ENEE 408M: Project Specification

University of Maryland, College Park, `Spring 2022`
Prof. Shuvra S. Bhattacharyya

1 Collaboration Policy

Students will work in teams. Students who are grouped together in the same team are allowed to discuss the project requirements, and to collaborate on all aspects of code development, testing, debugging, and documentation, unless otherwise specified as part of specific requirements for the project. Each team is responsible for a single set of deliverables, which represents the collective effort of the team.

2 Overview

In this project, you will develop an embedded face detector system (EFDS), which identifies images or image regions that contain human faces. In your experimentation with the EFDS, you will explore trade-offs among key implementation metrics, including real-time processing speed, detection rate (true-positive rate), and the rate of false positives. Your EFDS will be based on a well-known face detection algorithm called the Viola-Jones (VJ) algorithm [1]. This algorithm provides a highly configurable structure that is mapped naturally into a dataflow modeling framework, and that can be tuned along various dimensions to explore implementation trade-offs in a powerful way.

There are two main parts of the project design — the EFDS itself, and the *training module* for the EFDS. The EFDS is designed to run on the Raspberry Pi device, and will be developed using both native compilation on `enee408m.ece.umd.edu` (for prototyping and testing), and cross-compilation (for performance evaluation, further testing, and demonstration). The purpose of the training module is to optimize parameters in the EFDS, and to assist the system designer in configuring the EFDS across a range of useful operating points (trade-off options). The training module is designed to run only on `enee408m.ece.umd.edu`; it is not intended to be part of the deployed EFDS.

As the primary reference for developing both the training module and the EFDS, you are referred to the classic paper by Viola and Jones [1] and the more concise preliminary version of this work [2]. Both of these papers will be made available on Canvas for easy reference.

The scalable structure of the EFDS, which is based on the VJ algorithm, is illustrated in Fig. 1.

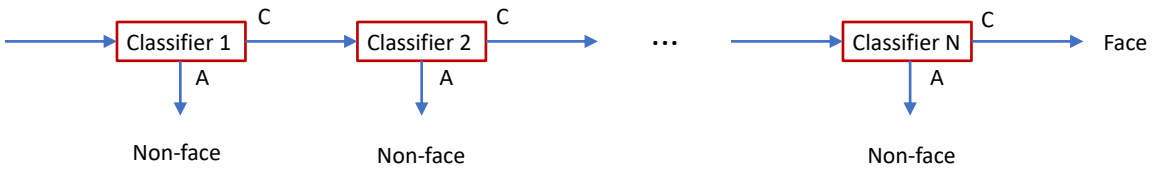

Figure 1: An illustration of the EFDS.

The system involves a parameterized number of classifiers N, where N is a key parameter that can be used to influence the implementation trade-offs described above. You will implement the EFDS (Fig. 1) as a dataflow graph in Welter-C++ with an actor type (C++ class) defined to represent a single classifier, and a separate classifier object (actor) instantiated for each of the N classifiers. As shown in the figure, each classifier has a single input port, which we denote by I, and two output ports C and A. The symbols I, C,

Mode/ Ports	I	A	C
`Configure`	0	0	0
`Read`	-1	0	0
`Classify`	0	0	0
`True`	0	0	1
`False`	0	1	0

Table 1: The dataflow table for the classifier actor.

and A stand for *input*, *continue*, and *abort*, respectively.

On each firing, a classifier actor consumes a single token from I, where the token type is called Pointer To Image Sub-window (PTIS). You need to implement this token type as part of the project. A *sub-window* is a rectangular image region over which face detection is being performed.

The classifier actor has five modes: `configure`, `read`, `classify`, `false`, and `true`. The `configure` mode is used to load parameters into the actor's state from one or more files, where the set of files to read from (or the set of associated file names) is managed as an actor parameter. No tokens are consumed in this mode. The `configure` mode transitions unconditionally to the `read` mode.

The `read` mode consumes a single PTIS token from the I input, and produces no tokens. If the consumed token value is a a non-null pointer, then the actor is transitioned to the `classify` mode; otherwise, it is transitioned to the `false` mode.

The `classify` mode operates on the image sub-window W referenced by the token that was consumed by the previous actor firing (which was in the `read` mode). The VJ-algorithm-based classifier encapsulated by the actor is applied to determine whether or not W should be rejected as being a non-face region or whether the enclosing EFDS should continue considering W as a region that possibly encapsulates a face. If W is rejected, then the actor is transitioned to the `false` mode; otherwise, it is transitioned to the `true` mode. Details on the decision algorithm to be implemented in the `classify` mode can be found in References [2, 1].

The `false` and `true` modes are very simple. Neither mode consumes any tokens, and each mode simply produces a copy of the token value that was consumed during the most recent firing in the `read` mode. This copy is encapsulated in a single output token. The `false` mode outputs the token on the A output, while the `true` mode outputs the token on the C output. Both of these modes transition unconditionally back to the `read` mode, which will be used to start processing the next input sub-window.

Table 1 shows the dataflow table for the classifier actor, and Fig. 2 shows the mode transition graph.

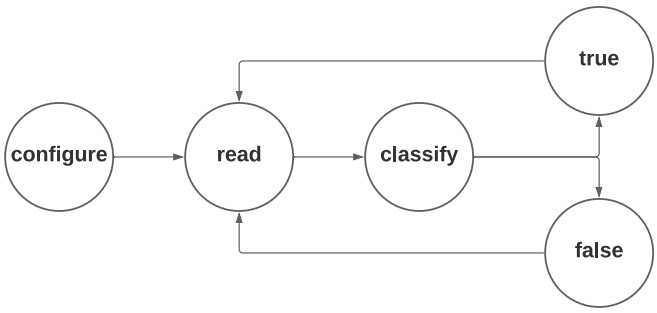

Figure 2: The MTG for the classifier actor.

The classifier actor does not have any error mode; instead, if an error is detected during the `classify` mode, then the actor simply transitions to the `false` mode.

Note that our description of Fig. 1 in terms of dataflow concepts (actors, firings, etc.) and PTIS tokens is not part of the VJ algorithm; instead, these are details of the model-based process that you are using to design and implement the EFDS based on the VJ algorithm.

You do not need to implement the training module using Welter or dataflow concepts. You should implement the training module as a C++ program that outputs a set of configuration files for the classifiers that are derived from the training process defined by the VJ algorithm. The configuration files will then be read by the classifiers, in their respective `configure` mode firings, when the EFDS is initialized prior to an execution of the system on a sequence of input sub-windows.

As implied above, you can design your system with the assumption that sub-windows will be provided as input; you do not need to decompose a larger input image into sub-windows. You will be provided with datasets consisting of many image sub-windows that you can use to train and test your EFDS implementations.

For both the training module and the EFDS, you can use components from the Welter-C++ library, and you can use components from the following C/C++ libraries, which are available on `enee408m.ece.umd.edu`:

```
/usr/lib/gcc/x86_64-linux-gnu/9/include
/usr/include/x86_64-linux-gnu
/usr/include
```

Apart from functionality that you use from these four libraries (including Welter), and any additional components that are provided specifically for your use in the project, you should write all of the code for your training module and EFDS yourselves — among the members of your project team — *without* using, adapting or referring to any other source code or libraries. This means that you should also *not* make any use of OpenCV code or libraries. In the project, the objective is to build up the image processing functionality yourselves, thereby gaining deeper experience with embedded image processing implementation, and also deeper insight and control over implementation trade-offs.

Two issues in design and implementation of embedded image processing systems are real-time performance (e.g., the maximum allowable frame rate) and image analysis accuracy. For the purposes of this project, *performance* is the average latency for identifying an image sub-window as corresponding to a face. Positive detection of a face requires executing all of the classifiers in the system, so the performance can be viewed as the average time to execute the entire chain of classifiers. When evaluating the latency on a benchmark set of input sub-windows (input instances), you would disregard the input instances that are aborted (produce outputs on any classifier A ports), consider the total execution time T_{pos} required for all of the other N_{pos} input instances, and compute the quotient T_{pos}/N_{pos}. Assuming your benchmark set contains at least one input instance that is identified as a face, this quotient gives the performance value for your EFDS implementation on the given benchmark set.

You will evaluate system accuracy in terms of both the detection rate and false positive rate, as described above. When combined with latency, this leads to a three-dimensional design evaluation space.

You have considerable freedom in this project on how to implement the EFDS. This flexibility will enable you to experiment extensively with different design options in a manner that permits exploration of trade-offs among face detection performance, detection rate, and false positive rate. The flexibility includes determining the number of classifiers to use (i.e., the value of N), how many features to use for each classifier, and how to implement the classifiers (subject to the design rules associated with their dataflow-based interface). Note that, based on the VJ algorithm, the numbers of features for the different classifiers are determined by the target detection and false positive rates that you assign to the different classifiers; thus, different ways of assigning the target rates influences the number of features, as well as the performance of the overall system.

In other words, you do not need to follow the same overall/cascade training process as what is described in [2, 1]. You *are* required to apply the boosting concept to develop each strong classifier, but you do not have to stick to any fixed process for designing the cascade of classifiers. The cascade design process is part of the trade-off exploration that is part of the project.

You are encouraged to use Bash scripts to help automate the process of design space exploration that you employ in the project. Any Bash scripts and C++ code that you develop for such automation should be submitted as part of your project deliverables, and explained in your project report. You should place the scripts in a top-level project directory called `scripts`, which is at the same level as your `src` and `test` directories. Your overall project will therefore include C++, Bash and `cmake` code (`CMakeLists.txt` files), as well as text files, such as `README.txt` and `test-desc.txt` files, that document the different parts of your project's directory tree. *All scripting should be done in the form of Bash scripts.*

A major objective in the design space exploration aspect of the project is for you to investigate different designs for the EFDS along with their trade-offs in terms of performance and accuracy. You should summarize your findings in the project report. Document in your report each EFDS implementation that you explored and what you learned from it — e.g., what is its effectiveness in terms of performance and accuracy, and what, if any insights did you gain as you developed, experimented with, and analyzed it? Additional guidelines and requirements for the project report will be given in a separate handout.

3 Deliverables

The deliverables for the project must be archived, using `dxpack`, in an archive file called `efds-project.tar.gz`. The project should be submitted through the submission directory of the team's designated *corresponding team member* (*CTM*). More details about the expected contents and form of the deliverables will be specified in a separate handout.

The archive file `efds-project.tar.gz` must be submitted by 11:30PM on Monday, May 9, 2022.

Document Version

Authored by Shuvra S. Bhattacharyya and Lei Pan. Last updated on March 15, 2022.

References

[1] P. Viola and M. Jones, "Robust real-time face detection," *International Journal of Computer Vision*, vol. 57, no. 2, pp. 137–154, 2004.

[2] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2001.