WORKFLOW FOR TRAINING AND SERVING DEEP LEARNING MODELS FOR IMAGE CLASSIFICATION AND OBJECT DETECTION - APPLICATION TO FAULT DETECTION ON ELECTRIC POLES

Christopher Coello[1], Rafael Sanchez[2], Sindre de Lange[2], Joachim Halvorsen[2], Marco Bertani-Økland[2], Viktor Myrvang[1], Ståle Heitmann[1]*

[1]*Elvia AS, Hamar, Norway*
[2]*Computas AS, Oslo, Norway*
* *christopher.coello@elvia.no*

Keywords: DEEP LEARNING, REST API, FAULT DETECTION

Abstract

This work presents the development of a workflow that allows for training, testing and serving deep learning models that can use any image coming from the image repository and can enrich these images with metadata derived from the results provided by the deep learning (DL) vision models. Using this workflow, an electric pole image classification model and a missing top cap object detection model were trained and served on millions on helicopter inspection images. In addition to obtaining very accurate models, we observe that such a modular workflow have reduced both the time from idea to prototype and the time from prototype to product.

1 Introduction

As Norway's largest distribution system operator (DSO), Elvia carries out daily a wide range of activities to operate, maintain, document and expand the electric grid. Some of these activities require picture documentation, for situation analysis and decision support. A non-exhaustive list of activities are: the overhead line inspection using aerial photography carried out using helicopter; documentation of installation of smart meters by standardised photography of the asset and the electric cabinet; validation of cable installation between a junction box and a house by submission of a defined number of standard pictures [1].
Historically, those different activities have been carried out by different divisions of the company, which have stored and processed these images in their respective storage and application systems. Consequently, the use of these images was exclusively possible within the system operating the task related to the image content.
An internal initiative was set up in March 2020 in order to set up a new IT ecosystem that had the objective to handle all images taken and/or processed by the different activities of the company. Interacting with this new IT ecosystem, the authors of this study developed a workflow that allows for training, testing and serving deep learning models that can use any image coming from the image repository and can enrich these images with metadata derived from the results provided by the deep learning (DL) vision models.
In this paper, Section 2 will present the different modules of the IT image ecosystem, Section 3 will present how this modules are used to achieve training and serving of DL models, Section 4 presents two use-cases that followed such workflow, while Section 5 will present the results obtained for those two use-cases in terms of model performance and easiness of implementation.

2 Methodology

A schematic of the overall IT architecture of the modules and workflows is shown in Figure 1. The three main components used for executing the task of training and serving DL models are: the *louvre-db* and its different modules/APIs, the *ai-vision-api* and the *Azure Custom Vision* service. The three components are described below in more detail.

2.1 louvre-db

The *louvre-db* ecosystem is a system for storage and distribution of images. It has been developed following a microservice-based architecture. The details of the architecture will not be presented, but its main objective is to ingest, process and make available images from/to all systems across the company. Within this ecosystem, there are three components that are important in order to facilitate the DL workflow which are the focus of this paper: ImageAPI, PluginHandler and ImageEnhanceAPI.

2.1.1 ImageAPI: The ImageAPI exposes the metadata database and allows the users to run queries against it. The metadata

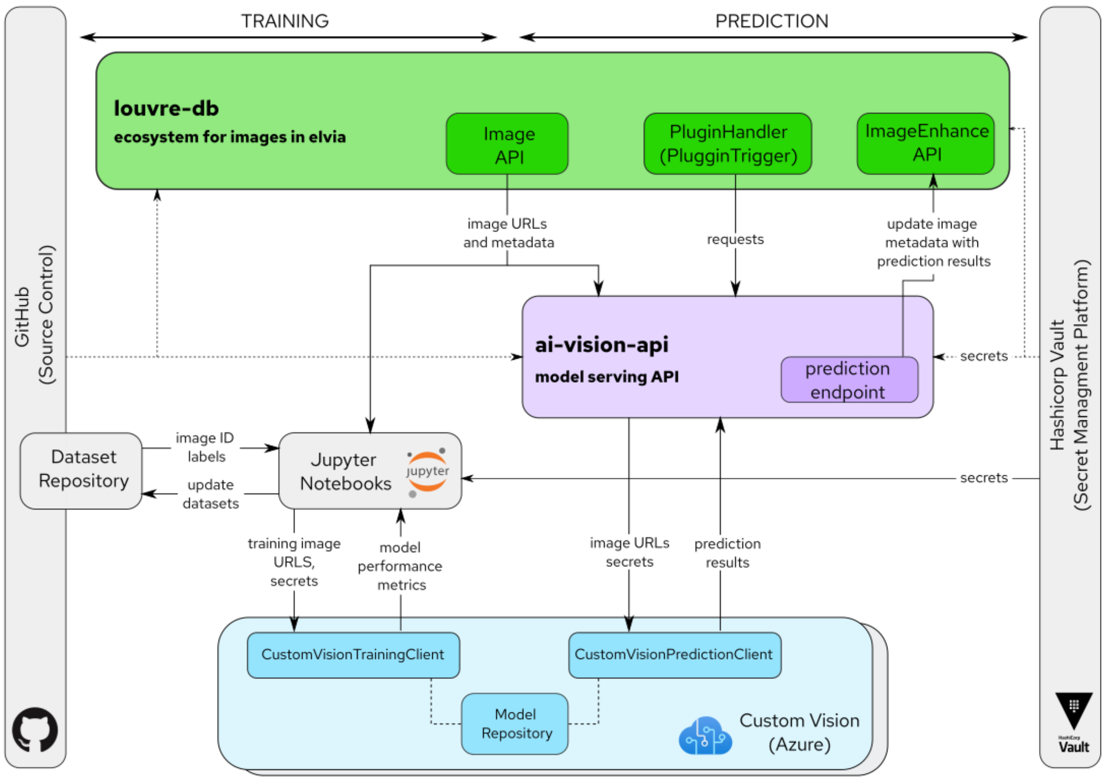

Fig. 1. Overall architecture implementation for the training (left) and the serving (right) of DL models.

database contains URLs to the images present in the storage system, together with all other possible metadata that are connected to the images, for example EXIF tags, GeoLocation tags, domain-related tags, and more. The interaction with the Azure Search motor is done using GraphQL query language.
As part of the image ingestion process, every image is resized to (and made available in) different variants: original, standard and thumbnail; each of which can be queried individually. The response to a query for a given image will typically contain the unique resource identifier (sas-uri) for each of its aforementioned variants, together with other metadata.

2.1.2 PluginHandler: The PluginHandler component is responsible for communicating with the different plugins, which in turn perform operations related to images within *louvre-db*. PluginHandler triggers a plugin by sending it webhooks when events happen. Examples of events are the uploading of new images tagged with a specific image category, or the change of a metadata field in the metadata database.
Each plugin is defined by a combination of a unique name, an event trigger, a filter and a URL to the endpoint that should be triggered. Both the event trigger and the filter can be a combination of several individual event triggers (or filters, respectively). Anybody with access can define plugins on their own.
An interesting feature is the possibility to introduce dependencies between any two plugins, e.g. the order in which they run. This can be achieved by modifying a metadata entry as part of running the first plugin, and then defining changes in that same metadata entry as the event trigger for the second plugin.

2.1.3 ImageEnhanceAPI: ImageEnhanceAPI simply takes care of changes in the metadata as a result of the processing done by the plugins. It updates the metadata of a given image, keeping in records the original metadata before modification for possible retrieving at a later stage.

2.2 *ai-vision-api*

ai-vision-api has the objective to coordinate the communication between the images and metadata present in *louvre-db* and the DL models. It was designed to be used when training as well as serving the DL models.
ai-vision-api is both a Python package and a Flask-based web application, and it can be customised based on the requirements of the models it serves. In terms of figure 1 and paragraph 2.1, *ai-vision-api* functions as a plugin.
ai-vision-api was developed paying special attention to decoupling the DL framework-specific modules from the rest of the code. By doing this, shifting between DL frameworks would not only be possible, but straightforward.
Development, deployment and operation of *ai-vision-api* is made following DevOps principles. *ai-vision-api* instances run as pods in a Kubernetes cluster on Microsoft Azure. The secure operation of *ai-vision-api* involves the retrieval of secrets from Elvia's secret management service.
The DL model training (paragraph 3.1) and serving (paragraph

3.2) workflows will explain in detail how the API is used to achieve these two steps.

2.3 Custom Vision

The DL framework used is the Custom Vision service ([2]) from Microsoft Azure Cognitive Services. This deep learning service allows to train, store and serve DL models for doing prediction on images. The user of the service can choose between classification (image -> label) and object detection (image -> bounding box) architectures. *ai-vision-api* interacts with Custom Vision through its Python SDK, more specifically with the training (and its CustomVisionTrainingClient class) and prediction (and its CustomVisionPredictionClient class) clients.

3 Deep Learning Workflows

By using the modules presented previously, the objective is to set up reproducible and easy-to-use workflows to train, validate and serve deep-learning models. In the next two sections, the workflow for training/validating new models and the workflow for serving trained models in production are described.

3.1 Training DL models

Training a DL classification (resp. object detection) model requires: a dataset containing a set of images with the corresponding label (resp. bounding box) of the image; a DL model architecture; and a machine to train the model (i.e. change the weights of the model) using the data in the dataset.

3.1.1 DL model: When using the Custom Vision service, the architecture and initial weights are predefined. For multilabel or multiclass classification, the user can choose among different domains (General, General A1, Food, Landmarks, Retail) when setting up and training a model. These domains are, we believe, an abstraction covering both the model architecture and the initial weights of the model (transfer learning). For object detection, the domains are General, General A1, Logo and Products on Shelves. In addition, small-size architectures can be chosen (Compact domains), which are intended to be exported and executed on mobile devices.

3.1.2 Dataset: Because the DL architecture is fixed, one of the only means to improve the quality of predictions is to improve the training dataset. Jupyter Notebooks (JNs) were extensively used for the training workflow because they allow flexibility and customisation. On a single platform, tasks such as visualizing images, plotting validation metrics and interacting with APIs are all feasible.
Training datasets are stored in GitHub as CSV files, and a set of JNs allows to incrementally improve these datasets following the schematic of Figure 2. Dataset improvement can take several forms:

- increasing the number of images in the dataset,
- defining strict rules for each labelled category,

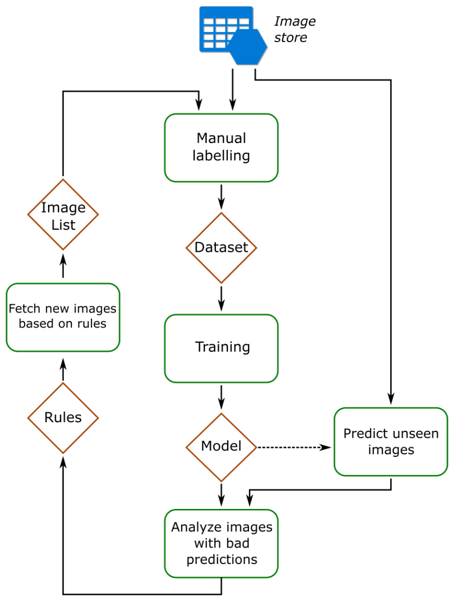

Fig. 2 Schematic of the incremental improvement of the dataset used to train the model. Green rectangles: process executed using Jupyter Notebooks, brown diamonds: artefacts.

- defining fine granularity in the labels to allow possible grouping of classes,
- cleaning errors in the labelling

For versioning and reproducibility purposes, each new iteration of the training dataset is named with a unique, short hash value generated based on its contents. Once the training dataset is sufficiently optimised, a new model version is rolled out in Custom Vision via its training client. The *ai-vision-api* prediction endpoints were designed to search for and use a particular model name in Custom Vision, therefore allowing model updates (i.e. re-training with more data) without having to update the API itself.

3.2 Serving

When a new image satisfying the rules of a given model plugin is uploaded to *louvre-db*, PluginHandler sends a webhook to the corresponding *ai-vision-api* model endpoint, containing the unique image identifier of the new image. Based on this information, *ai-vision-api* first queries ImageAPI to obtain the sas-uri of the new image's variant most suitable for prediction. It then calls the Custom Vision prediction service and provides it with the sas-uri together with additional information (i.e which model to use) via the Custom Vision prediction client. Custom Vision fetches the actual image file and finally sends back prediction results. After a validity check on the results, *ai-vision-api* updates the metadata linked to that image and model with the prediction results; this is done by sending a metadata update request to ImageEnhanceAPI.

3.2.1 Scaling: In order for the prediction endpoints to cope with traffic spikes and request queues, *ai-vision-api* implements Waitress [4]. Waitress is a Python WSGI server that can

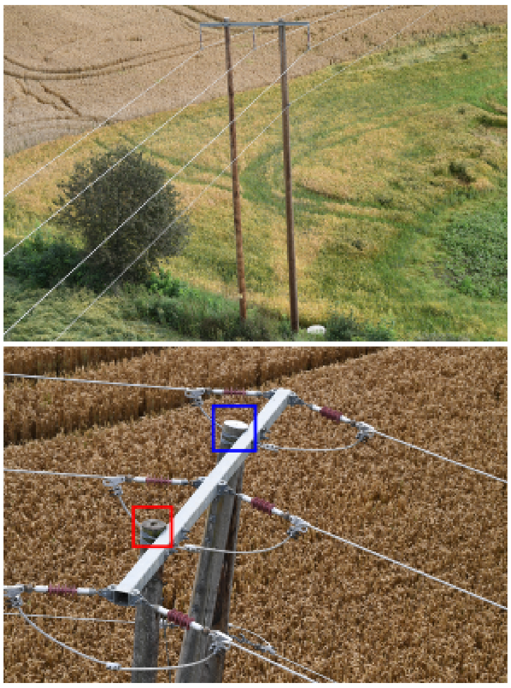

Fig. 3 Example of aerial helicopter images. On the bottom images, the bounding box of the "missing whole top cap" (red) and the "whole top cap" (blue) are visible.

handle requests in an asynchronous, multi-threaded manner. The default number of threads is four. In addition to *ai-vision-api* supporting multi-threading, the number of *ai-vision-api* instances will get adjusted automatically in Kubernetes, e.g. new instances will be spun up when the load is high.

4 Experiments

While developing the infrastructure around the *ai-vision-api*, one classification and one object-detection use-case were used to test the workflows. These use-cases were selected in collaboration with domain-experts using aerial helicopter images. Previous efforts (i.e [3]) has shown the feasibility of the object-detection use-case. Once trained, both models were configured as plugin within the *louvre-db* with different triggers.

4.1 PoleCat: detecting pole type

The objective is to detect the type of electric pole present in aerial helicopter images (Figure 3). Three classes of poles were targeted, as these covers almost the entirety of the poles in Elvia's grid (label names are given in parenthesis): TREE (TRE), STEEL (STÅL) and CONCRETE (BETONG) poles. A fourth class (MULTIPLE) that classified images with several poles was additionally tested, but not used in the final model. The class distribution is unbalanced as around 96% poles are of class TREE.
Four experiments were done with balanced (all classes with roughly 500 images) and unbalanced (TREE classes with 4000, all other classes with 500 images) datasets, with and without the MULTIPLE class. Based on validation metrics (see paragraph 4.3), the final dataset used three unbalanced classes: TREE (4161), STEEL (467), CONCRETE (596). A semi-automatic process (see Figure 2) is in place to gradually add new images in the two classes with low count.
The multiclass classification model with General A1 domain was used in Custom Vision client. The output of the model is a probability level for each class which must be transformed into one label. For a given image, the label for the class with the highest probability over the predefined threshold is the predicted label. The probability threshold is the only hyperparameter to be chosen for this model.
This plugin triggers when a new image with tag "helicopter inspection" is uploaded to the *louvre-db*.

4.2 TopCap Finder: finding missing top-cap

The objective is to detect and place a bounding box around the missing top caps. An example of missing top cap is shown in the bottom image of Figure 3. Based on the workflow presented in paragraph 3.1, the initial rules for labeling considered 10 different tags with a fine level of granularity: whole visible black top cap, whole visible white top cap, partially visible white top cap, partially missing top cap, whole missing top cap, etc... This granularity was based on the color, angle of the camera that allowed or not to see the whole top cap, and possible cables on overlaying the object of interest. After a significant number of experiments where categories were iteratively refined and grouped in numerous ways, the final balanced dataset used to train the published model contained two classes of bounding boxes: 'whole missing top cap' (1188) and 'whole top cap' (1182).
The object-detection with General A1 domain was used in Custom Vision client. In addition to the probability threshold, the amount of overlap between the predicted bounding box and the actual bounding box quantifies the accuracy of the positioning of the predicted bounding box. Whereas the probability threshold is of importance when predicting new bounding boxes, the overlap threshold is a hyperparameter that is only needed for obtaining validation metric.
This plugin triggers when the metadata corresponding to *PoleCat* changed and the prediction result from *PoleCat* model was of class TREE.

4.3 Metrics

The Custom Vision client returns the K-fold cross-validation precision and recall metrics. We remind that precision quantifies the fraction of predicted classifications that were correct whereas recall is a fraction of classes that were correctly predicted. The F_1 score is the harmonic mean of the precision and recall:

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall} \tag{1}$$

5 Results

The prediction results for the classification and object detection use-cases are briefly presented below, together with a general discussion about the API and its re-usability.

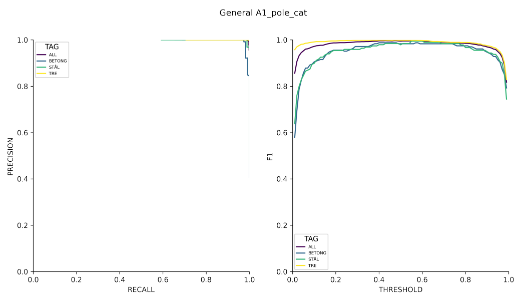

Fig. 4 Precision/recall (left) and F_1 score (right) graphs for varying thresholds for all the labels and per label.

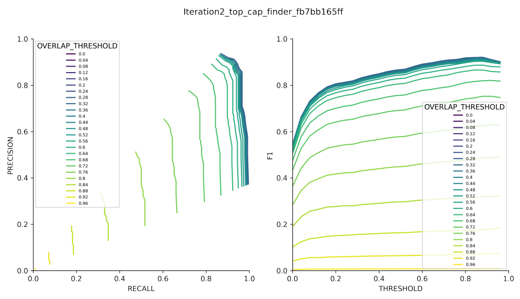

Fig. 5 Precision/recall (left) and F_1 score (right) graphs for varying thresholds and varying overlap thresholds for all the labels and per label.

5.1 Detecting poles

Figure 4 shows the precision/recall (left) and F_1 score (right) curves for the classification use-case (PoleCat). There is one curve for all labels together (ALL) and one curve per class.

5.2 Detecting missing top cap

Figure 5 shows the precision/recall (left) and F_1 score (right) curves for the object detection use-case (TopCap Finder). The curves are estimated for the class 'whole missing top cap', which is in this particular use case the only one we are interested in. Interestingly, when removing the class 'whole visible top cap', the results were worse (not shown) than with two classes. We believe that's because both objects (missing top cap and a visible top cap) are co-located (always on top of a tree mast), and the algorithm manages to differentiate better between them when trained with two classes instead of only one.

5.3 Accelerating innovation and re-usability

The development of such workflows and APIs directly decreases the time from ideas to prototype to most-valuable-product (MVP) to product. In the initial phase of prototype testing, the training workflow allows to almost solely focus on building a training dataset, which can be done by using JNs but in fine could be done by the domain experts themselves. Only minor changes are needed in order to include a first prototype directly against the production base and test the model at scale. The experience acquired when implementing these use case allowed us to set up some standard processes as the one shown in Figure 2. Each of the processes needed to obtain a solid and robust prototype are documented and executed using JNs. A clear advantage of using this JNs approach is that the customisation and re-usability from use-case to use-case is made very simple (for example, change the GraphQL query for selecting the images).
Another beneficial consequence of the *ai-vision-api* is the transfer from prototype in the development environment to a product in production is accelerated and simplified. Once the dataset is finalized, the model is retrained in the production environment with the same dataset (ad defined by its CSV source in GitHub).
Finally, the image ecosystem allows for images to be re-used outside the original purpose for which the image was taken. The classification use-case presented in paragraph 4.1 is a good example of such re-use of image: while the original aerial images were taken to detect failures in the overhead lines, the images could be easily re-used to increase the quality of our grid documentation. Indeed, once the pole type prediction are compared with the existing pole type in our NIS system: if a difference exists, then a manual process could be initiated to check this difference and either adjust the documentation or give feedback to the model that it misclassified a pole type.

6 Conclusion

In this paper, we presented the IT modules and workflows used in Elvia to train and serve DL vision models. In addition to the good prediction levels obtain in one classification and one object-detection use-case, we clearly see the benefit of this ecosystem in the ability to bridge the gap of innovation from idea to product and to reuse images outside the original purpose for which the image was taken.

References

[1] 'Krav til kabelrør - Eidsiva', accessed 24 February 2021, https://www.eidsiva.no/eidsivanett.no/graving-og-trefelling/stromoppkobling/
[2] 'Azure Custom Vision', accessed 24 February 2021, https://azure.microsoft.com/en-us/services/cognitive-services/custom-vision-service/
[3] Nguyen, V.N., Jenssen, R., Roverso, D.: 'Automatic autonomous vision-based power line inspection: A review of current status and the potential role of deep learning', Electrical Power and Energy Systems, 2018, 99, pp 107-120
[4] 'Waitress', accessed 26 February 2021, https://docs.pylonsproject.org/projects/waitress/en/stable/
[5] 'Flask', accessed 26 February 2021, https://palletsprojects.com/p/flask/