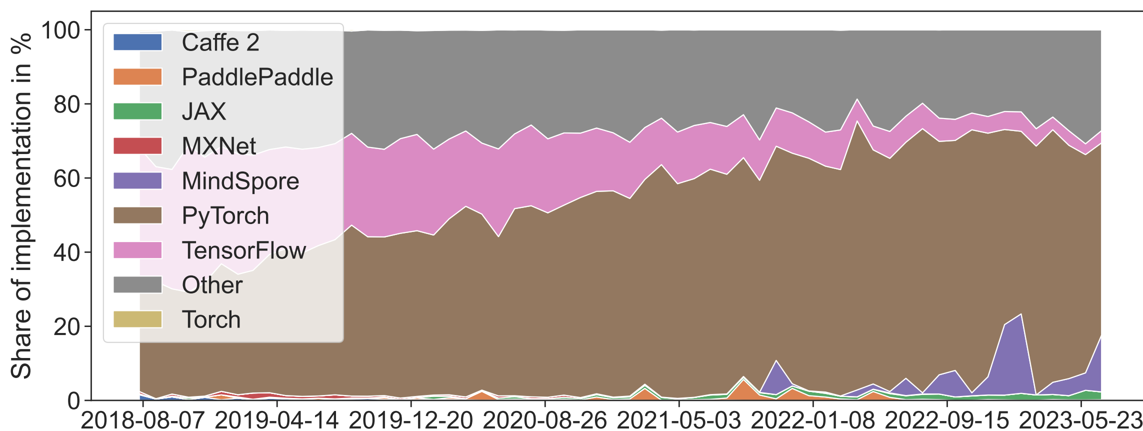


Figure 4.2.: Proportions of frameworks used in paper with published open source code collected by Papers with Code in the last five years [Pap23b].

Analyzing the worldwide Google search trends, displayed in Figure 4.3, shows a similar result. The search popularity of PyTorch increased in recent years while TensorFlow shrunk. Keras is listed here, as this data captures the raw search topics users entered. On the first of January 2022, Google adjusted its data collection system, which resulted in a visible spike in search popularity. The combined search popularity of Keras and TensorFlow is greater than PyTorch's.

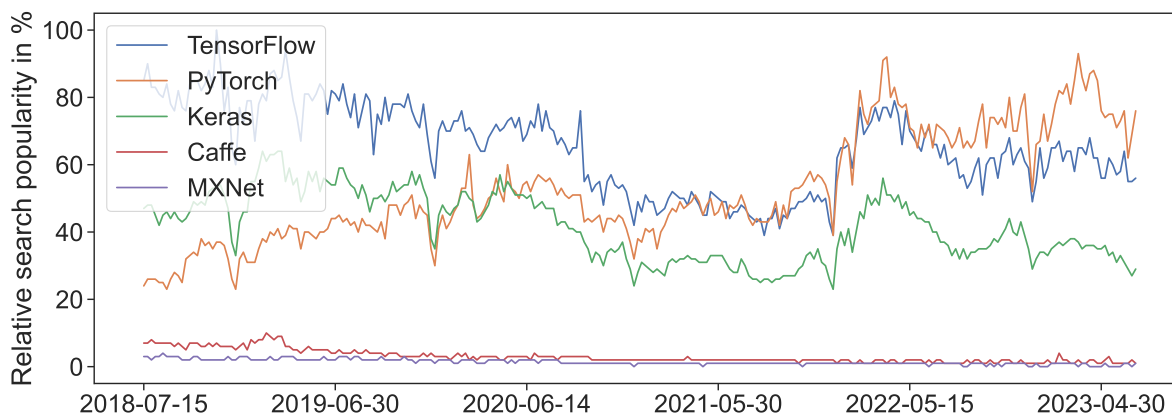


Figure 4.3.: Worldwide Google Search Trend results for TensorFlow, PyTorch, Keras, Caffe, and MXNet from the last five years [Goo23]. The values are relative to the highest value in the time span. On the first of January 2022, Google adjusted its data collection system, which resulted in a visible spike in search popularity.

Finally, it can be said that TensorFlow (Keras) and PyTorch are the most popular DL frameworks in research and industry [DPS+21]. Generally, PyTorch is preferred over TensorFlow in academia [DPS+21]. In the following, the differences between these two frameworks will be presented.

4.2.2. PyTorch vs. TensorFlow

PyTorch was released by Facebook in 2016 and is based on the ideas of Torch [PGM+19]. TensorFlow was released in 2015 and is developed by Google [AAB+15]. Fundamentally, both use a data flow graph to describe computation (computational graph). In the case of TensorFlow, a static computational graph is used. It assumes the input data is always the same, and the graph can be reused. This allows for applying graph optimization, increasing the computation performance [PGM+19, DPS+21]. Paszke *et al.* [PGM+19] say that a static graph reduces the ease of use, debugging, and flexibility of usable computation types. PyTorch follows the principle of dynamic eager execution [PGM+19]. It uses a dynamic graph, as operations are executed immediately as they are called in

the code. This makes debugging a lot easier, as the execution is much closer to the actual code written. In 2017, Google announced integrating eager dynamic execution in TensorFlow [Eag17] and switched completely to eager execution with the release of TensorFlow 2 and the integration of Keras in 2019 [Ten23].

Considering the performance between TensorFlow without eager execution and PyTorch, Dai *et al.* [DPS+21] compared multiple DL models and found that TensorFlow is on average 8% faster on CNN-based networks. Novac *et al.* [NCN+22] used TensorFlow 2 with eager execution and found PyTorch to need 25% less time on their CNN-based methods. Those tests were run on old GPUs (Tesla P100 and GeForce GTX 1070 were released in 2016). Benchmarks on newer GPUs with Tensor Cores are needed to properly evaluate both frameworks' performance.

```python
# PyTorch
import torch
class ClassificationModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.model = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=1, out_channels=32, kernel_size=(5, 5), stride=(1, 1)),
            torch.nn.ReLU(),
            torch.nn.Conv2d(in_channels=32, kernel_size=(5, 5), stride=(1, 1)),
            torch.nn.ReLU(),
            torch.nn.Flatten(),
            torch.nn.Linear(in_features=12800, out_features=64),
            torch.nn.ReLU(),
            torch.nn.Linear(in_features=64, out_features=10),
        )

    def forward(self, x):
        logits = self.model(x)
        return logits

model = ClassificationModel()
```

```python
# TensorFlow
import tensorflow as tf

model = tf.keras.models.Sequential(
    tf.keras.layers.Conv2D(filters=32, activation="relu",kernel_size=(5, 5), stride=(1, 1),
                            input_shape=(28, 28, 1)),
    tf.keras.layers.Conv2D(filters=32, activation="relu", kernel_size=(5, 5), stride=(1, 1)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=64, activation="relu"),
    tf.keras.layers.Dense(units=10, activation="relu"),
)
```

Listing 1: Definition of neural network architecture in PyTorch and in Tensorflow 2.

Both PyTorch and TensorFlow are written in C++. To offer an easier-to-use API for programmers, the functionalities are wrapped in Python. Internally, Python calls the C++ API to keep the performance of the functions. Listing 1 shows the creation of a

NN architecture in PyTorch and TensorFlow 2 (Keras) with the mentioned Python API. A combination of two convolutional layers and two linear layers is used. TensorFlow and PyTorch create layers by creating instances of the corresponding class and adding them in the correct order to a list (Sequential instance). The order in which they are added determines the data flow through the layers. The main difference is that PyTorch uses an extra class to wrap the model and override the `forward` method, which defines how the model should process the data. In TensorFlow, the data processing is defined through the layer structure. Additionally, the name of the activation function is passed as an argument and does not have to be explicitly initialized like in PyTorch.

Comparing the training of both models, TensorFlow automates much of this process. As displayed in Listing 2, TensorFlow has the `compile` and `fit` methods, which prepare the model for training and runs the training. Essentially, it wraps everything that has to be defined in PyTorch explicitly. TensorFlow has built-in metrics calculation, which can be specified in the `compile` method. In PyTorch, those metrics must be implemented by hand or external libraries like *TorchMetrics* [Wel23] have to be used.

```
# PyTorch
training_loader = torch.utils.data.DataLoader(training_set, batch_size=128, shuffle=True)
validation_loader = torch.utils.data.DataLoader(validation_set, batch_size=128, shuffle=False)

loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

for epoch in epochs:
    for data in training_data:
        inputs, labels = data
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_fn(outputs, labels)
        loss.backward()
        optimizer.step()
```

```
# TensorFlow
optimizer = keras.optimizers.Adam(learning_rate=0.001, momentum=0.9)
model.compile(
    loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy']
)
model.fit(
    x_train, y_train, epochs=50, batch_size=128 ,validation_data=(x_val, y_val)
)
```

Listing 2: Training of neural network model in PyTorch and in TensorFlow.

TensorFlow offers an easy-to-use API for creating and training NN models compared to PyTorch. As it does a lot of the work internally, fewer errors can occur when writing the code. On the other hand, this makes debugging harder, as the exceptions occur somewhere in the internal code of TensorFlow. PyTorch code is easier to debug, as the

essential instructions for creating and training the model are in the file. Thereby, exceptions are easier to track down. Section 4.2.1 showed that PyTorch is more often used in research. As "PathoLearn" is primarily used by medical students and professionals, it makes sense to use the framework, which is most used in research, to familiarise them with it.

Lightning To Reduce Boilerplate

As shown, PyTorch uses a lot of boilerplate code to create and train a NN. *Lightning* [FT19] is a framework built on top of PyTorch, which tries to reduce this by wrapping PyTorch with custom classes and methods. Listing 3 displays the training workflow with Lightning of the same PyTorch model as presented in Listing 1 and 2. Everything is wrapped inside a `LightningModule`, where the PyTorch model is initialized in the constructor. Methods like `training_step` and `validation_step` are overridden to define step-specific statements, like calculating the model output, the loss, and metrics. Explicitly calling the backpropagation or optimizer step is not needed anymore, as the overridden methods hook between those steps internally. Another optimization is using a `Trainer` object, which is comparable to TensorFlow's `compile` and `fit` methods. Training-specific parameters like the number of epochs can be passed directly to the trainer, removing the need to program the training loop explicitly. The deep integration of TorchMetrics allows for easy metrics calculation and logging. The accuracy is calculated as displayed in Listing 3 for each training and validation step. Additionally, those metrics can be logged to a defined logger (in this case, a CSV file) for each epoch, step, or both.

Finally, the Lightning framework focuses the code writing on the necessary elements to train and evaluate a NN. This is comparable to TensorFlow, but Lightning keeps the flexibility and control of the model training of PyTorch. This results in readable code, allowing readers to understand the NN model configuration easily and quickly identify the impact of code changes. Wrapping PyTorch with custom functionality poses the risk of reducing training performance. Depending on the configuration, Lightning is tenths to hundreds of milliseconds slower per epoch than PyTorch [Ben23, FT19]. Considering the offered flexibility and integration of, e.g., more straightforward metric calculation, aggregation, and logging, this seems to be an acceptable tradeoff.

```python
import lightning as L
import torch.nn.functional as F
import torchmetrics
from lightning.pytorch.loggers import CSVLogger
from lightning.pytorch import Trainer

# Initialize datasets and ClassificationModel Class

class LightningModel(L.LightningModule):
    def __init__(self):
        super().__init__()
        self.model = ClassificationModel()
        self.train_acc = torchmetrics.Accuracy(task="multiclass", num_classes=10)
        self.valid_acc = torchmetrics.Accuracy(task="multiclass", num_classes=10)

    def _shared_step(self, batch, batch_idx):
        input, label = batch
        logits = self.model(input)
        loss = F.cross_entropy(logits, label)
        return loss, logits, label

    def training_step(self, batch, batch_idx):
        loss, logits, label = self._shared_step(batch, batch_idx)
        self.train_acc(logits, label)
        self.log("train_acc", self.train_acc, on_epoch=True, on_step=False)
        return loss

    def validation_step(self, batch, batch_idx):
        loss, logits, label = self._shared_step(batch, batch_idx)
        self.valid_acc(logits, label)
        self.log("valid_acc", self.valid_acc, on_epoch=True, on_step=False)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=1e-3)
        return optimizer

model = LightningModel()
logger = CSVLogger("logs", name="log")
trainer = L.Trainer(max_epochs=10, logger=logger)
trainer.fit(model, train_loader, validation_loader)
```

Listing 3: Training of neural network model with Lightning [FT19].

4.3. Comparison of Existing End-To-End Artificial Intelligence Platforms

This section compares software platforms covering the entire ML lifecycle (MLOps tools). Moreschi [MRL+23] created a map of different software tools that cover specific lifecycle steps. As displayed in Figure 4.4 the landscape of available software solutions is large. Considering the ML lifecycle, *End-to-End Full-stack MLOps tools*, *OPS*, and *CI/CD*

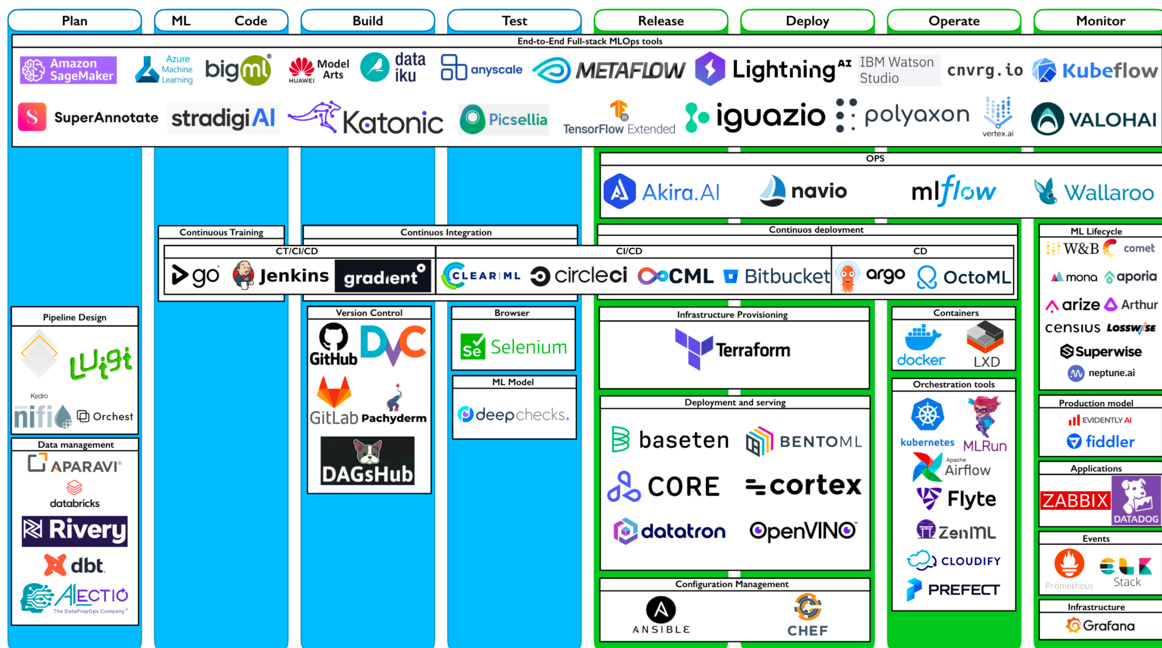


Figure 4.4.: Landscape of software tools grouped into their category that they solve in the machine learning lifecycle [MRL+23].

are the most relevant categories that cover most of the lifecycle steps. Table 4.1 was created by analyzing each listed software tool, additional research papers and reviews [RPC+22, SNKP22, HM22, TBF+22, KKH23], and own research. PathoLearn uses only open source software that can be self-hosted, so the MLOps tool should also fulfill these requirements [Nee21]. This allows the flexibility to change the MLOps software to meet special requirements and host the software on self-owned hardware. Additionally, the previous sections resulted in the selection of PyTorch as the DL framework for training AI models. Except for BigML [Big23] and Picsellia [Com23], every tool can cover the lifecycle of PyTorch models. Self-hosting the tool is often a paid feature. ClearML [Cle19] (CLE), Iterative [Dev23a] (ITE), MLflow [MLf23] (MLF), Polyaxon [Mou18] (POL) and Ray [Pro23a] (RAY) are the only tools that are open source and offer a free self-hosting option. Therefore, only these five tools will be compared in more detail.

Software Tool	Reference	Open Source	Self-hosted	PyTorch
Akira Ai	[MLO23]	✘	✘	✔
Amazon SageMaker	[Mac23b]	✘	✘	✔
BigML	[Big23]	✘	✔(paid)	✘
ClearML	[Cle19]	✔	✔	✔
cnvrg.io	[Ful23]	✘	✔(paid)	✔
Dataiku	[Dat23]	✘	✔	✔
DataRobot	[Mac23a]	✘	✔(paid)	✔
Huawei ModelArts	[Mod23a]	✘	✘	✔
IBM Watson Studeio	[Wat23]	✘	✘	✔
Iterative	[Dev23a]	✔	✔	✔
Katonic MLOps Platform	[Kat23]	✘	✔(paid)	✔
Microsoft Azure Machine Learning	[Azu23]	✘	✘	✔
MLflow	[MLf23]	✔	✔	✔
MLReef	[Col23]	✘	✔(paid)	✔
Navio	[Eas23]	✘	✘	✔
Picsellia	[Com23]	✘	✔(paid)	✔(pre-trained)
Polyaxon	[Mou18]	✔	✔	✔
Ray (Anyscale)	[Pro23a]	✔	✔	✔
Valohai	[Val23]	✘	✔(paid)	✔
Vertex AI	[Ver23]	✘	✘	✔
Wallaroo	[Wal23]	✘	✔(paid)	✔

Table 4.1.: List of different MLOps tools. For each software tool, the reference, whether the code is open source, can be self-hosted, and whether it supports PyTorch is documented.

The general requirements of the MLOps platform in the context of PathoLearn must be defined before comparing the different software tools. Recupito *et al.* [RPC+22] described numerous features MLOps tools should have. These features were grouped into the categories *General Features*, *Data Management Features*, and *Model Management Features*. Based on this, the features required for the extension of PathoLearn were created. Table 4.2 shows the list of requirements, with a description for their purpose. Some requirements were very specific and not listed, as they are irrelevant for the extension. On the other hand, additional requirements were added, which are indicated with a (+).

Multiple users can use the extension at the same time. Therefore, scalability is an essential factor. The MLOps tool should offer horizontal scalability by adding more servers with GPUs to allow the training of multiple models simultaneously. Horizontal scalability is also relevant in data management. Instead of adding more storage to a single server, multiple servers can increase the overall performance. Therefore, this requirement was additionally added. This requires an easy-to-use API or Software Development Kit (SDK) to create and fetch datasets and distribute them between the servers. Another important feature is the metadata management and collection. When creating a dataset,

metadata should be collected about it, e.g., dataset size and number of files. The metadata of a model training should include the software packages required to run the training or store which dataset is used. This can be extended to the model serving. This topic was not explicitly listed by Recupito *et al.* [RPC+22] and therefore added, as model serving is an important feature for the extensions. It should be scalable and offer an easy-to-use API or SDK. This should be combined with metadata collection to collect model performance and runtime parameters. Most of the features are used indirectly

Feature Category	Feature	Description
General	Open Source	The software code is public and available for use, modification, and distribution.
	Horizontal Scalability	It should be possible to scale the software horizontally.
	Self-hosting	Self-hosting the software should be possible without costing money.
	Metadata management/collection	Metadata management is used to collect data during the complete ML pipeline.
	Isolation/loosely coupling	Components can be developed and deployed independently and depend on each other to the least extent practicable.
	UI	The tool should offer its own User Interface or Dashboard. Specifically for managing the entire Machine Learning lifecycle and visualizing metrics.
Data Management	Data storage	Either a built-in database to store raw data, experiments, models, and metadata should exist or support external storage solutions.
	Horizontal Scalability (+)	Besides vertical scalability, the data storage should also support horizontal scalability to improve the data management performance.
	Metadata management/collection	Metadata management is used to collect data and can be used to determine which data is used to train a model.
	API / SDK	An easy-to-use API or SDK must exist to allow data management.
Model Management	Library support	It should support PyTorch / PyTorch Lightning.
	Model tracking	Intermediate ML model performance can be tracked and logged to maintain reproducibility and gain insight.
	Model registry	A centralized repository used to standardize the definition, storage, and access of features for training and serving, which is accessible via an API.
	Metadata management/collection	Metadata management is used to record ML model, the performance, and runtime parameters.
	API / SDK	An easy-to-use API or SDK must exist to create and manage models.
	Model serving (+)	Model serving should be offered scalable and with an easy-to-use API or SDK.

Table 4.2.: The required features the MLOps software tool must have to cover the features needed for PathoLearn extensions. These are mostly based on Recupito *et al.* [RPC+22]. Some features were removed as they are irrelevant to the extension. Those marked with a (+) were added. The features description was updated to fit the context of PathoLearn.

through the user interface (UI) of PathoLearn. Using the API and SDK of the MLOps tool, abstraction layers must be created to visualize the features in the UI. To still offer all features without adding additional abstraction layers, the MLOps tool should have an independent UI. It should allow to monitor and manage most of the features listed.

For every of the five software tools, it was analyzed whether it supports the specific feature defined in Table 4.2. As can be seen in Table 4.3, a ✔ indicates that the tool supports this feature, (✔) is a partial fulfillment, and ✘ means that this feature is not offered or it can not be freely used.

		Iterative	MLflow	Polyaxon	Ray	ClearML
General	Open Source	✔	✔	✔	✔	✔
	Horizontal Scalability	(✔)	✘	✔	✔	✔
	Self-hosting	✔	✔	✔	✔	✔
	Metadata management/collection	✔	✔	✔	✔	✔
	Isolation/loosely coupling	✔	✔	✔	✔	✔
	UI	✘	✔	✔	(✔)	✔
Data Management	Data storage	✔	✔	✔	✔	✔
	Horizontal Scalability	✔	✔	✔	✔	✔
	Metadata management/collection	✔	✔	✔	✔	✔
	API / SDK	(✔)	(✔)	✘	✔	✔
Model Management	Library support	✔	✔	✔	✔	✔
	Model tracking	✔	✔	✔	✔	✔
	Model registry	✔	✔	✘	✔	✔
	Metadata management/collection	✔	✔	✔	✔	✔
	API / SDK	✔	✔	✔	✔	✔
	Model serving	✔	(✔)	✘	✔	✔

Table 4.3.: Comparison of the different MLOps software tools for the predefined features. A ✔ indicates that the tool completely fulfills the feature. (✔) means partial fulfillment, and ✘ indicates no implementation or no free usage of this feature.

Iterative offers three independent open source modules: Data Version Control (DVC) [dvc23], Continuous Machine Learning (CML) [cml23], and MLEM [mle23]. These cover most of the features required. DVC and CML use git-based projects [dvc23, cml23] to enable version control for data and models. CML integrates the training into the CI/CD solutions of GitHub, GitLab, and Bitbucket. Horizontal scalability is only given by creating more runners to execute the CI/CD pipelines [Sel23]. A downside is that all the features are primarily exposed through a command line application. The offered Python SDK exposes only limited functionality. While metrics and plots can be defined and stored in DVC, they can only be visualized through external tools. A complete UI is available through a closed-source, non-free tool called *Studio* [Inc23a].

MLflow is a popular open source software that covers the ML lifecycle. A central tracking server collects and stores relevant information and can be visualized and managed through the UI. To enable parallel training on a server cluster, additional software like *Apache Spark* is needed [Con23a]. The Model registry stores trained models in a custom file format. The serving of a model is realized through a RESTful API. Multiple endpoints are created for monitoring and making predictions for each served model. This makes implementing custom UI elements in PathoLearn easy, as simple HTTP requests can be used to get predictions for input data provided by the user. The serving implementation does not explicitly consider scalability. Forwarding requests to a server cluster is not possible. Therefore, the SDK offers the ability to deploy models to other existing or custom model-serving software tools.

Polyaxon offers comparable functionality to the tools presented before. The UI is very sophisticated as it allows to search and filter metadata with a custom query language [Int12]. While the API and SDK can access many features, datasets can not be created directly and must already exist. Metrics and metadata are tracked and stored and can be directly visualized. Saving and deploying trained models is not possible. The model registry is only offered in the commercial product [Pol12], and model serving requires external tools [Ser23a].

Ray implements every feature required, loosely coupled into multiple libraries. Datasets can be created and managed scalable with *Ray Data* [Ray23b]. *Ray Train* covers everything considering training and has deep integrations with existing DL libraries, including PyTorch and PyTorch Lightning [Ray23d]. Another advantage over the other libraries is that scaling can be done besides Kubernetes with a custom cluster and worker nodes [Lau23]. This allows servers to be added easily afterward. This cluster can be reused for model serving with *Ray Serve* [Ray23c]. This creates a scalable way of training and serving models simultaneously. In contrast to Polyaxon's UI, *Ray Dashboard* exists primarily only for monitoring and debugging running experiments [Ray23a].

ClearML is the only MLOps software tool that fulfills every feature requirement. Especially its UI allows efficient model management. Besides visualizing metrics and metadata, training processes can be cloned, stopped, and restarted. Comparable to MLflow,

ClearML uses a central service called *ClearML Server* [Cle23b], which manages metric and metadata tracking, training scheduling, and data administration. Additionally, it handles server orchestration. Like Ray, ClearML resolves horizontal scalability with worker nodes. A *ClearML Agent* [Cle23a] can be installed on a server, which is automatically registered at the ClearML Server instance. The ClearML server keeps track of the present agents and distributes training jobs to them. The agent runs the training in a virtual environment, installing the needed software packages and downloading the used dataset. Metadata and metrics are reported back to the ClearML Server instance. Trained models can be stored in the model registry and used by the serving module *ClearML Serving* [Ser23b]. ClearML Serving integrates NVIDIA Triton [Ser23b]. NVIDIA Triton is an open source serving software that enables scalable model serving. Figure 4.5 displays the general architecture of the software. It supports multiple DL

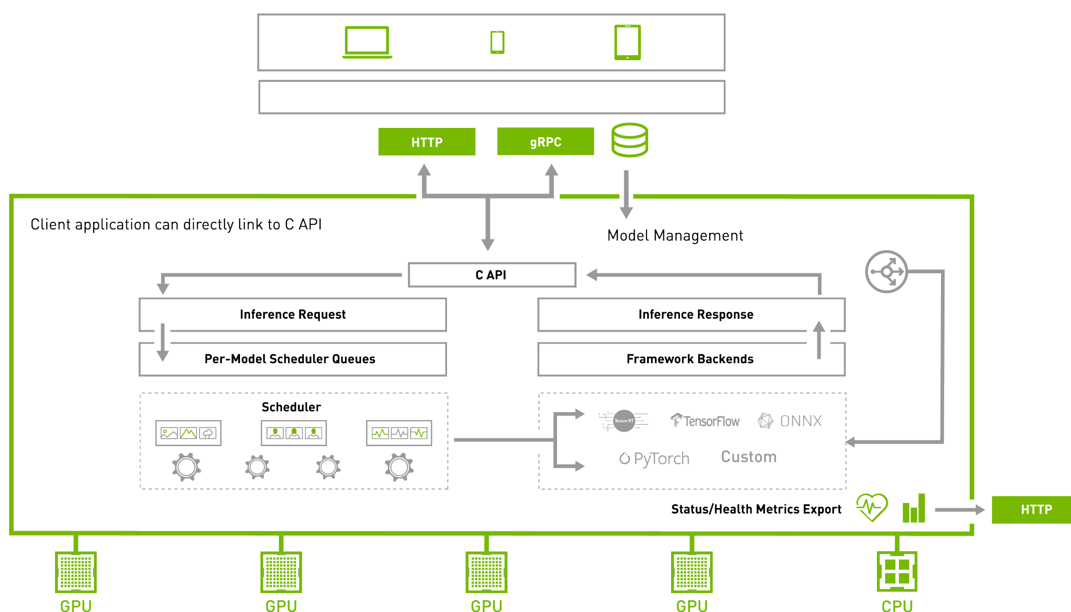


Figure 4.5.: Overview of the NVIDIA Triton architecture [Tri20].

frameworks and utilizes multiple GPUs and CPUs. Additionally, status and metrics are exported and can be accessed. As MLflow, ClearML Serving offers model predictions with REST endpoints. The requests are forwarded to the Triton server, and the prediction results are returned to the user. A downside of ClearML Serving is the missing integration with the existing UI. Everything is managed through the CLI application, and metrics are reported to Prometheus, which can be visualized in a Grafana dashboard [Ser23b].

To conclude the comparison, ClearML and Ray are the most fitting MLOps software tools for the defined requirements. After further analyzing both tools, Ray's API offers more complex functionality and control. While this is generally preferable, this complexity

is not needed for the use-case of PathoLearn. ClearML's API is easier to use. It offers core functions, e.g., starting training or loading datasets, and much is happening under the hood. As stated before, the tool should offer an extensive UI. Ray Dashboard is primarily only for monitoring the connected servers and managing the running jobs. In contrast to Ray, ClearML has a UI integrated that allows managing many of the ML lifecycle steps. Therefore, ClearML is the most fitting choice. The high-level API allows easy integration into PathoLearn, and the UI enables additional management options without the need to access the API. The following section presents the inner workings of ClearML in more detail.

4.3.1. ClearML

The core entities of ClearML are Projects, Tasks, and Models. A Project acts as a container for Tasks and Models to group them and create a logical structure [Pro23c]. Projects can be nested, resembling the hierarchical structure of a file system. A Task is generally everything that can be executed in some way on the server, e.g., a Python script, or storing data (artifacts), e.g., a trained model [Tas23]. Trained models are directly stored in a Model entity. Tasks and Models can be independently accessed through the UI or the SDK.

A feature of ClearML is the automatic analysis of the executed Python script. Listing 4 enables this feature by creating a new Task named *MyTask* inside the Project *MyProject*. The script code is stored in the Task in ClearML. Additionally, the imports are analyzed to define the libraries needed to run the script code. After starting the Python script, the used Python version is detected and stored in the Task. The Task is registered to be executed remotely (`task.execute_remotely(...)`) to facilitate the offered horizontal scalability through the ClearML Agent library. The Task is enqueued on the ClearML Server and fetched by an agent instance for execution. The Agent creates a new virtual environment, where the required packages stored in the Task will be installed, and the script code will be started. This creates a reproducible runtime environment for each Task, independent of the hardware and software installed on the Agent host.

```
from clearml import Task

# Initialize a new ClearML Task
task = Task.init(project_name="MyProject", task_name="MyTask")

# Run the script on a ClearML Agent instance
task.execute_remotely(queue_name="default", clone=False, exit_process=True)

# Remaining script
```

Listing 4: Example code for initializing and marking a ClearML Task for remote execution.

Considering the training process of NN architectures, visualizing and storing metrics is an important step in evaluating the network's performance. The ClearML Logger class [Log23] handles the reporting of metrics calculated in the script code to the Task. Like the automatic analysis of the Python script, it also identifies the metrics calculated in the code. This integrates with the metric logging functionality available in Lightning. Listing 3 displays how to log metrics with the `self.log()` method. As displayed in Figure 4.6, ClearML catches these method calls and stores the calculated metrics in the Task, which are visualized in the UI.

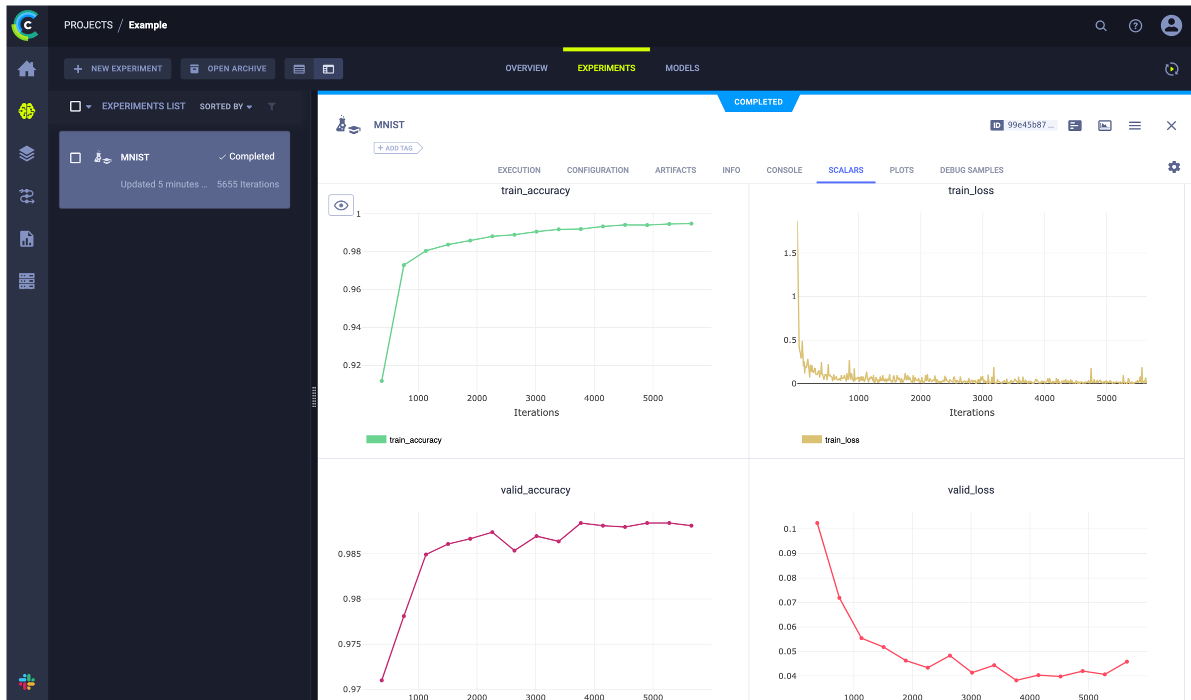


Figure 4.6.: Metrics of a neural network training visualized in the UI of ClearML.

The *ClearML Data* library creates, manages, and versions reusable datasets. A dataset is created in Listing 5. A ClearML Dataset instance is a special form of Task. Therefore, they can be grouped into Projects. Local files can be added to the dataset and uploaded. The internal structure of the uploaded folder will thereby be preserved. To identify that no more changes will be made to this version, the dataset must be marked as *final*.

As displayed in Figure 4.7 the uploaded dataset can be visualized in UI. Each dataset is identified by a unique string, enabling ClearML Agent instances to easily get a local copy for training through the provided SDK.

ClearML supports different scalable storage solutions to enable horizontal scalability [Sto23], e.g., Amazon S3 [Ama23], Google Cloud Storage [Clo23], and MinIO [Inc23b].

Unlike a hierarchical file system, these storage solutions use the "object storage" method. Each element saved is considered an object and does not have to follow a hierarchy but is stored on the same layer. In combination with a unique identifier, stored objects can be accessed easily and distributed between multiple servers, as no hierarchy has to be kept in sync between the servers. As shown in the previous Listing 5, a URI is defined where the dataset's data should be uploaded. In this case, to a storage that supports the Amazon S3 protocol. The ClearML server can also be configured to use the storage solution for everything that should be stored.

```
from clearml import Dataset

dataset = Dataset.create(dataset_name="MNIST", dataset_project="Datasets",
            description="Handwritten digits", dataset_tags=["Classification"], output_uri="s3://...")
# Add local folder to the dataset
dataset.add_files(folder_path)
# Upload the folder to the configured output url
dataset.upload()
# Mark the dataset as final
dataset.finalize()
```

Listing 5: Example code for creating a ClearML Dataset instance.

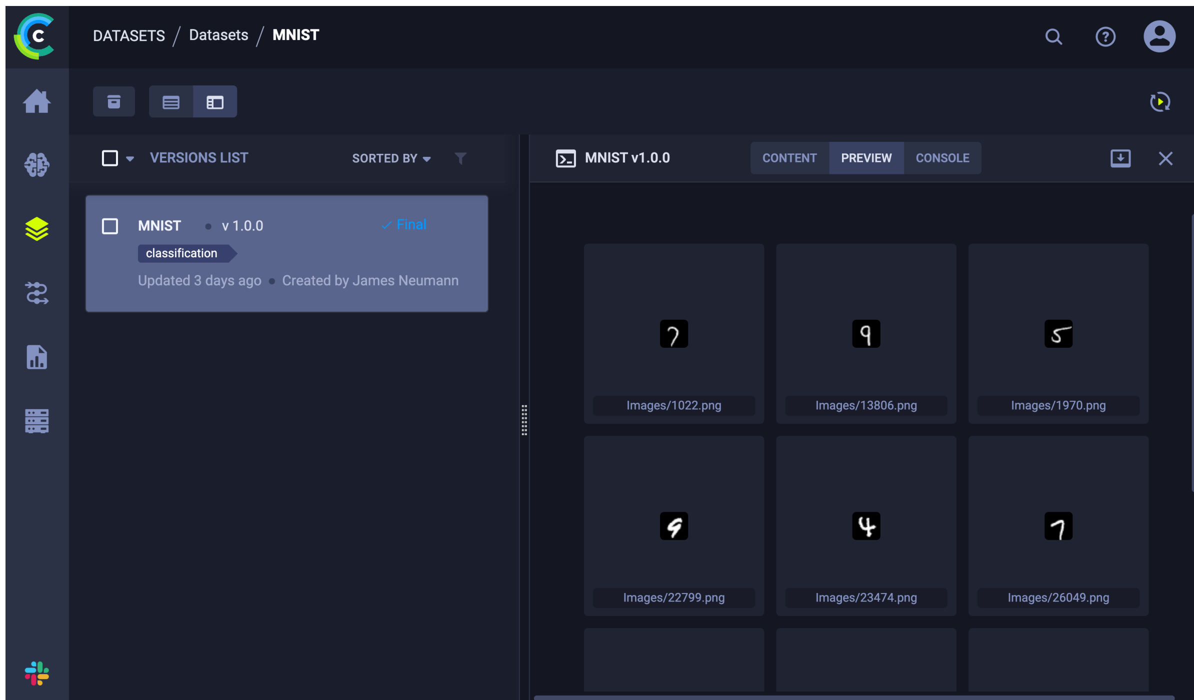


Figure 4.7.: A ClearML Dataset visualized in the UI.

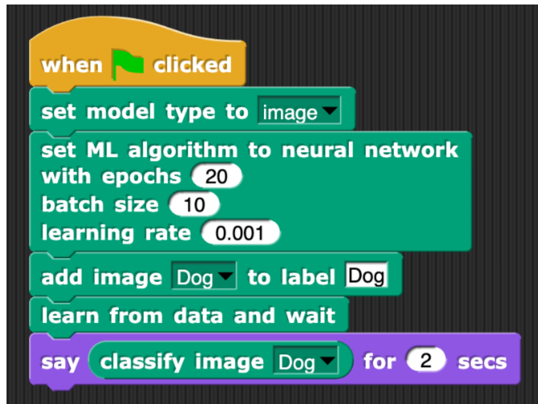
4.4. Software Tools for Teaching Artificial Intelligence

AI/ML also plays an important role in education. As AI becomes more present in every field, teaching students the foundational concepts becomes necessary to prepare them for their later workspace [GHPBB21]. As explained in the previous sections, creating AI models requires using a text-based programming language (e.g., Python). Therefore, students must not only understand the concepts of AI but also know the syntax and concepts of the specific programming language. This creates a significant boundary to teach and learn ML concepts effectively. To mitigate the additional boundary, visual programming languages are often used [GHPBB21]. The textual concepts of the programming languages are converted into visual elements, which can be dragged and dropped onto a canvas. Connections between those elements realize the execution flow. It has been shown that visual programming can support students in understanding the topic and especially focusing them on the specific problem, e.g., understanding how ML works, removing the necessity to understand the programming language used internally [GHPBB21, HCS21].

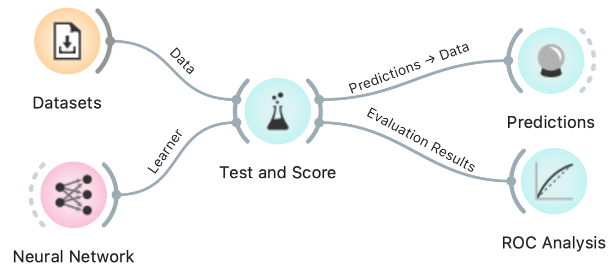
Gresse von Wangenheim *et al.* [GHPBB21] reviewed different visual tools for teaching ML. Figure 4.8 displays some examples. Generally, those tools use a combination of connected nodes to create an AI model. Different types of visual programming paradigms can be defined: *LearningML* [Lea23] uses a block-based approach, where blocks are connected, and the parameters are adjusted directly inside these blocks. *Orange* [Lju23] uses a Data flow-based approach, which offers much more customization, as nodes can be connected to (multiple) other nodes. In workflow-based visual programming, the nodes describe the workflow of training ML. *Google Teachable Machine* [Tea23] uses nodes to create the labels and training data in a dataset. Another node starts the training, and hyperparameters can be adjusted. In the last node, predictions can be made.

The analyzed tools offer different types of abstraction. Some allow configuration of the ML process, while others focus on quick results (see Google Teachable Machine Figure 4.8c). Gresse von Wangenheim *et al.* [GHPBB21] found that most of the tools are integrated into popular block-based languages like *Scratch* [MRR+10] or *Snap!* [Mön23]. While these tools offer easy-to-train ML models, they mostly do not support collaborative teamwork or sharing the result with others [GHPBB21]. Also, integrating these tools into different frameworks, like PyTorch, is often not directly possible.

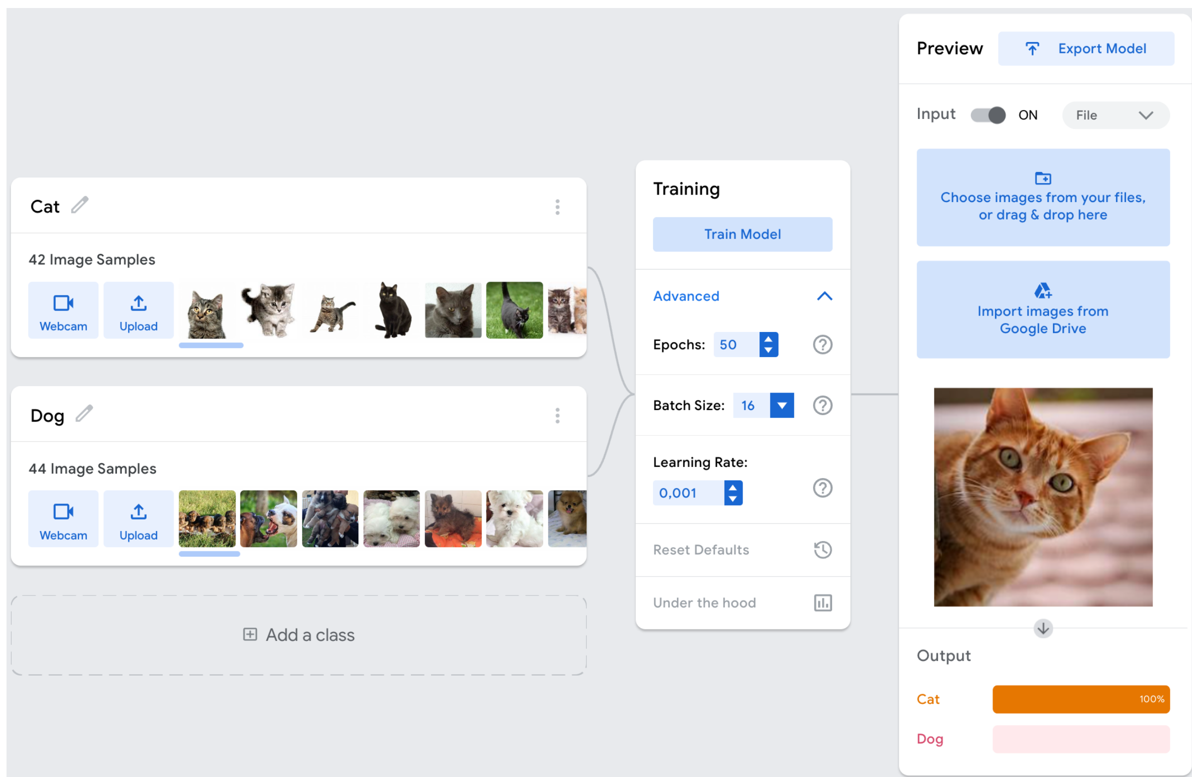
Finally, it can be said that many different tools exist that make teaching and learning ML more accessible. Visual programming is a viable approach to abstract text-based programming into a form where only the knowledge to solve the problem is needed. The visual programming language can thereby offer different levels of complexity. A node can represent a high-level concept of ML, e.g., a classification node. The user only needs to understand that this node represents an ML model that can classify images. What kind of classification model is used is not presented to the user. Alternatively, a node



(a) Block-based (LearningML) [Lea23]



(b) Data flow-based (Orange) [Lju23]



(c) Workflow-based (Google Teachable Machine) [Tea23]

Figure 4.8.: Different visual programming examples for teaching ML found by Gresse von Wangenheim *et al.* [GHPBB21]. LearningML (a) uses a block-based language, where blocks are connected, and adjustments are made directly inside the blocks. Orange (b) lets the user connect different nodes. The nodes describe the data flow from the datasets through training and evaluation. Lastly, Google Teachable Machine (c) uses a workflow-based approach.

can be a single convolutional layer where the user can configure every parameter, e.g., kernel size, stride, and padding. Looking at the different paradigms, a data flow-based or workflow-based visual programming language is closest to the actual behavior of training a NN. The training data passes through every layer until the output layer, where the predictions are evaluated. This allows users to understand the inner workings of neural networks while still being offered a high-level abstraction from text-based programming.

5. Requirements

Besides selecting a programming language and MLOps software tool for creating and training ML models, additional requirements specific to the extension of PathoLearn can be defined. The design and development of PathoLearn was based on a detailed requirements analysis [Nee21]. Initially, the stakeholders and target groups were defined. Then, the concrete requirements were documented in the form of user stories. The following sections present the updated stakeholders, target groups, and the extension's requirements.

5.1. Stakeholders and Target Groups

PathoLearn is used as an additional education tool in the lecture on digital pathology at the Hannover Medical School (MHH). Therefore, the stakeholders are generally the teachers and students of that course. Additionally, the extension can be used by researchers to develop AI. While the students learn about digital pathology, AI is not included in their curriculum. Thereby, as displayed in Table 5.1, it can be deduced that the extension must be easy to use and understandable, besides being complex enough to support researchers in their AI research.

Characteristic	Description
Number of participants	unlimited
Geographical distribution	Germany
Age	at least early adults
Gender	any
Educational Degree	lowest: qualification for university, highest: professor
Prior knowledge	Existence in the field of Pathology. Knowledge about AI is limited or not available.
Learning motivation	Ex- and Intrinsic. Related to AI, more intrinsic for researchers and teachers while more extrinsic for students.
Learning duration	different, depending on prior knowledge
Media competence	common computer and browser skills
Learning places	In the university, at home, in the lecture, etc.
Technical equipment	A computer or laptop with a modern browser (internet)

Table 5.1.: Characteristics of the target group. Per [Ker18].

5.2. User Stories

A user story (US) is an efficient way to document the requirements of the different stakeholders. This will capture the functionalities the extension must offer without specifying technical details. The format of the USs will be the same as used in Neemann [Nee21]:

As <a user role>, I want <to perform this action> so that <I can accomplish this goal>.

Table 5.2 displays the collected USs. Three different user roles can be defined: a general user, student or lecturer, or researcher. The general user captures all USs that all user roles require. Student or lecture capture USs specific for education and the researcher user role defines USs for research specific requirements.

As a User	
Dataset Management	
US-1	...I want to upload my own datasets so that I can use them for training.
US-2	...I want to create datasets from existing tasks inside PathoLearn so that I can use them for training.
US-3	...I want to delete datasets so that they can not be used for training anymore.
US-4	...I want to see metadata about the dataset so that I can identify what kind of dataset it is.
US-5	...I want to see example images of the dataset so that I can get an idea about the dataset's data.
Project Management	
US-6	...I want to create projects so that I can group experiments.
US-7	...I want to update projects so that I can update misspellings in their name or description.
US-8	...I want to delete projects so that I can neither access the project nor the containing experiments anymore.
Experiment Management	
US-9	...I want to create experiments so that I can organize my different artificial intelligence architectures.
US-10	...I want to delete experiments so that I can remove my artificial intelligence architectures and training results.
US-11	...I want to update experiments so that I can fix misspellings in their name and description.
Artificial Intelligence Management	
US-12	...I want to create artificial intelligence architectures so that I can train a model on my dataset.
US-13	...I want to create artificial intelligence architectures with other users simultaneously so that I can work in a group.
US-14	...I want to update my artificial intelligence architectures so that I can adjust specific elements.
US-15	...I want to delete my artificial intelligence architectures so that neither I nor other users can access them.

	US-16	...I want to get information about the current training status so that I know if my model is working properly.
	US-17	...I want to get information about the generated metrics so that I can identify if my model is performing as desired.
	US-18	...I want to perform predictions on my trained model so that I can see how good my model's predictions are.
As a student or lecturer		
Artificial Intelligence Management		
	US-19	...I want to use predefined artificial intelligence architectures so that I only have to provide a dataset.
As a researcher		
Artificial Intelligence Management		
	US-20	...I want to create artificial intelligence architectures with a lot of customizability so that I can adjust them to my research requirements.

Table 5.2.: Collected user stories grouped by user role and epic.

5.3. Functional Requirements

From USs, functional requirements can be extracted. The epics *Dataset Management*, *Project Management*, and *Experiment Management* are primarily for organizing the AI models that the users create. Therefore, basic create, read, update, and delete (CRUD) operations must be offered. More complex requirements result from the *Artificial Intelligence Management* USs.

As shown in Section 4.4, data flow-based visual programming is a good way to abstract from text-based programming in Python. This creates a fitting abstraction, as the target group has no or limited experience in programming (see Section 5.1). The results from Section 4.2 and Section 4.3 create additional requirements. As displayed in Figure 5.1, a visual programming editor must exist in PathoLearn (US-12, US-14, US-15, US-19, US-20). A PyTorch script is generated from the connected nodes and transferred to ClearML for training. While training the model, the status and calculated metrics should be continuously returned to PathoLearn for visualization (US-16, US-17, US-18).

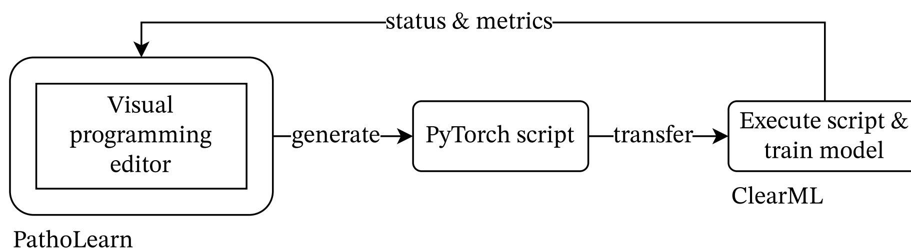


Figure 5.1.: General flow between the visual programming editor and the training.

Considering US-13, the visual programming editor needs user collaboration integrated. The result of Section 4.4 showed that this feature is not included in existing visual programming editors created for teaching ML. Figure 5.2 shows two schematic nodes which could exist in a visual programming editor. Users connected to the same editor instance in PathoLearn should always see the same state. Therefore, all interactable elements, like the configuration elements of the node, must be synchronized between every user. Creating, deleting, or moving a node should also be visible to every user simultaneously. Finally, if a user deletes or creates a node connection, every other user should receive a state update.

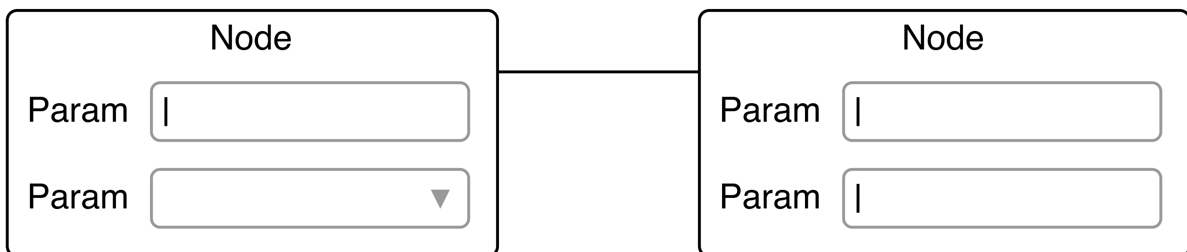


Figure 5.2.: Two connected schematic nodes of a visual programming editor. The nodes can contain different control elements, e.g., input fields or dropdowns, which allow configuration of the specific nodes.

5.4. Non-Functional Requirements

Neemann [Nee21] also defined non-functional requirements that PathoLearn should fulfill. As displayed in Table 5.3, these are also valid for the extension with additional requirements for the collaboration feature.

NF-4 defines an average response time of 150 ms. This is based on the time specified in the original non-functional requirements of PathoLearn [Nee21]. Miller [Mil68] stated that if the response time of a system is up to 100 ms, the user thinks that his actions result in an instantaneous reaction of the system. In the case of US-13, this can be extended. Every other user that is connected to the visual programming editor should also receive the action in up to 100 ms. This ensures that changes made to the nodes and connections in the editor are synchronized between every user, allowing collaborative working. Users generally have no knowledge about AI (see Table 5.1), so a focus should be on NF-7 and NF-8. The visual programming editor must be easy to use and the nodes available easy to understand.

Security

	NF-1	Users that are not registered have no access to the extension of PathoLearn.
Availability		
	NF-2	If software components of the extension fail, they should be restarted in a few minutes.
	NF-3	Software components should be horizontally scalable. Especially those components responsible for training
Performance		
	NF-4	The average response time should not be over 150 ms.
	NF-5	The average response of collaboration features should be less than 100 ms.
	NF-6	The extension should be reactive at every time.
Usability		
	NF-7	The extension should be easy to use.
	NF-8	The extension features should be easily understandable and fast to learn by users.

Table 5.3.: Non-functional requirements of the PathoLearn extension.

6. Implementation

This chapter covers the implementation of Patholearns' extension. First, the overall software architecture is presented, followed by explaining how authentication between the various software components is realized. The last part explains how NNs can be created, trained and served inside a collaborative visual programming editor.

6.1. General Software Architecture

The software architecture of PathoLearn can be divided into multiple (micro)services. As illustrated in Figure 6.1, six different services and a web frontend exist. Those with a light gray background were implemented to realize PathoLearns' extension. Both the *Learn* and *Slide* service, combined with most of the web frontend, was implemented in the bachelor thesis [Nee21]. The Learn service is responsible for all features related to teaching and learning digital pathology. The WSI that are uploaded are processed and stored in the Slide service. Both services offer a RESTful API, which the web frontend can use. Both backend services were implemented in Python and utilize the framework *FastAPI* [Fas23] for creating the APIs.

The remaining services were created based on the requirements defined in Chapter 5 and the results from Section 4.3. The *ClearML service* consists of the different introduced sub-services ClearML Server, ClearML Serving, and ClearML Agent (see Section 4.3.1). The *AI service* is used as a middleman between the ClearML REST-API and the web frontend. It stores and manages everything related to the AI lifecycle. Specifically, it is the backbone of the visual programming editor (see Section 6.3.1), offering different REST endpoints based on FastAPI. Additionally, the *Socket service* enables real-time collaboration with multiple users, which is explained in more detail in Section 6.3.3.

The numerous different services require proper authentication. The *Auth service* is a central authentication point for every service. It handles the login and register of users and verifies user sessions sent from the web frontend to one of the introduced services.

Lastly, the original implementation of PathoLearn utilizes Docker. Docker allows packaging software programs into containers containing every software requirement, including operating system libraries to run the program. Unlike virtual machines, the docker

containers use the server kernel through the docker engine [Nee21]. Therefore, docker containers are lightweight, as the hardware is not virtualized. This makes it possible to deploy the containers on any system that supports the docker engine. Additionally, tools like *Docker Swarm* [Swa23] and *Kubernetes* [Pro23b] allow easy scalability of containers on multiple servers. Thereby, the new services are also packaged into docker containers.

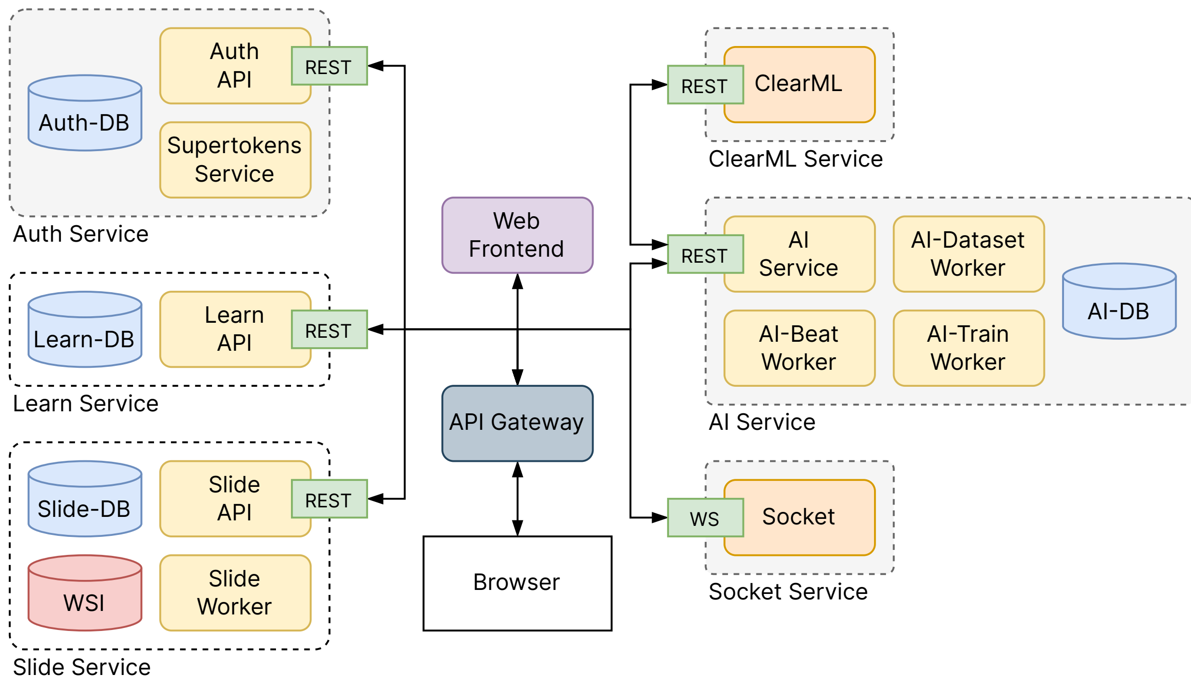


Figure 6.1.: Microservice architecture of PathoLearn. The services with light gray backgrounds were added during the development of Patholearns' extension. The ClearML service consists of multiple different ClearML services. For easier readability, those were combined into one.

6.2. Centralized Authentication

As shown in the previous section, PathoLearn was extended with additional services. In the original implementation of PathoLearn, the authentication was implemented in the Learn service [Nee21]. It secured the application using JSON Web Tokens (JWT) [JBS15]. A User would receive a JWT token after a successful login, which it sends along with every HTTP request he made to the REST API. The service would then validate the JWT and check the authorization. While this is a simple way to enable user authentication, it is integrated into the specific REST endpoints of the Learn service. Every other service would have to make authentication requests through this API,

creating an additional load on the service. Each service could integrate the same authentication with a shared database. While this would simplify the implementation, it would decrease the software's maintainability. Therefore, following the concept of microservice architectures, that a service should only realize a specific business function or requirement [AAE16], the Auth service was added for centralized authentication.

There are many open source authentication systems available (e.g., Ory [Ory23] or KeyCloak [Key23]). These systems are also called *identity and access management* (IAM) tools. OAuth 2.0, OpenID, and SAML are common IAM protocols. With these protocols, the user does not have to log in to every service it uses. A single authentication at the IAM system creates a session that can be reused for every application connected with the IAM. External Accounts, e.g., a Google account, can also be used. These features often result in complex setup and configuration of the IAM tool. Users of PathoLearn should create accounts with the offered registration and not use any external accounts. Therefore, only a simple IAM tool is needed.

SuperTokens [Sup23] is another open source authentication tool. Compared to the other existing ones, it is simpler to set up and easier to configure. It comprises three components: a frontend SDK, a backend SDK and *SuperTokens Core* (STC). The frontend SDK offers functions for the authentication workflow, e.g., login, register, and logout. The backend SDK integrates into FastAPI, automatically creating endpoints for the authentication workflow steps. The backend SDK communicates with the STC service, which checks if the credentials provided by the user are correct and creates or revokes sessions.

Figure 6.2 illustrates the steps involved in the sign-in process. The user enters his e- mail and password. The frontend sends a POST request with the credentials to the specific endpoint (`/auth/signin`) of the Auth API. It validates the input, and the backend SDK sends the request to the STC instance. It checks if the provided credentials are valid and returns the corresponding user ID stored in the user database. If the user exists, the SDK initializes a new session and requests an access and refresh token from STC. The Auth API returns the tokens, and the frontend sets these tokens as cookies. Due to security reasons, the access token has only a limited lifetime. If the access token validity expires, the refresh token is used to request a new pair of access and refresh tokens without the user needing to sign in again.

This protects unauthorized users from accessing PathoLearn through the front end. To further improve security, the REST endpoints of the different services must also be protected. FastAPI allows adding functions to endpoints (middleware) [Mid23]. These are executed before the actual endpoint function is called. Listing 6 displays how the `verify_session` middleware can be used to protect FastAPI endpoints. The middleware checks whether the cookies sent with the request contain a valid access token by accessing the Auth service. If it is invalid, a status code 401 is returned, and the endpoint function

is not executed. Otherwise, the endpoint function is run and can access a session object containing session information (e.g., user ID and tokens).

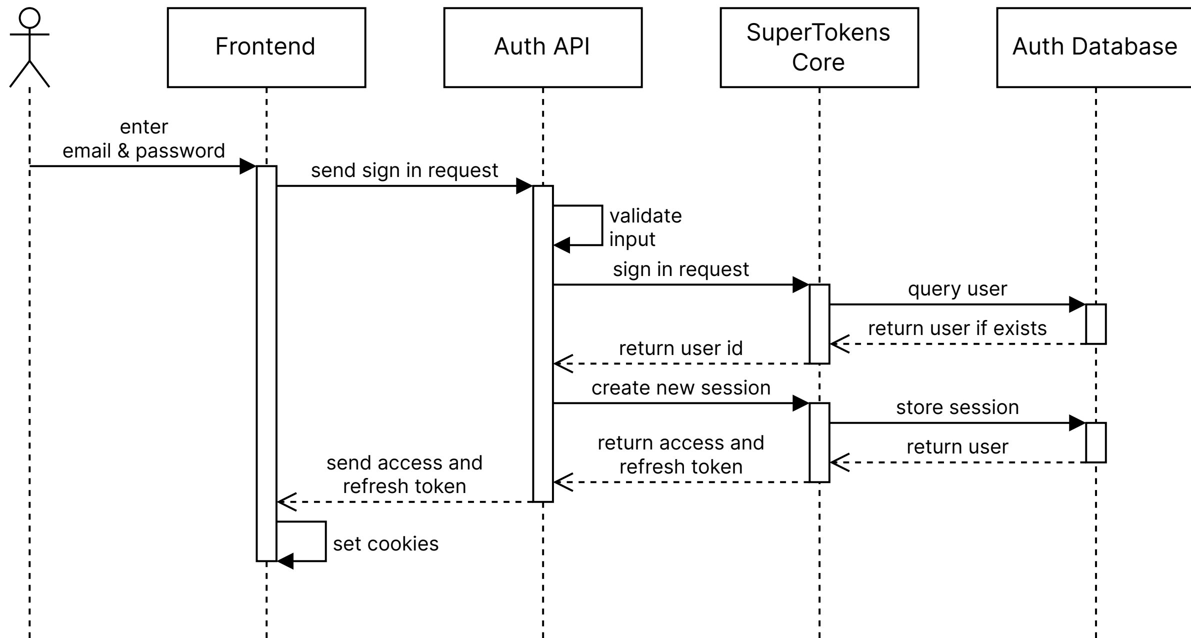


Figure 6.2.: Sequence diagram of the sign in process of a user.

The frontend SDK and the middleware used by every service enable the required centralized authentication. A downside of this approach is that service-specific user information needs to be stored in the respective service. The information must be connected to the user instance of the authentication service. This can be realized through the available session information, as presented in Listing 6, extracting the user ID and storing it with the additional information in its own database.

```python
from supertokens_python.recipe.session.framework.fastapi import verify_session
from supertokens_python.recipe.session import SessionContainer
from fastapi import Depends

@app.post('/train')
async def train_model(session: SessionContainer = Depends(verify_session())):
    user_id = session.get_user_id()
    # Remaining code
```

Listing 6: A FastAPI endpoint protected by the `verify_session` middleware, which enforces valid SuperTokens cookies to be sent with the request.

6.3. Creating Neural Network Architectures

After users successfully sign in, they can create NN architectures with a visual programming editor. As required by The USs regarding the *Project* and *Experiment Management*, users can create, update and delete projects. Inside a project, they can create, update and delete experiments (see Section A.2). The projects and experiments are not user-specific and can be viewed and modified by every other user. An experiment represents a neural network architecture. How users can create these architectures inside a visual programming editor collaboratively will be explained in the following sections.

6.3.1. Visual Programming Editor

As presented in Section 4.4, a visual programming editor is a fitting way to remove the boundary of knowing the syntax of text-based programming languages. *Rete.js* [Ret23] is a JavaScript framework for creating node-based editors. Custom nodes can be created to fit the requirements of the visual programming language. As displayed in Figure 6.3 different nodes are used to visually program a chatbot that returns a message depending on the user's input. Each node can have multiple *sockets*, which are used to create connections between nodes. Only sockets of the same type can be connected (e.g., only an action with an action or text with text).

Figure 6.3.: A very simple message bot created with Rete.js [Cha23]. Custom node and connection types can be created, allowing for conditional data processing and execution.

Considering US-20 and the results from Chapter 2 and Chapter 3, multiple nodes can be defined that must exist to create artificial intelligence models. Figure 6.4 displays those nodes, which are available in the visual programming editor. With the dataset node, users can select an existing dataset they want to use to train the NN model. For feature extraction, convolutional nodes (*Conv2D*), linear nodes (*Linear*), and pooling nodes (*Pooling*) can be added. Between those nodes, it is possible to connect batch normalization or dropout nodes. The *Flatten* node is specifically for transforming two-dimensional data to one-dimensional data so that linear nodes can process the data. Section 3.1 introduced the residual block. The used skip connections require branching between the connected nodes and a way to merge the branches again. This is realized through the *Add* node. It accepts two incoming connections from nodes, whose data is then added. In the case of the U-Net architecture, the branches are not added but con-

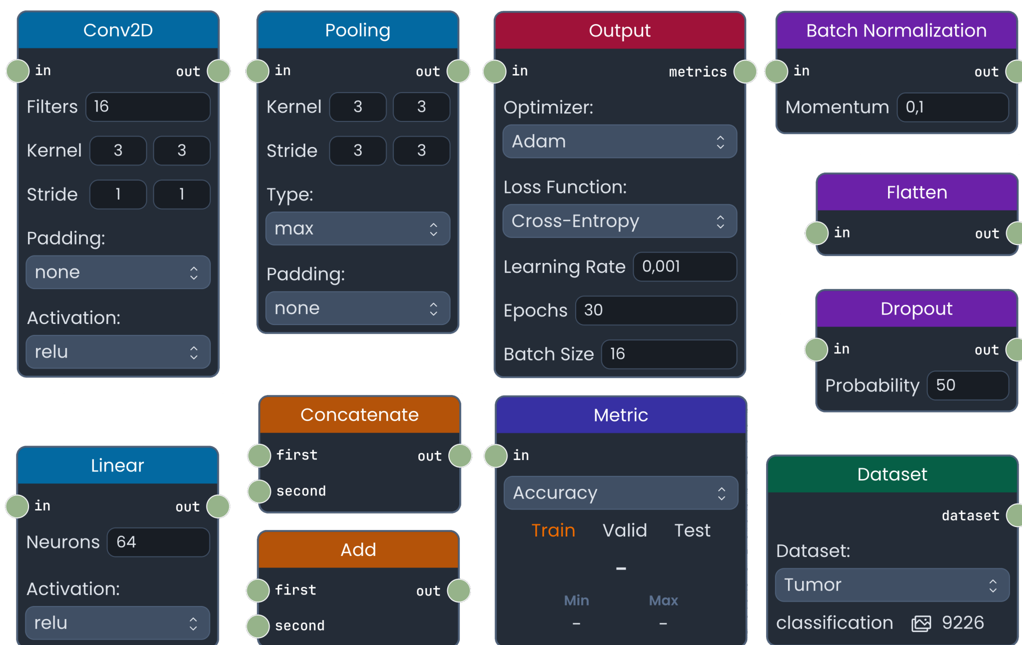


Figure 6.4.: The different nodes implemented in the visual programming editor.

catenated (see Section 3.3). This can be done with the *Concatenate* node. Additionally, with a combination of those nodes, the inception module (see Section 3.1) can be realized (see Figure A.8). The *Output* node manages the various training parameters, like the loss and optimizer function that should be used. Lastly, the *Metric* node supports various metrics calculated for the train, validation, and test dataset.

Each node contains multiple so-called *control* elements. These are comparable to the parameter adjustment elements in block-based languages (see Section 4.4). Each control element changes node-specific parameters to allow high customizability in creating the NN architecture as required by US-20.

A user test (see Section 7) of the visual programming editor showed that it was difficult to understand each node's functionality and how they should be connected. To overcome this, a detailed explanation text was added for each node. These explanations are based on Chapter 2 and Chapter 3. Figure 6.5 illustrates the explanation of the Linear node. The content is stored in Markdown documents. This allows easy formatting and including math formulas through the library *MathJax* [Con23b]. First, a general introduction to the Node's inner workings and general application cases is given. Afterward, the different Node parameters are explained, and tips on which parameter values are generally a good choice. Different webpages are linked in the *additional information* paragraph if users want to deepen their knowledge of the Node. A link to the respective PyTorch documentation page is included if the Node represents a PyTorch layer (see Section 6.4.2).

Figure 6.5.: The detailed information displayed for the Linear node.

Figure 6.6 displays an example NN architecture created with the visual programming editor and the nodes presented before. Offering such a variety of nodes also presents many cases of possible errors. Therefore, different validation procedures are in place.

The main one is the validation of connections. As explained in the introduction, Rete.js allows for typed sockets. Each node defines the types of input and output sockets it supports. For example, the Conv2D-node processes and outputs only two-dimensional data. Therefore, if a user tries to connect this node to a linear node, which only accepts and outputs linear data, the user will receive a warning message that this connection can not be created and a Flatten node has to be added in between.

Therefore, if a user tries to connect this node to a Linear node, which only supports linear data as input and output, it will get a warning message that this is impossible and a Flatten node has to be added in between. Besides validating connections, nodes are also checked. The editor should only contain a single NN architecture. Therefore, users are prevented from adding multiple Dataset- and Output-nodes, as these are the start and end nodes of every architecture. Additionally, a path must exist from the Dataset node to the Output node while passing through at least one node that extracts features. Another likely error is missing the final Linear node that maps the features to the selected dataset's defined classes. Thereby, this layer is automatically added if missing.

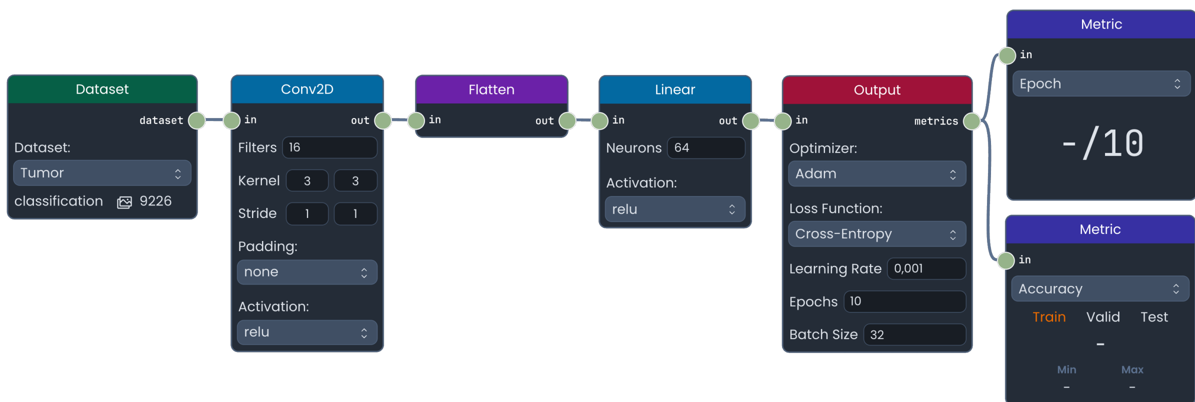


Figure 6.6.: A neural network architecture created with the visual programming editor in PathoLearn.

6.3.2. Predefined Neural Network Architectures

Besides the nodes that allow highly customized architectures, predefined architecture nodes are available as defined in US-19. For classification tasks, the in Section 3.1 mentioned state-of-the-art architectures are available (see Figure 6.7). Multiple versions are available for ResNet and VGG (e.g., resnet18, resnet11, vgg11, vgg16) that the user can select. Additionally, it can be selected whether a pre-trained model should be used (*General*) or not (*No*) (see Section 3.4). Only linear layers can be connected to

the output of the node (*fc*) to allow learning of task-specific features, as explained in section 3.4. The classification architectures are directly available in PyTorch and can be

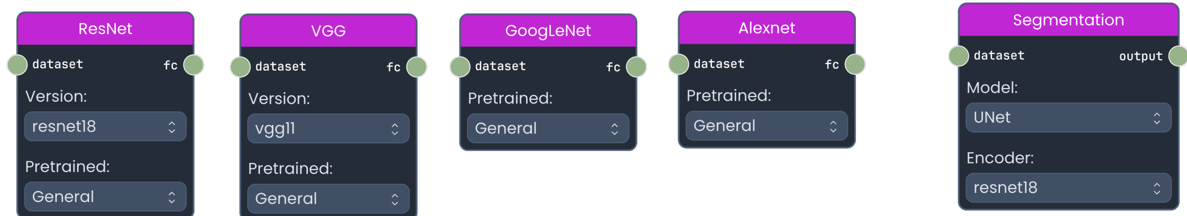


Figure 6.7.: The predefined architecture nodes available in PathoLearn.

dynamically loaded based on their name (see Listing 7).

```
model = torchvision.models.get_model("resnet18", weights="DEFAULT")
```

Listing 7: Loading the ResNet-18 architecture with pre-trained weights in PyTorch.

Object detection and segmentation tasks are handled differently. The use of specialized layers, e.g., deconvolution and ROI pooling (see Section 3.2 and Section 3.3), makes it difficult to create complete architectures with individual nodes in the visual programming editor. Therefore, the *Segmentation* node covers different segmentation architectures. As segmentation is a specialized object detection task, only segmentation tasks can be realized in the visual programming editor due to how segmentation datasets are created (see Section 6.4.1). Section 3.3 presented that these architectures utilize backbones to learn features on the image. Therefore, the Segmentation node allows the selection of an encoder model for feature detection. Internally, the library *Segmentation Models* is used [Iak23]. It is comparable to the `torchvision` library (see Listing 8) and offers different architecture specifically designed for segmentation tasks.

```
import segmentation_models_pytorch as smp
model = smp.create_model(arch="unet", encoder_name="resnet18", in_channels=3, classes=2)
```

Listing 8: Loading the U-Net architecture with the ResNet-18 architecture as feature encoder in PyTorch.

To further ease the process of creating a NN in the visual programming editor, the user can generate a complete architecture by only choosing a dataset and the desired complexity of the network (see Figure 6.8). The Dataset node, the Architecture node, the Output node, and the Metric nodes are added to the editor based on the user's selection (see Figure 6.9).

With this setup, users can create arbitrary complex architectures, either with multiple specialized nodes or predefined architectures. The visual editor allows users to create, move, connect, or delete nodes. To make these features accessible for collaborative working, as required by US-13, additional features need to be implemented.

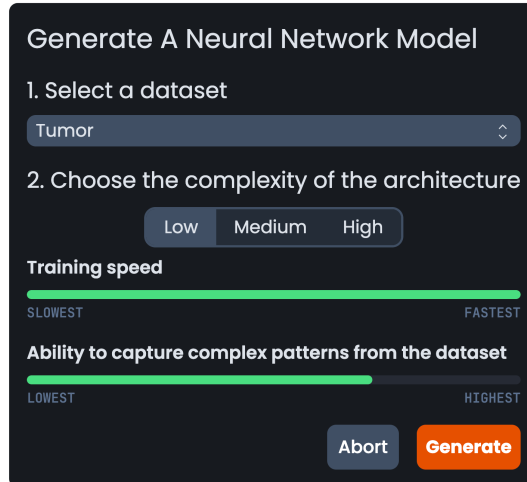


Figure 6.8.: Interface for generating a Neural Network model based on the selected dataset and chosen properties.

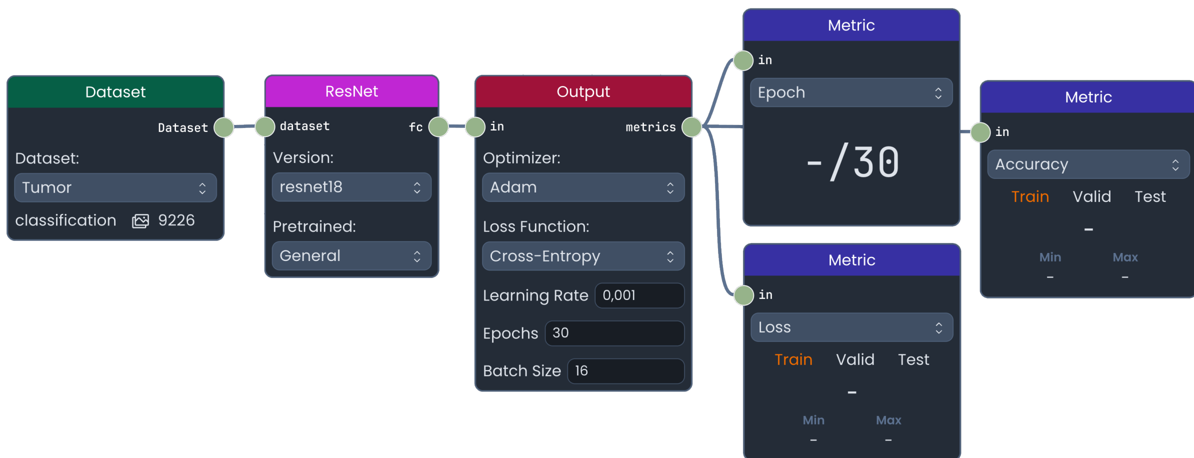


Figure 6.9.: A neural network architecture generated with the interface displayed in Figure 6.8.

6.3.3. Collaboration

Collaborative working in groups requires synchronizing the visual programming editor between every user in the group. Each user should receive the changes made by another user in real-time without manually requiring to refresh the editor.

HTTP Polling, HTTP Long Polling, Server Send Events (SSE), and Websockets are methods to enable real-time communications. HTTP Polling requires the users' client to repeatedly request the server for updates. This can waste the server's resources, as the request is also executed if no updates are available. This also does not enable real-time updates, as the update frequency depends on the defined polling rate and the latency of the communication between the server and the client. Instead, it would be preferable that the server can push an update directly to clients. This is impossible as the web browser initiates every communication with the server [SOM11]. HTTP Long Polling tries to overcome this. Instead of the server directly returning the response to the request, it holds it. If an update occurs or after a long time with no updates, the server sends the response and closes the request [SOM11]. While this reduces the number of requests made to the server, it must keep the connections for every connected client open. Each connection allocates resources for the duration of the polling request. SSE improves this further by allowing servers not to close the request after the first update but to keep the connection open until it is manually closed. If the client has established a connection, the server can send indefinitely many events to which the client listens [Usi23].

Polling methods and SSE only allow uni-directional communication from the server to the client. This limits the possibilities in multi-user collaboration applications, as user changes cannot be published with the same connection. Instead, an additional request is necessary.

The WebSocket protocol solves this, as it enables bi-directional communication [MF11]. It is built on TCP and is designed to work over HTTP ports 80 and 443. A handshake comparable to the TLS handshake is used to establish a connection. The client makes an HTTP request with the HTTP upgrade header set to inform the server that it would like to change (upgrade) the connection from the HTTP protocol to the WebSocket protocol [MF11]. The server acknowledges, and every further communication over the TCP connection is switched to the binary bi-directional WebSocket protocol.

The native WebSocket implementation in the browser allows simple sending and receiving of data between server and client. Many different open source WebSocket libraries exist that build on top of the WebSocket protocol and implement additional features, like automatic reconnecting if the connection is aborted, broadcasting messages to all users, and often support concepts of *Rooms* or *Channels*, which allow sending messages only to a subset of users connected to the WebSocket. This is especially useful in the

case of US-13, where users should work in groups. Therefore, the update message should only be sent to users connected to the same visual editor instance.

Socket.IO is a popular JavaScript library offering many quality-of-life features like auto reconnecting and Rooms [Soc23]. One of Socket.IO's main selling features is an HTTP Long Polling fallback. If the WebSocket connection establishment fails due to browser incompatibility, it uses HTTP Long Polling instead [Soc23]. Today, this is mostly not needed anymore, as every browser nowadays natively supports the WebSocket protocol [Web23]. An alternative implementation is *µWebSockets.js* [UNe23]. It uses a C++ implementation under the hood, which increases the message throughput and reduces the overall server resources needed compared to Socket.IO [Sim23]. On the other hand, it does not offer as many additional features as Socket.IO. This is resolved by *Soketi*. Soketi is an open source WebSocket server built on top of µWebSockets.js [Sok23]. It implements the *Pusher Channels Protocol* (PCP) [Pus23b]. Pusher is comparable to Socket.IO, as it offers APIs that abstract from the native WebSocket API and implements additional features [Pus23a]. Like Socket.IO, it uses its own protocol (PCP) to realize these features. While the protocol and client SDKs for different programming languages are open source, the server is not. Using Soketi as the server-side protocol implementation combined with the official Pusher client SDKs creates a performant and feature-rich WebSocket implementation. The Socket Service displayed in Figure 6.1 therefore uses a Soketi server instance running in its own Docker container, and the other services wanting to utilize WebSockets use the respective client SDK.

User Connection

As mentioned, the group concept must be implemented in the WebSocket integration. The PCP offers five different channel types: public channels, private channels, private encrypted channels, presence channels, and cache channels. Every user who knows the channel's name can subscribe to public channels. Private channels mitigate this, as channel permission must be authorized, with optional message encryption enabled in private encrypted channels. Presence channels extend private channels by adding the feature to store information about who is subscribed to this channel. A cache channel saves and delivers the last sent message to newly connected users.

The information about who is connected to the current visual programming editor is essential for group collaboration. Users should always know who is connected and who is possibly missing. Therefore, users connect to the same presence channel. Figure 6.10 visualizes the process of connecting to the presence channel. The WebSocket connection is initialized after the user connects to the visual programming editor. Since the client authentication is configured, the Pusher client makes an HTTP-POST request to the Auth API. The endpoint validates the user session (see Section 6.2) and extracts user information (e.g., name). The WebSocket client of the API generates a new token for

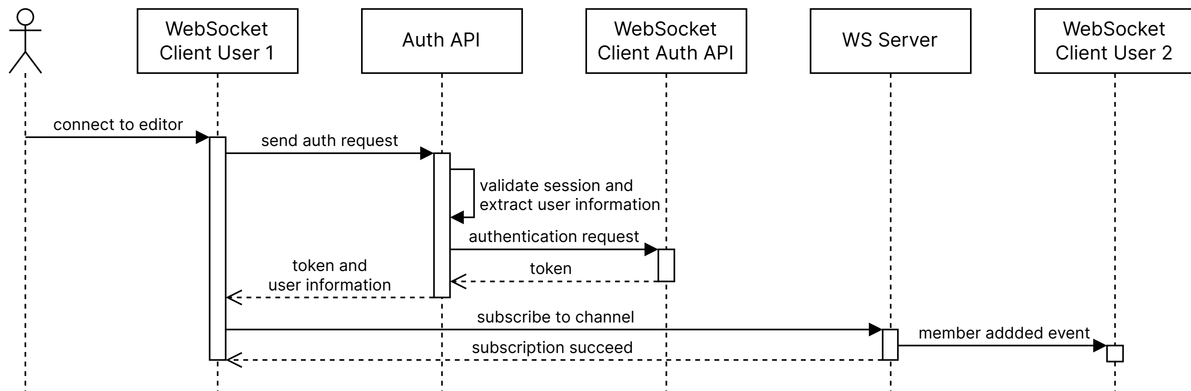


Figure 6.10.: Sequence diagram of the WebSocket workflow if a new user connects to the visual editor.

the user. Additionally, the API appends a random color to the user information. The token with the information is returned to the user's WebSocket Client, requesting a subscription to the channel at the WebSocket server with the token and data afterward. The server registers the users and sends an event to every other group member that a new user joined the channel. This event contains the user information so that each user can display the name and color of the new member.

This workflow ensures that only authorized users can join the channel. The color assigned to each user facilitates easy identification of connected users, particularly when multiple users share the same name.

Visual Programming Editor Syncronization

Users can connect to the visual programming editor and see which other users are also connected. As Section 6.3.1 explained, users can add and manipulate nodes and connections. Therefore, these changes must be published to every other user. This can be done with Pushers client events. Listing 9 shows how a client can subscribe and trigger events. Specifically, the `client-mouse-moved` event synchronizes the mouse cursors between users. If a user moves their mouse, the event is triggered with the updated mouse position and the client ID as payload data. Every other client receives the event and updates the virtual cursor position of the user with the specific ID.

To synchronize editor elements, the plugin system of Rete.js is utilized [Plu23]. Plugins communicate through signals, which are passed through every registered plugin. These signals contain different editor events depending on the action or change made in the editor. If a user, for example, creates a connection between nodes, a `connectioncreated` event is signaled. Rete.js does not have a plugin for integrated real-time synchronization. Therefore, a custom plugin was implemented to process every relevant editor event and

```
// Subscribe to an event
channel.bind('client-mouse-moved', (event: MouseMoveEvent) => {
    applyMouseMove(event);
});

// Trigger an event
channel.trigger('client-mouse-moved', {
    id: 'c3c5a87b-d6fd-483f-9b96-f0d1605c7654',
    x: 612.9931388063334,
    y: 257.26468859216976,
});
```

Listing 9: Example of subscribing to an event in a Pusher channel and triggering an event with data sent along.

trigger client events. The signals also contain the change data and can be sent with the client events to enable all other clients to apply the changes to their editor.

While this enables synchronization between users, a concurrency issue can arise. Conflicts and inconsistencies can occur if two users, for example, modify the same node simultaneously, possibly creating a race condition. To mitigate these problems, the concept of locks is applied. Locks limit access to resources when there are many threads of execution. In a multithreaded application, a lock prevents other threads from accessing the resource (e.g., a file or memory) while a thread modifies it. In this use case, the resources are the elements of the visual programming editor, and the threads are the connected users. If a user selects a node, a WebSocket event is sent to notify other users that this user now locks this node, turning off any interaction with this node for the users. The visualization of those locks is displayed in Figure 6.11. Two different users lock the Dataset and Conv2D node. Each node is outlined with the user's color, and their name is displayed at the bottom of the node. Besides drawing each user's cursor, control elements are also outlined if users select and change them. This allows each user to be able to follow the actions of other users precisely.

The presented mechanism ensures synchronization between already connected users. New users need to be synchronized to the newest state if they connect. Therefore, besides sending WebSockets events, additional HTTP requests are sent to the AI API with the updates and locks, which are then stored in the database to preserve the current state of the editor. This state can be downloaded by new users, ensuring it is synchronized with the other users.

Locks in multithreading applications bring the risk of deadlocks. A deadlock is a situation where no thread, including itself, can not proceed because each waits for another thread to finish or unlock a resource. In the case of WebSockets, this can occur if a user suddenly disconnects, not releasing its locks. To overcome this, each other connected user receives a WebSocket event if a user disconnects, releasing the locks of that user.

Additionally, after a new user connects, it compares the list of connected client IDs with those stored in the lock status, removing all locks that can not match the existing IDs.

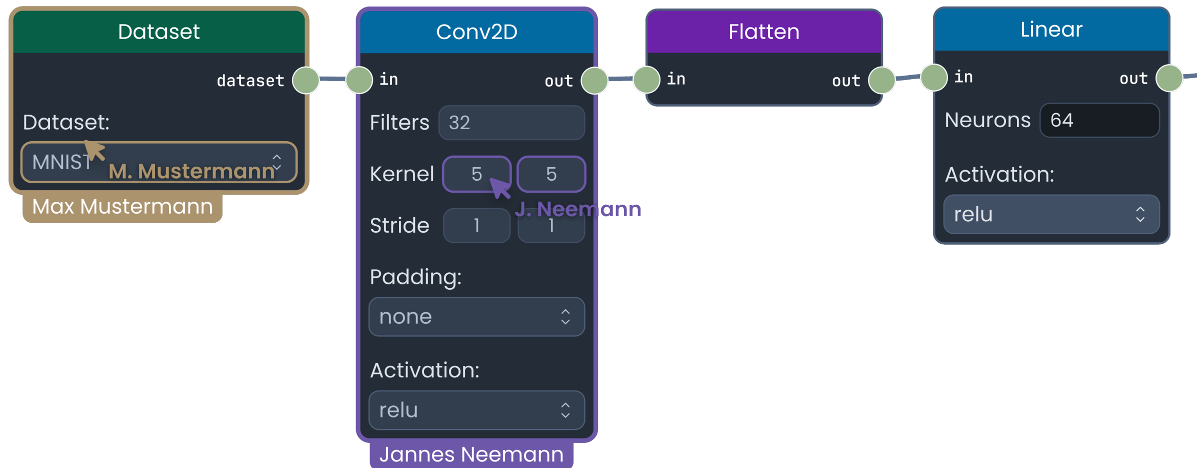


Figure 6.11.: Visualization of user locking nodes and control elements.

6.4. Training Neural Network Models

After users created a NN architecture collaboratively, it needs to be trained. Additional steps are required to make this possible. First, the creation of datasets is explained, and afterward, how the nodes are parsed to executable PyTorch code. The NN training workflow and metric visualization are presented in the last section.

6.4.1. Creating Datasets

Datasets are the core element to train NN models. As displayed in Figure 6.6, a dataset node is the start node of every model created in the visual editor. Users can select which dataset they want to use for training. US-1 requires users to be able to upload new datasets. To realize this feature, a standardized format is needed. In the case of a classification dataset, users are required to upload a compressed folder, where each contained folder name corresponds to a class, and each folder should only include images assigned to that class.

As US-4 requires metadata extraction, the uploaded dataset has to be processed and analyzed to create the relevant metadata before it is uploaded to the centralized dataset store. Which kind of metadata is stored for each dataset is presented in Section A.3.

This results in two upload steps: the upload done by the user to the server and the upload done by the server to the store. As datasets are generally quite large, these uploads can take time. To let the users not wait until both uploads are completed, the processing and uploading on the server are done in the background by the *AI API Worker*, and the user gets information about the current dataset status. Figure 6.12 displays the different steps involved in creating a dataset. After the user enters the required information, the AI API saves the dataset information in the database and starts a new Celery-Job. The AI API Worker executes this job and first sends an update event through the WebSocket Server to the client that the dataset is now being processed and also updates the status in the database. Afterward, the archive file is extracted, and the metadata is generated and stored in the database. Lastly, a ClearML dataset is initiated, as explained in Section 4.3.1, and the files are uploaded to the store. A final status change is executed, marking the dataset to be ready to use for training.

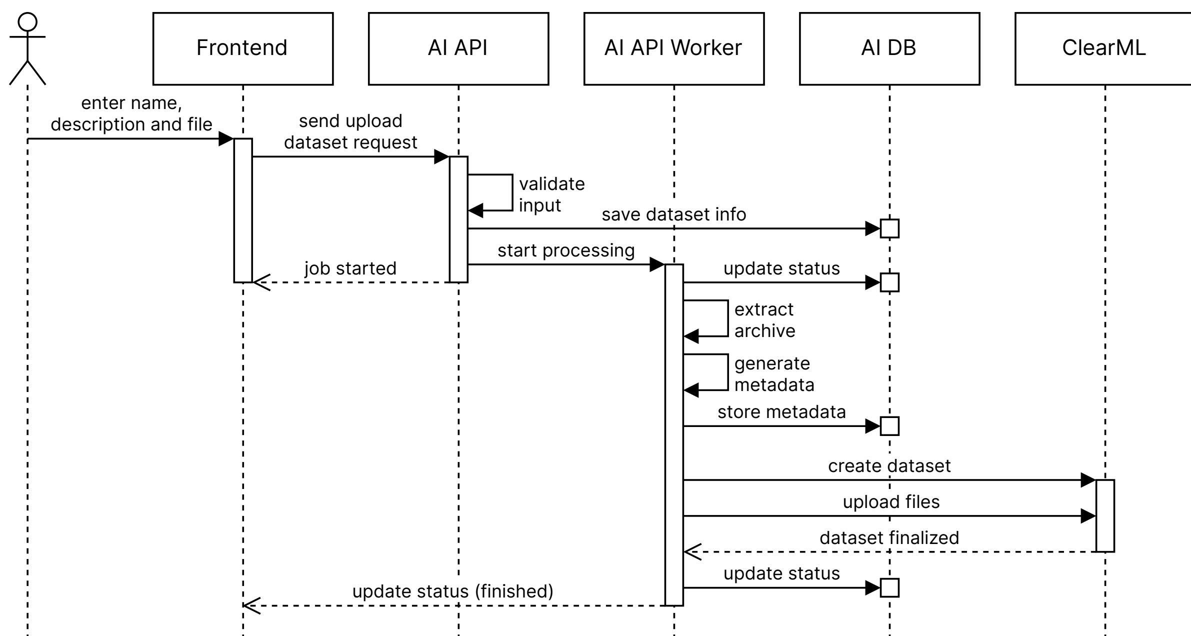
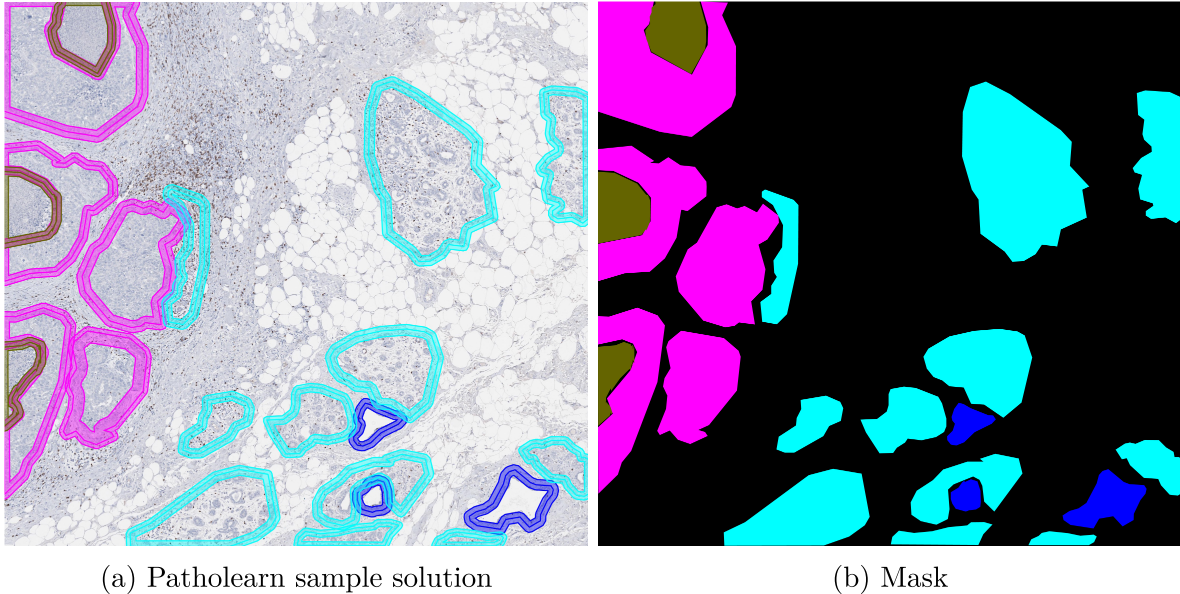


Figure 6.12.: Sequence diagram of creating a dataset.

Besides uploading classification datasets, segmentation datasets can be created through the available tasks in PathoLearn. The annotations created by students and teachers on the WSI represent the masks required for training segmentation models. Therefore, users can select tasks from PathoLearn from which a dataset will be generated. In the current implementation, only the tasks that utilize polygon annotations can be selected. Users can select which annotations should be used for training: the sample solution, their solution or both. If they want to include their solution, they can select whether only the correct annotations or all should be included. Considering the sample solution,

only the actual polygon of an annotation is used, not the inner and outer thresholds. Figure 6.13 illustrates a sample solution of a PathoLearn task (a) and the resulting mask (b). The mask is generated from the annotation coordinates and their *annotation class* (see Chapter 1). The color saved for the class is reused in the mask.



(a) Patholearn sample solution (b) Mask

Figure 6.13.: The sample solution of a PathoLearn task (a) and the resulting mask (b).

As Section 2.5.2 presented, WSIs can not be used directly for training due to their size. The WSIs of the selected tasks are split into quadratic patches. The user can decide which patch size to use: 128, 256 or 512. The WSI viewer implemented in PathoLearn uses the *Deep Zoom* file format [Nee21]. It stores the WSI at different resolutions (levels), and each level is split into 256×256 patches. Not every level dimension is divisible by 256, so some patches at the image's border are smaller. Due to the fixed and inconsistent patch size, the patches need to be regenerated. The full resolution of the WSI is reconstructed from the Deep Zoom and then split into the selected patch size. Border patches that would not have full-resolution reuse some image data from the previous patches. As described in Section 2.5.2, the patching removes the contextual information of annotations. To minimize this, users can choose a downscaling factor. This factor is applied before patching the image and mask. Therefore, the higher the downscaling factor, the more contextual information is preserved on each patch, but fewer patches are generated. Each image and mask patch receive the same file name to ensure the correct mask-image combination is used during training. As the annotation class is color-encoded in the mask image, the image and mask patches can be stored in two separate folders. Those folders are uploaded to ClearML, as described with the classification dataset (see Figure 6.12).

6.4.2. Parsing Visual Programming Editor Nodes

Parsing the NN architecture users create involves converting the nodes in the visual programming editor into Python/PyTorch code that the server can run. As Section 4.2.2 presented, Lightning should be used besides PyTorch.

The nodes and connections created in the Node Editor can be viewed as a directed acyclic graph (DAG) $\mathcal{G} = (\mathcal{A}, \mathcal{V})$, with $\mathcal{A}$ being the set of nodes and $\mathcal{V}$ the directed edges. Two nodes $v, u \in \mathcal{V}$ have a directed connection $(v, u) \in \mathcal{A}$, and no cycles $(v, v) \notin \mathcal{A}$ can be created. This allows the use of graph-based algorithms for parsing the graph to PyTorch models. As displayed in Figure 6.6, the general structure of a graph created by the user consists of a dataset node as the start node, an indefinite amount of nodes forming the NN architecture, and the end of the graph is always formed by an output node, connected to an arbitrary amount of metric nodes.

```
class DataModule(pl.LightningDataModule):
    def __init__(self, batch_size: int = 32):
        super().__init__()
        self.batch_size = batch_size

    def prepare_data(self):
        ClassificationDataset()

    def setup(self, stage: str):
        self.full_dataset = ClassificationDataset()
        self.train_dataset, self.val_dataset, self.test_dataset = random_split(self.full_dataset,
                [0.8, 0.1, 0.1])

    def train_dataloader(self):
        return DataLoader(self.train_dataset, batch_size=self.batch_size,
            num_workers=multiprocessing.cpu_count(),shuffle=True, drop_last=True)

    def val_dataloader(self):
        return DataLoader(self.val_dataset, batch_size=self.batch_size,
            num_workers=multiprocessing.cpu_count(), shuffle=False, drop_last=False)

    def test_dataloader(self):
        return DataLoader(self.test_dataset, batch_size=self.batch_size,
            num_workers=multiprocessing.cpu_count(), shuffle=False, drop_last=False)
```

Listing 10: Datamodule used for classification tasks.

As explained in Section 4.2.2, PyTorch, combined with Lightning, still has boilerplate code that can be reused for every architecture users create. Therefore, templates represent these repeating code fragments, and architecture-specific elements are replaced or injected through Pythons template strings [War02]. Section A.4 introduces the template used for the dataset class of a classification task. For segmentation tasks, only additional steps are added to load the mask patches and map the mask colors to numeric labels. In both cases, the only dynamic element that is changed during runtime is the ClearML dataset ID, which depends on the selected dataset in the Dataset node. As explained

in Section 2.5.1, is the selected dataset split into a training, validation and test dataset. To ease the process, the `DataModule` class is used from the Lightning library [Lig23]. It offers predefined methods for initially setting up the datasets and creating the required PyTorch `DataLoader` objects. As displayed in Listing 10 is the `setup` function used to initialize the dataset object and split it into the three sub-datasets. 80% of the data is used for training, 10% for validation, and 10% for testing. The corresponding methods use these datasets and, in the Output node, configured batch size to create the `DataLoader` objects. Without further configuration, the DataModule instance can be directly passed to the Lightning-Trainer instance.

In PyTorch, the layers of a model can be grouped inside a `torch.nn.Sequential` class. Like every other layer, the input data can be passed to it, acting as a single entry point to the model. Internally, the data is passed sequentially in the order the layers were added, representing the entire forward pass of the network (see Listing 11).

```
model = torch.nn.Sequential(
    torch.nn.Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1)),
    torch.nn.ReLU(),
    torch.nn.Conv2d(16, 16, kernel_size=(5, 5), stride=(1, 1)),
    torch.nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0),
    torch.nn.Flatten(start_dim=1, end_dim=-1),
    torch.nn.Linear(in_features=256, out_features=10, bias=True),
)
prediction = model(x) # forward pass
```

Listing 11: Example of the PyTorch `Sequential` class for encapsulating the layers of the network.

In Section 6.3.1 introduced, the available nodes in the visual programming editor are parsed to the PyTorch equivalent layers and code. Chapter 2 and Chapter 3 also presented the relevant parameters that can be adjusted in the different building blocks. As required by US-20, these parameters can be adjusted in the respective nodes (see Figure 6.4). Figure 6.14 illustrates examplatory the parsing result of a Conv2D node. The parameters are directly used as parameters for initializing the corresponding PyTorch classes. While the layer's activation function is selected in the node, in PyTorch, this requires an additional class instantiation. For padding, the user can either select *none* or *same* padding (see Section 2.6.2).

Every PyTorch layer inherits from the `torch.nn.Module` class, which enables easy implementation of custom layers. This is necessary, as Pooling nodes also support the same padding options as the Conv2D nodes, but PyTorch does not offer pooling layers with same padding. To overcome this, a `MaxPool2DSame` class and a `AvgPool2DSame` class were implemented to act as a drop-in replacement for the `MaxPool2D` and `AvgPool2D` classes, if the user selected same padding (see Section A.7).

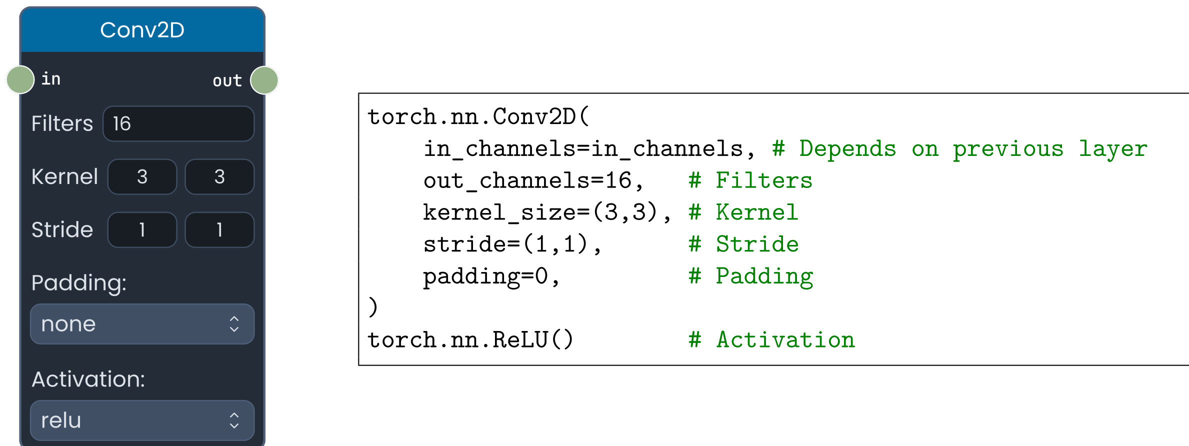


Figure 6.14.: A convolutional node and the parsed PyTorch elements. The parameter values of the node are used as the parameters for the corresponding PyTorch layers.

The Add and Concatenate Node are mostly realized through explicit PyTorch function calls, not as `torch.nn.Module` layer [Tor23b, Tor23c]. Custom layers were implemented to ease the parsing, which realize the same functionality (see Section A.8). Both nodes merge branches in the data flow of the neural network, which increases the complexity of parsing $\mathcal{G}$. With a depth-first approach, the parsing algorithm finds all paths from the Dataset node to the Output node. This removes all nodes from $\mathcal{G}$ that are present in the visual programming editor but do not have any connections. Additionally, all branches are detected. A forward pass is simulated while traversing the graph to ensure the input data can be passed through the network successfully. A data sample is created based on the selected dataset. After each layer is parsed, the sample is applied to it, and the output is used as the new sample for the next layer. Suppose multiple layers use the previous layer output as input and create a branch. In that case, each branch is recursively parsed, as it could also contain additional branches, and the output of each branch is tracked to ensure their dimensions match at the Add and Concatenate node.

Considering the realization of the inception module in PathoLearn (see Figure A.8), multiple Concatenate nodes are used. Listing 12 shows the parsed PyTorch model. The multiple Concatenate nodes were optimized to a single `Concatenate` layer. Each branch in the architecture uses its own `torch.nn.Sequential` container. This way, an arbitrary nested neural network can be created and trained.

Besides the inception module, Figure A.9 illustrates an implementation of the residual block presented in Section 3.1, which uses the `Add` layer and an additional branch to create the skip connection.

The resulting PyTorch model is integrated into a `LightningModule` (see Listing 3). Additionally, the metric nodes are parsed to the corresponding TorchMetric classes and added to the relevant positions in the template (see Section A.6).

```
self.model = torch.nn.Sequential(
    torch.nn.Conv2d(3, 192, kernel_size=(1, 1), stride=(1, 1), padding="same"),
    torch.nn.ReLU(),
    Concatenate(
        torch.nn.Sequential(
            torch.nn.Conv2d(192, 96, kernel_size=(1, 1), stride=(1, 1), padding="same"),
            torch.nn.ReLU(),
            torch.nn.Conv2d(96, 128, kernel_size=(3, 3), stride=(1, 1), padding="same"),
            torch.nn.ReLU(),
        ),
        torch.nn.Sequential(
            torch.nn.Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1), padding="same"),
            torch.nn.ReLU(),
        ),
        torch.nn.Sequential(
            torch.nn.Conv2d(192, 16, kernel_size=(1, 1), stride=(1, 1), padding="same"),
            torch.nn.ReLU(),
            torch.nn.Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1), padding="same"),
            torch.nn.ReLU(),
        ),
        torch.nn.Sequential(
            MaxPool2dSame(kernel_size=(3, 3), stride=(1, 1), padding=0, dilation=1, ceil_mode=False),
            torch.nn.Conv2d(192, 32, kernel_size=(1, 1), stride=(1, 1), padding="same"),
            torch.nn.ReLU(),
        ),
    ),
)
```

Listing 12: The PyTorch model of the inception module displayed in Figure A.8.

Finally, the Output node is parsed, adding the selected loss and optimizer function to the Lightning Module (see Section A.6) and initializing a `Trainer` instance. The script can be executed after adding the required import statements.

6.4.3. Training workflow

The ClearML Agent presented in Section 4.3.1 is utilized in training the parsed NN. Each script includes the necessary code to run it on an Agent instance (see Listing 4). Figure 6.15 illustrates the different steps involved in the training workflow. The user starts the training, and the AI API parses the NN, as described in the previous section. The resulting script is sent to the *AI-Train Worker*, which starts a new Celery-Job and executes the script, enqueuing the ClearML Task at the ClearML Server. Then a new periodic Celery-Job is spawned at the *AI-Beat Worker*. ClearML does not offer WebSocket capabilities. Therefore, clients must utilize polling to check whether the ClearML Task is finished. This would create a lot of load on the server. To overcome this, the AI-Beat Worker acts as a polling middleman for every client. Every five seconds,

it loads the task information from the ClearML Server and sends WebSocket events to the connected clients, informing whether the task status has changed. Each metric node displays metric-specific values. It displays the minimum, maximum, and current metric values. To enable automatic updates of those nodes, the worker fetches the newest metric values from ClearML and sends them with a WebSocket event.

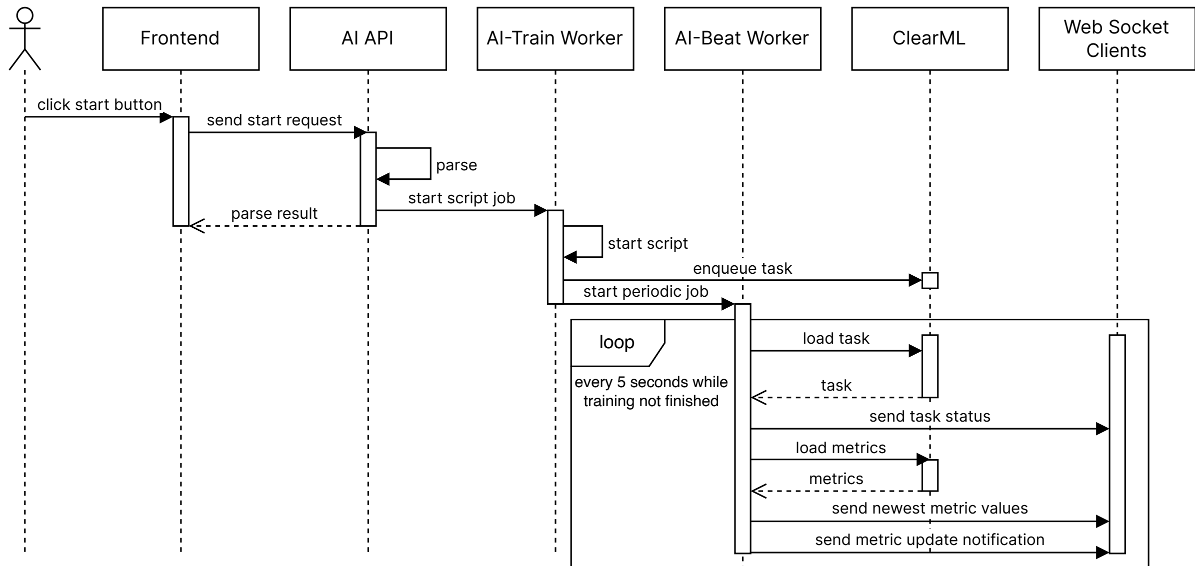


Figure 6.15.: The workflow of training a neural network architecture created by users.

As the metric nodes only display the current value, users can show the progression of metrics over time through diagrams. Figure 6.16 illustrates how this is visualized in PathoLearn. The users's selected metrics are displayed as graphs over the training iterations for the training, validation and test dataset. The AI-Beat Worker sends an update notification that new data is available through a WebSocket event to enable continuous updates. The new data can not be sent directly with this event, as it can be too big. Therefore, clients fetch the metrics directly from the server after receiving the update notification. Besides metrics visualization, the ClearML internal logs are also displayed (see Figure 6.17). Internal training details are displayed, enabling easier debugging if errors occur. Like the metric diagrams, the logs are also periodically updated.

If the ClearML Task status changes to a state where no progress on the training is made (e.g., completed or failed), the periodic Celery-Job is canceled, and no more updates are sent to the client.

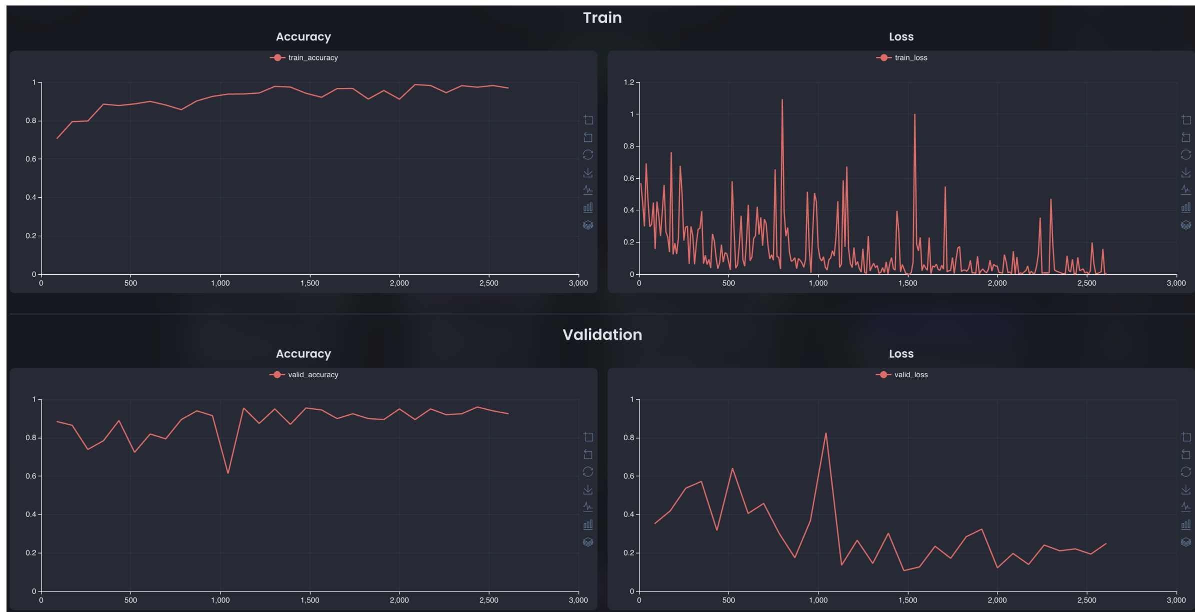


Figure 6.16.: The metrics page displays the users' selected metrics in graphs over the training iterations.

Logs

```
                       Epoch 0: 100%|██████████| 87/87 [00:05<00:00, 15.11it/s, v_num=0]
                       Epoch 1:   8%|▉         | 7/87 [00:01<00:12,  6.25it/s, v_num=0]

12.09.2023, 11:10:01   GPU available: True (cuda), used: True
                       TPU available: False, using: 0 TPU cores
                       IPU available: False, using: 0 IPUs
                       HPU available: False, using: 0 HPUs
                       2023-09-12 11:09:57,067 - clearml - INFO - Dataset.get() did not specify alias. Dataset information will not be automatically logged in ClearML Ser
                       ver.
                       Missing logger folder: /data/.clearml/venvs-builds/3.10/code/lightning_logs
                       2023-09-12 11:09:57,476 - clearml - INFO - Dataset.get() did not specify alias. Dataset information will not be automatically logged in ClearML Ser
                       ver.
                       LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

                         | Name           | Type                | Params
                       -------------------------------------------------------
                       0 | model          | ClassificationModel | 11.2 M
                       1 | valid_accuracy | MulticlassAccuracy  | 0
                       2 | train_accuracy | MulticlassAccuracy  | 0
                       3 | test_accuracy  | MulticlassAccuracy  | 0
                       -------------------------------------------------------
                       11.2 M    Trainable params
                       0         Non-trainable params
                       11.2 M    Total params
                       44.710    Total estimated model params size (MB)
                       Epoch 0:  31%|███       | 27/87 [00:02<00:04, 12.86it/s, v_num=0]

12.09.2023, 11:09:57    ::: Using Cached environment /data/.clearml/venvs-cache/2a5ec68aac5ba7e4ec4282161ddd2bad.5a571c19680ab335c00a13594d3548f7.b36f742a97fbd7fd9073ac7ad
                       64612c8  :::


                       Adding venv into cache: /data/.clearml/venvs-builds/3.10
                       Running task id [be9ede3368c74c44b3b6f2440a0e5e21]:
                       [.]$ /data/.clearml/venvs-builds/3.10/bin/python -u /data/.clearml/venvs-builds/3.10/code/builder_train.py
                       Summary - installed python packages:
                       pip:
                       - aiohttp==3.8.5
                       - aiosignal==1.3.1
```

Figure 6.17.: The logs page displaying ClearML internal logs of the training.

6.5. Serving Neural Network Models

As defined in US-18, the trained NN should be served to the users. Model serving is also referred to as using the model ”in production,” which generally means to make predictions. Section 4.3 already introduced ClearML Serving, which offers an easy-to-use and scalable approach to serving NN models through REST APIs. Making predictions on a trained PyTorch model requires the structure of the neural network to exist. The exported file only contains the parameter values for each layer, which can be loaded into the architecture. *TorchScript* overcomes this problem by serializing the PyTorch models into an optimized, environment-independent format [Tor]. This enables the model to run on other, possibly more performant programming languages, e.g., C++, to improve the prediction speed. The *Open Neural Network Exchange* (ONNX) standard is an open format to represent NN models [ONN23]. It defines various building blocks (operators) and a common file format. An ONNX model only contains the necessary mathematical operations to do inference (prediction). Therefore, ONNX models can run on any platform supported by the ONNX platform (including mobile phones and browsers). PyTorch models can be directly converted to ONNX models. The only downside of using the ONNX operators is that specialized or custom operations are not supported. Additionally, very small numerical errors can occur due to the different operator implementations [Tor23d]. The performance evaluation performed in Section A.11 showed that this numerical error is so small that it does not influence the final result in any way. Considering the serving performance, it was shown that there is no significant difference in prediction speed between both formats. Because the ONNX format aims to create a unified standard for NN model representation independently of the DL frameworks, it will be used as the serving format in PathoLearn.

PyTorch and Lightning support direct export to the ONNX format. Listing 13 shows the code used for exporting. As the NN is converted to the operators of ONNX, it is required to define the expected input dimensions. The `input_sample` defines an example input. The `dynamic_axes` object defines which data elements can have a dynamic size. In this case, the batch size is variable, allowing users to get predictions on multiple images. Additionally, the width and height of the image can be arbitrary, as it will be resized to the required size anyway. Only the number of image channels is fixed, defined by the dataset the model was trained on. Finally, a ClearML *OutputModel* is initiated, creating a Model instance and storing the ONNX model in the ClearML Server.

After exporting and saving the model, a new ClearML Serving endpoint is created. Due to the missing SDK in ClearML Serving, the endpoint can only be created through the command-line interface. Listing 14 shows an example command that creates a new endpoint. The options sent along the command are nearly equivalent to those specified in the ONNX export in Listing 13. Axis that have a dynamic axis are encoded with a -1. To create a unique REST endpoint for every trained model, the ClearML Task ID is used.

```
trained_model = LightningModel.load_from_checkpoint(
    trainer.checkpoint_callback.best_model_path, model=model
)
trained_model.freeze()
trained_model.eval()
trained_model.to_onnx(
    "model.onnx",
    input_sample=torch.randn(1, 3, 256, 256, requires_grad=False),
    export_params=True,
    input_names=["input"],
    opset_version=17,
    output_names=["output"],
    dynamic_axes={
        "input": {0: "batch_size", 2: "width", 3: "height"},
        "output": {0: "batch_size", 2: "width", 3: "height"},
    },
)
OutputModel(task=task, name="model", framework="onnx").update_weights("model.onnx")
```

Listing 13: Code example of exporting a trained Lightning model to the ONNX format.

Additionally, a script file can be specified through the `--preprocess` options. It enables the data preprocessing before sending it to the Triton instance and postprocessing on the prediction result. The preprocessing involves resizing the image and normalizing it. In the case of a classification model, the softmax function is applied in the postprocessing step to get classification probabilities from the model (see Listing 25).

```
clearml-serving model add --engine triton --input-size -1 $channels -1 -1 --input-type float32
↪   --input-name input --output-size -1 $num_classes --output-type float32 --output-name output
↪   --endpoint $clearml_task_id, --preprocess $preprocessing_path --model-id $model_id --aux-config
↪   platform="onnxruntime_onnx" default_model_filename="model.bin"
```

Listing 14: Example command for creating a new ClearML Serving endpoint. The ID of the ClearML Task ID is the endpoint's name.

After initializing the new endpoint, it can take some time until the Triton instance registers the new model. Therefore, a Celery-Job is started that checks every five seconds if the endpoint is accessible. If so, a WebSocket event is sent to inform the user and update the UI. Figure 6.18 shows the UI presented to a user to make predictions. A dropzone allows the user to upload an image. Alternatively, a random image can be selected from the dataset the model was trained on. This image is uploaded to the AI API, which fetches the dataset metadata and sends a request to the ClearML serving endpoint. In the case of a classification task, the probability for each class is determined (see softmax activation function in Section 2.3.3) and returned to the UI through the AI API. The UI combines the probabilities with the class names stored in the dataset metadata and displays probabilities per class (see Figure 6.19).

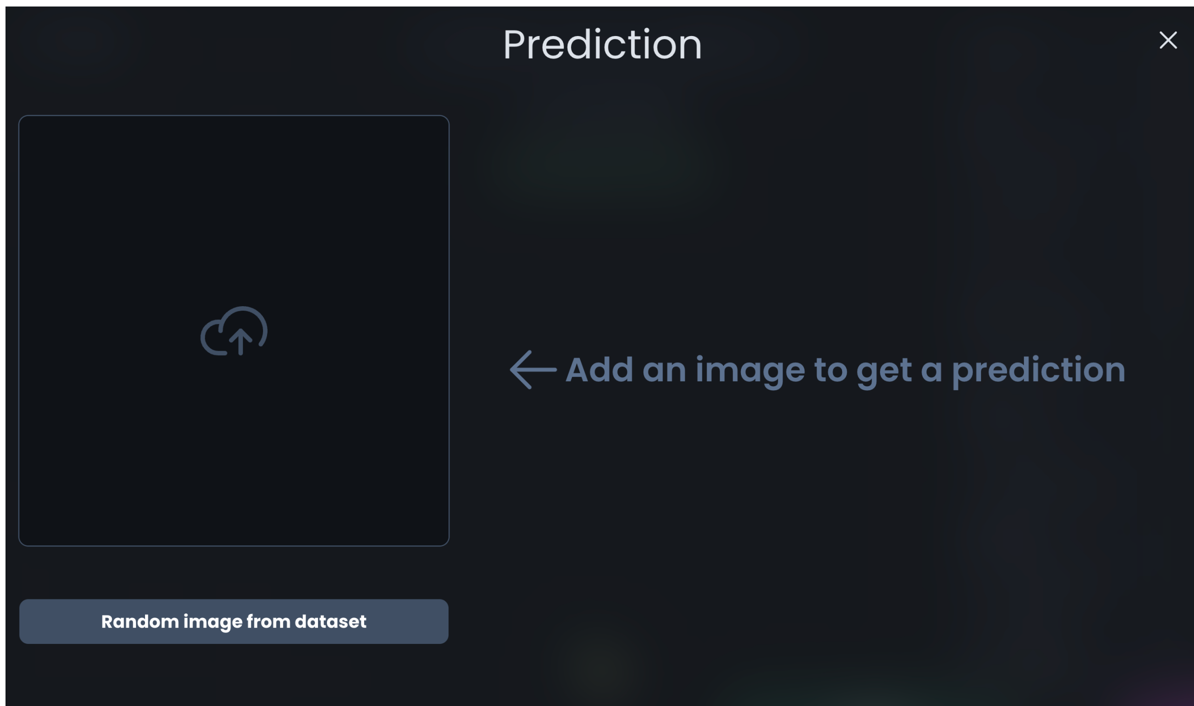


Figure 6.18.: Prediction UI enabling users to add an image to get a prediction.

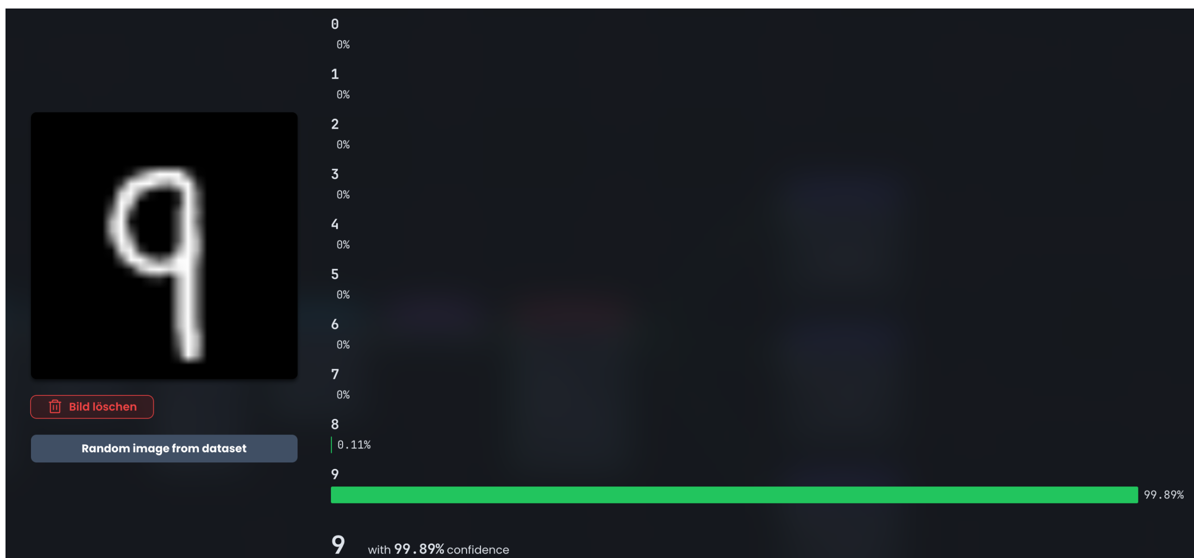


Figure 6.19.: Visualized prediction result for the uploaded image. As a classification task was trained, probabilities for each class were calculated.

Segmentation tasks are handled differently. The probability is calculated for each pixel and class. The pixel receives the class that has the highest probability. The color for that class is extracted from the dataset metadata and stored in the pixel position. Finally, the entire mask is returned to the UI and visualized as illustrated in Figure 6.20. The annotation classes are displayed to allow users to understand which color is which class.

Figure 6.20.: Visualized prediction result for the uploaded image. As a segmentation task was trained, a segmentation mask is returned.

7. User Test

To evaluate the extension of PathoLearn, a user test was conducted. The primary goal of this test was to determine whether the implemented features worked as intended and whether they enjoyed them. Also, it should be checked whether the participants' knowledge of AI improved through using PathoLearn. A working group at the MHH was asked to participate. Additionally, external cooperation partners from different universities in Luxembourg, Spain, and Italy were invited to participate.

7.1. Execution

As PathoLearn is primarily a tool for teaching, the general idea was to create an environment of a lecture. In this lecture, the participants should learn about AI and test their knowledge in PathoLearn. All participants were invited to an online meeting or could join physically at the MHH. First, a presentation (lecture) was held, which introduced AI and how it works. Additionally, the building blocks of artificial intelligence were presented, and how these are represented in PathoLearn. Some example experiments with different datasets and NNs were created beforehand in PathoLearn. These explained how the different nodes can be connected, how multiple users can work together, how the training can be started and monitored, and how predictions on the trained model can be made. It was emphasized to create groups to test the collaboration feature thoroughly. The participants had one week to test PathoLearn with the provided datasets or their own uploaded datasets. On the final day, a closing meeting was held to gather feedback and discuss application use cases of PathoLearn.

Figure 7.1 illustrates the server infrastructure used for realizing the user test. Three servers were used to host the different services. The *Moriarty* server was for management and only ran the ClearML Server instance. *Holmes* had a ClearML Agent instance running and was the primary training server as it had the most powerful GPU. Finally, the *Watson* server ran the PathoLearn software and ClearML Serving. All servers were connected through a gigabit ethernet connection. A hosted proxy server was added because the servers are located inside a private network and cannot be accessed from the outside. The servers connected to the proxy and all requests from the browser were passed through the proxy. Due to the relatively slow upload speed of 50 megabits of the private network, the overall system latency for the user was higher.

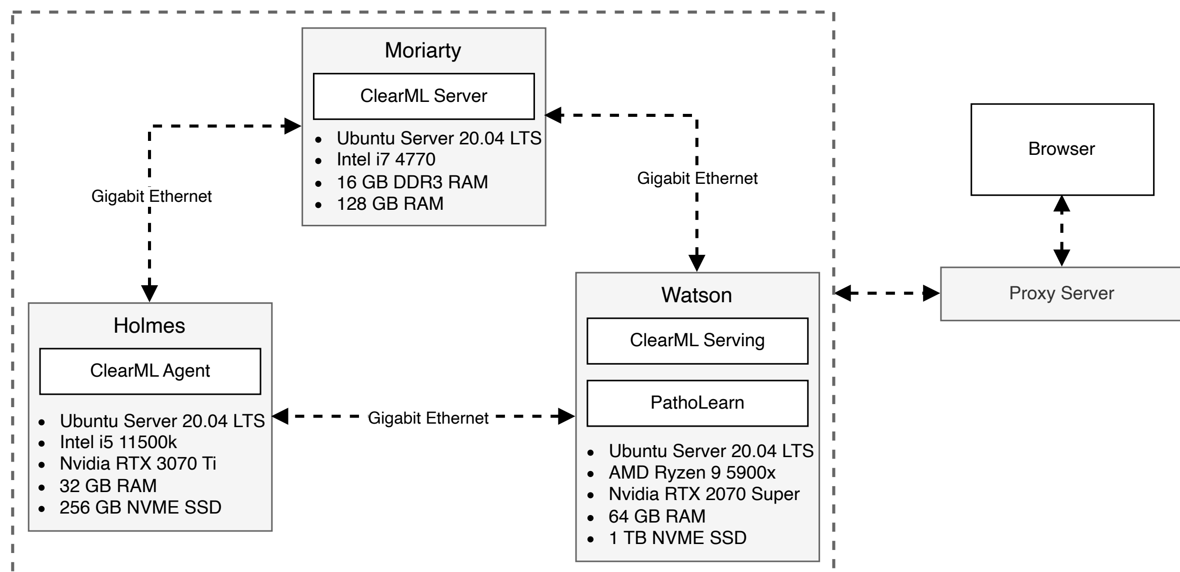


Figure 7.1.: The infrastructure used for the user test.

7.2. Surveys

Besides collecting feedback through the discussions, two anonymous surveys were conducted. The participants were asked to complete the first survey before using PathoLearn and should not include the information gathered in the lecture. The primary goal was to identify the demographics of the group and their general knowledge of AI (see Section A.13.1). In the second survey (see Section A.13.2), the participants were asked to evaluate whether the lecture and software improved their knowledge of AI and if they could see PathoLearn being used for teaching and researching AI. Additionally, they should rate how they liked the software's overall understandability, the UI's look, and the offered features. Finally, they could suggest functionality they found missing, describe any problems encountered, and give final feedback.

7.3. Results

Five participants completed the first survey (see Table A.6). Four have a medical background (digital pathology, neuropathology, oncology), and one has a biological one. One participant was a medical student. The remaining are either professors, PhD students, or postdoctoral researchers. All participants have heard of AI and used software that utilizes AI. They also had a general idea of how artificial intelligence works, but only two had already created AI. Everyone thinks AI is relevant or very relevant in their current

or later job, but especially relevant in their current studies and research projects. They also believe that the relevance of AI will increase in the future.

In the second survey, four participants answered the questions (see Table A.7). All stated that their understanding of AI improved through the lecture and using PathoLearn. Considering using PathoLearn in teaching and research, the participants unanimously agreed that PathoLearn is very usable in the teaching scenario, but some ($n = 2$) think that some additional adjustments have to be made to make PathoLearn usable for actual research. NF-8 requires that PathoLearn is easy to understand. All thought that the UI is intuitive and easy to understand. The main issue was the general lack of knowledge about creating NN. The participants found it hard to understand what the different nodes and parameters do. Therefore, they said that at the current state, an instructional lecture about NN like the one held is essential. To resolve this issue, participants wished that each node could display a short explanation of what it does and how to use it. Also, users would like to get information about the different parameters in the nodes and suggestions about which to change to increase the performance. This request was implemented after the user test (see Section 6.3.1).

The possibility to work collaboratively was enjoyed by everyone, especially by the teachers and students, who could see this being actively used in a lecture. There was a suggestion to extend this further by integrating a chat system.

One topic that participants noted was the reporting and informing of errors. The current implementation only notifies users of an error. It was wished that the errors were presented to the user in more detail by providing feedback on how to solve them.

The presented way of making predictions on the trained model was positively accepted and easy to understand for everyone. It was only wished to make predictions on multiple images simultaneously and present the results in a different way.

8. Requirements Fulfillment

Chapter 5 defined user stories that the End-To-End AI platform extension must fulfill. Based on these, the in Chapter 6 implementation was presented. Table 8.1 shows that the extension fulfills all USs. Additional requirements resulting from the user test were also implemented (see Chapter 7). Besides the USs, additional non-functional requirements

User-Story	Fulfillment	Reference
US-1	✔	Section 6.4.1
US-2	✔	Section 6.4.1
US-3	✔	Section A.5
US-4	✔	Section 6.4.1, Section A.3, Section A.5
US-5	✔	A.5
US-6	✔	Section A.2
US-7	✔	Section A.2
US-8	✔	Section A.2
US-9	✔	Section A.2
US-10	✔	Section A.2
US-11	✔	Section A.2
US-12	✔	Section 6.3.1
US-13	✔	Section 6.3.3
US-14	✔	Section 6.3.1
US-15	✔	Section 6.3.1
US-16	✔	Section 6.4.3
US-17	✔	Section 6.4.3
US-18	✔	Section 6.5
US-19	✔	Section 6.3.2
US-20	✔	Section 6.3

Table 8.1.: Fulfillment of the defined USs in the implemented extension of PathoLearn.

were defined in Section 5.4. NF-1 is fulfilled through the centralized authentication service (see Section 6.2). As the extension's services utilize Docker, the same properties concerning availability (NF-2 and NF-3) are given as the original services of PathoLearn [Nee21]. Services are automatically restarted if they crash. Multiple instances can

be spawned through *Docker Swarm* to ensure horizontal scalability. The small user test showed no performance issues, even with the higher latency. With better network connectivity, the overall performance could even be increased.

Considering NF-5, a small-scale load test of the collaborative features was conducted (see Section A.14). It showed that the system is capable of handling many simultaneous events with multiple connections while having response times below 100 ms. Larger response times seemed to be primarily caused by the client and not the WebSocket server. A larger load test must be executed to measure the overall response time of the system in general and the response time of the collaborative feature specifically.

Additionally, the user test showed that the extension was generally easy to use, and all participants liked the overall user interface. On the other hand, the visual programming editor's concrete functionality was difficult to understand without prior knowledge of AI (see Section 7.3). Therefore, NF-7 and NF-8 could need additional improvements (see Chapter 9).

9. Conclusion and Future Work

This thesis aimed to evaluate and implement an End-To-End AI platform for the open source teaching software PathoLearn. Students should get first contact with creating and using AI, as it becomes increasingly important in their daily jobs. Additionally, researchers should be able to create AI for their research without writing code. To identify the concepts required to train NNs, the foundations of ML and especially CNNs were laid. As digital pathology is primarily based on WSIs, the building blocks of NNs based on the three computer vision tasks, classification, object detection, and image segmentation, were presented. Additionally, it was evaluated that PyTorch is the best-fitting programming framework for creating and training NNs. As the AI lifecycle also includes dataset management and model serving, available software tools that cover the entire lifecycle were compared. ClearML came out on top with the most flexible and complete suite of software libraries and SDKs. It was researched that a visual programming editor is a common way to remove the necessity of knowing how to write code. The existing visual programming editors for teaching ML and AI were either poorly maintained or did not offer to be integrated into other software. Therefore, a custom visual programming editor was implemented. The possibility to work collaboratively in (remote) groups through real-time editor synchronization is a key feature that is not present in the other visual programming editors. This focus on collaboration is also increased through continuous updates of the training status and metric values. Group members can make predictions on their trained model by simply uploading an image to cover the final step of the AI lifecycle.

The user test showed that everyone liked the overall user interface and the general idea of the visual programming editor. The chosen design and offered functionality were accepted by everyone, especially the ability to work together. Besides this, the main downside of the current implementation is the big entry hurdle. Without a general knowledge of the inner workings of AI, it is very hard to create your architectures in the editor. Especially for medical students at the MHH who do not have a lecture on AI will have less motivation to use this software, as students generally want to get through the exams and not want to do additional work not included in the curriculum. Teachers of different fields suggested that the theory should be taught in an additional lecture or even be included in the curriculum, as students must be aware of AI's relevance in their jobs and, therefore, gather a general understanding of it.

To reduce the hurdle, info texts for each node were added. This could be further extended through a more complex feedback system comparable to the one existing in the task-solving of PathoLearn [Nee21]. It could continuously analyze the architecture and give feedback and tips on where it could be improved or which kind of node would fit best next. This can be extended by a step-by-step tutorial, which iteratively creates an architecture while explaining the reason and functionality of the node added. This can also be extended to the training workflow. Currently, the system does not give extensive feedback if the training fails. The concrete error is only displayed on the training logs page. The user test showed that this reduces the motivation to continue using the software. Therefore, the feedback system could check the error occurring and generate concise feedback explaining the error and how to solve it. Besides errors, implementing a system that analyzes training metrics to detect common effects like overfitting or underfitting could propose possible parameter or architecture changes to improve the results.

Datasets for classification can only be created by uploading a folder in a pre-defined format. This could be improved through deeper integration into PathoLearn, instructing users through different steps: give the dataset a name, write down the available classes, and add images to each class. This could improve the understanding of how datasets work, as they actively add images to different classes. This can also be extended to the existing tasks in PathoLearn. Currently, they can only be used for creating segmentation datasets. To improve the connection between solving tasks in PathoLearn and training NNs on the solved tasks, users could select the annotation groups they want to classify and the annotations that should be included. Based on this, the patches are extracted from the WSI, creating a dataset.

A deeper integration with the tasks in PathoLearn can also be realized through the model serving. For students, a new type of task could be implemented. A served model is used to make predictions on images. The image and the prediction are presented to a student, and it has to decide whether the NN made a correct prediction. This would, on the one side, sensitize the user to using NN in their workflow, and, on the other, they could verify their knowledge on the topic. Teachers could also benefit from NNs by using a model to pre-generate annotations on the WSI. This could reduce the time required to create a task, as teachers must only modify the generated annotations and add missing ones. As the predictions are patch-based, the challenge of merging patches has to be resolved for this kind of integration.

The collaborative feature is a primary component of the visual programming editor. Currently, all projects and experiments can be accessed by all authenticated users. This could be changed and extended to the course system in PathoLearn. Teachers could not only create tasks but also experiments in their courses. This way, everything is encapsulated in a course, which better represents the structure of a lecture. Additionally, students could create their own projects and experiments. Instead of being publicly

available for everyone, they can invite specific users in the system to their project to grant them access and utilize the collaborative features. As suggested in the user test, this could be further improved by introducing a chat in the visual programming editor or even allowing them to communicate through voice chat.

Creating and training NNs is often about changing a few layers or parameters. In the current implementation, a new experiment would have to be created to realize this. A version system could be implemented that allows users to clone an architecture to create a new version inside the same experiment. This allows training a version and simultaneously preparing a new version for training. This way, users can compare the performance of different versions to identify which architecture or parameters are the best. This could also be extended to make the best version publicly available. Other users could copy the architecture or make predictions on the trained model.

Concluding, this thesis showed how an end-to-end AI platform can be implemented with a collaborative visual programming editor in the existing software PathoLearn. It builds the foundation for creating and training NNs based on images. The high-level user interface removes the need to understand and write text-based code. Additionally, they do not need to think about server management for training and how the model serving is realized, as everything is happening under the hood. They can focus on improving their architecture with other group members simultaneously if the performance is unsatisfactory. Besides this, the general software architecture allows easy integration of the presented improvements and additional features.

A. Appendix

A.1. Comparison of Pre-Trained and Not Pre-Trained CNNs

Section 3.4 introduced the method of using pre-trained models, which used large datasets, to improve performance on other, potentially smaller, datasets. This section outlines a small-scale experiment comparing pre-trained and non-pre-trained models to determine if pre-trained models demonstrate superior performance.

A.1.1. Dataset

The Breast Cancer Histopathological Database (BreakHis) [SOPH16] dataset is more complex, as it contains RGB images of 700×460 pixels. The 1994 images are also split into 70% training (1397), 10% validation (199), and 20% test (399) images. The dataset contains four benign classes (adenosis, fibroadenoma, phyllodes tumor, papillary carcinoma) and four malignant classes (ductal carcinoma, lobular carcinoma, mucinous carcinoma, papillary carcinoma). For the training, a binary classification was realized to determine whether an image is benign or malignant.

A.1.2. Model Configuration

The ResNet-18 architecture was used as the NN model. Three different models were trained. The first does not use pre-trained weights. The second uses the offered weights from PyTorch, where the ImageNet dataset was used for training [RDS+15, Mod23b]. The following will refer to it as *general*. Lastly, the third model used weights trained on the Breast Histopathology Images (BHI) [JM16] to evaluate whether medical datasets can improve the performance of other medical datasets. The dataset contains $227,524$ images with 50×50 pixels, where $198,738$ are benign and $78,786$ are malignant. Therefore, the dataset is a lot larger than the BreakHis dataset. The ResNet-18 architecture was trained with the BHI dataset, and the weights were exported and loaded into the model for the BreakHis dataset training (see Section 3.4).

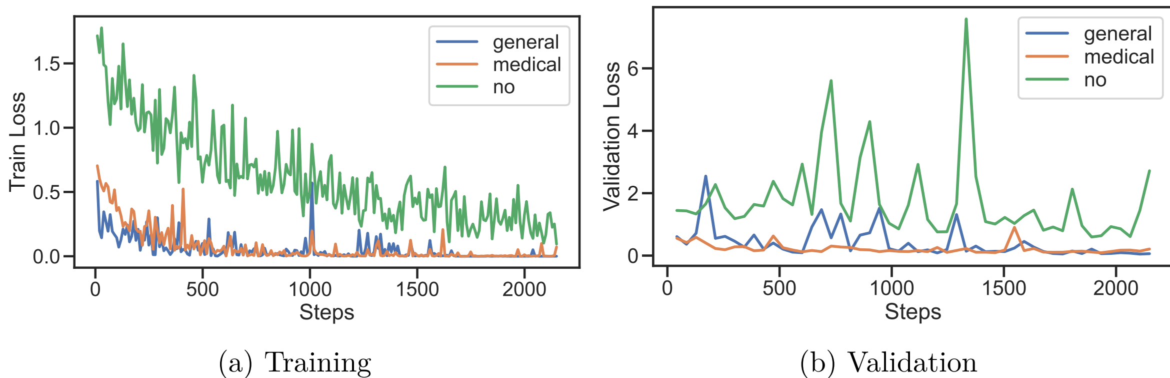
All three models were trained for 50 epochs with a batch size of 32. The cross-entropy loss function was combined with Adam and a learning rate of 0.001. Every training was run through ClearML with a ClearML Agent. The hardware and software configurations can be found in Table A.1.

OS	Ubuntu Server 20.04 LTS
CPU	AMD Ryzen9 5900x, 12 cores, 24 threads, up to 4.80 GHz
Memory	64GB DDR4, 3200 MHz
GPU	Nvidia RTX 2070 Super 8GB
Storage	1 TB NVME PCIE 3.0 SSD (OS and software installation) 2 TB SATA SSD (Data storage)
Docker	24.0.5
ClearML Server	1.11.0
ClearML Agent	1.5.2
ClearML Serving	1.3.0
Python	3.10
PyTorch	2.0.1+cu118
Lightning	2.0.7

Table A.1.: Hardware and software specification of the server.

A.1.3. Results

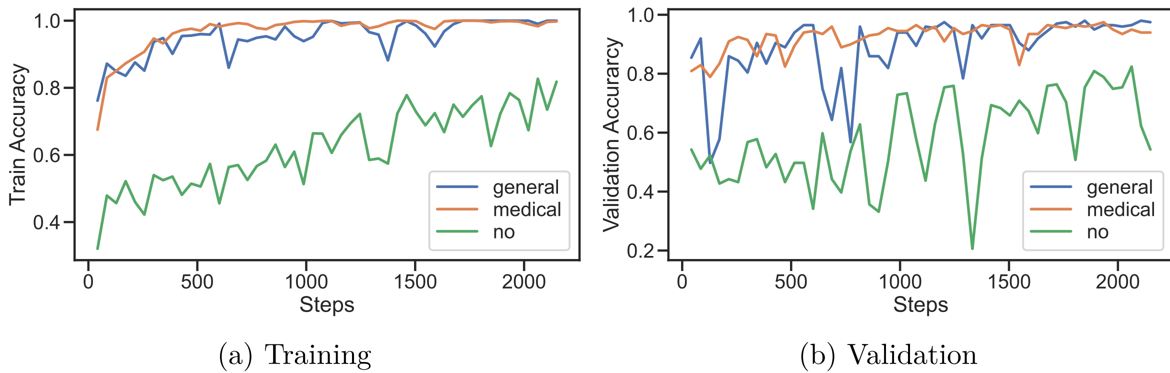
Figure A.1 shows the different loss values for all three models for the training and validation dataset. It can be seen that both the medical and general model training converges quickly, while the model with no pre-trained weights has higher loss values with slower convergences. Additionally, the model shows higher fluctuation, especially on the validation dataset.



(a) Training (b) Validation

Figure A.1.: The loss values for both the training (a) and validation (b) datasets.

Comparable results can be seen in Figure A.2. The pre-trained models have a much higher initial accuracy than the model with no pre-trained weights. Due to the faster training convergence, the accuracy also saturates more quickly. The fluctuation in the loss values is also reflected in the accuracy. While the model improves its accuracy continuously, it only showed a maximum training accuracy of 82.70% and 82.41% on the validation dataset. The difference in accuracy becomes more noticeable on the test dataset, where it measured 67.17%, and the medical and general model reached 98.74% and 99.25%, respectively. Comparing the medical and general models, no significant difference can be observed. Their accuracy and loss values do not differ much.



(a) Training (b) Validation

Figure A.2.: The accuracy values for both the training (a) and validation (b) datasets.

Finally, it can be said that using pre-trained models significantly increases accuracy and training speed. Using a dataset from another or similar domain seems to make no difference. This could be due to the ImageNet dataset containing various classes, and therefore, many complex features can be learned and reused. Medical or domain-specific weights could influence the results if the dataset is more complex (e.g., more classes and complex images).

A.2. The Project and Experiment Page

The project and experiment pages are for managing different visual programming instances. A project acts as a container for multiple experiments. As displayed in A.3 the different projects are listed with their name and description. Additionally, users can create new projects with the buttons available. Users can modify or delete the project with the three dots in the top right corner.

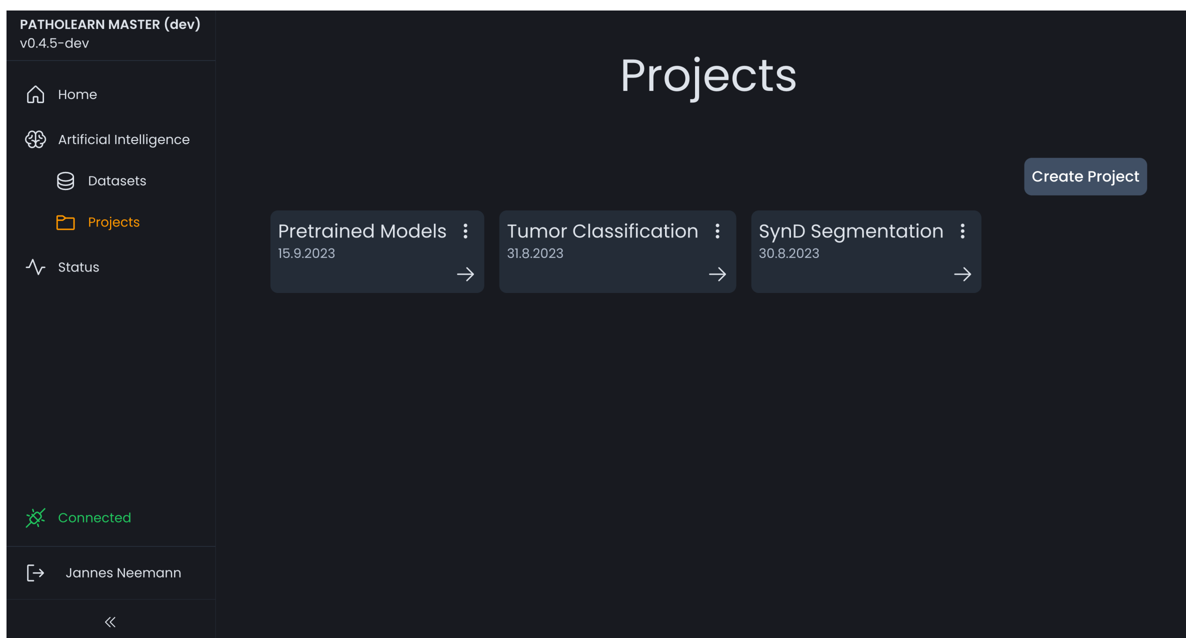


Figure A.3.: The project page in PathoLearn.

Selecting a project with the arrow navigates the user to the experiment page of the project. It uses the same layout, listing each experiment (see Figure A.4). Only an additional element is present for each experiment where it is visualized, whether a created model is currently training, is finished, or the training failed. The visual programming editor for the specific experiment is opened by pressing the arrow (see Figure A.5).

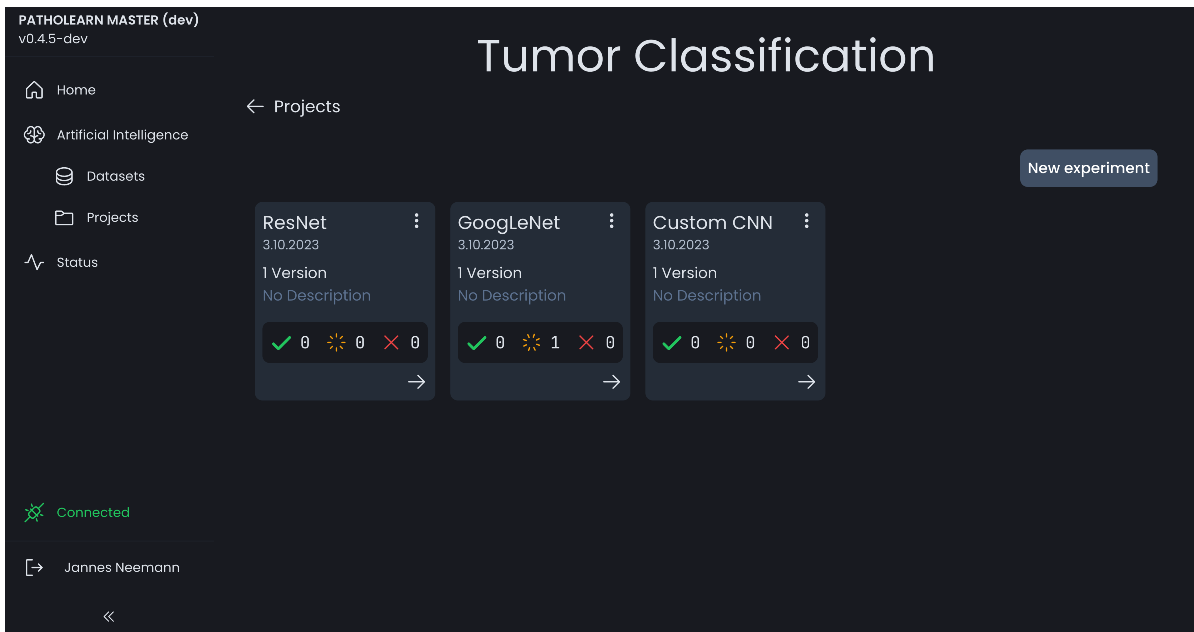


Figure A.4.: The experiment page in PathoLearn.

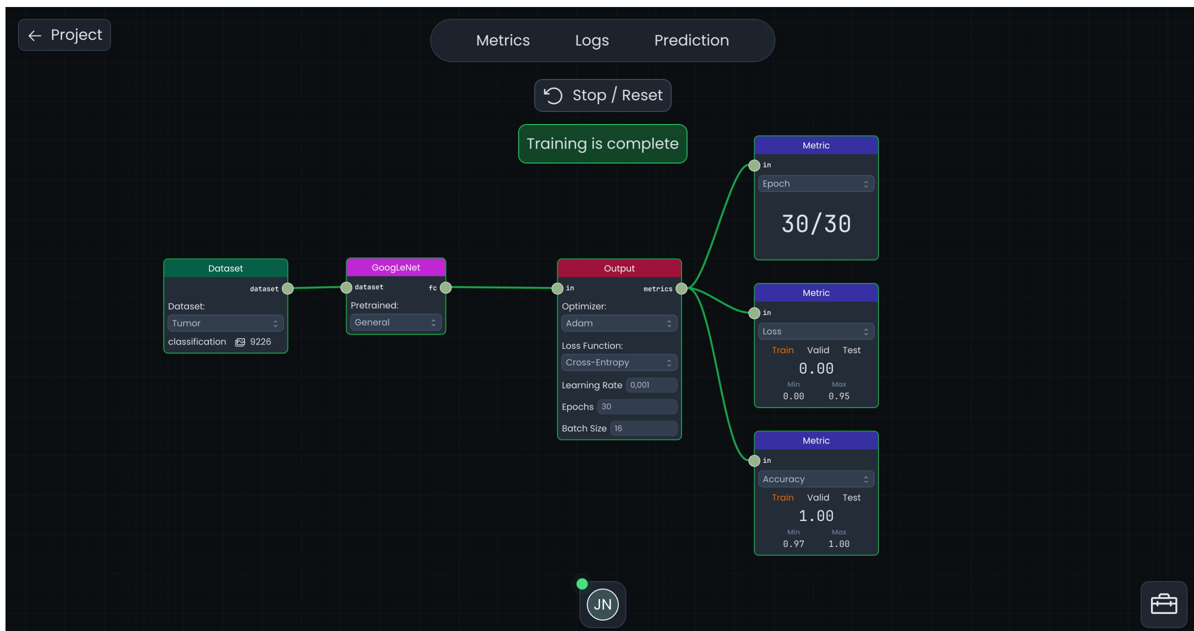


Figure A.5.: The visual programming editor in PathoLearn.

A.3. Dataset Metadata

Table A.2 shows the metadata stored to a dataset. Depending on the dataset type, different metadata is stored. The data is primarily used for pre-processing and training a neural network.

Metadata	Description
`dataset_type`	Indicates the dataset type. Either a `classiciation` or `segmentation` dataset.
`is_grayscale`	Whether the images are grayscale or not.
`dimension`	Stores the width and height of the images.
`class_map`	Maps the human-readable class names, e.g., benign or malignant, to their internal number required for the training process. Additionally, it can be used to map the prediction back to the name. The mask color is stored for segmentation datasets to generate a prediction mask correctly.
`patch_size`	Used by segmentation datasets. Indicates the size of each image path.
`patch_magnification`	Used by segmentation datasets. Indicates with which resize factor the slide was stored and then patched.
`task_ids`	Used by segmentation datasets. Stores the IDs of the PathoLearn tasks used for generating the segmentation dataset.

Table A.2.: The different metadata elements stored for a classification or segmentation dataset.

A.4. Dataset Template Code

Listing 15 displays the template code used for creating the PyTorch code for a classification dataset. Most of the code is reused for all classification tasks. Only the ClearML dataset ID is replaced to ensure the correct dataset is used for training.

```
class ClassificationDataset(Dataset):
    def __init__(self):
        dataset = ClearmlDataset.get(dataset_id="${dataset_id}")
        dataset_path = dataset.get_local_copy()
        class_folder_list = glob.glob(f"{dataset_path}/*")

        metadata = dataset.get_metadata()
        self.class_map = metadata['class_map']
        self.dimension = metadata['dimension']
        self.is_grayscale = metadata['is_grayscale']
        self.y = []
        self.x = []
        for class_path in class_folder_list:
            class_name = class_path.split("/")[-1]

            for img_path in glob.glob(f"{class_path}/*.[jp][pn]g"):
                self.y.append(self.class_map[class_name])
                self.x.append(img_path)

        if self.is_grayscale:
            normalize = A.Normalize((0.1307,), (0.3081,))
        else:
            normalize = A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225))
        resize_dimension = self.dimension
        if resize_dimension["x"] > 256:
            resize_dimension["x"] = 256
        if resize_dimension["y"] > 256:
            resize_dimension["y"] = 256
        self.transform = A.Compose([
                A.Resize(height=resize_dimension["y"], width=resize_dimension["x"]),
                normalize, ToTensorV2(),
            ])

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        img_path = self.x[idx]
        target = self.y[idx]
        image = Image.open(img_path)
        if self.is_grayscale:
            image = image.convert("L")
        else:
            image = image.convert("RGB")
        image = np.array(image)
        if self.transform:
            image = self.transform(image=image)['image']
        return image, torch.tensor(target)
```

Listing 15: PyTorch dataset template used for classification tasks.

A.5. The Dataset Page

The extension offers a page where all created datasets are listed (see Figure A.6). The dataset's type and current processing status are displayed for each listed dataset. On this page, new datasets can be created Clicking one of the right arrows opens the detailed

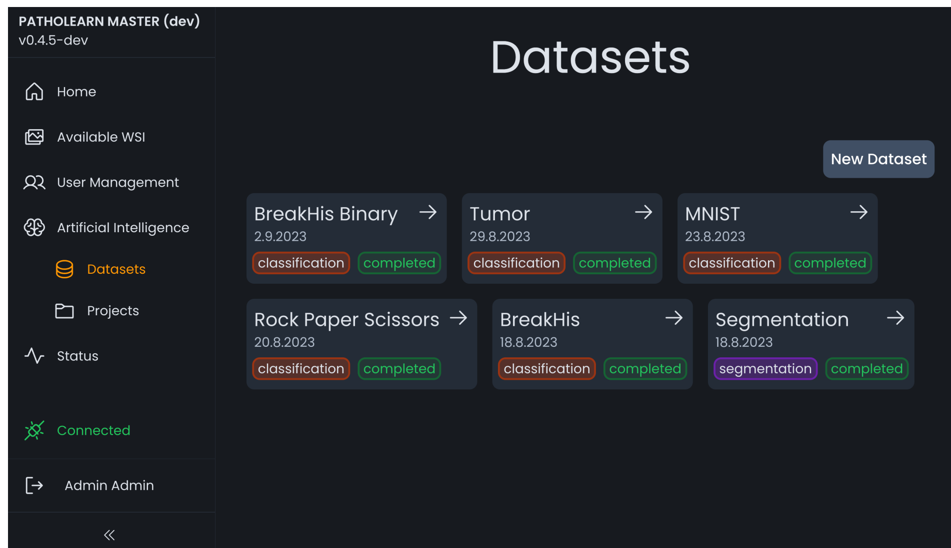


Figure A.6.: The datasets page, where datasets are listed.

dataset page. As displayed in Figure A.7, example images of the dataset are listed. The metadata generated based on Table A.2 is also displayed. The dataset can also be deleted. This is only possible if the dataset is not used by any experiment.

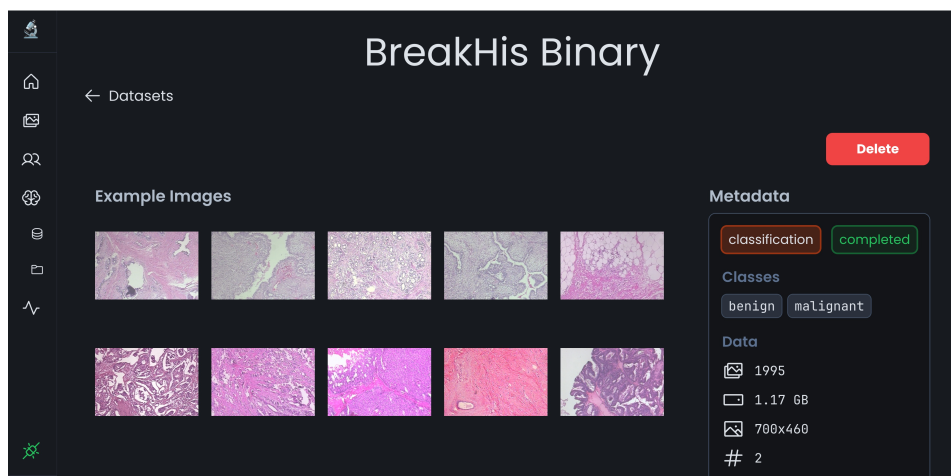


Figure A.7.: The detailed dataset page.

A.6. Lightning Model for Classification Tasks

Listing 16 displays the template code for the Lightning classification model. Besides the learning rate, the metrics selected by the users are injected. The loss function is replaced in the shared step method, accompanied by the selected optimizer in the `configure_optimizers` method.

```
class LightningModel(pl.LightningModule):
    def __init__(self, model):
        super().__init__()

        self.learning_rate = $learning_rate

        self.model = model

        self.save_hyperparameters(ignore=["model"])

$metrics_constructors

    def forward(self, x):
        return self.model(x)

    def _shared_step(self, batch):
        features, true_labels = batch
        logits = self(features)
        loss = ${loss}(logits, true_labels)
        predicted_labels = logits
        return loss, true_labels, predicted_labels

    def training_step(self, batch, batch_idx):
        loss, true_labels, predicted_labels = self._shared_step(batch)
        self.log("train_loss", loss)

        # To account for Dropout behavior during evaluation
        self.model.eval()
        with torch.no_grad():
            _, true_labels, predicted_labels = self._shared_step(batch)
$metrics_train_updates
        self.model.train()
        return loss

    def validation_step(self, batch, batch_idx):
        loss, true_labels, predicted_labels = self._shared_step(batch)
        self.log("valid_loss", loss)
$metrics_valid_updates


    def test_step(self, batch, batch_idx):
        loss, true_labels, predicted_labels = self._shared_step(batch)
        self.log("test_loss", loss)
$metrics_test_updates


    def configure_optimizers(self):
        optimizer = ${optimizer}(self.parameters(), lr=self.learning_rate)
        return optimizer
```

Listing 16: Lightning template for a classification model.

A.7. PyTorch Pooling Layers with Same Padding

The pooling layer in PyTorch does not support same pooling. Therefore, custom layers for maximum and average pooling were implemented that calculate the padding so that the input and output dimensions match [Kuv22].

```
class MaxPool2dSame(torch.nn.MaxPool2d):
    def calc_same_pad(self, i: int, k: int, s: int, d: int) -> int:
        return max((math.ceil(i / s) - 1) * s + (k - 1) * d + 1 - i, 0)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        ih, iw = x.size()[-2:]

        pad_h = self.calc_same_pad(i=ih, k=self.kernel_size[0], s=self.stride[0], d=self.dilation[0])
        pad_w = self.calc_same_pad(i=iw, k=self.kernel_size[1], s=self.stride[1], d=self.dilation[1])

        if pad_h > 0 or pad_w > 0:
            x = torch.nn.functional.pad(x, [pad_w // 2, pad_w - pad_w // 2,
                                            pad_h // 2, pad_h - pad_h // 2])
        return torch.nn.functional.max_pool2d(x, self.kernel_size, self.stride, self.padding,
            self.dilation, self.ceil_mode, self.return_indices)
```

Listing 17: Custom maximum pooling layer that implements same padding.

```
class AvgPool2dSame(torch.nn.AvgPool2d):
    def calc_same_pad(self, i: int, k: int, s: int, d: int) -> int:
        return max((math.ceil(i / s) - 1) * s + (k - 1) * d + 1 - i, 0)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        ih, iw = x.size()[-2:]

        pad_h = self.calc_same_pad(i=ih, k=self.kernel_size[0], s=self.stride[0], d=self.dilation[0])
        pad_w = self.calc_same_pad(i=iw, k=self.kernel_size[1], s=self.stride[1], d=self.dilation[1])

        if pad_h > 0 or pad_w > 0:
            x = torch.nn.functional.pad(x, [pad_w // 2, pad_w - pad_w // 2,
                                            pad_h // 2, pad_h - pad_h // 2])
        return torch.nn.functional.avg_pool2d(x, self.kernel_size, self.stride, self.padding,
            self.ceil_mode, self.count_include_pad, self.divisor_override)
```

Listing 18: Custom average pooling layer that implements same padding.

A.8. PyTorch Layers for Addition and Concatenation

PyTorch has no Add and Concatenate Layer natively built in. As the parsing algorithm uses PyTorch layers as node representation, an *Add* and a *Concatenate* layer were implemented. In both layers, an additional check is implemented that covers the case where the input channels of the first layer of the first input and the first layer of the second input do not match. This is needed to realize a residual block, as the node from which a branch is created is used as an input for the Add node. Depending on the previous node, the input channel size can differ from the one used in the branch's first node.

```
class Add(torch.nn.Module):
    def __init__(self, *modules):
        super().__init__()
        self.sum_modules = torch.nn.ModuleList(modules)

    def forward(self, x):
        first_in = self.sum_modules[0][0].in_channels
        second_in = self.sum_modules[1][0].in_channels

        if first_in != second_in:
            out = self.sum_modules[0](x)
            module_sum = out + sum(module(out) for module in self.sum_modules[1:])
            return module_sum
        return sum(module(x) for module in self.sum_modules)
```

Listing 19: The *Add* layer that adds the output values of multiple layers together.

```
class Concatenate(torch.nn.Module):
    def __init__(self, *modules) -> None:
        super().__init__()
        self.concate_modules = torch.nn.ModuleList(modules)

    def forward(self, x):
        first_in = self.concate_modules[0][0].in_channels
        second_in = self.concate_modules[1][0].in_channels

        if first_in != second_in:
            out = self.concate_modules[0](x)
            outputs = [out]
            outputs += [module(out) for module in self.concate_modules[1:]]
            return torch.cat(outputs, dim=1)
        return torch.cat([module(x) for module in self.concate_modules], dim=1)
```

Listing 20: The *Concatenate* layer concatenates the output values of multiple layers together.

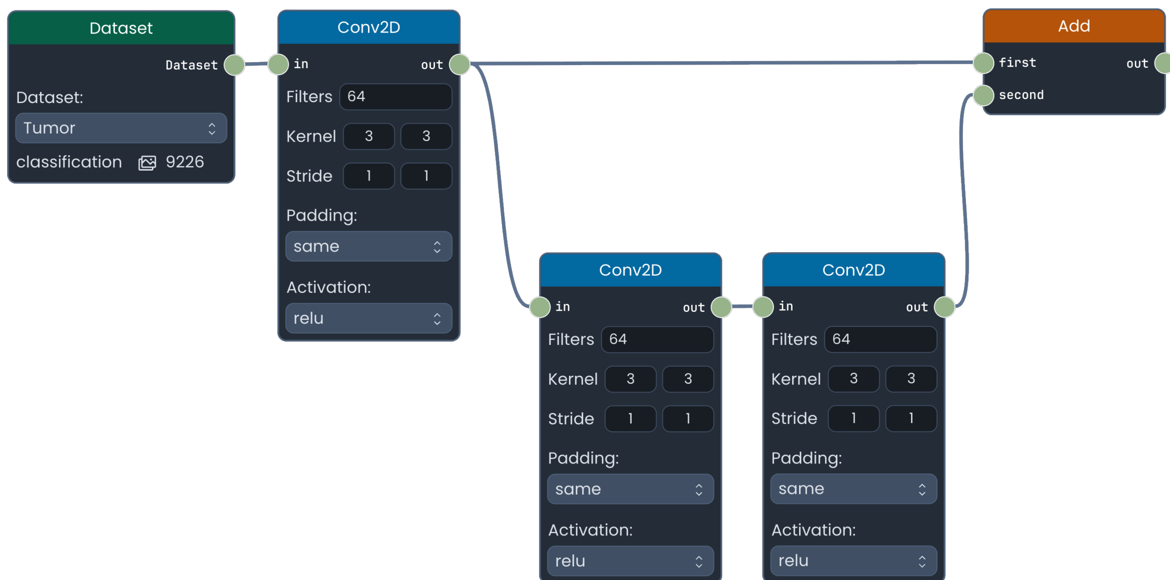
A.9. The Inception Module Realized in PathoLearn

Figure A.8 illustrates an implementation of the inception module in PathoLearn. The *Concatenate* Node only accepts two inputs. Therefore, multiple of those are required.

Figure A.8.: The inception module realized in PathoLearn.

A.10. The Residual Block Realized in PathoLearn

Figure A.9 illustrates a simple residual block realized in PathoLearn and the resulting PyTorch layer code.




```
self.model = torch.nn.Sequential(
    Add(
        torch.nn.Sequential(
            torch.nn.Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding="same"),
            torch.nn.ReLU(),
        ),
        torch.nn.Sequential(
            torch.nn.Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding="same"),
            torch.nn.ReLU(),
            torch.nn.Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding="same"),
            torch.nn.ReLU(),
        ),
    ),
)
```

Figure A.9.: A simple residual block realized in PathoLearn (top) and the resulting PyTorch model (below).

A.11. Evaluating the Best Neural Network Model Serving Format

Section 6.5 introduced two formats for serving trained NNs: TorchScript and ONNX. A load test was conducted to determine the best-fitting format in combination with ClearML Serving and Triton. To evaluate the performance, two datasets of different complexity were used. Additionally, a NN with few layers was used to train the less complex dataset and a NN with many layers for the more complex dataset.

A.11.1. Training Environment

The ClearML Serving instance uses Triton with GPU support. Both run inside Docker containers, including additional containers for request management and metric collection. All containers run on a single server. Table A.3 shows the server's hardware and software specifications. For training, a ClearML Agent was utilized.

OS	Ubuntu Server 20.04 LTS
CPU	AMD Ryzen9 5900x, 12 cores, 24 threads, up to 4.80 GHz
Memory	64GB DDR4, 3200 MHz
GPU	Nvidia RTX 2070 Super 8GB
Storage	1 TB NVME PCIE 3.0 SSD (OS and software installation) 2 TB SATA SSD (Data storage)
Docker	24.0.5
ClearML Server	1.11.0
ClearML Agent	1.5.2
ClearML Serving	1.3.0
Python	3.10
PyTorch	2.0.1+cu118
Lightning	2.0.7

Table A.3.: Hardware and software specification of the server.

A.11.2. Datasets

The MNIST dataset contains images of handwritten numbers from 0 to 9 [Den12]. Each grayscale image is 28×28 pixels. It contains 68992 images, split into 70% training (48294), 10% validation (6899), and 20% test (13799) images. The Breast Cancer

Histopathological Database (BreakHis) [SOPH16] dataset is more complex, as it contains RGB images of 700×460 pixels. The 1994 images are also split into 70% training (1397), 10% validation (199), and 20% test (399) images. It contains eight classes (adenosis, ductal carcinoma, fibroadenoma, lobular carcinoma, mucinous carcinoma, papillary carcinoma, phyllodes tumor, and tubular adenoma).

A.11.3. Neural Network Architecture and Training Configuration

Listing 21 shows the NN architecture used for training the MNIST dataset. A simple combination of convolutional and pooling layers was used. A single linear layer maps the features to the ten classes.

```
torch.nn.Sequential(
    torch.nn.Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1)),
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False),
    torch.nn.ReLU(),
    torch.nn.Conv2d(16, 16, kernel_size=(5, 5), stride=(1, 1)),
    torch.nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False,),
    torch.nn.Flatten(start_dim=1, end_dim=-1),
    torch.nn.Linear(in_features=256, out_features=10, bias=True),
)
```

Listing 21: Neural Network configuration for the MNIST dataset.

The BreakHis dataset was trained with a ResNet-50 architecture, as displayed in Listing 22. It used pre-trained weights of the ImageNet dataset, and the feature collector was overridden to classify the eight classes. Both used the Cross-Entropy loss function in combination with the Adam optimizer and a learning rate of 0.001. The BreakHis NN architecture was trained for 30 epochs with a batch size of 32. As the MNIST NN architecture is simpler and the images smaller, the training was only run for 15 epochs with a batch size of 128. During the training, the model with the lowest validation loss was saved and, after the training, converted into the TorchScript and ONNX format. Both converted models were uploaded to the ClearML server, and new ClearML serving endpoints were created for four models.

```
self.model = torchvision.models.get_model(name="resnet50", weights="DEFAULT")
self.model.fc = torch.nn.Sequential(torch.nn.Linear(in_features=2048, out_features=8, bias=True))
```

Listing 22: Neural Network configuration for the BreakHis dataset.

A.11.4. Metric Gathering

To evaluate the performance, a load test was performed. Grafana k6 is an open source load testing tool that tests REST-APIs with single-user or multi-user interaction [K6D23]. Listing 24 shows the script used to load test the different ClearML serving endpoints. The `DATASET` and `FORMAT` constants configure the dataset and format to use. The `options` object contains the dataset metadata and the model IDs for the different formats.

A single-user and a multi-user test is performed for each dataset and format. Listing 23 shows the command for running a single-user test for 100 iterations on the BreakHis dataset and the ONNX format. The `--vus` option defines the number of virtual users to create, and the `--iterations` options the number of iterations the script will be executed. The multi-user test uses 10 virtual users and 1000 iterations. If equally distributed between the virtual users, each should perform 100 requests.

```
k6 run --out csv=breakhis_onnx_1_100.csv --vus 1 --iterations 100 script.js
```

Listing 23: Command for running a single-user test for 100 iterations on the BreakHis dataset with the ONNX format. The results are stored in a CSV file.

Grafan k6 gathers a variety of different metrics [Bui23]. As only the performance should be evaluated, only those metrics concerning the HTTP request speed are of interest. The `http_req_waiting` metric represents the time the client waits for a response from the server. It measures the time the server needs to process the request and return the predictions. Therefore, it is the most fitting metric for evaluating the performance of serving endpoints with different formats, as it excludes factors like the time needed to establish the request and the time needed to upload the image.

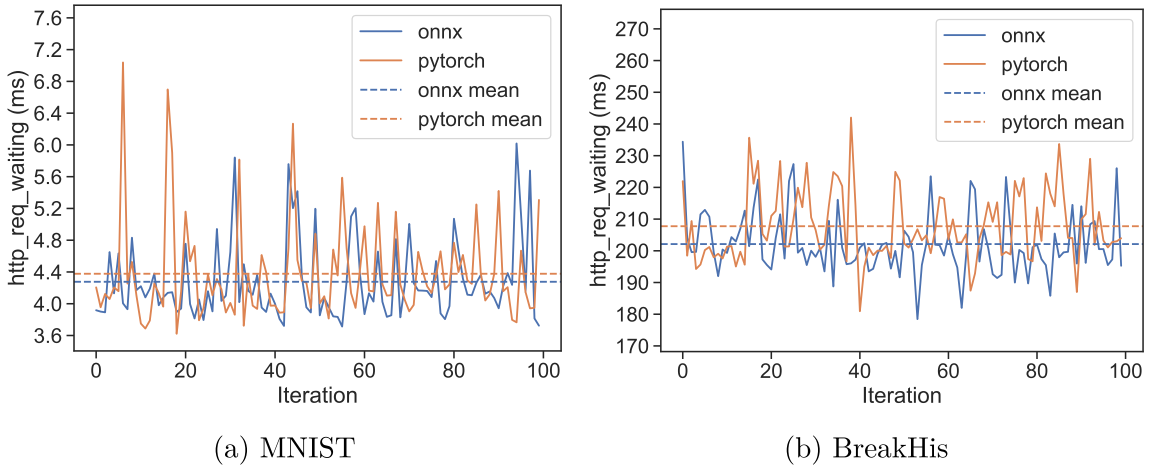
Besides response duration, the numerical error is determined. Five images of each dataset are used to get predictions for the PyTorch and ONNX format. Afterward, the difference between each class probability represents the prediction error. The minimum, maximum, and average error is determined for each test image.

```
import http from 'k6/http';
import { sleep } from 'k6';
import { check } from 'k6';

const DATASET = 'mnist';
const FORMAT = 'pytorch';

const options = {
  mnist: {
    image: './mnist.bin',
    metadata: {
      class_map: { 0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9 },
      dimension: { x: 28, y: 28 },
      is_grayscale: true,
    },
    pytorch: '97460cea38db4c868eaf1ae899a21b17',
    onnx: '4764941541f545829b4e0f096065b5b9',
  },
  breakhis: {
    image: './breakhis.bin',
    metadata: {
      class_map: {
        adenosis: 0,
        ductal_carcinoma: 1,
        fibroadenoma: 2,
        lobular_carcinoma: 3,
        mucinous_carcinoma: 4,
        papillary_carcinoma: 5,
        phyllodes_tumor: 6,
        tubular_adenoma: 7,
      },
      dimension: { x: 700, y: 460 },
      is_grayscale: false,
    },
    pytorch: '64efc65890b8493fb896ef38985ee47e',
    onnx: 'ed197849ec134041925a5c802b821950',
  },
};
const image = open(options[DATASET].image);

export default function () {
  const modelId = options[DATASET][FORMAT];
  const url = `http://xxx.xxx.xxx.xxx:8080/serve/${modelId}`;
  const payload = JSON.stringify({
    image: image,
    metadata: options[DATASET].metadata,
  });
  const params = {
    headers: {
      'Content-Type': 'application/json',
    },
  };
  const res = http.post(url, payload, params);
  console.log(res.json()['probabilities'][0]);
  check(res, {
    'is status 200': (r) => r.status === 200,
    'has probabilities': (r) =>
      r.json()['probabilities'][0].length ===
        Object.keys(options[DATASET].metadata.class_map).length,
  });
  sleep(1);
}
```

Listing 24: Script for load testing the ClearML Serving REST-endpoint with Grafana k6.

A.11.5. Results

Figure A.10 and Figure A.11 show the results for the single-user and multi-user tests respectively. Generally, the MNIST shows on both tests, as expected, a much smaller request duration than the BreakHis dataset. Additionally, the durations fluctuate between the iterations, especially at the multi-user test. Comparing the average durations between the PyTorch and ONNX formats, it can be seen that both formats perform similarly. In the single-user test, the ONNX format is slightly faster and slightly slower in the multi-user test.



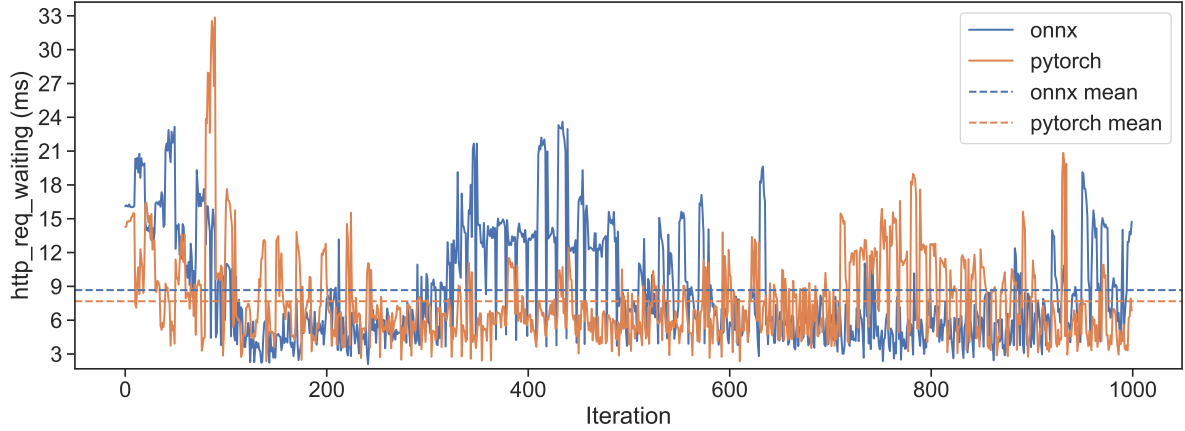
(a) MNIST (b) BreakHis

Figure A.10.: Results of the single-user test for 100 iterations.

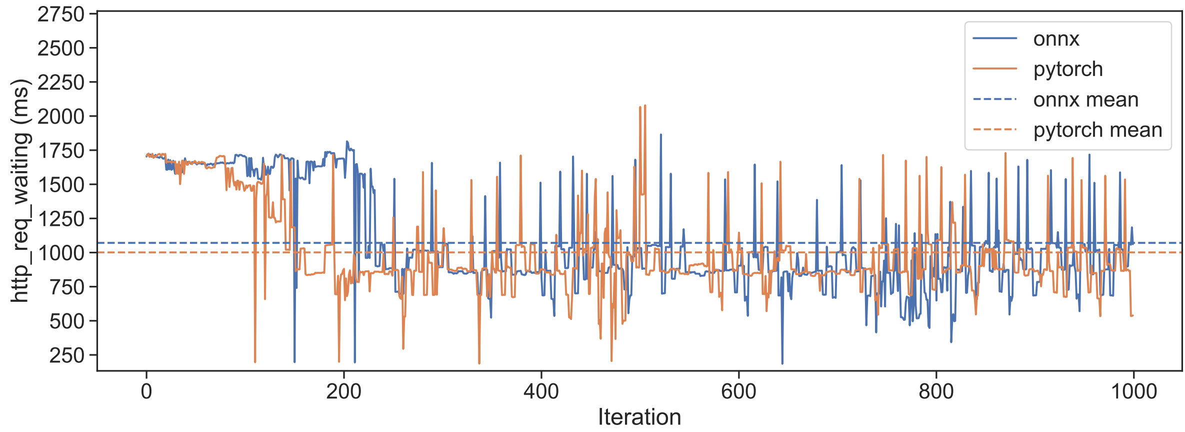
Table A.4 and Table A.5 displays the gathered minimum, maximum, and average error. On both datasets, the ONNX predictions differ slightly from the PyTorch predictions. For every test image, a prediction error can be observed. The largest absolute error for the MNIST dataset was 4.768372×10^{-7} and for the BreakHis dataset 2.048910×10^{-8}. This numerical error is so small that it does not influence the final classification result.

image	min	max	mean
1	$-5.820766 \times 10^{-10}$	7.372575×10^{-18}	$-7.435819 \times 10^{-11}$
2	0.000000	3.126388×10^{-13}	4.238145×10^{-14}
3	0.000000	2.728484×10^{-12}	2.822677×10^{-13}
4	0.000000	1.746230×10^{-8}	1.753643×10^{-9}
5	-4.768372×10^{-7}	4.768372×10^{-7}	4.195565×10^{-12}

Table A.4.: Prediction difference between PyTorch and ONNX models of the MNIST dataset. The differences between the ten class probabilities were used to calculate the maximum, minimum, and mean for every image.



(a) MNIST



(b) BreakHis

Figure A.11.: Results of the multi-user test with 10 virtual users and for 1000 shared iterations.

image	min	max	mean
1	0.000000	6.217249×10^{-14}	8.654806×10^{-15}
2	-1.004082×10^{-9}	4.092726×10^{-12}	$-1.253233 \times 10^{-10}$
3	$-9.822543 \times 10^{-11}$	2.048910×10^{-8}	2.913836×10^{-9}
4	$-1.818989 \times 10^{-11}$	0.000000	$-2.461401 \times 10^{-12}$
5	$-2.473826 \times 10^{-10}$	3.637979×10^{-10}	$-1.067235 \times 10^{-11}$

Table A.5.: Prediction difference between PyTorch and ONNX models of the BreakHis dataset. The differences between the ten class probabilities were used to calculate the maximum, minimum, and mean for every image.

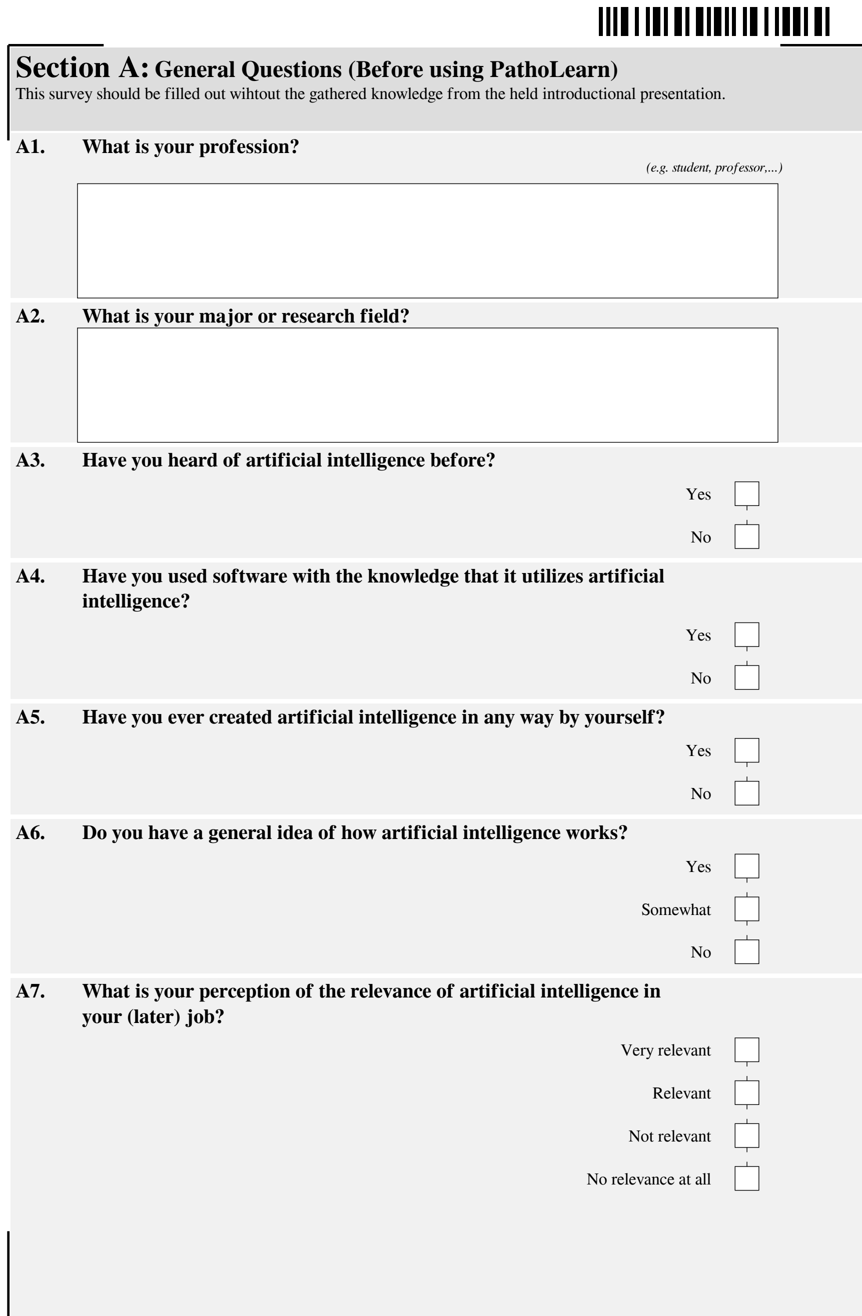
A.12. Examplatory Pre- and Postprocessing script

```
from typing import Any

import numpy as np
from PIL import Image
import torch
from io import BytesIO
import base64
import albumentations as A
from albumentations.pytorch import ToTensorV2


class Preprocess(object):
    def __init__(self):
        pass

    def preprocess(self, body: dict, state: dict, collect_custom_statistics_fn=None) -> Any:
        image = Image.open(BytesIO(base64.b64decode(body["image"]))).convert("RGB")
        dataset_metadata = body["metadata"]
        dimensions = dataset_metadata["dimension"]
        image_numpy = np.array(image)
        if dimensions["x"] > 256:
            dimensions["x"] = 256
        if dimensions["y"] > 256:
            dimensions["y"] = 256
        transform = A.Compose(
            [
                A.Resize(height=dimensions["y"], width=dimensions["x"]),
                A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
                ToTensorV2(),
            ]
        )
        data = transform(image=np.array(image_numpy))["image"]
        return np.array([data.numpy()]).astype(np.float32)

    def postprocess(self, data: Any, state: dict, collect_custom_statistics_fn=None) -> dict:
        if not isinstance(data, np.ndarray):
            return dict(digit=-1)
        data = torch.tensor(data)
        propabilities = torch.nn.functional.softmax(data, dim=1)
        return dict(propabilities=propabilities.tolist())
```

Listing 25: Examplatory script that executes pre- and postprocessing steps before the prediction and after for a classification model.

A.13. Surveys

A.13.1. Survey before Using Patholearn



Section A: General Questions (Before using PathoLearn)
This survey should be filled out wihtout the gathered knowledge from the held introductional presentation.

A1. What is your profession?
(e.g. student, professor,...)

A2. What is your major or research field?

A3. Have you heard of artificial intelligence before?
Yes ☐
No ☐

A4. Have you used software with the knowledge that it utilizes artificial intelligence?
Yes ☐
No ☐

A5. Have you ever created artificial intelligence in any way by yourself?
Yes ☐
No ☐

A6. Do you have a general idea of how artificial intelligence works?
Yes ☐
Somewhat ☐
No ☐

A7. What is your perception of the relevance of artificial intelligence in your (later) job?
Very relevant ☐
Relevant ☐
Not relevant ☐
No relevance at all ☐

A8. **What is your perception of the relevance of artificial intelligence in your current studies or research projects?**

- Very relevant ☐
- Relevant ☐
- Not relevant ☐
- No relevance at all ☐

A9. **Do you think the relevance of artificial intelligence will increase in the future?**

- Yes ☐
- Yes, but not much ☐
- No, not really ☐
- No relevance at all ☐

A.13.2. Survey after Using Patholearn

Section A: General Questions (After using PathoLearn)

This survey should be filled out after using PathoLearn.

A1. Did your understanding of how artificial intelligence works improve?

(e.g. student, professor,...)

- Yes ☐
- Slightly ☐
- No, why ☐

A2. Could you see the tool being used for teaching artificial intelligence?

- Yes ☐
- Undecided, why? ☐
- No, why? ☐

A3. Could you see the tool being used to research artificial intelligence in the medical research field or your research project?

- Yes ☐
- Undecided, why? ☐
- No, why? ☐

A4. **How easy was PathoLearn to understand for you?**

- Easy ☐
- Medium ☐
- Hard ☐

A5. **On a scale from 1-5, how did you enjoy using PathoLearn (5 being the best)?**

- 1 ☐
- 2 ☐
- 3 ☐
- 4 ☐
- 5 ☐

A6. **On a scale of 1-5, how did you like the possibility of working with colleagues simultaneously (5 being the best)?**

- 1 ☐
- 2 ☐
- 3 ☐
- 4 ☐
- 5 ☐

A7. **On a scale from 1-5, how did you like the overall look of the UI (5 being the best)?**

- 1 ☐
- 2 ☐
- 3 ☐
- 4 ☐
- 5 ☐

A8. **On a scale from 1-5, how did you like the presented way of testing the trained artificial intelligence models (Prediction)**

1 ☐
2 ☐
3 ☐
4 ☐
5 ☐

A9. **What kind of functionallity did you find missing or wished to be included (an explanation would be great)?**

A10. **Did you encounter any problems?**

A11. **Do you have any final feedback?**

A.13.3. Survey Answers before Using PathoLearn

A1.	**What is your profession?**
User	Answer
1	*Imaging analyst*
2	*PhD Student*
3	*Professor*
4	*medical student*
5	*PostDoc*
A2.	**What is your major or research field?**
User	Answer
1	*Digital pathology*
2	*Immunology, Oncology*
3	*Oncoimmunology*
4	*neuropathology*
5	*Biology*
A3.	**Have you heard of artificial intelligence before?**
User	Answer
1	Yes
2	Yes
3	Yes
4	Yes
5	Yes
A4.	**Have you used software with the knowledge that it utilizes artificial intelligence?**
User	Answer
1	Yes
2	Yes
3	Yes
4	Yes
5	Yes
A5.	**Have you ever created artificial intelligence in any way by yourself?**
User	Answer
1	Yes
2	No
3	No
4	No
5	Yes

A6.	**Do you have a general idea of how artificial intelligence works?**
User	Answer
1	Yes
2	Yes
3	Yes
4	Yes
5	Somewhat
A7.	**What is your perception of the relevance of artificial intelligence in your (later) job?**
User	Answer
1	Very relevant
2	Relevant
3	Very relevant
4	Relevant
5	Relevant
A8.	**What is your perception of the relevance of artificial intelligence in your current studies or research projects?**
User	Answer
1	Very relevant
2	Very relevant
3	Very relevant
4	Very relevant
5	Relevant
A9.	**Do you think the relevance of artificial intelligence will increase in the future?**
User	Answer
1	Yes
2	Yes
3	Yes
4	Yes
5	Yes, but not much

Table A.6.: Answers of users to the first survey.

A.13.4. Survey Answers after Using PathoLearn

A1.	**Did your understanding of how artificial intelligence works improve?**
User	Answer
1	Yes *(No comment)*
2	Yes *(No comment)*
3	Yes *(No comment)*
4	Slightly *(No comment)*
A2.	**Could you see the tool being used for teaching artificial intelligence?**
User	Answer
1	Yes *(No comment)*
2	Yes *(No comment)*
3	Yes *(No comment)*
4	Yes *(No comment)*
A3.	**Could you see the tool being used to research artificial intelligence in the medical research field or your research project?**
User	Answer
1	Undecided, why? *(No comment)*
2	Yes *(No comment)*
3	Yes *(No comment)*
4	Undecided, why? *(No comment)*

A4.	How easy was PathoLearn to understand for you?
User	Answer
1	Medium *Die grundsätzliche Oberfläche ist sehr intuitiv (erinnert mich etwas an Scratch). Bei den einzelnen Knoten wäre es aber hilfreich, wenn man noch eine Infobox hätte, was dieser Schritt prinzipiell bewirkt, sonst kommt man mit wenig Hintergrundwissen und einfach mal ausprobieren nicht besonders weit.*
2	Medium *(No comment)*
3	Hard *(No comment)*
4	Easy *(No comment)*
A5.	**On a scale from 1-5, how did you enjoy using PathoLearn (5 being the best)?**
User	Answer
1	4
2	4
3	5
4	4
A6.	**On a scale of 1-5, how did you like the possibility of working with colleagues simultaneously (5 being the best)?**
User	Answer
1	5
2	5
3	3
4	3
A7.	**On a scale from 1-5, how did you like the overall look of the UI (5 being the best)?**
User	Answer
1	5
2	5
3	4
4	5

A8.	**On a scale from 1-5, how did you like the presented way of testing the trained artificial intelligence models (Prediction)?**
User	Answer
1	5
2	4
3	5
4	4

A9.	**What kind of functionality did you find missing or wished to be included (an explanation would be great)?**
User	Answer
1	*1. kleine ausklappbaren Infoboxen mit Hintergrundinfos zu den einzelnen Tools 2. Option, das man direkt von Knoten zu Code und andersherum springen kann und so auch lernen kann, was dahinter steckt 3. Chatfunktion, um gemeinsames Arbeiten noch effektiver zu machen*
2	*- I would like to evaluate the performance of neural networks in tasks like cell segmentation and classification. - I found missing some suggestions of which networks and which parameters are the most suitable for the task I chose*
3	*A vignette explaining main functions*
4	*For prediction part, it would be more helpful if I could add more than one picture at once, and it would be better if there is also a table next to the results showing the predictions directly.*

A10.	**Did you encounter any problems?**
User	Answer
1	*- "training failed" tritt manchmal auf und ohne Hintergrundwissen hat man gar keine Idee, wo man etwas ändern muss. Hier wäre die Benennung häufiger Ursachen hilfreich. Nachdem das Training fehlgeschlagen ist, konnte ich keine Bearbeitung an dem Projekt vornehmen, um es überarbeitet erneut zu starten. - ansonsten sind mir keine Probleme aufgefallen (sogar die touch-Funktion meines Laptops funktioniert ganz normal)*
2	*no*
3	*Only problems related to the fact that it's a difficult subject/task. It requires training*
4	*No*

A11. Do you have any final feedback?	
User	Answer
1	*Sehr schöne Oberfläche. Es macht Spaß unterschiedliche Einstellungen auszuprobieren und zu schauen, was passiert. Die Möglichkeit bei Prediction eigene Bilder hochzuladen gefällt mir sehr gut (und besonders der Versuch absichtlich schwierige Bilder zu erstellen und zu schauen, wann/ob die AI scheitert). Allerdings kann auch schnell Frust aufkommen, wenn man etwas länger ausprobiert, es nicht funktioniert und man nicht weiß, woran es liegen könnte. Hier wären zusätzliche Hintergrundinfos nötig, die aber natürlich auch mittels Kontakt zu einem Dozierenden vermittelt werden könnten.*
2	*I found PathoLearn a very interesting and innovative tool, suitable for both teaching and research. Moreover, the interface was very user-friendly.*
3	*No*
4	*It is good in general and enough for using in teaching, but I think it needs some improvements if you want to use it in actual research.*

Table A.7.: User responses to the second survey.

A.14. Evaluating the Visual Programming Editor Real-Time Performance

As stated in the non-function requirements (see Section 5.4), the changes made in the visual programming editor should be synchronized to all other users in up to 100 ms. To evaluate if the current implementation fulfills this, a test is conducted.

A.14.1. Environment

The software architecture uses a custom WebSocket server, which processes the incoming WebSocket events and distributes them to the specified channels (see Section 6.1). Thereby, the network connection to and from the server is deciding for the latency of the visual programming synchronization. As mentioned in Section 7.1, a proxy server is required to make PathoLearn accessible from outside of the private network. The test was conducted inside the private network to overcome the increased latency caused by the limited upload speed and the additional request over the proxy server.

A single server hosts PathoLearn. Five different devices (clients) were used, each simulating a user connected to the same visual programming editor. The devices are directly connected to the router via ethernet or Wi-Fi. Therefore, they can have up to gigabit speed to and from the server. Each device's connection speed was throttled to simulate a real-world scenario, where the distance between the server and a client requires multiple hops. The throttling feature of the Chrome browser was utilized [Thr23] and set to the average connection speed in Germany from February 2023 (download: approx. 83 Mbit/s, upload: approx. 28 Mbit/s) [Ave23].

A.14.2. Procedure

The clients were connected to the same visual programming editor. To evaluate the performance, a client can act as a sender, which sends WebSocket events. Table A.8 lists the 10 different events used. A sender sends these events to the WebSocket server every second, distributing them to the other clients. As these events are sent nearly simultaneously, it is simulated that multiple users concurrently change elements in the visual programming editor.

The latency of the events is measured by tracking the Round-Trip Time (RTT) from the sender to the client and back to the sender. If the sender sends an event, a timestamp is sent with that event. After the client receives an event, it sends a specific response event back to the sender. The sender receives the response and calculates the time difference

between the stored timestamp and the current time. As the RTT is calculated, the difference is halved.

The events were sent every second for 200 iterations to simulate a load over time and accompany external factors, like random network speed degradation and client load spikes.

Event	Bytes	Description
`client-node-locked`	278	Indicating to other users that the sender selected that node and is now locked.
`client-node-unlock`	223	Indicating to other users that the sender unselected that node and is now unlocked.
`client-control-lock`	225	Indicating to other users that the sender selected a control element of a node and is now locked.
`client-control-unlock`	230	Indicating to other users that the sender unselected a control element of a node and is now unlocked.
`client-control-changed`	281	Indicating to other users that the sender changed the value of a control element.
`client-node-dragged`	402	Indicating to other users that the sender changed a node position.
`client-node-created`	1503	Indicating to other users that the sender created a new node.
`client-node-removed`	221	Indicating to other users that the sender removed a node.
`client-connection-created`	336	Indicating to other users that the sender created a new connection between two nodes.
`client-connection-removed`	233	Indicating to other users that the sender removed a connection between two nodes.

Table A.8.: List of used WebSocket events used to synchronize the visual programming editor between users.

A.14.3. Results

In the first test, a single sender was used. Considering the 200 iterations and the 10 events per iteration, each client received 2000 events. Figure A.12 illustrates the box plots of each client for each event. Generally, it can be seen that all clients have, for all events, a median duration considerably smaller than 100 ms. Considering each client individually, the median event durations are generally close together. Comparing the clients, a larger difference in the results can be observed. `client_5` shows a large spread in the data, ranging from values below 10 ms to values larger than 200 ms. This can also be seen in the large interquartile range (IQR). On the other hand, `client_4` has a very small data spread. Excluding the outliers, it did not show durations larger than 40 ms. The whiskers of the clients, which were calculated as 1.5 times the IQR, showed that all clients could achieve durations lower than 20 ms. Additionally, it can be seen that the relative duration of each event is comparable between clients, e.g., the `client-node-locked` has everywhere the smallest median, and the `client-connection-removed` has the largest median. Some outliers of events show nearly the same duration. This is due to the simultaneous sending of the different events and a current network degradation or system load spike.

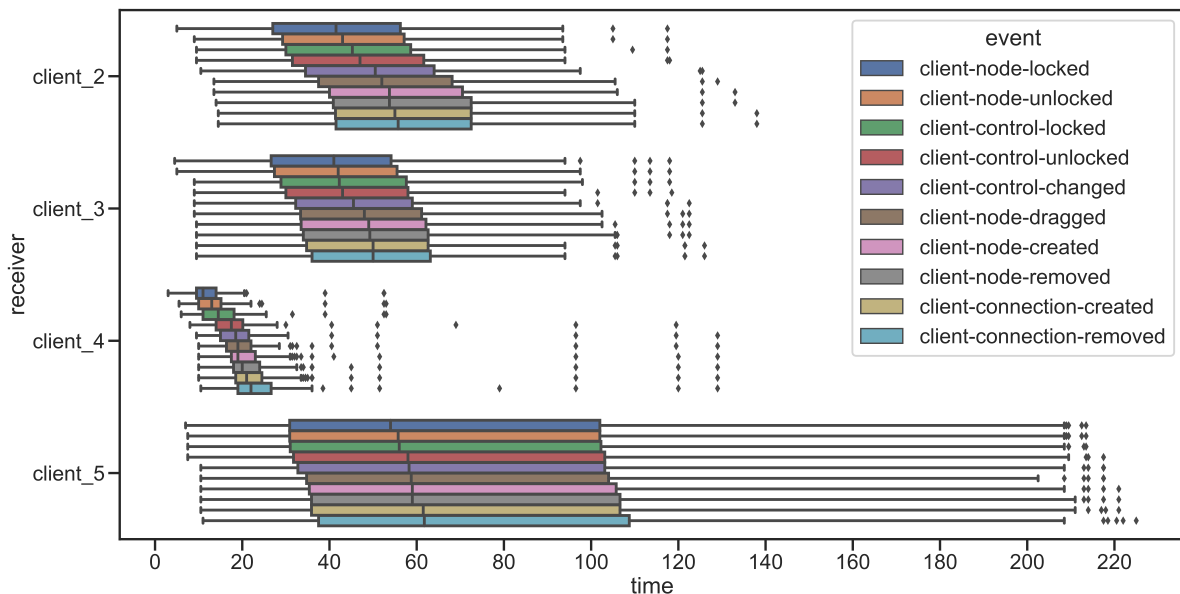


Figure A.12.: The box plots of each event duration for each client. The `client_1` was used as a sender. The whiskers extend to 1.5 times the interquartile range.

The second test used `client_1` and `client_2` as senders. Therefore, the remaining clients receive 4,000 events each, and the sender clients receive 2,000 from the other sender. Figure A.13 displays the box plots of the different clients. A similar result as

the single sender test can be seen. The high data spread of `client_5` is also present. The increased event amount also produced more outliers. `client_4` again has a small data spread and comparable outlier values. Also, `client_3` shows comparable results. On the other hand, `client_2` does not show any outliers, and the IQR is smaller. Lastly, `client_1` shows the smallest data spread. The outliers are close to the whiskers, which are all under 50 ms.

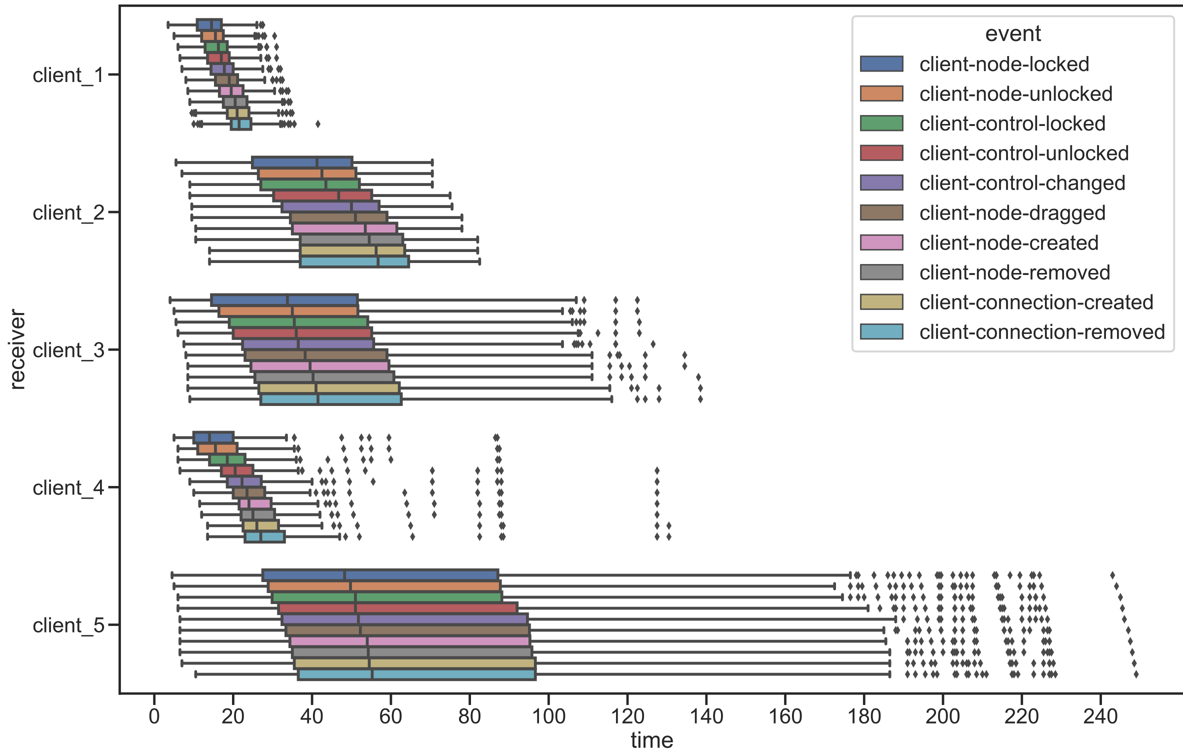


Figure A.13.: The box plots of each event duration for each client. The `client_1` and `client_2` were used as senders. Both senders also received the other sender events. For the other clients, double the data could be collected due to the two senders. The whiskers extend to 1.5 times the interquartile range.

Additionally, the average durations of the events per client were calculated to determine whether it can be generally said that events reach clients in under 100 ms. This is illustrated in A.14. The results are based on the test with two senders. Generally, the average durations show comparable results to the box plots. It can be especially seen that all clients have an average duration below 75 ms. Additionally, the hierarchical event duration between clients is even more noticeable. As outliers greatly influence the average calculation, those resulting from the box plots calculation in A.13 were removed, and the updated average durations are displayed in A.15. All clients only show slight improvements, except `client_5`, where it improved by up to 10 ms.

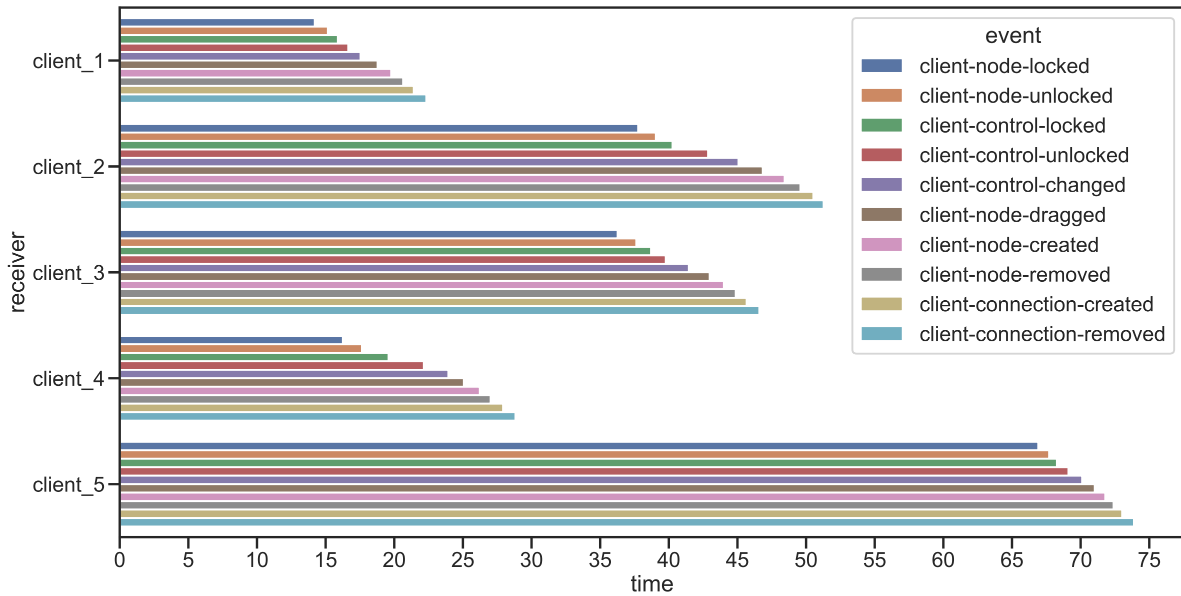


Figure A.14.: The average duration of each event for each client. The `client_1` and `client_2` were used as senders. Both senders also received the other sender events. For the other clients, double the data could be collected due to the two senders.

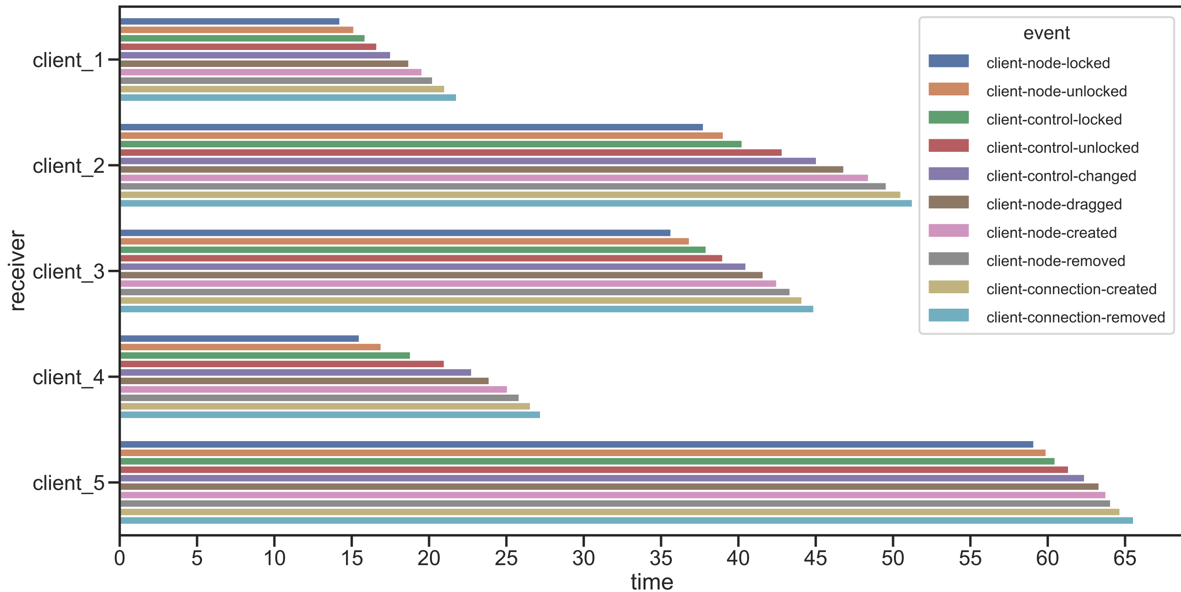


Figure A.15.: The average duration of each event for each client. The `client_1` and `client_2` were used as senders. Both senders also received the other sender events. For the other clients, double the data could be collected due to the two senders. Outliers were removed.

Looking at the average durations, it can be confirmed that these are under the required 100 ms. Whether this is valid for all clients can not be said, as the results seem to depend on the used client highly. The used software (e.g., operating system, browser version or network driver version), as well as hardware components (e.g., network card or WiFi chip), could influence the clients' response times. This is supported by the fact that `client_3` and `client_4` had the slowest hardware components, and `client_1` and `client_4` the fastest. Additionally, the RTT is only an approximation of the duration. Due to the acknowledgment event, the duration is always influenced by the time required until the sender receives it. This can either reduce or increase the measured duration for the initial event. While the network connection between the clients is throttled, it does not simulate a real-world scenario where clients need to route through potential multiple network devices. Finally, it can be said that the WebSocket server of PathoLearn is capable of handling multiple events at once without performance degradation. Event durations larger than 100 ms seem to be caused by the client side, not the server side. To prove this, an additional test is required to determine the actual influence of different clients and realistic network connections.

Bibliography

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

[AAE16] Nuha Alshuqayran, Nour Ali, and Roger Evans. A Systematic Mapping Study in Microservice Architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51, November 2016. `doi:10.1109/SOCA.2016.15`.

[AAH+22] Mohamed Amgad, Lamees A. Atteya, Hagar Hussein, Kareem Hosny Mohammed, Ehab Hafiz, Maha A. T. Elsebaie, Ahmed M. Alhusseiny, Mohamed Atef AlMoslemany, Abdelmagid M. Elmatboly, Philip A. Pappalardo, Rokia Adel Sakr, Pooya Mobadersany, Ahmad Rachid, Anas M. Saad, Ahmad M. Alkashash, Inas A. Ruhban, Anas Alrefai, Nada M. Elgazar, Ali Abdulkarim, Abo-Alela Farag, Amira Etman, Ahmed G. Elsaeed, Yahya Alagha, Yomna A. Amer, Ahmed M. Raslan, Menatalla K. Nadim, Mai A. T. Elsebaie, Ahmed Ayad, Liza E. Hanna, Ahmed Gadallah, Mohamed Elkady, Bradley Drumheller, David Jaye, David Manthey, David A. Gutman, Habiba Elfandy, and Lee A. D. Cooper. NuCLS: A scalable crowdsourcing, deep learning approach and dataset for nucleus classification, localization and segmentation. *GigaScience*, 11:giac037, May 2022. URL: `http://arxiv.org/abs/2102.09099`, `arXiv:2102.09099`, `doi:10.1093/gigascience/giac037`.

[AAK17] Rainer Alt, Gunnar Auth, and Christoph Kögler. Innovationsorientiertes IT-Management mit DevOps. In Rainer Alt, Gunnar Auth, and

Christoph Kögler, editors, *Innovationsorientiertes IT-Management mit DevOps: IT im Zeitalter von Digitalisierung und Software-defined Business*, essentials, pages 21–32. Springer Fachmedien, Wiesbaden, 2017. URL: https://doi.org/10.1007/978-3-658-18704-0_3, doi:10.1007/978-3-658-18704-0_3.

[AAS20] Arohan Ajit, Koustav Acharya, and Abhishek Samanta. A Review of Convolutional Neural Networks. In *2020 International Conference on Emerging Trends in Information Technology and Engineering (Ic-ETITE)*, pages 1–5, 2020. doi:10.1109/ic-ETITE47903.2020.049.

[ACG+09] Frederico A. C. Azevedo, Ludmila R. B. Carvalho, Lea T. Grinberg, José Marcelo Farfel, Renata E. L. Ferretti, Renata E. P. Leite, Wilson Jacob Filho, Roberto Lent, and Suzana Herculano-Houzel. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. 513(5):532–541, 2009. arXiv:19226510, doi:10.1002/cne.21974.

[Ado21] Adoption stage of AI in healthcare in the EU 2021, 2021. URL: https://www.statista.com/statistics/1312566/adoption-stage-of-ai-in-healthcare-in-the-eu/.

[ale22] alexeyo26. News - Cognitive Toolkit - CNTK, August 2022. URL: https://learn.microsoft.com/en-us/cognitive-toolkit/news.

[Ama23] Amazon Simple Storage Service S3 – Cloud Online-Speicher, 2023. URL: https://aws.amazon.com/de/s3/.

[AMAZ17] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017. doi:10.1109/ICEngTechnol.2017.8308186.

[AŠ20] Viktar Atliha and Dmitrij Šešok. Comparison of VGG and ResNet used as Encoders for Image Captioning. In *2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pages 1–4, April 2020. doi:10.1109/eStream50540.2020.9108880.

[ASP17] Ammar Ahmad Awan, Hari Subramoni, and Dhabaleswar K. Panda. An In-depth Performance Characterization of CPU- and GPU-based DNN Training on Modern Architectures. In *Proceedings of the Machine Learning on HPC Environments*, MLHPC'17, pages 1–8, New York, NY, USA, November 2017. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3146347.3146356, doi:10.1145/3146347.3146356.

[Ave23] Average monthly internet speed Germany 2023, 2023. URL: https://www.statista.com/statistics/1338657/average-internet-speed-germany/.

[Azu23] Azure Machine Learning: Machine-Learning-as-a-Service — Microsoft Azure, 2023. URL: https://azure.microsoft.com/de-de/products/machine-learning.

[BCG+21] Srinadh Bhojanapalli, Ayan Chakrabarti, Daniel Glasner, Daliang Li, Thomas Unterthiner, and Andreas Veit. Understanding Robustness of Transformers for Image Classification. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 10211–10221. IEEE Computer Society, October 2021. URL: https://www.computer.org/csdl/proceedings-article/iccv/2021/281200k0211/1BmJS9EzORq, doi:10.1109/ICCV48922.2021.01007.

[BD18] Ebubekir BUBER and Banu DIRI. Performance Analysis and CPU vs GPU Comparison for Deep Learning. In *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*, pages 1–6, October 2018. doi:10.1109/CEIT.2018.8751930.

[Ben23] Benchmark performance vs. vanilla PyTorch — PyTorch Lightning 2.1.0dev documentation, 2023. URL: https://lightning.ai/docs/pytorch/latest/benchmarking/benchmarks.html.

[BFH+18] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL: http://github.com/google/jax.

[Big23] BigML.com - machine learning made easy, 2023. URL: https://bigml.com/.

[Bis94] Chris M. Bishop. Neural networks and their applications. 65(6):1803–1832, 1994. URL: https://aip.scitation.org/doi/10.1063/1.1144830, doi:10.1063/1.1144830.

[Bui23] K6 - built-in metrics, 2023. URL: https://k6.io/docs/using-k6/metrics/reference/.

[C+15] François Chollet et al. Keras. https://keras.io, 2015.

[CAB+21] Jerome Y. Cheng, Jacob T. Abel, Ulysses G. J. Balis, David S. McClintock, and Liron Pantanowitz. Challenges in the Development, Deployment, and Regulation of Artificial Intelligence in Anatomic Pathology. *The American Journal of Pathology*, 191(10):1684–1692, October

2021. URL: https://www.sciencedirect.com/science/article/pii/S0002944020305083, doi:10.1016/j.ajpath.2020.10.018.

[Caf23] Caffe2, 2023. URL: http://caffe2.ai/.

[Cha23] Chatbot example - Rete.js, 2023. URL: https://retejs.org/examples/chatbot.

[CHP21] Konstantinos Chatzilygeroudis, Ioannis Hatzilygeroudis, and Isidoros Perikos. Machine Learning Basics. In Parisa Eslambolchilar, Andreas Komninos, and Mark Dunlop, editors, *Intelligent Computing for Interactive System Design*, pages 143–193. ACM, 1 edition, 2021. URL: https://dl.acm.org/doi/10.1145/3447404.3447414, doi:10.1145/3447404.3447414.

[CJW+18] Hye Yoon Chang, Chan Kwon Jung, Junwoo Isaac Woo, Sanghun Lee, Joonyoung Cho, Sun Woo Kim, and Tae-Yeong Kwak. Artificial Intelligence in Pathology. *Journal of Pathology and Translational Medicine*, 53(1):1–12, December 2018. URL: https://synapse.koreamed.org/articles/1152382, doi:10.4132/jptm.2018.12.16.

[CKF11] Ronan Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A Matlab-like Environment for Machine Learning. In *NIPS 2011*, 2011. URL: https://www.semanticscholar.org/paper/Torch7%3A-A-Matlab-like-Environment-for-Machine-Collobert-Kavukcuoglu/3449b65008b27f6e60a73d80c1fd990f0481126b.

[Cle19] ClearML. Clearml - your entire mlops stack in one open-source tool, 2019. Software available from http://github.com/allegroai/clearml. URL: https://clear.ml/.

[Cle23a] ClearML Agent — ClearML, 2023. URL: https://clear.ml/docs/latest/docs/clearml_agent.

[Cle23b] ClearML Server — ClearML, 2023. URL: https://clear.ml/docs/latest/docs/deploying_clearml/clearml_server/.

[CLL+15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems, December 2015. URL: http://arxiv.org/abs/1512.01274, arXiv:1512.01274.

[Clo23] Cloud Storage, 2023. URL: https://cloud.google.com/storage?hl=de.

[CM08] Ami Citri and Robert C. Malenka. Synaptic Plasticity: Multiple Forms, Functions, and Mechanisms. 33(1):18–41, 2008. URL: https://www.nature.com/articles/1301559, doi:10.1038/sj.npp.1301559.

[cml23] Cml · continuous machine learning, 2023. URL: https://cml.dev/doc.

[Col23] Collaborative Machine Learning development — MLReef, 2023. URL: https://www.mlreef.com.

[Com23] The Complete Computer Vision Platform — Picsellia, 2023. URL: https://www.picsellia.com//.

[Con23a] Concepts — MLflow 2.5.0 documentation, 2023. URL: https://mlflow.org/docs/latest/concepts.html#scalability-and-big-data.

[Con23b] MathJax Consortium. MathJax, 2023. URL: https://www.mathjax.org/.

[CPS13] Abhimanyu Chopra, Abhinav Prashar, and Chandresh Sain. Natural Language Processing. 1(4), 2013.

[Dat23] Dataiku — Everyday AI, Extraordinary People, 2023. URL: https://www.dataiku.com/.

[DCM+21] Andrea Duggento, Allegra Conti, Alessandro Mauriello, Maria Guerrisi, and Nicola Toschi. Deep computational pathology in breast cancer. *Seminars in Cancer Biology*, 72:226–237, July 2021. URL: https://www.sciencedirect.com/science/article/pii/S1044579X20301784, doi:10.1016/j.semcancer.2020.08.006.

[Den12] Li Deng. The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. *IEEE Signal Processing Magazine*, 29(6):141–142, November 2012. doi:10.1109/MSP.2012.2211477.

[Dev23a] Developer tools for Machine Learning — Iterative, 2023. URL: https://iterative.ai/.

[Dev23b] TensorFlow Developers. TensorFlow. Zenodo, July 2023. URL: https://zenodo.org/record/8118033, doi:10.5281/zenodo.8118033.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *The Journal of Machine Learning Research*, 12(null):2121–2159, July 2011. URL: https://dl.acm.org/doi/10.5555/1953048.2021068.

[DPS+21] Hulin Dai, Xuan Peng, Xuanhua Shi, Ligang He, Qian Xiong, and Hai Jin. Reveal training performance mystery between TensorFlow and PyTorch in the single GPU environment. *Science China Information Sciences*, 65(1):112103, December 2021. doi:10.1007/s11432-020-3182-1.

[dvc23] Data version control · dvc, 2023. URL: https://dvc.org/doc.

[Eag17] Eager Execution: An imperative, define-by-run interface to TensorFlow, October 2017. URL: https://ai.googleblog.com/2017/10/eager-execution-imperative-define-by.html?m=1.

[Eas23] Easily manage, deploy and monitor Machine Learning models. — craftworks' navio, 2023. URL: https://www.craftworks.ai/navio.

[EGHS16] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. DevOps. *IEEE Software*, 33(3):94–100, May 2016. doi:10.1109/MS.2016.68.

[Fas23] FastAPI, 2023. URL: https://fastapi.tiangolo.com/.

[FLHI+18] Vincent Francois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An Introduction to Deep Reinforcement Learning. 11(3-4):219–354, 2018. URL: http://arxiv.org/abs/1811.12560, arXiv:1811.12560, doi:10.1561/2200000071.

[FT19] William Falcon and The PyTorch Lightning team. PyTorch Lightning, March 2019. URL: https://github.com/Lightning-AI/lightning, doi:10.5281/zenodo.3828935.

[Ful23] Full Stack Machine Learning Operating System — cnvrg.io, 2023. URL: https://cnvrg.io/.

[Fun18] William K. Funkhouser. Pathology. *Molecular Pathology*, pages 217–229, 2018. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7150310/, doi:10.1016/B978-0-12-802761-5.00011-0.

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

[GDDM14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation, October 2014. URL: http://arxiv.org/abs/1311.2524, arXiv:1311.2524, doi:10.48550/arXiv.1311.2524.

[GHPBB21] Christiane Gresse von Wangenheim, Jean C. R. Hauck, Fernando S. Pacheco, and Matheus F. Bertonceli Bueno. Visual tools for teaching machine learning in K-12: A ten-year systematic mapping. *Education and Information Technologies*, 26(5):5733–5778, September 2021. doi:10.1007/s10639-021-10570-8.

[Gir15] Ross Girshick. Fast R-CNN, September 2015. URL: http://arxiv.org/abs/1504.08083, arXiv:1504.08083.

[GLGN+08] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel Computing Experiences with CUDA. *IEEE Micro*, 28(4):13–27, July 2008. doi:10.1109/MM.2008.57.

[GM04] Kalanit Grill-Spector and Rafael Malach. The Human Visual Cortex. *Annual Review of Neuroscience*, 27(1):649–677, 2004. doi:10.1146/annurev.neuro.27.070203.144220.

[Gol19] Sunila Gollapudi. Artificial Intelligence and Computer Vision. In Sunila Gollapudi, editor, *Learn Computer Vision Using OpenCV: With Deep Learning CNNs and RNNs*, pages 1–29. Apress, 2019. doi:10.1007/978-1-4842-4261-2_1.

[Goo23] Google Trends, 2023. URL: https://trends.google.com/trends/explore?date=today%205-y&q=%2Fg%2F11bwp1s2k3,%2Fg%2F11gd3905v1,%2Fg%2F11c1r2rvnp,%2Fg%2F11g6ym8nbt&hl=de.

[GZM20] Christian Garbin, Xingquan Zhu, and Oge Marques. Dropout vs. batch normalization: An empirical study of their impact to deep learning. *Multimedia Tools and Applications*, 79(19):12777–12815, May 2020. doi:10.1007/s11042-019-08453-9.

[Gé19] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media, Inc, second edition edition, 2019. URL: https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/.

[HCS21] Yue Hu, Cheng-Huan Chen, and Chien-Yuan Su. Exploring the Effectiveness and Moderators of Block-Based Visual Programming on Student Learning: A Meta-Analysis. *Journal of Educational Computing Research*, 58(8):1467–1493, January 2021. doi:10.1177/0735633120945935.

[Heb49] Donald Olding Hebb. *The Organization of Behavior: A Neuropsychological Theory.* Wiley, 1949. URL: https://pure.mpg.de/rest/items/item_2346268_3/component/file_2346267/content.

[HGDG18] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask R-CNN, January 2018. URL: http://arxiv.org/abs/1703.06870, arXiv:1703.06870, doi:10.48550/arXiv.1703.06870.

[HM22] Nipuni Hewage and Dulani Meedeniya. Machine Learning Operations: A Survey on MLOps Tool Support. 2022. URL: http://arxiv.org/abs/2202.10169, arXiv:2202.10169, doi:10.48550/arXiv.2202.10169.

[HPS20] Matthew G. Hanna, Anil Parwani, and Sahussapont Joseph Sirintrapun. Whole Slide Imaging: Technology and Applications. *Advances In Anatomic Pathology*, 27(4):251–259, July 2020. doi:10.1097/PAP.0000000000000273.

[Hub59] D. H. Hubel. Single unit activity in striate cortex of unrestrained cats. 147(2):226–238.2, 1959. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1357023/, arXiv:14403678.

[HW59] D. H. Hubel and T. N. Wiesel. Receptive fields of single neurones in the cat's striate cortex. 148(3):574–591, 1959. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1363130/, arXiv:14403679.

[HZJM22] Khoa Ho, Hui Zhao, Adwait Jog, and Saraju Mohanty. Improving GPU Throughput through Parallel Execution Using Tensor Cores and CUDA Cores. In *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 223–228, July 2022. doi:10.1109/ISVLSI54635.2022.00051.

[HZRS14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. volume 8691, pages 346–361. 2014. URL: http://arxiv.org/abs/1406.4729, arXiv:1406.4729, doi:10.1007/978-3-319-10578-9_23.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, December 2015. URL: http://arxiv.org/abs/1512.03385, arXiv:1512.03385, doi:10.48550/arXiv.1512.03385.

[Iak23] Pavel Iakubovskii. Qubvel/segmentation_models.pytorch, September 2023. URL: https://github.com/qubvel/segmentation_models.pytorch.

[Inc23a] Iterative Inc. Iterative Studio, 2023. URL: https://studio.iterative.ai.

[Inc23b] MinIO Inc. MinIO — High Performance, Kubernetes Native Object Storage, 2023. URL: https://min.io.

[Int12] Introduction to Querying Metadata and Artifacts - Polyaxon quick start tutorial - Core Concepts, April 2012. URL: https://polyaxon.com/docs/intro/query-metadata-artifacts/.

[IS15a] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, March 2015. URL: http://arxiv.org/abs/1502.03167, arXiv:1502.03167.

[IS15b] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, March 2015. URL: `http://arxiv.org/abs/1502.03167`, `arXiv:1502.03167`, `doi:10.48550/arXiv.1502.03167`.

[IZG18] Andrej Ilievski, Vladimir Zdraveski, and Marjan Gusev. How CUDA Powers the Machine Learning Revolution. In *2018 26th Telecommunications Forum (TELFOR)*, pages 420–425, November 2018. `doi:10.1109/TELFOR.2018.8611982`.

[JBS15] Michael B. Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT). RFC 7519, May 2015. URL: `https://www.rfc-editor.org/info/rfc7519`, `doi:10.17487/RFC7519`.

[JM16] Andrew Janowczyk and Anant Madabhushi. Deep learning for digital pathology image analysis: A comprehensive tutorial with selected use cases. *Journal of Pathology Informatics*, 7:29, 2016. `doi:10.4103/2153-3539.186902`.

[JNS22] Biswajit Jena, Gopal Krishna Nayak, and Sanjay Saxena. Convolutional neural network and its pretrained models for image classification and object detection: A survey. *Concurrency and Computation: Practice and Experience*, 34(6):e6767, 2022. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6767`.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding, June 2014. URL: `http://arxiv.org/abs/1408.5093`, `arXiv:1408.5093`, `doi:10.48550/arXiv.1408.5093`.

[K6D23] K6 Documentation, 2023. URL: `https://k6.io/docs`.

[Kan03] Laveen N. Kanal. Perceptron. In *Encyclopedia of Computer Science*, pages 1383–1385. John Wiley and Sons Ltd., 2003. URL: `https://dl.acm.org/doi/book/10.5555/1074100`.

[Kat23] Katonic MLOps Platform, 2023. URL: `https://katonic.ai/`.

[KB17] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017. URL: `http://arxiv.org/abs/1412.6980`, `arXiv:1412.6980`, `doi:10.48550/arXiv.1412.6980`.

[KBKT17] Brady Kieffer, Morteza Babaie, Shivam Kalra, and H. R. Tizhoosh. Convolutional neural networks for histopathology image classification: Training vs. Using pre-trained networks. In *2017 Seventh International Conference on Image Processing Theory, Tools and Applications (IPTA)*, pages

1–6, November 2017. URL: https://ieeexplore.ieee.org/document/8310149.

[Ker18] Michael Kerres. Mediendidaktik: Konzeption und Entwicklung digitaler Lernangebote. In *Mediendidaktik*. De Gruyter Oldenbourg, March 2018. URL: https://www.degruyter.com/document/doi/10.1515/9783110456837/html?lang=de, doi:10.1515/9783110456837.

[Ker23] Keras: The high-level API for TensorFlow — TensorFlow Core, 2023. URL: https://www.tensorflow.org/guide/keras.

[Key23] Keycloak, 2023. URL: https://www.keycloak.org/.

[KKH23] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. Machine Learning Operations (MLOps): Overview, Definition, and Architecture. *IEEE Access*, 11:31866–31879, 2023. doi:10.1109/ACCESS.2023.3262138.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL: https://papers.nips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[Kuv22] Prajot Kuvalekar. Answer to ”padding='same' conversion to PyTorch padding=#”, August 2022. URL: https://stackoverflow.com/a/73332370.

[Lau23] Launching an On-Premise Cluster — Ray 2.6.1, 2023. URL: https://docs.ray.io/en/latest/cluster/vms/user-guides/launching-clusters/on-premises.html#on-prem.

[LBBH98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. doi:10.1109/5.726791.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015. URL: https://www.nature.com/articles/nature14539, doi:10.1038/nature14539.

[LBOM12] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient BackProp. In Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade: Second Edition*, Lecture Notes in Computer Science, pages 9–48. Springer, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-35289-8_3.

[Lea23] Lml - artificiai intelligence made eas, 2023. URL: https://web.learningml.org/en/home-spanish-en-translation/.

[LG00] S. Lawrence and C.L. Giles. Overfitting and neural networks: Conjugate gradient and backpropagation. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 1, pages 114–119 vol.1, July 2000. doi:10.1109/IJCNN.2000.857823.

[LH19] Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization, January 2019. URL: http://arxiv.org/abs/1711.05101, arXiv:1711.05101, doi:10.48550/arXiv.1711.05101.

[Lig23] LightningDataModule — PyTorch Lightning 2.0.9 documentation, 2023. URL: https://lightning.ai/docs/pytorch/stable/data/datamodule.html.

[Lju23] Bioinformatics Laboratory Ljubljana, University of. Data Mining, 2023. URL: https://orangedatamining.com/.

[LLR+22] Xintong Li, Chen Li, Md Mamunur Rahaman, Hongzan Sun, Xiaoqi Li, Jian Wu, Yudong Yao, and Marcin Grzegorzek. A comprehensive review of computer-aided whole-slide image analysis: From datasets to feature extraction, segmentation, classification and detection approaches. *Artificial Intelligence Review*, 55(6):4809–4878, August 2022. doi:10.1007/s10462-021-10121-0.

[Log23] Logger — ClearML, 2023. URL: https://clear.ml/docs/latest/docs/fundamentals/logger/.

[LSD15] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully Convolutional Networks for Semantic Segmentation, March 2015. URL: http://arxiv.org/abs/1411.4038, arXiv:1411.4038, doi:10.48550/arXiv.1411.4038.

[LSJR16] Jason D. Lee, Max Simchowitz, Michael I. Jordan, and Benjamin Recht. Gradient Descent Only Converges to Minimizers. In *Conference on Learning Theory*, pages 1246–1257. PMLR, June 2016. URL: https://proceedings.mlr.press/v49/lee16.html.

[LSM+20] Timothy P. Lillicrap, Adam Santoro, Luke Marris, Colin J. Akerman, and Geoffrey Hinton. Backpropagation and the brain. *Nature Reviews Neuroscience*, 21(6):335–346, June 2020. URL: https://www.nature.com/articles/s41583-020-0277-3, doi:10.1038/s41583-020-0277-3.

[Mac23a] Machine Learning Operations (MLOps), 2023. URL: https://www.datarobot.com/platform/mlops/.

[Mac23b] Machine Learning – Amazon Web Services, 2023. URL: https://aws.amazon.com/de/sagemaker/.

[MF11] Alexey Melnikov and Ian Fette. The WebSocket Protocol. Request for Comments RFC 6455, Internet Engineering Task Force, December 2011. URL: `https://datatracker.ietf.org/doc/rfc6455`, `doi:10.17487/RFC6455`.

[Mic23] Microsoft/CNTK. Microsoft, July 2023. URL: `https://github.com/microsoft/CNTK`.

[Mid23] Middleware - FastAPI, 2023. URL: `https://fastapi.tiangolo.com/tutorial/middleware/`.

[Mil68] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, Part I), pages 267–277, New York, NY, USA, December 1968. Association for Computing Machinery. URL: `https://dl.acm.org/doi/10.1145/1476589.1476628`, `doi:10.1145/1476589.1476628`.

[MIL17] MILA and the future of Theano, 2017. URL: `https://groups.google.com/g/theano-users/c/7Poq8BZutbY/m/rNCIfvAEAwAJ`.

[Min23] Mindspore-ai/mindspore. MindSpore, July 2023. URL: `https://github.com/mindspore-ai/mindspore`.

[mle23] Mlem · simplifying machine learning model deployment, 2023. URL: `https://mlem.ai/doc`.

[MLf23] MLflow - A platform for the machine learning lifecycle, 2023. URL: `https://mlflow.org/`.

[MLO23] MLOps Platform for EDGE AI and Enterprise AI, 2023. URL: `https://www.akira.ai`.

[Mod23a] ModelArts AI Development Platform — Huawei Cloud, 2023. URL: `https://www.huaweicloud.com/intl/en-us/product/modelarts.html`.

[Mod23b] Models and pre-trained weights — Torchvision 0.15 documentation, 2023. URL: `https://pytorch.org/vision/stable/models.html`.

[Mön23] Jens Mönig. Snap! Build Your Own Blocks, July 2023. URL: `https://github.com/jmoenig/Snap`.

[Mou18] Mourad Mourafiq. Polyaxon: Cloud native machine learning platform. Web page, 2018. Software available from polyaxon.com. URL: `https://github.com/polyaxon/polyaxon`.

[MRL+23] Sergio Moreschi, Gilberto Recupito, Valentina Lenarduzzi, Fabio Palomba, David Hastbacka, and Davide Taibi. Toward End-to-End MLOps Tools Map: A Preliminary Study based on a Multivocal Literature Review, April 2023. URL: http://arxiv.org/abs/2304.03254, arXiv:2304.03254.

[MRR+10] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education (TOCE)*, 10:16, November 2010. doi:10.1145/1868358.1868363.

[NCN+22] Ovidiu-Constantin Novac, Mihai Cristian Chirodea, Cornelia Mihaela Novac, Nicu Bizon, Mihai Oproescu, Ovidiu Petru Stan, and Cornelia Emilia Gordan. Analysis of the Application Efficiency of TensorFlow and PyTorch in Convolutional Neural Network. *Sensors*, 22(22):8872, January 2022. URL: https://www.mdpi.com/1424-8220/22/22/8872, doi:10.3390/s22228872.

[NDB+19] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchý. Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: A survey. *Artificial Intelligence Review*, 52(1):77–124, June 2019. doi:10.1007/s10462-018-09679-z.

[Nee21] Jannes Neemann. *Entwicklung einer Lernsoftware für das Fach Pathologie*. Bachelor Thesis, 2021. URL: http://nbn-resolving.de/urn:nbn:de:bsz:960-opus4-19729, doi:10.25968/opus-1972.

[Nor23] Normalize — Torchvision 0.15 documentation, 2023. URL: https://pytorch.org/vision/stable/generated/torchvision.transforms.Normalize.html.

[ONN23] ONNX — Home, 2023. URL: https://onnx.ai/.

[Ory23] Ory - API-first Identity Management, Authentication and Authorization. For Secure, Global, GDPR-compliant Apps — Ory, 2023. URL: https://www.ory.sh/.

[OSP+22] Parita Oza, Paawan Sharma, Samir Patel, Festus Adedoyin, and Alessandro Bruno. Image Augmentation Techniques for Mammogram Analysis. *Journal of Imaging*, 8(5):141, May 2022. URL: https://www.mdpi.com/2313-433X/8/5/141, doi:10.3390/jimaging8050141.

[Pad23] PaddlePaddle/Paddle. PaddlePaddle, July 2023. URL: https://github.com/PaddlePaddle/Paddle.

[Pap23a] Papers with Code - About Papers With Code, 2023. URL: https://paperswithcode.com/about.

[Pap23b] Papers with Code - Papers With Code : Trends, 2023. URL: https://paperswithcode.com/trends.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[Plu23] Plugin system - Rete.js, 2023. URL: https://retejs.org/docs/concepts/plugin-system.

[Pol12] Polyaxon management - Model Registry, April 2012. URL: https://polyaxon.com/docs/management/model-registry/.

[Pre19] Preferred Networks Migrates its Deep Learning Research Platform to PyTorch, December 2019. URL: https://www.preferred.jp/en/news/pr20191205/.

[Pro23a] Productionizing and scaling Python ML workloads simply, 2023. URL: https://www.ray.io/.

[Pro23b] Produktionsreife Container-Orchestrierung, 2023. URL: https://kubernetes.io/de/.

[Pro23c] Projects — ClearML, 2023. URL: https://clear.ml/docs/latest/docs/fundamentals/projects.

[Pus23a] Pusher — Leader In Realtime Technologies, 2023. URL: https://pusher.com/.

[Pus23b] Pusher. Pusher Channels Protocol, 2023. URL: https://pusher.com/docs/channels/library_auth_reference/pusher-websockets-protocol/undefined/docs/channels/library_auth_reference/pusher-websockets-protocol/.

[Qia99] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145–151, January 1999. URL: https://www.sciencedirect.com/science/article/pii/S0893608098001166, doi:10.1016/S0893-6080(98)00116-6.

[Ray23a] Ray Dashboard — Ray 2.6.1, 2023. URL: https://docs.ray.io/en/latest/ray-observability/getting-started.html.

[Ray23b] Ray Data: Scalable Datasets for ML — Ray 2.6.1, 2023. URL: https://docs.ray.io/en/latest/data/data.html.

[Ray23c] Ray Serve: Scalable and Programmable Serving — Ray 2.6.1, 2023. URL: https://docs.ray.io/en/latest/serve/index.html.

[Ray23d] Ray Train: Scalable Model Training — Ray 2.6.1, 2023. URL: https://docs.ray.io/en/latest/train/train.html.

[RDGF16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection, May 2016. URL: http://arxiv.org/abs/1506.02640, arXiv:1506.02640.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge, January 2015. URL: http://arxiv.org/abs/1409.0575, arXiv:1409.0575, doi:10.48550/arXiv.1409.0575.

[Ret23] Rete.js - JavaScript framework for visual programming, 2023. URL: https://retejs.org/.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation, May 2015. URL: http://arxiv.org/abs/1505.04597, arXiv:1505.04597.

[RHGS16] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks, January 2016. URL: http://arxiv.org/abs/1506.01497, arXiv:1506.01497.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. 323(6088):533–536, 1986. URL: https://www.nature.com/articles/323533a0, doi:10.1038/323533a0.

[RM51] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400–407, September 1951. URL: https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-22/issue-3/A-Stochastic-Approximation-Method/10.1214/aoms/1177729586.full, doi:10.1214/aoms/1177729586.

[RM87] David E. Rumelhart and James L. McClelland. Learning Internal Representations by Error Propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*, pages 318–362. MIT Press, 1987. URL: https://ieeexplore.ieee.org/document/6302929.

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. 65(6):386–408, 1958. arXiv:13602029, doi:10.1037/h0042519.

[RPC+22] Gilberto Recupito, Fabiano Pecorelli, Gemma Catolino, Sergio Moreschini, Dario Di Nucci, Fabio Palomba, and Damian A. Tamburri. A Multivocal Literature Review of MLOps Tools and Features. In *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 84–91, August 2022. doi:10.1109/SEAA56994.2022.00021.

[RRS+22] João Pedro Mazuco Rodriguez, Rubens Rodriguez, Vitor Werneck Krauss Silva, Felipe Campos Kitamura, Gustavo Cesar Antônio Corradi, Ana Carolina Bertoletti de Marchi, and Rafael Rieder. Artificial intelligence as a tool for diagnosis in digital pathology whole slide images: A systematic review. *Journal of Pathology Informatics*, 13:100138, January 2022. URL: https://www.sciencedirect.com/science/article/pii/S2153353922007325, doi:10.1016/j.jpi.2022.100138.

[Sah20] Shagan Sah. Machine Learning: A Review of Learning Types, 2020. URL: https://www.preprints.org/manuscript/202007.0230/v1, arXiv:2020070230, doi:10.20944/preprints202007.0230.v1.

[Sel23] Self-hosted Runners, 2023. URL: https://cml.dev/doc/self-hosted-runners.

[Ser23a] Integrations - serving, 2023. URL: https://polyaxon.com/integrations/serving/.

[Ser23b] Serving — ClearML, 2023. URL: https://clear.ml/docs/latest/docs/clearml_serving/.

[SFH21] Ola Spjuth, Jens Frid, and Andreas Hellander. The machine learning life cycle and the cloud: Implications for drug discovery. *Expert Opinion on Drug Discovery*, 16(9):1071–1079, September 2021. doi:10.1080/17460441.2021.1932812.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, January 2014. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[Sim23] Simplified stock exchange as pubsub benchmark. uNetworking AB, August 2023. URL: https://github.com/uNetworking/pubsub-benchmark.

[SIVA16] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning, August 2016. URL: `http://arxiv.org/abs/1602.07261`, `arXiv:1602.07261`.

[SLJ+14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions, September 2014. URL: `http://arxiv.org/abs/1409.4842`, `arXiv:1409.4842`.

[SNKP22] Georgios Symeonidis, Evangelos Nerantzis, Apostolos Kazakis, and George A. Papakostas. MLOps - Definitions, Tools and Challenges. In *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0453–0460, January 2022. `doi:10.1109/CCWC54503.2022.9720902`.

[Soc23] Socket.IO, 2023. URL: `https://socket.io/`.

[Sok23] Soketi, 2023. URL: `https://docs.soketi.app/`.

[SOM11] Eric Stratmann, John Ousterhout, and Sameer Madan. Integrating long polling with an MVC framework. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, WebApps'11, page 10, USA, June 2011. USENIX Association. URL: `https://dl.acm.org/doi/10.5555/2002168.2002178`.

[SOPH16] Fabio A. Spanhol, Luiz S. Oliveira, Caroline Petitjean, and Laurent Heutte. A Dataset for Breast Cancer Histopathological Image Classification. *IEEE Transactions on Biomedical Engineering*, 63(7):1455–1462, July 2016. `doi:10.1109/TBME.2015.2496264`.

[SRG+16] Hoo-Chang Shin, Holger R. Roth, Mingchen Gao, Le Lu, Ziyue Xu, Isabella Nogues, Jianhua Yao, Daniel Mollura, and Ronald M. Summers. Deep Convolutional Neural Networks for Computer-Aided Detection: CNN Architectures, Dataset Characteristics and Transfer Learning. *IEEE Transactions on Medical Imaging*, 35(5):1285–1298, May 2016. `doi:10.1109/TMI.2016.2528162`.

[Sto23] Storage — ClearML, 2023. URL: `https://clear.ml/docs/latest/docs/integrations/storage`.

[Sup23] SuperTokens, Open Source User Authentication, 2023. URL: `https://supertokens.com/`.

[SVI+15] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision, December 2015. URL: http://arxiv.org/abs/1512.00567, arXiv:1512.00567, doi:10.48550/arXiv.1512.00567.

[Swa23] Swarm mode overview, August 2023. URL: https://docs.docker.com/engine/swarm/.

[SZ15] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, April 2015. URL: http://arxiv.org/abs/1409.1556, arXiv:1409.1556, doi:10.48550/arXiv.1409.1556.

[TAA+16] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre-Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Mélanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian Goodfellow, Matt Graham, Caglar Gulcehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrancois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert T. McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A Python framework for fast computation of mathematical expressions, May 2016. URL: http://arxiv.org/abs/1605.02688, arXiv:1605.02688, doi:10.48550/arXiv.1605.02688.

[Tas23] Tasks — ClearML, 2023. URL: https://clear.ml/docs/latest/docs/fundamentals/task.

[TBF+22] Matteo Testi, Matteo Ballabio, Emanuele Frontoni, Giulio Iannello, Sara Moccia, Paolo Soda, and Gennaro Vessio. MLOps: A Taxonomy and a Methodology. *IEEE Access*, 10:63606–63618, 2022. doi:10.1109/ACCESS.2022.3181730.

[TC23] Juan Terven and Diana Cordova-Esparza. A Comprehensive Review of YOLO: From YOLOv1 and Beyond, August 2023. URL: http://arxiv.org/abs/2304.00501, arXiv:2304.00501, doi:10.48550/arXiv.2304.00501.

[Tea18] Caffe2 Team. Caffe2 and PyTorch join forces to create a Research + Production platform PyTorch 1.0, May 2018. URL: http://caffe2.ai/blog/2018/05/02/Caffe2_PyTorch_1_0.html.

[Tea23] Teachable Machine, 2023. URL: https://teachablemachine.withgoogle.com/.

[Ten23] TensorFlow 1.x vs TensorFlow 2 - Behaviors and APIs — TensorFlow Core, 2023. URL: https://www.tensorflow.org/guide/migrate/tf1_vs_tf2.

[TH12] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude. In *COURSERA: Neural Networks for Machine Learning*, 4, pages 26–31. 2012. URL: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[Thr23] Throttling, February 2023. URL: https://developer.chrome.com/docs/devtools/settings/throttling/.

[TM20] Jonas Teuwen and Nikita Moriakov. Chapter 20 - Convolutional neural networks. In S. Kevin Zhou, Daniel Rueckert, and Gabor Fichtinger, editors, *Handbook of Medical Image Computing and Computer Assisted Intervention*, The Elsevier and MICCAI Society Book Series, pages 481–501. Academic Press, 2020. URL: https://www.sciencedirect.com/science/article/pii/B9780128161760000259, doi:10.1016/B978-0-12-816176-0.00025-9.

[TOA+19] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. Chainer: A Deep Learning Framework for Accelerating the Research Cycle, August 2019. URL: http://arxiv.org/abs/1908.00213, arXiv:1908.00213.

[Tor] TorchScript — PyTorch 2.0 documentation. URL: https://pytorch.org/docs/stable/jit.html.

[Tor23a] Torch Package Reference Manual. Torch, July 2023. URL: https://github.com/torch/torch7.

[Tor23b] Torch.add — PyTorch 2.0 documentation, 2023. URL: https://pytorch.org/docs/stable/generated/torch.add.html.

[Tor23c] Torch.cat — PyTorch 2.0 documentation, 2023. URL: https://pytorch.org/docs/stable/generated/torch.cat.html?highlight=cat.

[Tor23d] Torch.onnx — PyTorch 2.0 documentation, 2023. URL: https://pytorch.org/docs/stable/onnx.html.

[Tri20] Triton Inference Server, March 2020. URL: https://developer.nvidia.com/triton-inference-server.

[TZZ23] Yingjie Tian, Yuqi Zhang, and Haibin Zhang. Recent Advances in Stochastic Gradient Descent in Deep Learning. *Mathematics*, 11(3):682, January 2023. URL: https://www.mdpi.com/2227-7390/11/3/682, doi:10.3390/math11030682.

[UNe23] uNetworking/uWebSockets.js. uNetworking AB, August 2023. URL: https://github.com/uNetworking/uWebSockets.js.

[Usi23] Using server-sent events - Web APIs — MDN, February 2023. URL: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events.

[UvGS13] J. R. R. Uijlings, K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders. Selective Search for Object Recognition. *International Journal of Computer Vision*, 104(2):154–171, September 2013. doi:10.1007/s11263-013-0620-5.

[Val23] Valohai — Take ML places it's never been, 2023. URL: https://valohai.com/.

[Ver23] Vertex AI, 2023. URL: https://cloud.google.com/vertex-ai?hl=de.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL: https://papers.nips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[Wal23] Wallaroo.AI, 2023. URL: https://wallaroo.ai/.

[War02] Barry Warsaw. PEP 292 – Simpler String Substitutions — peps.python.org, 2002. URL: https://peps.python.org/pep-0292/.

[Wat23] Watson Studio - Resources and Tools, 2023. URL: `https://developer.ibm.com/components/watson-studio/`.

[Web23] WebSocket - Web APIs — MDN, March 2023. URL: `https://developer.mozilla.org/en-US/docs/Web/API/WebSocket`.

[Wel23] Welcome to TorchMetrics — PyTorch-Metrics 1.0.1 documentation, 2023. URL: `https://torchmetrics.readthedocs.io/en/stable/`.

[WKW16] Karl Weiss, Taghi M. Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big Data*, 3(1):9, May 2016. URL: `https://doi.org/10.1186/s40537-016-0043-6`.

[WLLT21] Lei Wang, Zhengchao Liu, Ang Liu, and Fei Tao. Artificial intelligence in product lifecycle management. 114(3):771–796, 2021. `doi:10.1007/s00170-021-06882-1`.

[WS19] Shuai Wang and Zhendong Su. Metamorphic Testing for Object Detection Systems, 2019. URL: `http://arxiv.org/abs/1912.12162`, `arXiv:arXiv:1912.12162`, `doi:10.48550/arXiv.1912.12162`.

[XYFP23] Mingle Xu, Sook Yoon, Alvaro Fuentes, and Dong Sun Park. A Comprehensive Survey of Image Augmentation Techniques for Deep Learning. *Pattern Recognition*, 137:109347, May 2023. URL: `https://www.sciencedirect.com/science/article/pii/S0031320323000481`, `doi:10.1016/j.patcog.2023.109347`.

[YMW+20] Jining Yan, Lin Mu, Lizhe Wang, R. Ranjan, and Albert Zomaya. Temporal Convolutional Networks for the Advance Prediction of ENSO. *Scientific Reports*, 10:8055, May 2020. `doi:10.1038/s41598-020-65070-5`.

[Zei12] Matthew D. Zeiler. ADADELTA: An Adaptive Learning Rate Method, December 2012. URL: `http://arxiv.org/abs/1212.5701`, `arXiv:1212.5701`.

[Zha19] Jiawei Zhang. Basic Neural Units of the Brain: Neurons, Synapses and Action Potential, 2019. URL: `http://arxiv.org/abs/1906.01703`, `arXiv:arXiv:1906.01703`, `doi:10.48550/arXiv.1906.01703`.

[ZJ17] Wang Zhiqiang and Liu Jun. A review of object detection based on convolutional neural network. In *2017 36th Chinese Control Conference (CCC)*, pages 11104–11109, July 2017. `doi:10.23919/ChiCC.2017.8029130`.

[ZQD+21] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A Comprehensive Survey on Transfer Learning. *Proceedings of the IEEE*, 109(1):43–76, January 2021. `doi:10.1109/JPROC.2020.3004555`.