Some Tips to Strengthen Your Project Report

This document suggests a few ideas to perfect your thesis and project reports. It includes three sections: Coding, Model description, and Experimental analysis.

A. Good Coding Practices of Software Engineering

1. Writing clean and modular code
2. Writing efficient code
3. Code refactoring
4. Adding meaningful documentation

- Clean code:
  - Code that is readable, simple, and concise.
  - Clean code is crucial for collaboration and maintainability in software development.
  - Practice sufficient in-line commentary. Good comments are essential for quality code development and delivery. They made it easy for bug fixing and code sharing.

- Modular code:
  - Code that is logically broken up into functions and modules. Modular code that makes your code more organized, efficient, and reusable.
  - Do not write one long script for completing all the operations, such as data loading, feature extraction, transformation, model training, model testing, and visualization of the results. Always, a divide-and-conquer approach is useful for readability and debugging.

- Module: A file. Modules allow code to be reused by encapsulating them into files that can be imported into other files.

- Style guide: It is important to follow Python Enhancement Proposal (PEP) 8 – Style Guide for Python Code

- Including code in reports/thesis: Do not include the source code in the middle of the report. You can have them in the appendix if you wish while providing useful markers in the main text at appropriate places that point to the attached code snippet. The code snippet (not a screenshot) must be added in actual coding font and color, not in regular text, to greatly improve readability.

  Follow Code listing - Overleaf, Online LaTeX Editor to directly import code snippets into your report's/ thesis's appendix.

Refactoring Code

- Restructuring your code to improve its internal structure without changing its external functionality. This gives you a chance to clean and modularize your program after you've got it working.

- Since it is not easy to write your best code while you are still trying to just get it working, allocating time to do this is essential to producing high-quality code. Despite the initial time and effort required, this pays off by speeding up your development time in the long run.

- You become a much stronger programmer when you are constantly looking to improve your code. The more you refactor, the easier it will be to structure and write good code the first time.

Example #1: A simple example of modular programming is given below with dummies.

```
#------------------------ Header Comment ----------------------#
# Write general information about the script                   #
# the authors involved and copyright details, if any           #
#--------------------------------------------------------------#
# describe the purpose of this function and
# its input-output parameters

Task_1 <- function() {
     ⋮
}

# describe the purpose of this function and
# its input-output parameters
Task_2 <- function() {
     ⋮
}
          ⋮

# describe the purpose of this function and
# its input-output parameters
Task_n <- function() {
     ⋮
}

# Write function calls as required
```

Example #2: Check Appendix

B. Model Description

1. It is important to have impactful visualization for describing the data flow, proposed model, and the results.

2. You may get some ideas and inspiration from the following sample diagrams.

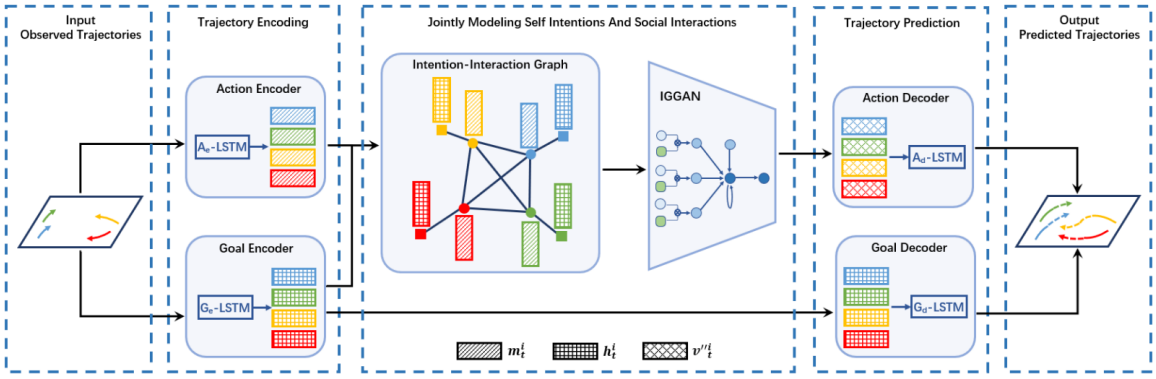


Fig. 2. The architecture of our proposed method. This model is a hierarchical network consisting of two LSTM based sequence-to-sequence modules, which are used to reason the destinations and the future movements respectively. To model self intentions jointly with social interactions, Intention-Interaction Graph (IIG) is designed. And Interaction Gated Graph Attention Networks (IGGAN) is proposed to aggregate information in IIG, which achieves reason the influence degree of both neighboring pedestrians and predetermined destinations.

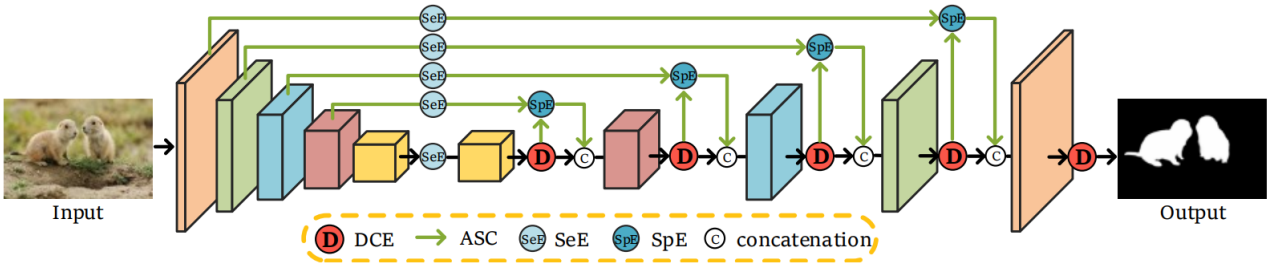


Fig. 2. The architecture of our proposed dense context exploration network (DCENet). DCE: dense context exploration module. ASC: attentive skip-connection. SeE: semantic enhancement block. SpE: spatial enhancement block.

Ack: IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, 2021.

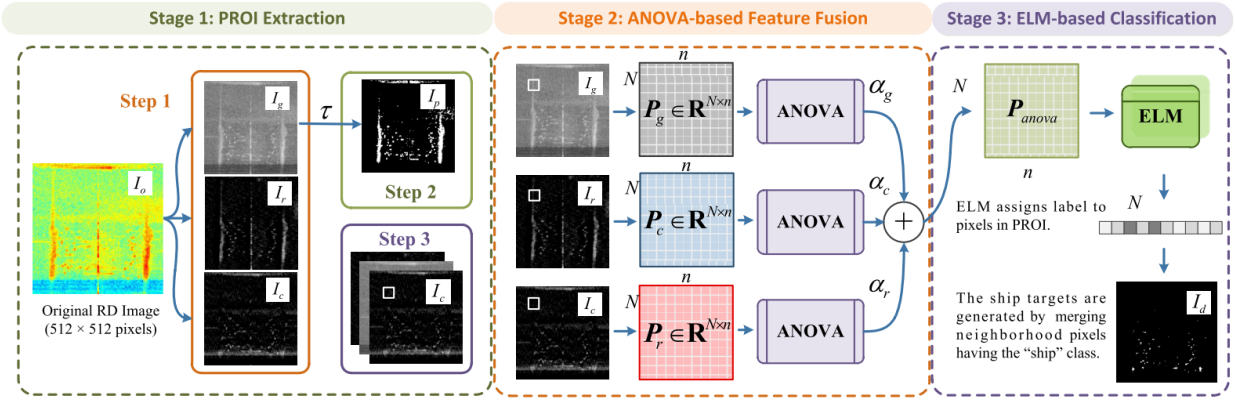


Fig. 2. Overview of the proposed ship detection method. Stage 1 aims to extract the PROI (τ is the threshold calculated from I_g; if pixel intensity is greater than τ, then the pixel is the PROI), Stage 2 performs feature fusion, and Stage 3 focuses on ship detection in the PROI. The white rectangular boxes in Stages 1 and 2 indicate the moving window used to extract the image patches. All the variables denoted in this figure are described in Section II.

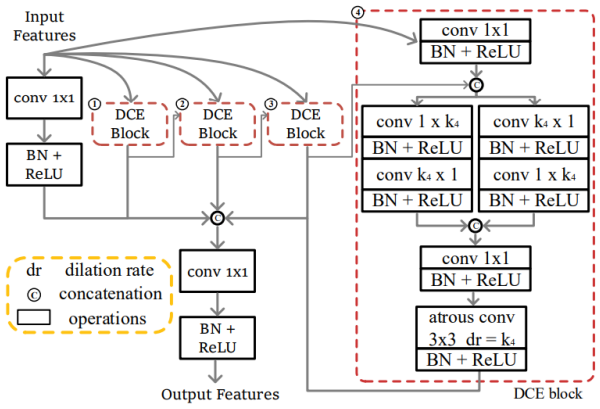


Fig. 3. Detailed illustration of our proposed dense context exploration (DCE) module.

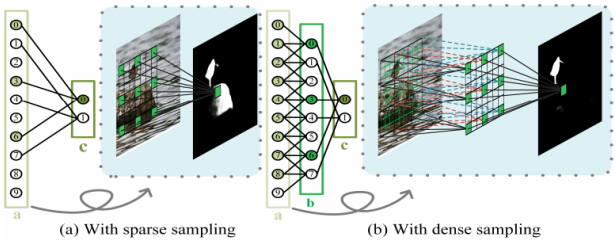
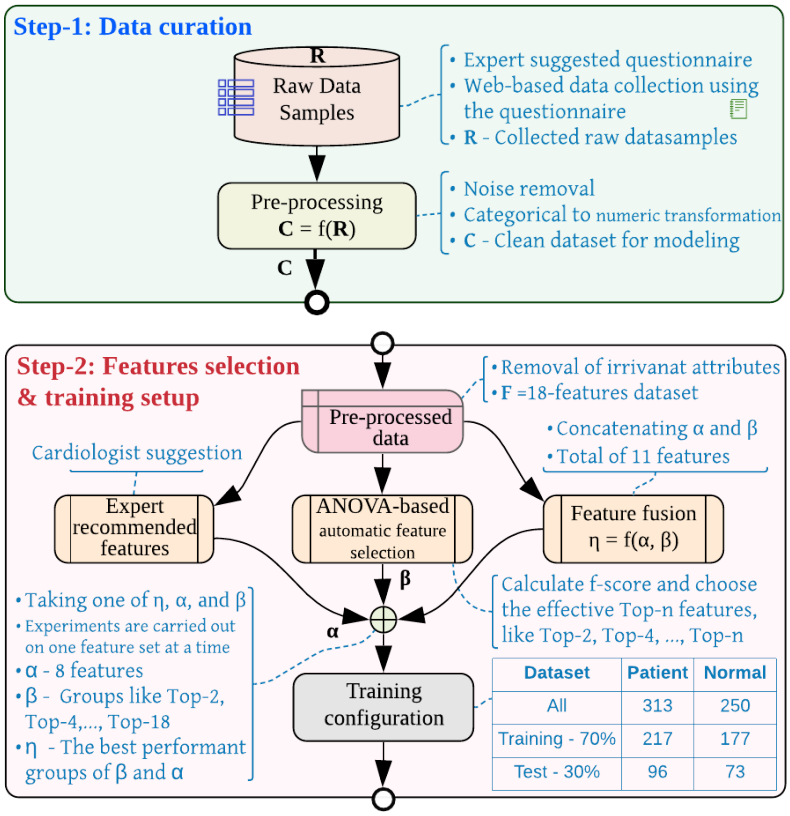


Fig. 4. Illustration of different sampling strategies: (a) traditional sparse sampling in atrous convolution and (b) dense sampling in our DCE block.

Ack: IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, 2021.



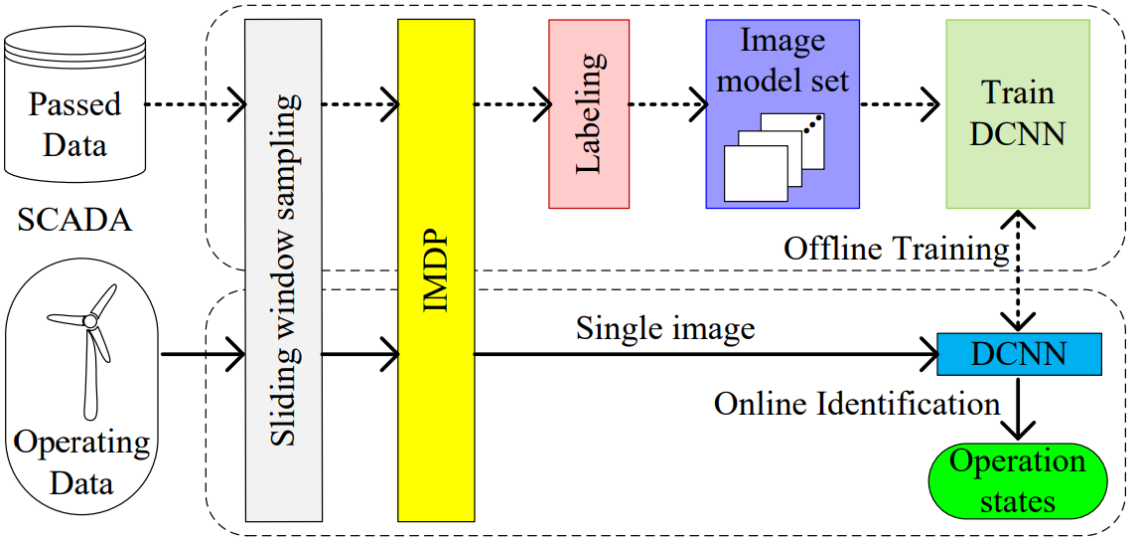


Fig. 1: The data flow diagram of the proposed framework for the state identification.
Ack: IEEE TII 2020.

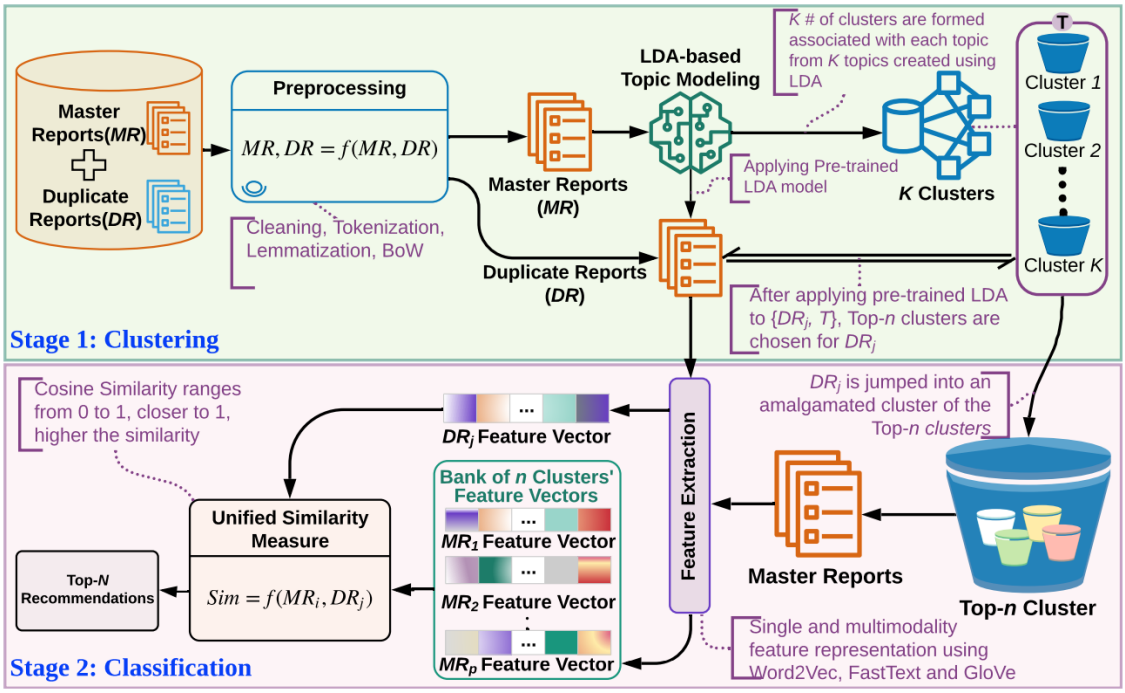


Fig. 3: The proposed double-tier topic modeling and classification model for duplicate bug report detection.

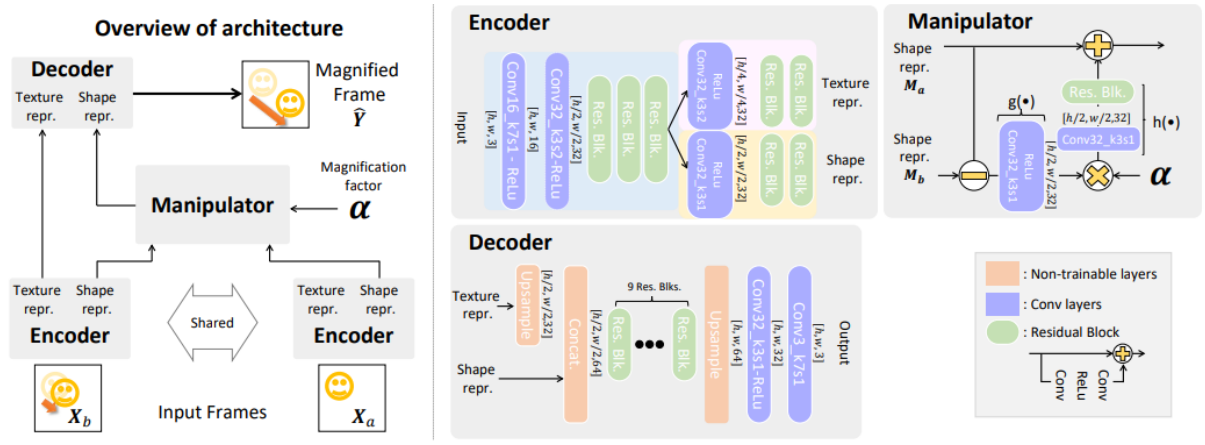


Fig. 4. The network architecture of Tae-Hyun O. et al., ECCV 2018: (a) Overview of the architecture. Our network consists of 3 main parts: the encoder, the manipulator, and the decoder. During training, the inputs to the network are two video frames, (X_a, X_b), with a magnification factor α, and the output is the magnified frame $\hat{Y}$. (b) Detailed diagram for each part. Conv<c>_k<k>_s<s> denotes a convolutional layer of c channels, k × k kernel size, and stride s.

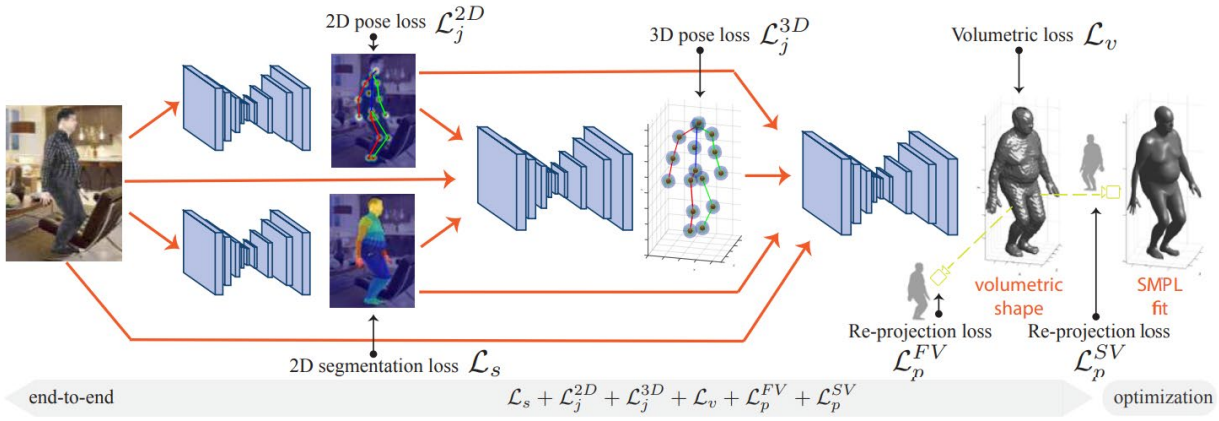


Fig. 5: BodyNet: End-to-end trainable network for 3D human body shape estimation from Gul V. et al., ECCV 2018. The input RGB image is first passed through subnetworks for 2D pose estimation and 2D body part segmentation. These predictions, combined with the RGB features, are fed to another network predicting 3D pose. All subnetworks are combined into a final network to infer volumetric shape. The 2D pose, 2D segmentation, and 3D pose networks are first pre-trained and then fine-tuned jointly for the task of volumetric shape estimation using multi-view re-projection losses. We fit the SMPL model to volumetric predictions for evaluation.

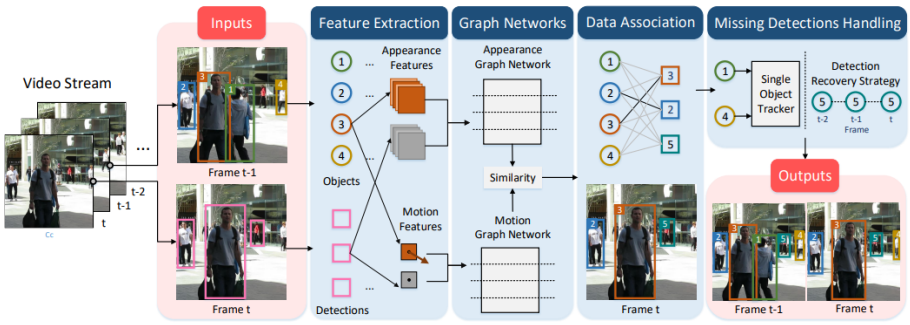


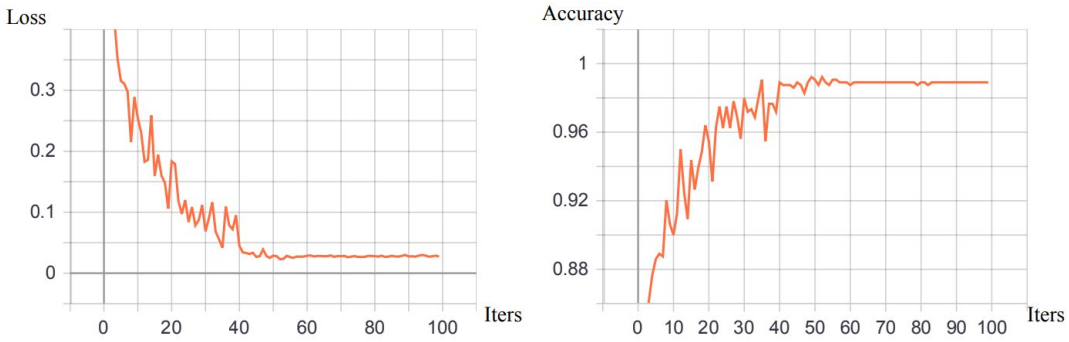
Figure 2. Pipeline of our MOT model. Inputs are the frame $t-1$, the frame t, the object set $\mathcal{O}_t$ from frame t and the detection set $\mathcal{D}_t$ at frame t. Outputs are tracking results. For example, there are four objects from frame $t-1$ and three detections from frame t, and their features are extracted and sent into the graph networks. Afterwards, data association and missing detections handling are performed.

C. Experimental Analysis

1. **Dataset:**
   a. Provide a summary of the benchmark datasets used to train and test your model.
   b. When you compare your performance with the existing models (in the literature), make sure that you follow the same dataset(s): the same train and test sets (or the same split ratio) as in the existing models.
   c. Describe how you handled class-imbalance conditions (if there was a such case).
   d. Describe how you addressed data scarcity (if there was a such case) and overfitting.

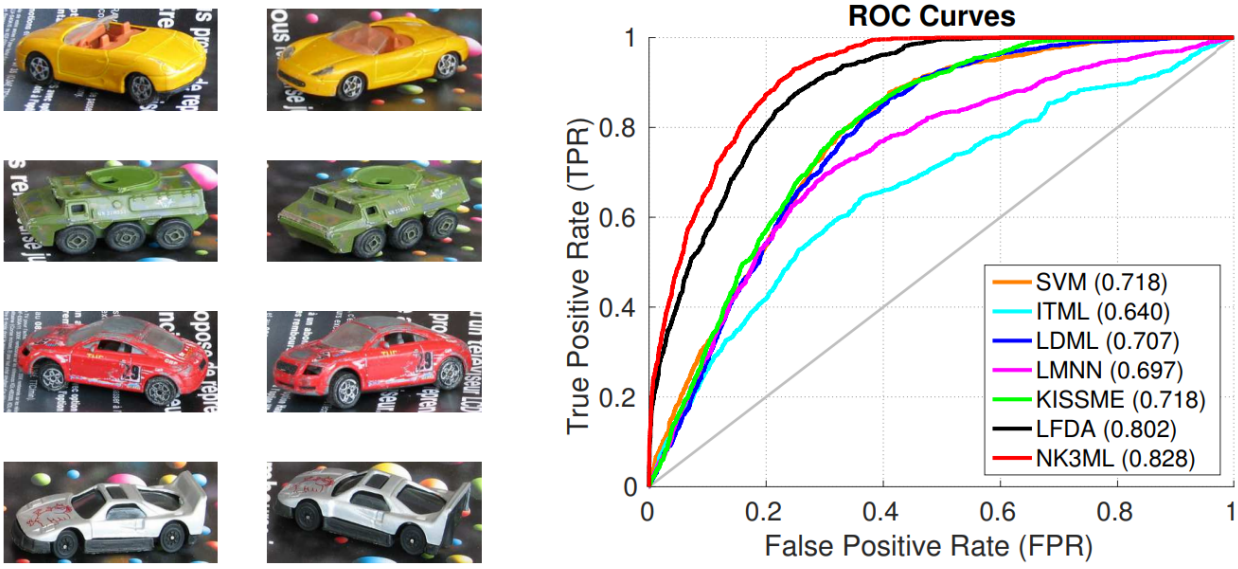
2. **Environment:** Describe the hardware and software platform used to build your model.

3. **Experimental findings:**
   a. The experimental study must include **quantitative** and **qualitative** (e.g., visualizing the results along with ground truths) analysis and interpretation of the results.
   b. Perform an **inference time** (per sample processing time) analysis of the model(s) on the benchmark test data set(s).
   c. Provide appropriate graphs, plots, and tables to summarize the results. Save the plots or simulation results .png by command, insert them in the appropriate places in the report, and index them. For example, in Python, the commands – `plt.savefig('my_plot.png', dpi=600)` or `plt.savefig('my_plot.eps', format='eps')` will help you save your plots on disk. A few examples are given on the next page.
   d. Elaborate the **insight** of the experimental study: Discuss the % of improvements you achieved compared to a baseline model. What was the reason for this improvement? how it was possible to achieve it.



(a) Loss change curve (b) Accuracy change curve

Fig. 2: Model training observation: Loss and accuracy vs iteration. Ack: IEEE TII 2020.



(a) (b)

Fig. 6: ToyCars dataset (a) Sample images (b) ROC curves and EER comparisons from Ali and Chaudhuri, ECCV 20108.

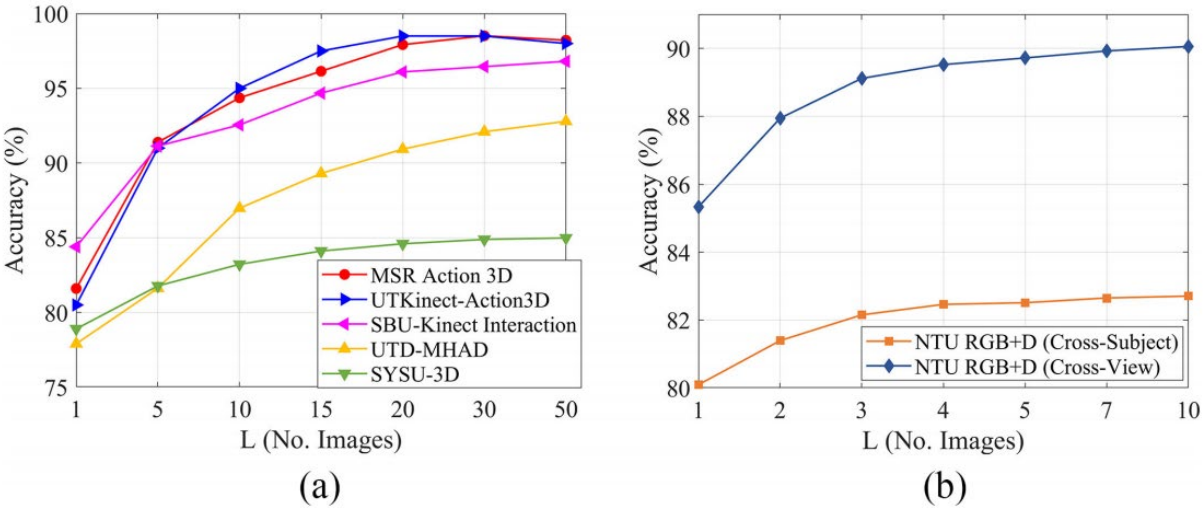
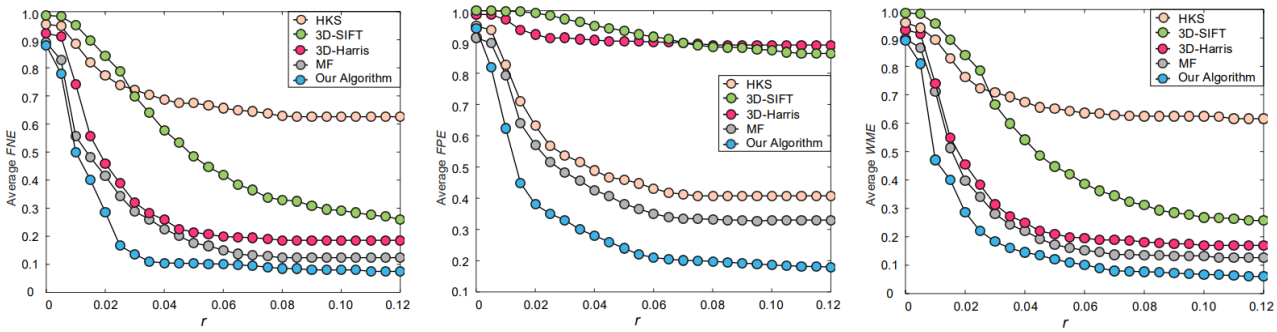


Fig. 7: Overall recognition accuracy of the proposed method on several datasets with different augmentation factors *L* from Thien H. et al., IEEE TII, 2020.



(a) FNE (b) FPE (c) WME

Fig. 8: Overall recognition accuracy of the proposed method on several datasets with different augmentation factors L. Ack: IEEE TM, 2020.

D. Clear your doubts

1. Do not hesitate to contact the instructor or your supervisor if you have any doubts.

E. Acknowledgment

- Some parts of Section A were adopted from https://classroom.udacity.com/
- PEP 8 – Style Guide for Python Code | peps.python.org

Appendix – Documentation

- What is documentation?
    - Elaboration of the statements or functions that come with or are embedded in the code.
    - It helps clarify complex parts of the program, making the code easier to navigate, and quickly conveying the integration and functionalities of different components of the program.
    - Different levels of documentation:
        - Line level – inline commentary
        - Module or function level – docstrings:
            - Docstrings are surrounded by triple quotes. The first line of the docstring is a brief explanation of the function's purpose
        - Project level – project documentation:
            - Provide a succinct detail for others to understand the purpose of your project and quickly get something working.
            - E.g., a README file of your project. At a bare minimum, it should provide the following details:
                i. What is this project all about and its purpose,
                ii. List of dependencies (software and hardware requirements, datasets, etc.),
                iii. Usage instructions (how to execute and test the program).
                iv. and provide sufficiently detailed instructions on how to use it.

Appendix – Code review

A. Once you or your team have completed the functionality of the program review the code by asking yourself/yourselves the following questions.

Is the code clean and modular?

- Can I/we understand the code easily?
- Does it use meaningful names and whitespace?
- Is there a duplicate code?
- Can I/we provide another layer of abstraction?
- Is each function and module necessary?
- Is each function or module too long?

Is the code efficient?

- Are there loops or other steps I/we can vectorize?
- Can I/we use better data structures to optimize any steps?
- Can I/we shorten the number of calculations needed for any steps?
- Can I/we use generators or multiprocessing to optimize any steps?

Is the documentation effective?

- Are inline comments concise and meaningful?

- Is there complex code that's missing documentation?
- Do functions use effective docstrings?
- Is the necessary project documentation provided?

Is the code well-tested?

- Does the code have high test coverage?
- Do tests check for interesting cases?
- Are the tests readable?
- Can the tests be made more efficient?

Is the logging effective?

- Are log messages clear, concise, and professional?
- Do they include all relevant and useful information?
- Do they use the appropriate logging level?

B. Modular Code Example

This example is taken from How to Write Clean and Modular Code by Pratik Raghuvanshi.

Task: Develop a program to curve a given list of test scores in three different ways to boost student's marks.

Solution 1: A non-modular way of coding

```
s = [88, 92, 79, 93, 85]
print(sum(s)/len(s))

s1 = []
for x in s:
    s1.append(x + 5)

print(sum(s1)/len(s1))

s2 = []
for x in s:
    s2.append(x + 10)

print(sum(s2)/len(s2))

s3 = []
for x in s:
    s3.append(x ** 0.5 * 10)

print(sum(s3)/len(s3))
```

Note: This code snippet is hard to understand and quite repetitive.

Solution 2: A modular way of coding.

```
import math
import numpy as np

def flat_curve(arr, n):
    return [i + n for i in arr]

def square_root_curve(arr):
    return [math.sqrt(i) * 10 for i in arr]

test_scores = [88, 92, 79, 93, 85]
curved_5 = flat_curve(test_scores, 5)
curved_10 = flat_curve(test_scores, 10)
curved_sqrt = square_root_curve(test_scores)

for score_list in test_scores, curved_5, curved_10, curved_sqrt:
    print(np.mean(score_list))
```

Note: This code snippet performs the same task as solution#1, but it is modular, clean, and easy to understand. However, this code also missing module-level comments and in-line commentary. You are required to add the comments.