Productive Reproducible Workflows for DNNs: A Case Study for Industrial Defect Detection

Perry Gibson, José Cano
School of Computing Science, University of Glasgow, UK

***Abstract*—As Deep Neural Networks (DNNs) have become an increasingly ubiquitous workload, the range of libraries and tooling available to aid in their development and deployment has grown significantly. Scalable, production quality tools are freely available under permissive licenses, and are accessible enough to enable even small teams to be very productive. However within the research community, awareness and usage of said tools is not necessarily widespread, and researchers may be missing out on potential productivity gains from exploiting the latest tools and workflows. This paper presents a case study where we discuss our recent experience producing an end-to-end artificial intelligence application for industrial defect detection. We detail the high level deep learning libraries, containerized workflows, continuous integration/deployment pipelines, and open source code templates we leveraged to produce a competitive result, matching the performance of other ranked solutions to our three target datasets. We highlight the value that exploiting such systems can bring, even for research, and detail our solution and present our best results in terms of accuracy and inference time on a server class GPU, as well as inference times on a server class CPU, and a Raspberry Pi 4.**

***Index Terms*—deep learning, docker, defect detection, pytorch, reproducibility, bonseyes**

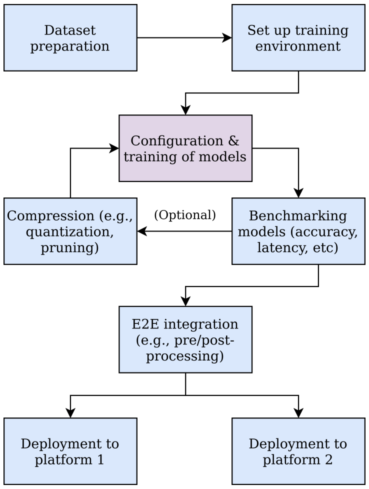
Dataset preparation
Set up training environment
Configuration & training of models
Compression (e.g., quantization, pruning)
(Optional)
Benchmarking models (accuracy, latency, etc)
E2E integration (e.g., pre/post-processing)
Deployment to platform 1
Deployment to platform 2

Fig. 1: Simplified representation of the workflow of developing a DNN application. Note that only one step directly involves choosing model architectures and training.

I. Introduction

Deep Learning is becoming a common component within applications for a number of domains, from computer vision [1]–[4], natural language processing [5], [6], scientific computing [7]–[9], and many more. To aid in the development of these applications, there are a wide range of libraries and tools, such as deep learning frameworks including PyTorch [10], TensorFlow [11], and MXNet [12]. However, beyond the creation of models themselves, there are a number of supplementary steps for creating applications that leverage Deep Neural Networks (DNNs). As shown in Figure 1, many of the steps involved in developing a DNN application do not directly involve DNN models (e.g., dataset preparation, setting up development and deployment environments). Researchers and industry practitioners generally take care when designing (or choosing) and training their models. However, many of these complementary steps may be overlooked or implemented in a more ad-hoc manner, especially within the research community. However, we argue that there are advantages to leveraging the growing set of tools and workflows for end-to-end DNN application development in research, even if the end goal is not production ready deployment. For instance, ensuring that datasets are in a consistent and easily usable format for DNN training, and that this transformation process is reproducible. Or when evaluating on more than one hardware platform, ensuring that the software environment is correctly set up with all of the required software dependencies, and ideally in a way which is reproducible. For the latter example, continuous integration (CI) and continuous deployment (CD) pipelines can fit this role, however can be time consuming and tedious to set up from scratch. Thus, in this position paper we discuss key tools which increased our productivity in developing an artificial intelligence (AI) application for visual industrial defect detection, and how integrating such tools into DNN development workflows can help both researchers and industry practitioners.

The contributions of this paper include the following:

- We describe several tools which we have used to increase the productivity of our DNN research, including PyTorch Lightning [13] and templates from the Bonseyes Marketplace Platform [14].
- We highlight how these tools were valuable to us in a case study for visual industrial defect detection, and how we used them to develop an end-to-end solution.
- We describe the three datasets we used to tackle our problem, and the models we trained. We present our best performing models in terms of accuracy and inference time, using an Nvidia A100 as our main evaluation platform, as well as presenting results on an x86 CPU, and a Rapsberry Pi 4.

II. Core DNN development/deployment tools

Achieving high accuracy on a target problem is generally the main motivating goal of any machine learning project, while latency becomes more important in systems research and when deploying to constrained devices in industrial use-cases. Machine learning researchers can explore a wide range of design choices, for example varying the neural architecture, changing aspects of the training process (e.g., learning rate, optimizer), and applying varying types of data augmentation.

However, although these aspects of the solution are pivotal, it is important that the supporting infrastructure to help solve the problem is not overlooked or chosen as an afterthought. For example: how is the raw training data to be translated into a format that the DNN can understand? Can this be easily reproduced? What is the software environment that a DNN will be trained in, and will it still work in future when packages are updated? What platforms will the DNN be deployed to, and how will this deployment be managed? These questions are important, thus in this paper we list a number of open source tools that we leveraged for our case study (discussed in Section III), and the value they can bring for deep learning application development. We do not list the most obvious and ubiquitous tools, for example PyTorch [10], which is the most popular deep learning framework used in research [15], or version control systems such as git. Instead we focus on systems and tools which we believe filled a niche that greatly increased our productivity in carrying out our case study discussed in Section III, and may not necessarily be well known or commonly used within the research community. In particular, instrumental to our success were systems and templates provided by the Bonseyes Marketplace [14], which were designed with these goals in mind [1]. The core tools we leveraged were as follows:

Segmentation Models PyTorch (SMP) [17]: a library which builds on top of PyTorch, and eases the development of DNN applications for computer vision problems related to image segmentation. Since our case study is for industrial visual defect detection, which is a sub-problem of image segmentation, exploiting this library enabled us to produce a range of solutions more quickly than if we had created our own solution from scratch. Frameworks and libraries for specific problem spaces, which build on top of lower level DNN frameworks (e.g., PyTorch and TensorFlow [11]) are becoming more popular, and researchers should be aware of them when approaching a new problem domain, since they may provide shortcuts to a solution, or at least provide a convenient set of benchmarks to compare against. As well as SMP for image segmentation, other examples of higher level DNN libraries include the TensorFlow Object Detection API [18] for object detection, and HuggingFace's Transformers library [19] for natural language processing and other tasks suited for Transformer-based [20] architectures.

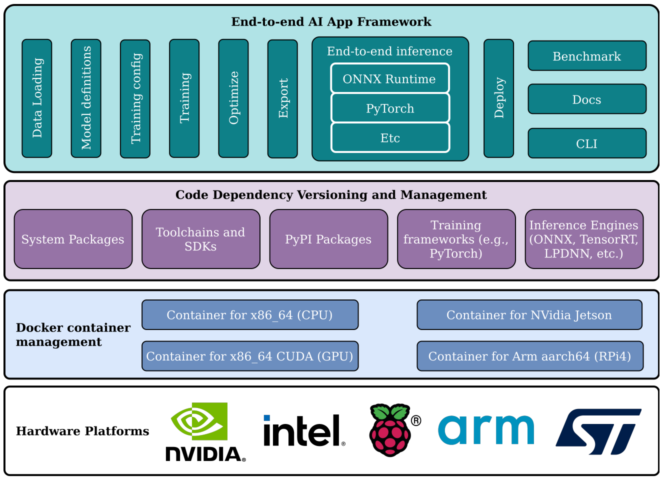
End-to-end AI App Framework
Data Loading
Model definitions
Training config
Training
Optimize
Export
End-to-end inference
ONNX Runtime
PyTorch
Etc
Deploy
Benchmark
Docs
CLI
Code Dependency Versioning and Management
System Packages
Toolchains and SDKs
PyPI Packages
Training frameworks (e.g., PyTorch)
Inference Engines (ONNX, TensorRT, LPDNN, etc.)
Docker container management
Container for x86_64 (CPU)
Container for x86_64 CUDA (GPU)
Container for NVidia Jetson
Container for Arm aarch64 (RPi4)
Hardware Platforms
NVIDIA.
intel.
arm

Fig. 2: Simplified representation of the features provided by the Bonseyes AI Asset template system, adapted (with permission) from the AI Asset Generator documentation [21].

PyTorch Lightning [13]: a wrapper library for PyTorch which reduces the amount of boilerplate code for defining model training. In addition, it also provides utilities for pruning and quantization, which are designed to be simpler to use than those provided by normal PyTorch alone. It also boasts the feature of enabling training across on multiple-GPUs, TPUs (Tensor Processing Units), CPUs, and IPUs (Intelligence Processing Units) without requiring changes in the code.

Bonseyes Datatools: a codebase template that allows developers to produce tools which convert raw data into a user-defined standard dataset format, including utilities for exploratory data analysis, visualization, and dataset tagging and versioning. As well as ensuring that dataset preparation is more reproducible, a secondary purpose of a datatool is to separate initial dataset preparation from the model training code, which aids re-usability for future projects.

Bonseyes AI Assets: a codebase template that aids developers in producing a tool for both training and deployment of DNNs. An AI Asset encapsulates all code and dependencies required for their solution, with an overview of its features shown in Figure 2. Datasets generated from user-defined Bonseyes datatools can be easily mounted on the AI Asset, simplifying the data loading process. Code for common activities such as benchmarking, report generation, and model conversion and inference using PyTorch, ONNX Runtime [22], and TensorRT [23] is provided, with support for more inference engines in development. The design philosophy is to provide as much boilerplate code as possible without forcing developers to make design choices they do not want to. Developers can use their deep learning framework of choice, and include any software dependencies they require. The motivation for having all of the tools in a single environment is so that it is easier to investigate performance degradation over the whole pipeline, even when deploying on other platforms. The trade-off here is increased disk storage for libraries.

[1] More information on the Bonseyes suite of tools can be found in the Bonseyes Platform documentation [16].

Bonseyes AI Asset CI/CD pipeline: Continuous Integration (CI) and Continuous Deployment (CD) are software engineering principles whereby code is regularly subjected to automated testing, with CD being particularly focused on ensuring that code works in a deployment environment. Although valuable, setting up these pipelines can be a very time consuming task and may not be a high priority for researchers who are focused on validating their ideas rather than producing production ready systems. However, the Bonseyes AI Asset includes a predefined CI/CD pipeline, which means that developers can reap the benefits of having their development and deployment environments be independently tested with each code commit without requiring the high initial set-up costs. Users must provide an x86-based server featuring an NVidia GPU and run a setup script which allows the server to receive and test new code commits, automatically testing for four platforms (x86+CUDA, x86-only, Nvidia Jetson, and Raspberry Pi, as seen in Figure 2). QEMU [24] emulation is used to test the Arm-based Jetson and Raspberry Pi platforms on the server, with Docker containers for each platform being generated and available for immediate deployment at the end of the process. The testing process is automatic, with developers being sent an email if their pipeline fails.

III. Case Study

As discussed in Section I, our goal was to produce an end-to-end AI application for the problem of industrial visual defect detection. In essence, the task is to take visual input (e.g., from a camera) of some industrial product (e.g., textiles, rolled steel, printed circuit boards, etc) and identify if there are any defects on the product (e.g., scratches, blemishes, smudges, etc). This information can then be used to improve product quality, and reduce waste. An example of this can be seen in Figure 3, where in Figure 3a we can see a photograph from some industrial product, and in Figure 3b we have a human annotated label of where in the image a defect is, shown in red. To solve this problem effectively we were required to have our data in a consistent format (Section III-A), have models which can efficiently process said data (Section III-B), and have other parts of our development and deployment workflow be as supportive as possible for our workflow (Section III-C).

A. Datasets and data processing

For our case study, we used three publicly available datasets to train and evaluate our system: DAGM2007 [25], KolektorSDD [26], and KolektorSDD2 [27]. Below is a brief overview of the three datasets:

- **DAGM2007** contains grayscale images for 10 classes of artificially generated patterns, with around 8% of them containing defects. The classes were designed to mirror real world problems, with 1150 images per class, and images of size 512 × 512.
- **KolektorSDD** is a small dataset of grayscale images collected from a real industrial environment. There are only 399 images, with around 8% of them containing defects, and the standard size of images for the dataset being 512 × 1408. A sample image from KolektorSDD is shown in Figure 3.
- **KolektorSDD2** is a larger dataset of color images of size 230×630. There are 3335 images with around 9% of the images containing defects.

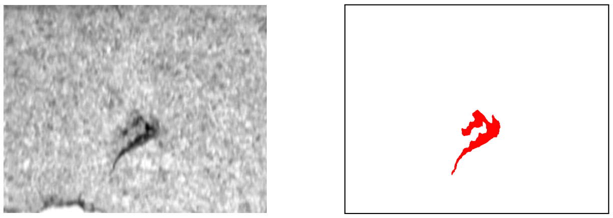
(a) Image data (b) Defect label

Fig. 3: Sample element from the KolektorSDD dataset [26], with the image data (a), and pixel-wise mask of defective regions highlighted in red (b).

The raw data of the three datasets are stored in different directory hierarchies, and represent their annotations in varying formats. Thus, we standardize our datasets to a common format, which simplifies our training and evaluation code later in the project. This is the purpose of the *Bonseyes Datatool* template, which provides utilities to create a conversion pipeline for raw data. We created three datatools, one for each of our datasets, which all converged on a common format. We represent a given dataset element with in the following format:

- Path to image data.
- Compressed matrix representing the defect annotation.
- The classification: defective or non-defective.

The datatool represents all of this data in a standardized JSON format, with raw image data stored in a simple directory hierarchy. Once the datatools have converted their respective datasets, we can then develop our training and evaluation pipeline. To achieve this, we leverage the Bonseyes AI Asset system, which as discussed in Section II provides a set of utilities and packages for developing AI applications. To ensure a consistent software environment we develop our solution in a Docker container provided by the AI Asset, adding any dependencies we require to the AI Asset's dependency file.

B. Model architectures and training

Bonseyes AI Assets do not enforce any strict requirements on how models are developed, simply providing a template to follow. Thus, for our models we leverage the SMP library [3], which provides model architectures for image segmentation. We can formulate surface defect detection as an image segmentation problem by representing the annotations (e.g., the ones seen in Figure 3b) as a mask matrix of 1s and 0s for defective and non-defective pixels respectively. Then when training we attempt to produce an output matrix which has maximum similarity with this matrix. To measure this similarity, we use the common metric of intersection-over-union (IoU), as shown in Figure 4, with an IoU-score of 0.0

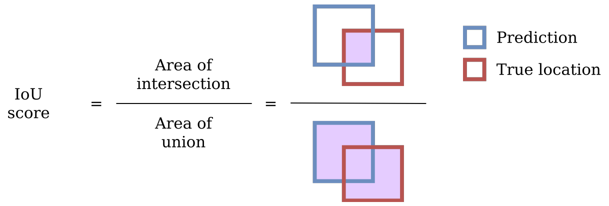

Fig. 4: Overview of the IoU-score which we optimize when training our DNN models.

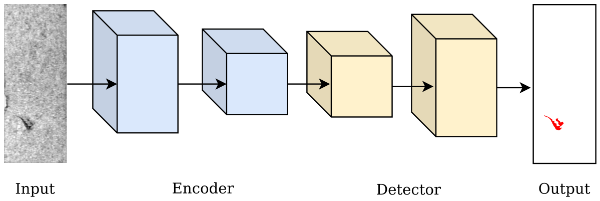

Fig. 5: Simplified example of an Encoder-Detector DNN architecture.

meaning that we predicted no defective pixels correctly, and 1.0 meaning that we predicted every defective pixel correctly. We leverage the fact that our three datasets are in the same format (as described in Section III-A) to simplify the definition of our models, since we only need to support one data format.

Using the SMP library, DNN model architectures are defined with two main components: an *encoder* model which processes raw image data, and a model which takes the output of the encoder to produce the image segmentation, which we refer to as the *detector*. Thus in this paper we call this an encoder-detector architecture, as shown in Figure 5. An advantage of this architecture is that for the encoder we can leverage pretrained ImageNet [28] models, such as ResNet50 [1], MobileNetV2 [29], and EfficientNet [30]. This can significantly reduce our training costs and the amount of training data we require, since our models do not need to learn from scratch how to identify low-level image features (e.g., edges, corners, textures, etc). The encoder model skips its final ImageNet classification layers, passing intermediate activations to the detector model. This means that data passed to the detector is easier to process than raw image data. SMP provides a number of state-of-the-art architectures we can choose for the detector, including Unet++ [31], MAnet [32], LinkNet [33], PAN [34], and more. In total, SMP can provide over 1000 unique encoder-detector pairs, and in our evaluation in Section IV we train a subset (62 models) and report on our best performing models.

Using a higher level library such as SMP rather than building our own architecture from scratch, or using a single model implementation published alongside a research paper (e.g., LinkNet, MANet) significantly increased our productivity, since we did not know ahead of time which architecture would provide the best performance, and having a tool such as SMP which allowed us to easily switch architectures meant we only had to integrate one codebase rather than several. In addition, for training our models we leveraged the PyTorch Lightning library [13], which further reduced the amount of boilerplate code we had to write for configuring the training procedures for our models. For future work, PyTorch Lightning will also reduce the effort required to further compress our models using techniques such as pruning and quantization, which in our experience can be more difficult to do when only using the utilities provided by the base PyTorch library.

Our DNN models only provide a mask matrix, hence to provide a final classification we apply a post-processing step integrated into the Bonseyes AI Asset *algorithm* class (which helps ensure that pre- and post-processing steps are performed consistently across frameworks). The post-processing step takes a user-defined threshold (e.g., 1%) and classifies the image as being defective if the proportion of pixels marked as defective by the model is greater than or equal to the threshold. Thus, we can measure the quality of our solution using both classification accuracy and IoU score, however it is sufficient to train our models using the IoU score alone.

C. Deployment, and complementary components

Generally DNNs are trained on HPC servers featuring GPUs, however when they are deployed they may also run on more constrained edge devices such as IoT devices, smartphones, drones, VR headsets, etc. When deploying on a new platform, tasks such as managing package dependencies and setting up the new environment can be very time consuming. Fortunately, as discussed in Section II, Bonseyes AI Assets contain a CI/CD pipeline to ease this deployment. Developers specify the versions of packages they are using for development purposes, and whenever they push a code commit the CI/CD pipeline builds the whole project for x86, Nvidia Jetson, and Raspberry Pi platforms. Occasionally, a package version for the x86 platform may not be available for another platform. In this case, developers will be sent an email about the issue and be able to specify a different package version for the platform, on in extreme cases a process to build the correct version of the package. When the developer is ready to test deployment on their target device, they need only to download the latest version of the Docker image for their target platform. Running CI/CD on a server with ISA emulation reduces the time required to setup a new platform, especially when steps like building packages can be more time consuming when compiling natively on constrained platforms such as the Raspberry Pi. This is especially valuable for machine learning researchers for whom testing on constrained edge devices may be considered merely supplementary results: not worth a large amount of effort, but whose inclusion will improve any evaluation they provide.

When running inference, deep learning frameworks such as PyTorch may not be the most optimal for running inference (since they are focused on training). Thus, Bonseyes AI Assets also include support for ONNX Runtime and TensorRT inference, with support for more backends in development.

IV. Results

In this section, we discuss our results from training a range of models separately on our 3 datasets: DAGM2007, KolektorSDD, and KolektorSDD2. In total we trained 62 models, and we include our tables of our 5 top performing models in terms of accuracy and inference time. For all models, we evaluate them using a validation dataset, and for DAGM2007 and KolektorSDD2 we use the officially provided held-out test datasets. For KolektorSDD, the dataset is too small for a held out test set (with only 52 examples of defects in the whole dataset). Therefore, we evaluate it using the test set of KolektorSDD2, from which we expect to see a performance degradation due to the images having different characteristics.

We report the inference times in three settings: 1) PyTorch running on an NVidia A100 GPU, 2) ONNX Runtime on a cloud-based Intel Broadwell series x86 CPU featuring 22 cores, and 3) ONNX Runtime on the CPU of a Rapsberry Pi 4 Model B. Note that we were unable to run models using EfficientNet-based encoders in the version of ONNX Runtime we tested due its lack of support for the models' "Swish" function. Hence we represent the inference time for those models in this setting with a '-'.

Tables I, II, and III show our top 5 models in terms of test set accuracy, as well as their inference time across our three settings. Across every model we trained, our median test set classification accuracies were 99.8%, 89.0%, and 97.55% for DAGM2007, KolektorSDD, and KolektorSDD2 respectively. The lower accuracy for KolektorSDD is expected, given that 1) our evaluation methodology tests on a completely different dataset (KolektorSDD2), and 2) how few examples are in the dataset relative to the other models. We observe that across all of our DNN models, the mean validation set accuracy for our KolektorSDD models is 98.8% (and was 99.7% and 97.4% for DAGM2007 and KolektorSDD2 respectively) which suggests that the models do learn well, however using KolektorSDD2 as a test set is unfair as the dataset is too different. Comparing against other published approaches for our 3 datasets, as ranked by Papers with Code [35]–[37], we observe that our best models get accuracies matching other highly ranked solutions. We note that models using EfficientNetB4 as the encoder architecture appear disproportionately in the top-5 models in terms of accuracy for the 3 datasets, and there is no clear winner for detector architectures, suggesting that the choice of encoder architecture has the greatest influence on final accuracy.

Tables IV, V, and VI show our top 5 models in terms of inference time on the NVidia A100 GPU, along with their accuracies, and inference time on other devices. On the A100, our models vary in inference time between 5.9ms and 34.0ms running on an NVidia A100 with PyTorch. We note that models with MobileNetV2 and ResNet34 as their encoder architectures are the only models that are in the top 5 in terms of inference time on the A100 across our three datasets, suggesting that as well as accuracy the encoder is the most important feature to consider for inference time. We note that our fastest models see accuracy penalties when compared to their counterparts in Tables I, II, and III. However, several of our fast models also get high accuracies. For example, in Table IV, MobileNetV2-Unet (Rank 2) and MobileNetV2-Pan (Rank 5) get nearly perfect accuracy on the test set, and for Table VI ResNet34-LinkNet (Rank 1) and MobileNetV2-Unet (Rank 2) get accuracies within 0.1% of the fifth best performing model in Table III.

We observe that the relative inference times of models and their scaling does not necessarily stay the same between settings (i.e., PyTorch on the A100 GPU, ONNX Runtime the x86 CPU, ONNX Runtime on the Raspberry Pi 4). For example in Table IV MobileNetV2-Pan (Rank 5) is almost $3.3\times$ faster than ResNet34-Unet (Rank 4) on x86+ONNX Runtime, whereas the models have almost identical inference times on the A100 using PyTorch, with a similar discrepancy seen on the Raspberry Pi 4. This tells us that relative inference time performance is not necessarily consistent between frameworks and devices. In future work, we will explore in greater detail these performance trade-offs and variances, how to make the best choice of model for a given hardware platform, and investigate further across-stack DNN optimizations [38] such as grouped convolutions [39] and quantization. Deep learning compilers such as TVM [40] and IREE [41] provide another dimension of DNN optimization, with approaches such as auto-tuning [42], auto-scheduling [43], and related systems [44] potentially bringing further performance improvements. Integration of these systems within an AI Asset could provide a more straightforward way to reap their benefits.

V. Conclusion

In conclusion, there are a wide range of tools available to increase the productivity of both deep learning engineers and researchers. Our paper highlighted that features such a continuous integration and deployment, which are rarely a priority for researchers, can bring a number of benefits and require little effort to set up if researchers embrace predefined workflows such as the open source tools provided by the Bonseyes Marketplace [14]. In addition, there are emerging higher level deep learning libraries (such as SMP [17]) that can improve productivity for specific domains, that should be exploited where possible. We discussed our experience using the latest tools to produce a solution for the problem of industrial defect detection, presenting results on an HPC server and a Raspberry Pi 4. For future work, we will seek to continue to use these tools and principles to improve the quality of our own research artifacts, and explore the utilities provided by both PyTorch Lightning and Bonseyes AI Assets for model compression such as pruning and quantization.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme within the framework of the BonsAPPs Project funded under grant agreement No 101015848. We thank Miguel De Prado for his insightful review and suggestions.

TABLE I: Top 5 models ranked by accuracy for DAGM2007 along with their inference times.

Rank	Test		Validation		Arch (Encoder-Detector)	Inf. time (ms)		
	Acc (%)	IoU	Acc (%)	IoU		A100	x86	RPi4
1	100.0	0.941	100	0.976	EfficientNetB4-LinkNet	18.9	-	-
2	100.0	0.939	100	0.975	EfficientNetB4-Pan	21	-	-
3	100.0	0.937	99.9	0.977	EfficientNetB4-MANet	23.5	-	-
4	100.0	0.917	100	0.971	MobileNetV2-Unet++	8.2	116	3351
5	100.0	0.914	99.9	0.967	MobileNetV2-Pan	6.9	36.6	853

TABLE II: Top 5 models ranked by accuracy for KolektorSDD along with their inference times.

Rank	Test		Validation		Arch (Encoder-Detector)	Inf. time (ms)		
	Acc (%)	IoU	Acc (%)	IoU		A100	x86	RPi4
1	90.2	0.892	100	0.931	InceptionV4-Unet++	34	1427.6	73372
2	90	0.363	98.8	0.926	MobileNetV2-Pan	8.6	58.9	2119
3	89.8	0.79	98.8	0.93	ResNet34-Unet++	11.6	608.0	30656
4	89.7	0.841	100	0.944	EfficientNetB4-MANet	29	-	-
5	89.6	0.892	98.8	0.895	EfficientNetB4-Unet	25.1	-	-

TABLE III: Top 5 models ranked by accuracy for KolektorSDD2 along with their inference times.

Rank	Test		Validation		Arch (Encoder-Detector)	Inf. time (ms)		
	Acc (%)	IoU	Acc (%)	IoU		A100	x86	RPi4
1	98.1	0.842	98.7	0.857	InceptionV4-Unet	23.7	152.3	5462
2	98	0.952	98.1	0.947	EfficientNetB4-Unet++	25	-	-
3	98	0.948	97.2	0.944	EfficientNetB4-Pan	21.8	-	-
4	97.9	0.882	97.9	0.879	EfficientNetB4-Unet	20.2	-	-
5	97.8	0.94	97.9	0.939	EfficientNetB4-LinkNet	21	-	-

TABLE IV: Top 5 models ranked by inference time on the NVidia A100 for DAGM2007 along with their accuracies, and inference times on other platforms.

Rank	Test		Validation		Arch (Encoder-Detector)	Inf. time (ms)		
	Acc (%)	IoU	Acc (%)	IoU		A100	x86	RPi4
1	94.1	0.848	97.4	0.941	MobileNetV2-LinkNet	5.9	43.3	766
2	99.9	0.908	99.9	0.968	MobileNetV2-Unet	6.4	73.4	2514
3	94.9	0.856	97.2	0.941	ResNet34-Pan	6.6	102.1	4459
4	94.5	0.844	97.1	0.939	ResNet34-Unet	6.8	120.2	4948
5	100	0.914	99.9	0.967	MobileNetV2-Pan	6.9	36.6	852

TABLE V: Top 5 models ranked by inference time on the NVidia A100 for KolektorSDD along with their accuracies, and inference times on other platforms.

Rank	Test		Validation		Arch (Encoder-Detector)	Inf. time (ms)		
	Acc (%)	IoU	Acc (%)	IoU		A100	x86	RPi4
1	89	0.696	100	0.899	MobileNetV2-LinkNet	7.9	80.2	1914
2	89	0.154	96.2	0.918	ResNet34-LinkNet	8	258.1	8757
3	64.4	0.002	98.8	0.924	ResNet34-Unet	8	256.8	12431
4	89	0.89	98.8	0.903	ResNet34-DeepLabV3	8.1	642.0	40772
5	89	0.89	98.8	0.925	ResNet34-Pan	8.2	227.1	12221

TABLE VI: Top 5 models ranked by inference time on the NVidia A100 for KolektorSDD2 along with their accuracies, and inference times on other platforms.

Rank	Test		Validation		Arch (Encoder-Detector)	Inf. time (ms)		
	Acc (%)	IoU	Acc (%)	IoU		A100	x86	RPi4
1	97.7	0.819	97.4	0.817	ResNet34-LinkNet	6	71.4	1908
2	97.7	0.94	97.6	0.937	MobileNetV2-Unet	6.1	60.0	1539
3	95.2	0.887	93.1	0.863	MobileNetV2-LinkNet	6.2	32.1	441
4	96.1	0.924	95.5	0.907	ResNet34-Unet	6.4	76.7	2927
5	97.4	0.937	97.9	0.939	MobileNetV2-DeepLabV3	6.6	106.8	4552

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 770–778.

[2] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, “Densely Connected Convolutional Networks,” in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 2261–2269.

[3] J. Long, E. Shelhamer, and T. Darrell, “Fully Convolutional Networks for Semantic Segmentation,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 3431–3440.

[4] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[5] Y. Kim, “Convolutional Neural Networks for Sentence Classification,” in *The Conference on Empirical Methods in Natural Language Processing*, 2014.

[6] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, “A Convolutional Neural Network for Modeling Sentences,” in *In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, 2014.

[7] T. Tröster, C. Ferguson, J. Harnois-Déraps, and I. G. McCarthy, “Painting with Baryons: Augmenting N-body Simulations with Gas Using Deep Generative Models,” *Monthly Notices of the Royal Astronomical Society: Letters*, vol. 487, no. 1, pp. L24–L29, 2019.

[8] J. N. Kutz, “Deep Learning in Fluid Dynamics,” *Journal of Fluid Mechanics*, vol. 814, pp. 1–4, Mar. 2017.

[9] A. C. Mater and M. L. Coote, “Deep Learning in Chemistry,” *Journal of Chemical Information and Modeling*, vol. 59, no. 6, pp. 2545–2559, Jun. 2019.

[10] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang *et al.*, “Automatic Differentiation in PyTorch,” in *NeurIPS Autodiff Workshop*, 2017.

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis *et al.*, “TensorFlow: A System for Large-Scale Machine Learning,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’16. USA: USENIX Association, Nov. 2016, pp. 265–283.

[12] T. Chen, M. Li, Y. Li, M. Lin, N. Wang *et al.*, “MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems,” *arXiv:1512.01274 [cs]*, Dec. 2015.

[13] W. Falcon and The PyTorch Lightning team, “PyTorch Lightning,” 2019. [Online]. Available: https://github.com/PyTorchLightning/pytorch-lightning

[14] T. Llewellynn, S. Koller, G. Goumas, P. Leitner, G. Dasika *et al.*, “BONSEYES: Platform for Open Development of Systems of Artificial Intelligence: Invited paper,” 2017, pp. 299–304.

[15] Papers with Code, “Papers with Code: Trends,” 2022. [Online]. Available: https://paperswithcode.com/trends

[16] Bonseyes, “Bonseyes Documentation,” 2022. [Online]. Available: https://beta.bonseyes.com/doc/

[17] P. Yakubovskiy, “Segmentation Models PyTorch,” 2020.

[18] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara *et al.*, “Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors,” in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul 2017, pp. 3296–3297.

[19] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue *et al.*, “HuggingFace’s Transformers: State-of-the-art Natural Language Processing,” *arXiv:1910.03771*, Jul. 2020.

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones *et al.*, “Attention is All You Need,” in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus *et al.*, Eds., 2017, pp. 5998–6008.

[21] Bonseyes Association, “Bonseyes AIAssetContainerGenerator · GitLab,” 2022. [Online]. Available: https://gitlab.com/bonseyes/artifacts/assets/aiasset_container_generator

[22] ONNX Runtime developers, “ONNX Runtime,” https://onnxruntime.ai/, 2021.

[23] Nvidia Corporation, “TensorRT: Programmable inference accelerator.” 2016. [Online]. Available: https://developer.nvidia.com/tensorrt

[24] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC)*, Apr. 2005, p. 41.

[25] D. Weimer, B. Scholz-Reiter, and M. Shpitalni, “Design of Deep Convolutional Neural Network Architectures for Automated Feature Extraction in Industrial Inspection,” *CIRP Annals*, vol. 65, no. 1, pp. 417–420, Jan. 2016.

[26] D. Tabernik, S. Šela, J. Skvarč, and D. Skočaj, “Segmentation-Based Deep-Learning Approach for Surface-Defect Detection,” *Journal of Intelligent Manufacturing*, May 2019.

[27] J. Božič, D. Tabernik, and D. Skočaj, “Mixed Supervision for Surface-defect Detection: from Weakly to Fully Supervised Learning,” *Computers in Industry*, 2021.

[28] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh *et al.*, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, 2015.

[29] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “MobilenetV2: Inverted Residuals and Linear Bottlenecks,” in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[30] M. Tan and Q. Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks,” in *International Conference on Machine Learning*, May 2019, pp. 6105–6114.

[31] Z. Zhou, M. M. Rahman Siddiquee, N. Tajbakhsh, and J. Liang, “UNet++: A Nested U-Net Architecture for Medical Image Segmentation,” in *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support*, ser. Lecture Notes in Computer Science, 2018, pp. 3–11.

[32] T. Fan, G. Wang, Y. Li, and H. Wang, “MA-Net: A Multi-Scale Attention Network for Liver and Tumor Segmentation,” *IEEE Access*, vol. 8, pp. 179 656–179 665, 2020.

[33] A. Chaurasia and E. Culurciello, “LinkNet: Exploiting Encoder Representations for Efficient Semantic Segmentation,” *arXiv:1707.03718*, Jun. 2017.

[34] H. Li, P. Xiong, J. An, and L. Wang, “Pyramid Attention Network for Semantic Segmentation,” *arXiv:1805.10180 [cs]*, Nov. 2018.

[35] Papers with Code, “Papers with Code - DAGM2007 Benchmark (Defect Detection),” 2022. [Online]. Available: https://paperswithcode.com/sota/defect-detection-on-dagm2007

[36] ——, “Papers with Code - KolektorSDD Benchmark (Defect Detection),” 2022. [Online]. Available: https://paperswithcode.com/sota/defect-detection-on-kolektorsdd

[37] ——, “Papers with Code - KolektorSDD2 Benchmark (Defect Detection),” 2022. [Online]. Available: https://paperswithcode.com/sota/defect-detection-on-kolektorsdd2

[38] J. Turner, J. Cano, V. Radu, E. J. Crowley, M. O’Boyle *et al.*, “Characterising Across-Stack Optimisations for Deep Convolutional Neural Networks,” in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2018, pp. 101–110.

[39] P. Gibson, J. Cano, J. Turner, E. J. Crowley, M. O’Boyle *et al.*, “Optimizing Grouped Convolutions on Edge Devices,” in *2020 IEEE 31st International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2020, pp. 189–196.

[40] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan *et al.*, “TVM: An automated end-to-end optimizing compiler for deep learning,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Oct. 2018, pp. 578–594.

[41] H.-I. C. Liu, M. Brehler, M. Ravishankar, N. Vasilache, B. Vanik *et al.*, “TinyIREE: An ML Execution Environment for Embedded Systems from Compilation to Deployment,” May 2022.

[42] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau *et al.*, “Learning to Optimize Tensor Programs,” in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi *et al.*, Eds. Curran Associates, Inc., 2018, pp. 3393–3404.

[43] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu *et al.*, “Ansor: Generating High-Performance Tensor Programs for Deep Learning,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 863–879.

[44] P. Gibson and J. Cano, “Reusing Auto-Schedules for Efficient DNN Compilation,” no. arXiv:2201.05587, Jan. 2022.