A Software Toolbox for Deploying Deep Learning Decision Support Systems with XAI Capabilities

Fabrizio Nunnari
fabrizio.nunnari@dfki.de
German Research Center for Artificial Intelligence (DFKI)
Saarbrücken, Germany

Daniel Sonntag
daniel.sonntag@dfki.de
German Research Center for Artificial Intelligence (DFKI),
Saarland Universtity, Oldenburg University
Saarbrücken, Germany

ABSTRACT

We describe the software architecture of a toolbox of reusable components for the configuration of convolutional neural networks (CNNs) for classification and labeling problems. The toolbox architecture has been designed to maximize the reuse of established algorithms and to include domain experts in the development and evaluation process across different projects and challenges. In addition, we implemented easy-to-edit input formats and modules for XAI (eXplainable AI) through visual inspection capabilities. The toolbox is available for the research community to implement applied artificial intelligence projects.

CCS CONCEPTS

• **Computing methodologies** → **Computer vision**; **Machine learning**; • **Applied computing** → **Health informatics**; • **Human-centered computing** → **Visualization systems and tools**; • **Software and its engineering** → **Development frameworks and environments**.

KEYWORDS

Software toolbox, design patterns, deep learning, object-oriented architecture, convolutional neural networks, explainable AI.

ACM Reference Format:
Fabrizio Nunnari and Daniel Sonntag. 2021. A Software Toolbox for Deploying Deep Learning Decision Support Systems with XAI Capabilities. In *Companion of the 2021 ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '21 Companion), June 8–11, 2021, Virtual Event, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3459926.3464753

1 MOTIVATION

Over the last several years, the quantity of research in the field of artificial intelligence employing deep neural networks has increased enormously. This led to the proliferation of open-source libraries, developed by both industry and academic research labs, and they now enable us to experiment with new neural architectures, the most popular among them being Tensorflow [12], PyTorch [5] or MXNet [23]. To facilitate and speed-up development processes, higher level frameworks have been proposed, such as Keras [10]. Nowadays, the most popular programming paradigm used to exploit the above-mentioned software is through interactive Jupyter notebooks [9].

Against this background, we face two main problems in applied artificial intelligence projects. First, all of these tools require solid programming skills, leaving domain experts (likely non-developers) without the possibility test, tune or extend a prediction model. Second, tools such as Jupyter notebooks are good tools for exploring different architectures and new solutions [14]. However, they force developers to release 1-shot scripts which merge into a single sequence all of the processing steps needed to develop and deploy a model. This inevitably leads to bad software engineering practices, such as in-code absolute paths and machine-dependent parameters.

These findings urge the need for a software toolbox where AI engineers (developers) and practitioners (domain experts) jointly work in a machine learning cycle (cf. CRISP data mining cycle and extensions [22]). As much as we need software engineers to contribute with new and reusable architectures, we have to give practitioners the autonomy to setup complex configurations for training, run them for long periods of time without the need of user interaction, and examine the results using intuitive interactive interfaces. An additional aspect is to provide XAI (eXplainable AI) capabilities on top of that.

Figure 1 depicts our envisioned pipeline for deploying convolutional deep learning architecture, factoring out the separation of duties and overlapping responsibilities and competences between the developers and the domain experts. In the remainder of this paper, after reporting on related work, we describe the details and the software engineering behind its development. The toolbox, named TIML, is publicly available at the DFKI IML Git repository[1]. The toolbox is implemented with the Python language and uses the Keras framework with Tensorflow backend. All image processing is based on the Pillow [11] package.

2 RELATED WORK

Tools offering a pure visual (no-coding) approach for training and using AI environments do exist (e.g., Kubeflow [24]), but they limit the possibility to experiment with coding for new architectures. Among commercial applications, the Deep Learning Toolbox [13] and Rapidminer (https://rapidminer.com/) offer both visual front-end and extension capabilities, but bind the user to costly licenses, closed sources, and those tools do not offer visual explanation tools.

[1] https://github.com/DFKI-Interactive-Machine-Learning/TIML

EICS '21 Companion, June 8–11, 2021, Virtual Event, Netherlands.

ACM ISBN 978-1-4503-8449-0/21/06…$15.00
https://doi.org/10.1145/3459926.3464753

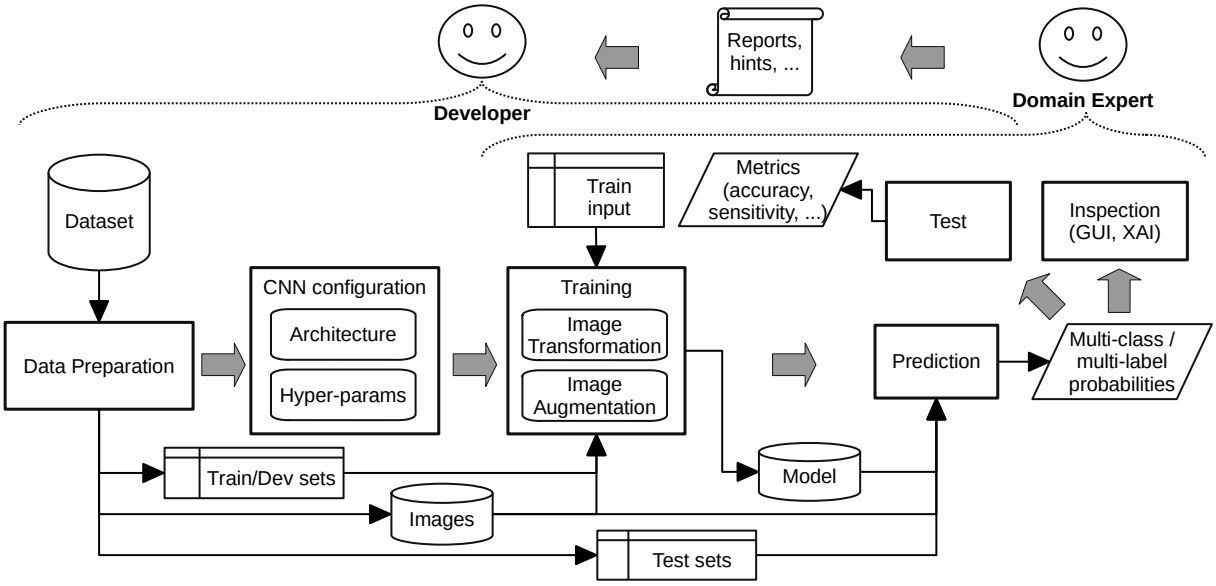

Figure 1: The pipeline for the development, test, deploy and use of convolutional neural networks.

Zacharias et al. [26] surveyed a number of tools and libraries for the development of deep learning tools for intelligent user interfaces. A comprehensive list can be found in [26] in table 1. All of them are low-level libraries offering programing API, thus unusable by domain experts without programming skills.

NVIDIA/DIGITS [15] is a toolkit that builds on the top of Tensorflow, Caffè, or Torch, offering classes and functions for a quick configuration of deep CNNs. It also offers a web interface for models' configuration. However, the interface is tailored to developers, not domain experts, and its last update is dated 2018.

In the field of cancer detection, the Cancer Imaging Phenomics Toolkit (CaPTk) [4] offers a collection of computer vision algorithms and predictive models for the diagnosis of brain, breast, and lung cancer. However, no support for deep learning methods is provided. The toolkit offers a set of command line tools, but no exploratory web interface.

The End2You toolkit [25] provides means to train end-to-end deep-learning models for video, audio and other multimedia data types such as physiological data streams. However, the deep learning architectures are hard-coded. There is the possibility to include custom models, but how to integrate them with the existing architecture is undocumented. The toolkit is controlled via command line interface, but there is no web-based support.

DLTK [17] and NiftyNet [7] are toolkits specialized in the pixel-level segmentation of medical images, but not classification. They offer only a programming API and therefore only suited for programmers.

MLflow [1] is a framework for the management of machine learning projects offering command line interface, REST API, and a web-based interface to monitor experiments progress. However, it acts more as a container of self-programmed projects, easing management, deploy and maintainance, without offering any pre-built machine learning implementation. It is a complement to TIML rather than a competitor.

From our survey, it appears that our toolkit is the only one offering, a software architecture facilitating the inclusion of new models, and at the same time, command line tools for the training and testing procedures, both command line and web interfaces for prediction tasks, and a web interface for the exploration of the results through XAI tools.

3 DEVELOPMENT PIPELINE

In the following, we exemplify the toolbox's development pipeline by an image classification example (see figure 1).

The **Data Preparation** stage is necessary because application projects all come with their own image formats and resolutions, data splits, and ground truth labelling idiosyncrasies. It is common practice to associate each of the train, validation, and test sets to a directory. However, when different splits are be needed, e.g., to perform 5-fold cross validation or to extract a test set from a train set, copying or rearranging image folders is both time and space consuming. Hence, in our toolbox we store all images in a single shared folder, and define the splits using lightweight CSV (comma-separated values) files, optionally containing the ground truth needed for training. When facing a new application scenario (or, e.g., imaging challenge such as ISIC), only a few scripts are needed from the software developers to normalize the data to meet the requirements of the applied AI project.

In the **CNN configuration** stage, developers configure new CNN architectures for specific domains. This can be tuning the hyper-parameters of some existing architecture, or defining new ones. As later explained in section 4, developers have here the freedom to propose new architectures without replicating common training and testing code.

Table 1: Example of a train input dataframe.

architecture	dataset	split	epochs	imgaug	batch size	img size	resize filter	color space	class weights
VGG16	ISIC-2019	pre	10	hflip_rot24	12	450	nearest	RGB	[0.2,0.8]
SC19	ISIC-2016	n=100	15	hflip_rot4	64	227	bilinear	HSV	compute

Training a classifier consists of mainly identifying the data source and launching the training process. We designed the *train input* format to be easily editable and understandable by developers as well as domain experts. It consists of a single CSV file, thus easily manageable with a spreadsheet editor (example in table 1). Each line of the train input is an independent train/test session allowing for batch test of any combination of data, architectures, and *data parameters*. The latter are not the hyper-parameters used to tune an architecture (which are not visible to the domain experts), but rather a set of data-manipulation parameters that can be directly understood and edited by domain experts.

With this strategy, searching for the optimal data parameters (either via grid search of random sampling) can be easily implemented by procedurally generate multiple lines for the train input starting from a template line. Similarly, to perform statistical tests on the variance of the results on a data set (e.g., x-fold cross-validation, test against random initialization), one can procedurally generate multiple lines with the same parameters' values but training and testing on different data splits. While it is true that these strategies involve the duplication of information across the lines of the input tables, the duplicated data is limited to textual information; memory and space footprints are limited with respect to the readability and easiness of inspection that this approach offers.

In *train input* tables (see table 1), the `architecture` column specifies a CNN configuration. The `dataset` column contains the name of the CSV file containing the list of image file names and ground truth labels. The `split` column specifies whether the validation and test sets are pre-split on different files or on-the-fly sampling is needed. The `epochs` column specifies the number of training epochs, while the `imgaug` column contains a preset for image augmentation, both affecting training time. The `batchsize` column is the training batch size, while `imgsize` is the (squared) resolution at which each input image will be rescaled, both affecting the quantity of GPU RAM needed. The `resize filter` specifies the resize sampling strategy (nearest, bilinear, bicubic, or lanczos). The `colorspace` specifies whether the images should be kept in their original `RGB` format or should be converted into HVS, LAB, or YCbCr. Finally, `classweights` specifies the weight factors for each class, which can also be automatically `computed` from the input dataset.

It was an explicit design choice to explore the training epochs, augmentation strategy, batch size, and input image size parameters because they all influence the time and hardware requirements for training. Those are aspects which must be controlled by domain experts when porting the system to new hardware platforms or when autonomously switching to datasets of different magnitudes. The resize filter and colorspace are supposed to have an impact on the performances only on different image distributions.

Training is performed via a command line tool, and its output is a set of binary Keras models together with test statistics and TensorBoard log-files to monitor the training history.

In our toolbox, the **Prediction** command line tool takes as input a model and a test set dataframe, and produces prediction files as a CSV file or as binary numpy vectors. Prediction can output in a single step (i.e., forward pass) not only the final classification layer, but also any combination of intermediate activation layers. This last feature is particularly important when the activation of an intermediate layer of a CNN can be used as feature vector for training further models. In this way, in further training steps we can read the binary vectors from disk, rather than running a forward propagation pass for every image, thus reducing both training time and memory footprint.

Testing consists of comparing the predictions of a model to the ground truth data. As the prediction has already been performed, this process doesn't require GPU-equipped machines. Our toolbox computes metrics such as accuracy, specificity, sensitivity, F1-score, ROC AUC, and more can be added to a centralized testing module. Since multi-class prediction is supported, metrics are reported both per-class and as average among classes. Computed metrics are written to JSON summary files and also to a CSV dataframe with the same format as the input training configurations, hence automatically creating comparison tables to observe the variation in performance together with the tuning of the training input. By using CSV files as both input and output formats, domain experts can (via spreadsheet editors) easily copy-paste valuable configuration results into summary tables, whose lines can be used to define new input tables for further experiments.

The manual **Inspection** of the model from the domain-experts is mandatory to properly verify the credibility of the system in real settings. Our toolbox includes a Flask [16] server to put trained models quickly at work. We configured a set of web pages allowing domain experts to upload single samples, or a batch of them, and verify classification results. On request, the interface shows the result of a visual explanation, i.e., the image activation areas that have driven the network to its final decision. Our framework integrates both the GradCAM [21] and the RISE [19] methods. Figure 2 shows an example in the domain of skin cancer classification. Finally, the server provides the same functionalities though an HTTP REST API, allowing for the integration with custom web applications. This is useful when deploying results to remote project partners while preserving the ownership of the binary trained model.

4 SOFTWARE COMPONENTS

Figure 3 shows the UML class diagram of the core components of this toolbox. The software is designed using "elements of reusable object-oriented software" design patterns [6].

Figure 2: Examples of the inspection web GUI.

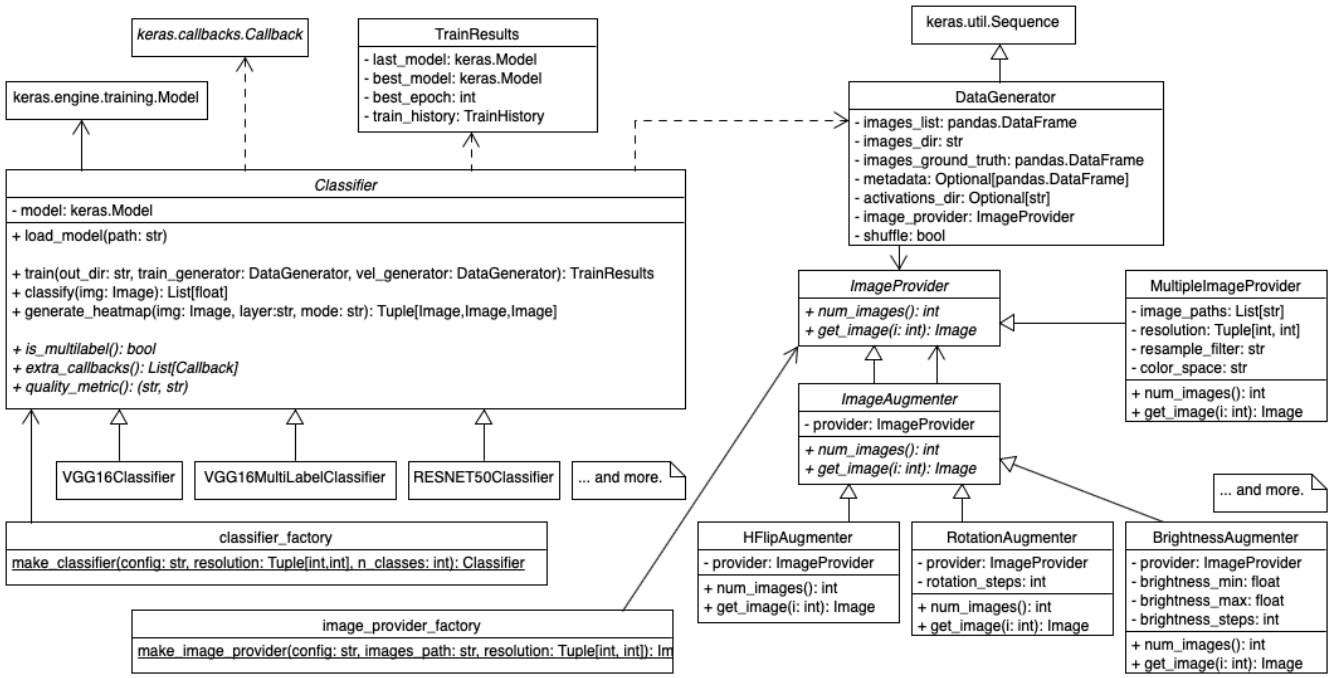

Figure 3: UML class diagram of the main components of our toolbox.

The **`Classifier`** is an abstract class containing all the code usually driving CNN engineering: a `train()` method to tune the hyper-parameters by minimizing a loss function, and a `classify()` method to compute the prediction on (a batch of) samples. Additionally, given an image, `generate_heatmap()` computes the XAI images: a grayscale saliency map, an RGB colored heatmap, and an overlap between the original sample and the heatmap.

To define a new classifier, software developers can implement subclasses and specify if the new classifier is a multi-class (argmax on softmax output) or a multi-label classifier ($p > 0.5$ on sigmoid output). The subclass can also specify a list of training callbacks (e.g., `ReduceLROnPlateau`) and the metric used to evaluate and save the best model before overfitting. The instancing of the Classifier is performed in the `make_classifier()` *FactoryMethod* [6], where several combination of hyper-parameters can be associated to a mnemonic ID (e.g., `VGG16-fc2048-droupout0.5-nadam`).

The **`DataGenerator`** class is an implementation of the `keras.utils.Sequence` interface (implementing the `len()` and the `__get_item__()` methods). Its basic function is to provide image pixels and the ground truth for training. Given a list of images, the constructor scans the source directory and verifies the existence of every sample, as well as the consistency of the ground truth data, in order to avoid time consuming failures at training time. The generator is

also counting each class occurrence, which is optionally used during training for compensating under-sampled classes (parameter `class_weight` of method `Model.fit_generator()`). The image format is automatically derived from the extension, thus allowing for mixed image formats in a single folder. Instead of images, the generator can alternatively read pre-cached numpy arrays, which is useful to speed up the training of the final classifiers when using ensembling methods. Ground truth data is optional, so that the very same generator can be used also for predictions.

Our toolbox provides a configurable **Image Augmentation** class set. A `DataGenerator` reads image pixels through an `ImageProvider`. The latter is the top-level interface for a *Decorator* design pattern [6]. An ImageProvider gives information of how many images are available (`num_images()`), and on request returns one of them (`get_image(i)`). The only concrete implementation (`MultipleImageProvider`) takes a list of image paths in the constructor and returns them on a given resolution. The abstract subclass `ImageAugmenter` implements the ImageProvider interface and at the same time holds a reference to another image provider (the *decorated* instance). Subclasses of ImageAugmenter can augment the images returned by their `provider` member. For example, the `HFlipAugmenter` returns the double of the images of its provider; each input image is returned both as-it-is and horizontally flipped. Software developers can define custom image augmentation chains and register them in the `make_image_provider()` factory method, thus allowing for the association of mnemonic config IDs to complex configurations. As an example, the configuration named `hflip_rot24`, where each image is both flipped and rotated 24 times, realizing a 48x augmentation, is implemented by:

```
paths: List[str] = [...]  # A list of image paths
hflip_rot24_provider =
    # Rotates an image 24 times
    RotatedImageAugmenter(rot_steps=24,
        # Flip images horizontally
        provider=HFlipImageAugmenter(
            # Reads from disk and rescale
            provider=MultipleImageProvider(images_path=paths,
                                           resolution=(224,224))))
```

5 CONCLUSIONS

We described the requirements, the software design and the implementation of a software toolbox of reusable components for the configuration and deployment of deep convolutional neural networks. The toolbox has already been used in the domain of skin cancer classification (e.g., ISIC challenges [2, 3]), medical image concept labeling (ImageCLEFmedical 2020 [18]), and other applied AI and team collaboration projects. With this tool, we provide practitioners with no coding abilities with a software package to quickly refine (image) classification pipelines by tuning (hyper-)parameters and get feedback as quickly as possible, staying focussed on the machine learning problem at hand. On the other hand, developers can quickly implement new CNN-based solutions thanks to a software architecture of reusable components implementing most of the routines common to image analysis experiments. The API documentation is available at the Git repository [8]. We hope this inspires future developments towards the adoption of recognized software design practices to increase the portability and reusability of development efforts across applied AI projects in different domains.In future developments, we plan the integration of image segmentation architectures (e.g., U-Net [20]) using the same architectural principles presented above, i.e., through the introduction of an abstract `Segmenter` class.

ACKNOWLEDGEMENTS

This work has been supported by pAItient (BMG) and Ophthalmo-AI (BMBF).

REFERENCES

[1] Andrew Chen, Andy Chow, Aaron Davidson, Arjun DCunha, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Clemens Mewald, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Avesh Singh, Fen Xie, Matei Zaharia, Richard Zang, Juntai Zheng, and Corey Zumar. 2020. Developments in MLflow: A System to Accelerate the Machine Learning Lifecycle. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*. ACM, Portland OR USA, 1–4. https://doi.org/10.1145/3399579.3399867

[2] Noel Codella, Veronica Rotemberg, Philipp Tschandl, et al. 2019. Skin Lesion Analysis Toward Melanoma Detection 2018: A Challenge Hosted by the International Skin Imaging Collaboration (ISIC). *arXiv:1902.03368 [cs]* (Feb. 2019).

[3] Noel C. F. Codella, David Gutman, M. Emre Celebi, et al. 2018. Skin lesion analysis toward melanoma detection: A challenge at the 2017 International symposium on biomedical imaging (ISBI), hosted by the international skin imaging collaboration (ISIC). In *2018 IEEE 15th International Symposium on Biomedical Imaging (ISBI 2018)*. IEEE, Washington, DC, 168–172. https://doi.org/10.1109/ISBI.2018.8363547

[4] Christos Davatzikos, Saima Rathore, Spyridon Bakas, et al. 2018. Cancer imaging phenomics toolkit: quantitative imaging analytics for precision diagnostics and predictive modeling of clinical outcome. *Journal of Medical Imaging* 5, 01 (Jan. 2018), 1. https://doi.org/10.1117/1.JMI.5.1.011018

[5] Facebook Open Source. 2021. PyTorch. https://pytorch.org/.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.

[7] Eli Gibson, Wenqi Li, Carole Sudre, et al. 2018. NiftyNet: a deep-learning platform for medical imaging. *Computer Methods and Programs in Biomedicine* 158 (May 2018), 113–122. https://doi.org/10.1016/j.cmpb.2018.01.025

[8] DFKI Interactive Machine Learning group. 2021. TIML: a Toolkit for Interactive Machine Learning. https://github.com/DFKI-Interactive-Machine-Learning/TIML.

[9] Jupyter community. 2021. Jupyter Notebook. https://jupyter.org/.

[10] Keras Special Interest Group. 2021. Keras. https://keras.io/.

[11] Fredrik Lundh and Alex Clark. 2021. Pillow Imaging Library. https://pillow.readthedocs.io/en/stable/.

[12] Martín Abadi, Ashish Agarwal, Paul Barham, et al. 2015. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. https://www.tensorflow.org/

[13] MathWorks. 2021. Deep Learning Toolbox. https://www.mathworks.com/products/deep-learning.html.

[14] Michael Michalko. 2011. *Creative thinkering: Putting your imagination to work*. New World Library.

[15] NVIDIA. 2021. DIGITS. https://github.com/NVIDIA/DIGITS.

[16] Pallets. 2021. Flask's documentation. https://flask.palletsprojects.com/.

[17] Nick Pawlowski, Sofia Ira Ktena, Matthew C. H. Lee, Bernhard Kainz, Daniel Rueckert, Ben Glocker, and Martin Rajchl. 2017. DLTK: State of the Art Reference Implementations for Deep Learning on Medical Images. *arXiv:1711.06853 [cs]* (Nov. 2017).

[18] Obioma Pelka, Christoph M Friedrich, Alba García Seco de Herrera, and Henning Müller. 2020. Overview of the ImageCLEFmed 2020 Concept Prediction Task: Medical Image Understanding. In *CLEF2020 Working Notes (CEUR Workshop Proceedings)*. CEUR-WS.org, Thessaloniki, Greece.

[19] Vitali Petsiuk, Abir Das, and Kate Saenko. 2018. RISE: Randomized Input Sampling for Explanation of Black-box Models. In *Proceedings of the British Machine Vision Conference (BMVC)*.

[20] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi (Eds.). Vol. 9351. Springer International Publishing, Cham, 234–241. https://doi.org/10.1007/978-3-319-24574-4_28

[21] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-CAM: Visual Explanations From Deep Networks via Gradient-Based Localization. In *The IEEE International Conference on Computer Vision (ICCV)*.

[22] Daniel Sonntag. 2008. On Introspection, Metacognitive Control and Augmented Data Mining Live Cycles. *CoRR* abs/0807.4417 (2008). arXiv:0807.4417 http://arxiv.org/abs/0807.4417

[23] The Apache Software Foundation (ASF). 2021. MXNet. https://mxnet.apache.org/. https://mxnet.apache.org/
[24] The Kubeflow Authors. 2021. Kubeflow. https://www.kubeflow.org/.
[25] Panagiotis Tzirakis, Stefanos Zafeiriou, and Bjorn W. Schuller. 2018. End2You – The Imperial Toolkit for Multimodal Profiling by End-to-End Learning. *arXiv:1802.01115 [cs]* (Feb. 2018). http://arxiv.org/abs/1802.01115
[26] Jan Zacharias, Michael Barz, and Daniel Sonntag. 2018. A Survey on Deep Learning Toolkits and Libraries for Intelligent User Interfaces. *arXiv:1803.04818 [cs]* (March 2018). http://arxiv.org/abs/1803.04818