Reproducibility Companion Paper: Comparative Evaluation of Deep Learning Architectures and Training Strategies for Diabetic Retinopathy Classification

Mohamed Lamine BenHabirech[1], Mourad Belhadj[1], Dalila Tamzalit[2], and Oussama Aiadi[1]

[1] Faculty of New Information and Communication Technologies (FNTIC), University of Kasdi Merbah Ouargla, Ouargla, Algeria
benhabirech.mohammedlamine@univ-ouargla.dz
belhadj.mourad@univ-ouargla.dz
aiadi.oussama@univ-ouargla.dz
[2] Nantes University, Nantes, France
dalila.tamzalit@univ-nantes.fr

Abstract. This companion paper accompanies our ICPR 2026 contribution *"Comparative Evaluation of Deep Learning Architectures and Training Strategies for Diabetic Retinopathy Classification"* and targets Track 2 (Reproducible Research Results) of the RRPR 2026 workshop. Rather than reporting new experimental findings, we document the reproducibility properties of the accompanying open-source toolkit, `dr_benchmark`, which implements the six replicated literature methods and the four-architecture scratch-versus-pretrained ablation study discussed in the main paper. We describe the installation procedure, the algorithmic implementation of each model family, the concrete effect of implementation-level parameters on result quality, guidance for integrating the toolkit's components into other training frameworks, and the limitations we are aware of together with planned improvements. Source code, a frozen Zenodo release, and instructions to reproduce every table in the main paper are publicly available under the MIT license.

Keywords: Reproducible research · Diabetic retinopathy · Deep learning · Transfer learning · Software reproducibility

1 Introduction

Reproducibility of deep learning results in medical imaging is hampered less by algorithmic ambiguity than by the accumulation of small, undocumented implementation choices: how a dataset is split, which random seed governs which stage of the pipeline, whether a classical machine learning head is checkpointed together with the neural backbone that feeds it, and so on. Our ICPR 2026 paper [2] compared seven transfer-learning architectures and six literature methods

for diabetic retinopathy (DR) classification across four public datasets (APTOS 2019, IDRiD, DDR, Messidor-2), and separately quantified the effect of ImageNet pretraining versus training from scratch on four backbones (VGG16, ResNet50, InceptionV3, AlexNet).

This companion paper follows the RRPR 2026 Track 2 guidance for papers that describe the reproducible-research qualities of an already-accepted ICPR submission. We use the four recommended axes for such a companion paper as the structure of the following sections: (i) algorithmic implementation details (Sect. 3), (ii) influence of parameters on result quality (Sect. 4), (iii) integration of the source code into other frameworks (Sect. 5), and (iv) known limitations, difficult cases, and future improvements (Sects. 6 and 7). We first describe the released artifact and the installation procedure (Sect. 2).

All claims in this paper are grounded in the released code, not in additional experiments; where we point out a discrepancy between the main paper's textual description and the code's default behaviour, we do so explicitly so that RRPR reviewers verifying that "the source code is correct and matches exactly what is described in the text" can locate the relevant lines directly.

2 The `dr_benchmark` Reproducibility Package

2.1 Repository overview

The toolkit used to produce every number in Tables 2 and 3 of the main paper is released as `dr_benchmark`[3], MIT-licensed, and archived on Zenodo (DOI: 10.5281/zenodo.20843975) so that a specific, citable version is preserved even if the GitHub repository changes [1]. The repository is organized as a small, dependency-light Python package:

```
src/
  Data.py # dataset download, preparation, Dataset/DataLoader
  Model.py # all model architectures + model_setup() factory
  Train.py # training loop, validation, checkpointing
  Test.py # evaluation, classification report, confusion matrix
experiments/
  run_experiment.py # command-line entry point
requirements.txt
```

This flat structure was a deliberate reproducibility choice: every experiment reported in the paper reduces to a single call to `run_experiment.py` with a documented set of command-line flags (Sect. 4.2), and every architecture is a self-contained `nn.Module` subclass in `Model.py` returned by a single factory function, `model_setup()`, so that the mapping between a row of Table 2/Table 3 and a class in the code is one-to-one and unambiguous.

[3] `https://github.com/Lamine-MOH/Comparative-Evaluation-of-DL-Architectures-and-Training-Strategies-for-DR-Classification`

2.2 Dependencies

The package pins its core numerical dependencies (Python $\geq$ 3.10, PyTorch 2.5.1, torchvision 0.20.1, NumPy 2.0.2, scikit-learn 1.5.2, pandas 2.2.2, Pillow 11.3.0) in `requirements.txt`. CUDA 12.8 is optional; every experiment also runs on CPU, which we verified explicitly because RRPR reviewers may not have access to the GPU used for the original runs. Two additional packages, `kagglehub` and `gdown`, are used exclusively for scripted dataset retrieval (Sect. 2.4).

2.3 Installation procedure

The following three commands reproduce the development environment used for the paper:

```
git clone https://github.com/Lamine-MOH/Comparative-Evaluation-of-DL-
    Architectures-and-Training-Strategies-for-DR-Classification.git
cd Comparative-Evaluation-of-DL-Architectures-and-Training-Strategies-for
    -DR-Classification
pip install -r requirements.txt
```

For GPU execution, PyTorch must be reinstalled with the CUDA-enabled wheel, as commented at the top of `requirements.txt`:

```
pip install torch==2.5.1+cu121 torchvision==0.20.1+cu121 \
  --index-url https://download.pytorch.org/whl/cu121
```

2.4 Dataset acquisition and preparation

All four datasets are retrieved programmatically rather than distributed with the repository, in line with the licensing terms of the source datasets and with the RRPR guidance to link to external repositories instead of redistributing data. `Data.py` exposes two functions:

```
from src.Data import dataset_download, dataset_prepare
path = dataset_download("Aptos") # or "IDRiD" / "DDR" / "Messidor-2"
dataset_path = dataset_prepare("Aptos", path)
```

`dataset_download` wraps `kagglehub` for APTOS 2019, DDR, and Messidor-2 (which therefore require a Kaggle API token) and `gdown` for IDRiD [9] (no authentication required). `dataset_prepare` then harmonizes each dataset's heterogeneous directory layout and label format into a single `Images/` folder plus a `labels.csv` file with two common columns, `file_name` and `diagnosis`, so that every downstream stage of the pipeline (loading, splitting, transforming) is dataset-agnostic.

3 Algorithmic Implementation Details

3.1 Scratch-versus-pretrained architecture family (Experiment 2)

For the ablation study of Sect. 4.2 of the main paper, each of the four backbones is implemented as two independent classes, e.g. `ResNet50Scratch` and `ResNet50Pretrained`, rather than as a single class with a boolean flag. This was a deliberate implementation choice: keeping the two variants structurally identical (same classifier head, same input resolution, same forward signature) while only the origin of the convolutional weights differs (randomly initialized vs. `torchvision` ImageNet weights) removes any possibility that an accidental architectural difference, rather than the initialization itself, explains the accuracy gap reported in Table 3. All eight classes are selected through the same factory:

```
model = model_setup(name="ResNet50Pretrained", num_classes=5, conf=conf)
```

where `conf` carries the optimizer and scheduler hyperparameters (Sect. 4). Task-dependent loss selection is centralized in `model_setup`: `BCEWithLogitsLoss` for the two binary tasks (`num_classes==2`) and `CrossEntropyLoss` for the five-class grading task, with `Adam` as the optimizer and `ReduceLROnPlateau` (monitoring validation loss) as the scheduler in both cases.

3.2 Literature-method reimplementations (Experiment 1)

The six replicated methods of Table 2 were reimplemented as five model classes (the CNN-SVD+ELM model unifies feature extraction and classification in one class):

Modified Xception [6]. `ModifiedXceptionModel` reproduces the deep-layer-aggregation idea of the original design using an EfficientNet-B3 multi-scale feature aggregator as the practical convolutional backbone, since the original Xception variant described in [6] is not distributed as a ready-made `torchvision` module; this substitution is documented in the class docstring and should be read together with Sect. 6.

Hybrid CNN-SVD+ELM [8]. `HybridCNNSVDELM` is a genuinely two-phase model. Phase 1 trains a four-block convolutional feature extractor (`CNNFeatureExtractor`, four `Conv-BN-ReLU-MaxPool-Dropout` blocks producing a 256-dimensional embedding) end-to-end with a differentiable linear classification head, exactly like a standard CNN, through `Train.train_model`. Phase 2 is invoked explicitly *after* phase-1 training converges, by calling `fit_svd_elm(train_loader, device)`: this extracts phase-1 embeddings for the whole training set, fits a `StandardScaler` then a `TruncatedSVD` (default 100 components, following the paper's 256→100 dimensionality reduction) and finally a closed-form Extreme Learning Machine [3] (500 hidden neurons by default, ReLU activation, Moore–Penrose pseudo-inverse solved with `numpy.linalg.pinv`). Once `fit_svd_elm` has run, `model.forward()` silently

switches from the differentiable phase-1 path to the non-differentiable NumPy path, returning the ELM decision scores wrapped back into a `torch.Tensor` so that the existing `Test.evaluate_model` function needs no modification to accept either type of model.

Hybrid MobileNetV2+SVM [11]. `MobileNetV2SVMClassifier` follows the same two-phase pattern. `MobileNetV2Backbone` wraps a `torchvision` MobileNetV2 [10] pretrained on ImageNet, replaces its classifier with a two-layer fully connected head (1280→256→256) mirroring the Keras architecture described in [11], and exposes a scalar regression output for phase-1 training. `fit_svm(train_loader, device)` then extracts the 256-dimensional `fc2` embeddings, standardizes them, and fits an RBF-kernel `sklearn.svm.SVC` ($C = 1.0$, `gamma="scale"`, `probability=True`) on the (rounded) integer DR grade. As with the CNN-SVD+ELM model, the internal `_svm_fitted` flag governs which forward path is used at inference time.

EfficientNet-B5 + CLAHE [7] and the VGG16/InceptionV3 preprocessing variants [4,5]. `EfficientNetB5DR`, `VGG16DR`, and `InceptionV3DR` follow the same single-phase, fully differentiable pattern as Sect. 4.1 but use the preprocessing/loss combinations reported in the corresponding literature methods; `BinaryDRLoss` implements the referral-vs-no-referral binary criterion used for the binary evaluation column of Table 2.

3.3 Reproducibility controls in the data pipeline

Three mechanisms in `Data.py` are the concrete implementation of the "identical training and testing splits" claim in Sect. 4.1 of the main paper: (i) `data_split` uses `sklearn.model_selection.train_test_split` with `stratify=dataset.labels` and a fixed `random_state` (default `seed=2026`) so that class proportions are preserved identically across every architecture evaluated on a given dataset; (ii) `create_dataloader` seeds its own `torch.Generator` and registers a `seed_worker` callback so that batch ordering and any worker-level NumPy/Python RNG state are also fixed; (iii) `get_random_samples_perClass`, used only for the qualitative figures in the main paper, uses an independent `numpy.random.Generator` seeded the same way, so that the illustrative samples in Fig. 1 of the main paper can be regenerated exactly.

4 Influence of Parameters on Result Quality

4.1 Pretraining as a parameter: the strongest lever in the codebase

The single implementation-level parameter with the largest, and most rigorously isolated, effect on result quality is the choice between the `*Scratch` and `*Pretrained` class for a given backbone (Sect. 4.1). Because both classes are architecturally identical except for the origin of the convolutional weights, Table 3

of the main paper *is itself* a parameter-influence study at the code level: switching this one parameter changes binary accuracy by up to 46 percentage points (InceptionV3, APTOS) and is never harmful, across all 64 architecture/dataset/-task combinations we ran. From a reproducibility-engineering standpoint, this means that a practitioner integrating any of these classes into a new project should default to the `*Pretrained` variant unless there is a specific reason (e.g. a non-natural-image domain far from ImageNet statistics) to expect otherwise.

4.2 Optimizer, scheduler, and CLI defaults

`run_experiment.py` exposes the pipeline's hyperparameters as command-line flags:

```
python experiments/run_experiment.py \
  --dataset Aptos --model MobileNetV2SVMClassifier \
  --num_classes 2 --epochs 30 --batch_size 16 \
  --img_size 224 --lr 1e-4
```

Two of these defaults are worth flagging explicitly for anyone attempting an exact, line-by-line reproduction of the ablation study protocol described in Sect. 4.2 of the main paper (batch size 32, maximum 50 epochs, early-stopping patience 20, LR reduction factor 0.5 with patience 10): the CLI default batch size is 16 and the CLI default epoch budget is 30, and `Train.train_model`'s own `early_stop_patience` default (10) is not overridden by `run_experiment.py`. The scheduler's patience is likewise fixed to 5 inside the `conf` dictionary built in `run_experiment.py`, rather than the 10 mentioned in the main text. None of this affects the correctness of the released results, which were produced with the flags matching the main paper's stated protocol, but a reproducer relying on the CLI defaults alone will not reproduce Table 3 exactly; we recommend always passing `--batch_size 32 --epochs 50` explicitly and, until a future release exposes it as a flag (Sect. 7), editing the two constants in `run_experiment.py` (`early_stop_patience` in the `train_model` call and `LR_SCHED_PAT` in `conf`) to match the paper's protocol precisely. We similarly note that `data_split(test_split_ratio=0.2, val_split=True, val_split_ratio=0.1)`, as called by `run_experiment.py`, produces an approximate 72/8/20 train/validation/test split rather than the 70/15/15 nominally described for Experiment 2, since `val_split_ratio` is applied to the 80% training partition rather than to the full dataset; this discrepancy is small enough not to explain the reported effect sizes, but we record it here in the interest of the exact code/text correspondence that RRPR reviewers are asked to check.

4.3 Internal hyperparameters of the classical-classifier hybrids

The CNN-SVD+ELM and MobileNetV2+SVM models expose their classical-stage hyperparameters as constructor arguments rather than CLI flags: `n_svd_features` (default 100), `n_elm_hidden` (default 500), and, for the SVM head, `svm_kernel`, `svm_C`, and `svm_gamma`. We did not run a dedicated sweep

over these values for the main paper, since they were fixed to the values reported by the original authors of each method to keep the replication faithful; based on the implementation, we can nonetheless state their expected qualitative effect precisely: increasing `n_svd_features` trades the noise-suppression benefit of the truncated decomposition for retained variance (`fit_svd_elm` logs the cumulative explained-variance ratio at fit time, which we recommend reproducers inspect on their own hardware/data split before trusting the 100-component default on a new dataset), and increasing `n_elm_hidden` increases the capacity of the closed-form ELM solver at the cost of a higher-dimensional pseudo-inverse and a higher risk of overfitting the (typically much smaller) training partitions such as IDRiD's 516 training images.

5 Integration of the Source Code into Other Frameworks

The `model_setup()` factory and the `DRDataset`/`create_dataloader` pair were designed to be reusable outside the specific `Train.py`/`Test.py` loop shipped in the repository. The single-phase architectures (`*Scratch`, `*Pretrained`, `ModifiedXceptionModel`, `EfficientNetB5DR`, `VGG16DR`, `InceptionV3DR`) are ordinary `torch.nn.Module` instances and can be dropped directly into any PyTorch-based training framework (e.g. PyTorch Lightning or a Hugging Face `Trainer` wrapped around a custom `nn.Module`) by consuming the `model` and `criterion` entries of the dictionary returned by `model_setup()` and ignoring the `optimizer`/`scheduler` entries if the host framework manages its own.

The two hybrid models (CNN-SVD+ELM, MobileNetV2+SVM) require more care, because they are only *partially* differentiable end-to-end. Integrating them into a generic training loop that assumes a single `forward`/`backward` call per batch for the whole training run will silently train only the phase-1 backbone and never invoke `fit_svd_elm`/`fit_svm`; the classical stage must be triggered as an explicit, separate step once phase-1 training has converged, exactly as `run_experiment.py` does not currently do automatically (Sect. 6). We flag this as a documented integration caveat rather than a bug: any downstream user is expected to call `model.fit_svd_elm(train_loader, device)` or `model.fit_svm(train_loader, device)` once, after phase-1 training, before evaluating or deploying the model.

Exporting the toolkit's models to inference-only formats such as ONNX is similarly only partially supported: the differentiable backbones export normally through `torch.onnx.export`, but the classical heads of the two hybrid models call into NumPy/scikit-learn inside `forward()` once fitted and therefore cannot be traced as part of a single ONNX graph. A practitioner who needs a single deployable artifact for one of the hybrid models should export the backbone alone and re-implement the (already very small, closed-form) scaler/SVD/ELM or scaler/SVM stage in the target inference runtime, or serialize the fitted scikit-learn objects with `joblib` and run them as a lightweight post-processing step next to the exported backbone.

6 Known Limitations and Difficult Cases

- **External dataset dependency.** Three of the four datasets are retrieved through `kagglehub` mirrors curated by a third-party Kaggle user rather than the original dataset hosts, and the fourth (IDRiD) is retrieved from a single Google Drive file ID hard-coded in `dataset_download`. Either dependency can break if the corresponding external resource is renamed, removed, or rate-limited, which is a common and largely unavoidable risk for reproducibility packages built on top of third-party dataset mirrors; we mitigate it by pinning the exact Kaggle dataset slugs and Drive file ID in the code so that at least the *intended* source is always unambiguous, even when it is temporarily unreachable.
- **Classical-model artifacts are not checkpointed.** `Train.train_model` checkpoints only `model.state_dict()`, i.e. the PyTorch backbone parameters. For the two hybrid models, the fitted `StandardScaler`, `TruncatedSVD`/ELM weights, or SVM support vectors are held only in memory and are lost when the process exits; reproducing a hybrid model's exact test-set predictions after a restart currently requires re-running `fit_svd_elm`/`fit_svm` on the same training split (which is itself seeded and therefore deterministic on CPU, see below) rather than reloading a saved artifact.
- **Statistical, not bit-exact, reproducibility on GPU.** The seeding strategy described in Sect. 3 fixes Python, NumPy, and PyTorch RNG state, but the code does not set `torch.backends.cudnn.deterministic=True` or disable `cudnn.benchmark`. On CPU, this yields bit-exact reproducibility; on GPU, non-deterministic cuDNN kernel selection means reruns are reproducible in distribution but not necessarily bit-for-bit, which we consider acceptable for the accuracy-level claims of the main paper but flag explicitly here.
- **Small-sample datasets remain intrinsically difficult.** IDRiD's 516 training and 81 held-out images (Sect. 3.1 of the main paper) is the setting in which every architecture we tested is most sensitive to the exact split, and where the from-scratch ablation collapses most severely (e.g. ResNet50 multiclass recall of 0.064, Sect. 4.2 of the main paper); this is a property of the dataset rather than of the toolkit, but it means IDRiD-based results in particular should be interpreted with wider uncertainty than APTOS- or DDR-based ones when reproduced on a different split seed.
- **One architectural substitution is not literal.** As noted in Sect. 3, `ModifiedXceptionModel` uses an EfficientNet-B3-based multi-scale aggregator rather than a literal Xception backbone, because a ready-made, license-compatible Xception implementation matching [6] exactly was not available in `torchvision` at implementation time. We consider this a faithful reimplementation of the deep-layer-aggregation *idea* rather than a literal reproduction of the original network, and we flag it here so that RRPR reviewers checking text/code correspondence are not surprised by the discrepancy.

7 Future Improvements

Based on the limitations above, we plan the following concrete improvements to the released toolkit:

1. Persist the fitted `StandardScaler`, `TruncatedSVD`/ELM weights, and SVM support vectors alongside the backbone checkpoint (e.g. via `joblib`, bundled into the same `.pt` file or a sibling `.joblib` file keyed by the same checkpoint path), so that a hybrid model can be reloaded for inference without re-running phase 2 training.
2. Expose the currently hard-coded scheduler patience and early-stopping patience as `run_experiment.py` CLI flags, and ship a small YAML/JSON configuration file per table of the main paper so that "one command per row of Table 2/3" becomes "one config file per row," removing the need for reproducers to consult this companion paper's Sect. 4 to align CLI defaults with the paper's stated protocol.
3. Add a `Dockerfile` pinning the full transitive dependency graph (not just the direct dependencies currently listed in `requirements.txt`) to further reduce environment-related drift, in support of a future RRPR/ICPR Reproducible Research Badge submission.
4. Add a lightweight continuous-integration workflow that runs a reduced-epoch smoke test of each of the twelve model classes on a small synthetic image set, to automatically detect breakage introduced by future dependency upgrades (e.g. `torchvision` API changes).
5. Cache CNN embeddings extracted for phase-2 fitting of the hybrid models, so that sweeping `n_svd_features`, `n_elm_hidden`, or SVM kernel/ C does not require re-running the (comparatively expensive) CNN forward pass over the training set each time.
6. Repeat the main paper's experiments over multiple random seeds and report variance, which would let the 20–30% cross-dataset variability discussed in the main paper's clinical-deployment discussion be expressed with confidence intervals rather than point estimates.

8 Conclusion

This companion paper documents the reproducibility properties of `dr_benchmark`, the open-source implementation behind our ICPR 2026 paper on DL architectures and training strategies for diabetic retinopathy classification. We described the installation and dataset-acquisition procedure, the concrete implementation of each of the twelve model classes involved in the two reported experiments, the effect of implementation-level parameters (most notably the pretrained-vs-scratch initialization) on result quality, guidance for integrating individual components into other training frameworks, and a transparent account of the toolkit's current limitations together with the improvements we plan for a future release. We hope this level of detail supports

both the RRPR reviewers evaluating the correspondence between our paper's text and its code, and any researcher wishing to build on, or independently verify, our results.

Data and code availability. Source code: `https://github.com/Lamine-MOH/Comparative-Evaluation-of-DL-Architectures-and-Training-Strategies-for-DR-Classification` (MIT license). Archived release: DOI 10.5281/zenodo.20843975. All four datasets used are public and retrievable via the scripted procedure of Sect. 2.4.

References

1. BenHabirech, M.L.: dr_benchmark: Comparative evaluation of dl architectures for diabetic retinopathy classification (source code). `https://github.com/Lamine-MOH/Comparative-Evaluation-of-DL-Architectures-and-Training-Strategies-for-DR-Classification` (2026), dOI: 10.5281/zenodo.20843975
2. BenHabirech, M.L., Belhadj, M., Tamzalit, D., Aiadi, O.: Comparative evaluation of deep learning architectures and training strategies for diabetic retinopathy classification. In: Proceedings of the 28th International Conference on Pattern Recognition (ICPR) (2026), to appear
3. Huang, G.B., Zhu, Q.Y., Siew, C.K.: Extreme learning machine: Theory and applications. Neurocomputing **70**(1-3), 489–501 (2006). `https://doi.org/10.1016/j.neucom.2005.12.126`
4. Islam, M.R., Hasan, M.A.M., Sayeed, A.: Transfer learning based diabetic retinopathy detection with a novel preprocessed layer. In: 2020 IEEE Region 10 Symposium (TENSYMP). pp. 888–891 (2020). `https://doi.org/10.1109/TENSYMP50017.2020.9230648`
5. Jiwani, N., Gupta, K., Sharif, M.H.U., Datta, R., Habib, F., Afreen, N.: Application of transfer learning approach for diabetic retinopathy classification. In: 2023 International Conference on Power Electronics and Energy (ICPEE). pp. 1–4 (2023). `https://doi.org/10.1109/ICPEE54198.2023.10060777`
6. Kassani, S.H., Kassani, P.H., Khazaeinezhad, R., Wesolowski, M.J., Schneider, K.A., Deters, R.: Diabetic retinopathy classification using a modified xception architecture. In: 2019 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT). pp. 1–6 (2019). `https://doi.org/10.1109/ISSPIT47144.2019.9001846`
7. Momeni Pour, A., Seyedarabi, H., Abbasi Jahromi, S.H., Javadzadeh, A.: Automatic detection and monitoring of diabetic retinopathy using efficient convolutional neural networks and contrast limited adaptive histogram equalization. vol. 8, pp. 136668–136673 (2020). `https://doi.org/10.1109/ACCESS.2020.3005044`
8. Nahiduzzaman, M., Islam, M.R., Islam, S.M.R., Goni, M.O.F., Anower, M.S., Kwak, K.S.: Hybrid cnn-svd based prominent feature extraction and selection for grading diabetic retinopathy using extreme learning machine algorithm. IEEE Access **9**, 152261–152274 (2021). `https://doi.org/10.1109/ACCESS.2021.3125791`
9. Porwal, P., Pachade, S., Kamble, R., Kokare, M., Deshmukh, G., Sahasrabuddhe, V., Meriaudeau, F.: Indian diabetic retinopathy image dataset (idrid): A database for diabetic retinopathy screening research. Data **3**(3), 25 (2018). `https://doi.org/10.3390/data3030025`

10. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.C.: Mobilenetv2: Inverted residuals and linear bottlenecks. 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) pp. 4510–4520 (2018). https://doi.org/10.1109/CVPR.2018.00474
11. Taufiqurrahman, S., Handayani, A., Hermanto, B.R., Mengko, T.L.E.R.: Diabetic retinopathy classification using a hybrid and efficient mobilenetv2-svm model. In: 2020 IEEE Region 10 Conference (TENCON). pp. 235–240 (2020). https://doi.org/10.1109/TENCON50793.2020.9293739