Report for the
Software Engineering Assessment

Prepared by: Kavita Kaur
Submission Date: 23rd July 2021

Table of Contents

Section 1: Problem Statement

The problem statement comprised of various segments as listed below:
Data Tasks;
Machine Learning Tasks;
Machine Learning library components following Pytorch;
Graphical User Interface (GUI) and
Test Suite;

This report will attempt to address the problem statement by closely addressing these segments.

Section 2: The Main Classes of the Code

In order to solve the given problem statement, I have organised the architecture into three main classes namely the Data, Classifier and Trainer classes.

The code can be assessed from https://github.com/kavitakaur/tarzan.

The next few sections will go into a little bit more detail of each of the classes.

2.1 Data Class

The Data Class mainly encompasses the Data tasks of the problem statement in terms of preparation of data for the subsequent training. It calls upon the DatasetFromImage class which inherits from a torch class. This allows the other Pytorch functions to recognise that it is a torch data type.

Another function here is the ‘split’ function that splits the data into train and test sets based on proportion, 0.8 is the default value.

The anonymization step is done here and designed in a modular fashion so that different options can be added in easily. Two options are provided for this function.

The data is also converted into Pytorch’s dataloader format via the ‘train_dataloader’ and ‘test_dataloader’ functions.

2.2 Classifier Class

The Classifier Class serves to instantiate the option of the neural network to use by inputting the specific Class of neural network to use.

The various neural networks are designed in their own individual classes. There are five options offered within the code – MLP, CNN, FNN, UNet and VGGNet classes as can be seen in the core.py file. Of these, MLP and CNN have been implemented with the various layers and the forward method.

The Classifier Class also allows users to set the Optimiser and Learning Rate Scheduler from the various options provided. Two options for optimiser – Adam and SGD while three options for learning rate scheduler – step, multistep and exponential have been implemented.

2.3 Trainer Class

The Trainer Class takes in arguments that are derived from the Data and Classifier Classes, namely the prepared data and the instantiated neural network with its corresponding components. This class largely addresses the Machine Learning Tasks segment of the problem statement.

The dataset is trained here and the various machine learning tasks are also carried out here. The number of epochs is set at 5 by default but it could be altered by users if necessary.

Interfaces for ‘threshold’, ‘segment’, ‘predict’ and ‘count_classes’ methods have been done with the ‘predict’ (character recognition) and ‘count_classes’ (counting number of different characters) methods implemented.

Testing is also done in this class and the accuracy of the trained model can be evaluated via the test method.

Additionally, users can opt to save the trained model for later use. This is advantageous especially if the user wishes to access to the trained model again after closing the application to do testing for example. The entire training steps won’t have to be repeated. Take note that to do this, user would have to ensure that the neural network instantiated has the same architecture as that used in training.

Section 3: Visualising the Classes of the Code

UML diagrams like the class and package diagram were used to better visualise the various classes and their interactions.

3.1 UML Class Diagram

Figure 1 below shows the UML Class Diagram of the classes used in this AI core engine.

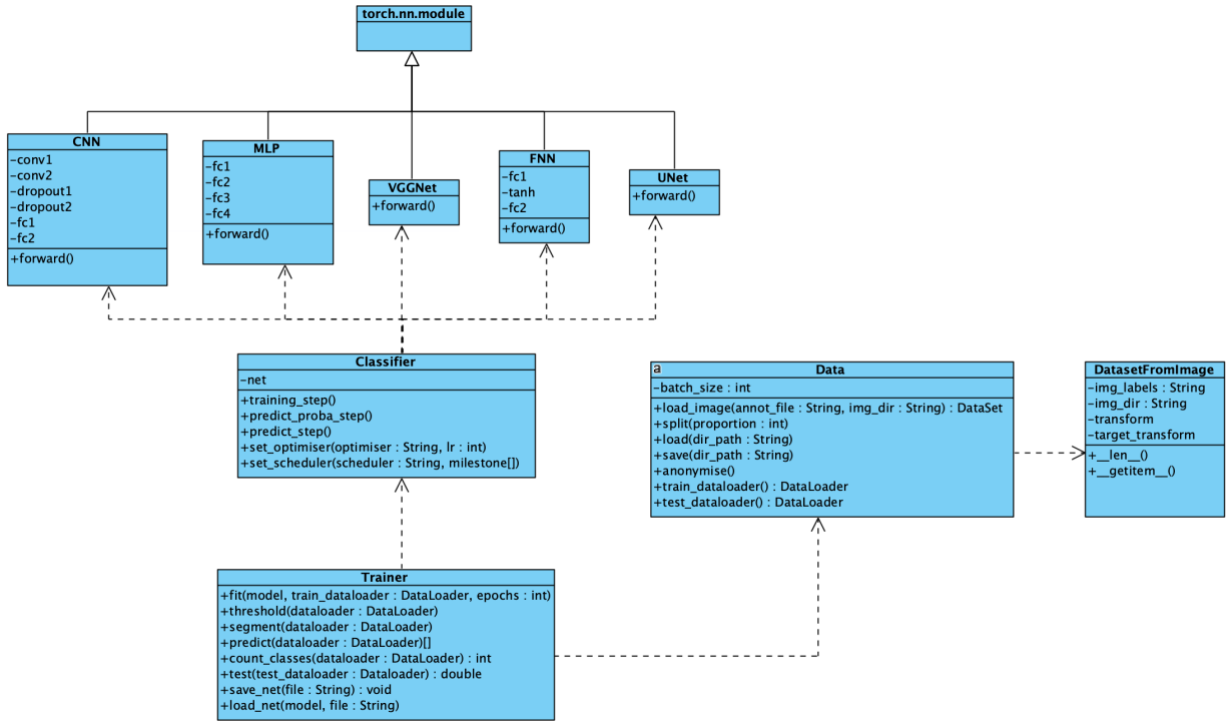

Figure 1. UML Class Diagram

We see that there are several dependency relationships between the various classes, where some of the elements or set of elements require/need/depend on other model element for specification or implementation. For example, the Trainer Class requires both the Data and Classifier Classes for its full implementation or operation.

3.2 UML Package Diagram

Figure 2 shows the package diagram of the various modules of the system and their interaction.

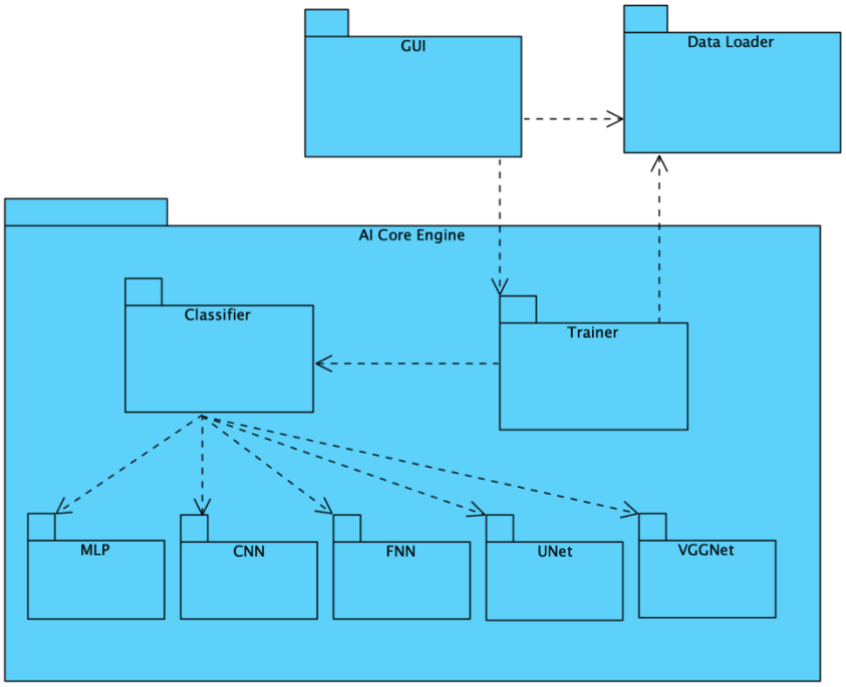

Figure 2. UML Package Diagram

Like in the class diagram, the arrows show the dependency relationships between the various modules. Here, I have also included the GUI and Data Loader module to illustrate the various interactions between each other and the Trainer class.

Section 4: The Code

Found in the file core.py within Tarzan folder and can be accessed via this link https://github.com/kavitakaur/tarzan/blob/master/tarzan/core.py.

To cater to the different options and combination, the code interfaces and implementations follows a plug-and-play design.

4.1 What the code does?

For tutorials on 1) how to predict images with an existing sample data and 2) how to save the parameters of trained model, refer to the README.md file on Github or the Appendices section, Appendix A and B of this report.

Section 5: Graphical User Interface (GUI)

Tkinter, Python's standard GUI framework, was used to design a simple GUI which allows users to use the above-designed AI core engine.

The code can be found within gui.py and can be accessed via this link https://github.com/kavitakaur/tarzan/blob/master/gui.py.

5.1 The Code – Key Features

I have arranged the code in a class framework with a Window Class, where the widgets used for the user interface is coded into. Of note are the Frame – container for grouping and organising the various widgets, Label – the text visible on the window, Entry – the input field and Button – the button.

Within this class are two methods – 'get_data' and 'ML_tasks' which connects the GUI to the data loader and AI core engine respectively.

5.2 What the code does?

Figure 3 shows the view of the Tkinter GUI that user will interact with. This is just a work-in-progress and adjustments can be made over time to add features and enhance the look of it.

It has been divided into two parts - "PART 1: LOAD DATA" and "PART 2: TRAIN DATA".

In the first part, users will have to provide two paths, one is the path to the annotation file while the other is to the image file. An example of what the input should be has been provided. Once done, they will click the "Get Data" button.

For the second part, they would input their choices of the DL model, optimiser and learning rate scheduler before clicking the "Start Training" button. The various options have been provided above the input boxes.

This is not the best user interface and is as such due to time limitation. One future change I would like to make is to replace the three input boxes under the "PART 2: TRAIN DATA" section with a dropdown menu or checkboxes so that it is easier for users to select the different options. This will also lead to lesser error made by the user in terms of typing errors.

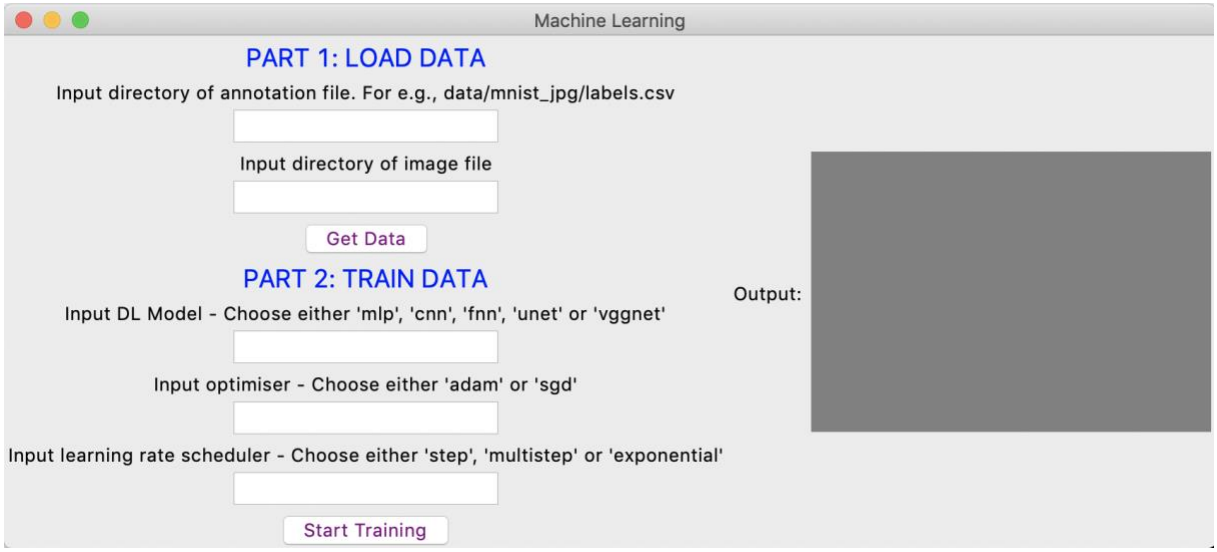

Figure 3. GUI

Figure 4 shows the same user interface but this time after the user have filled in the input boxes and run the program via the “Get Data” and “Start Training” buttons.

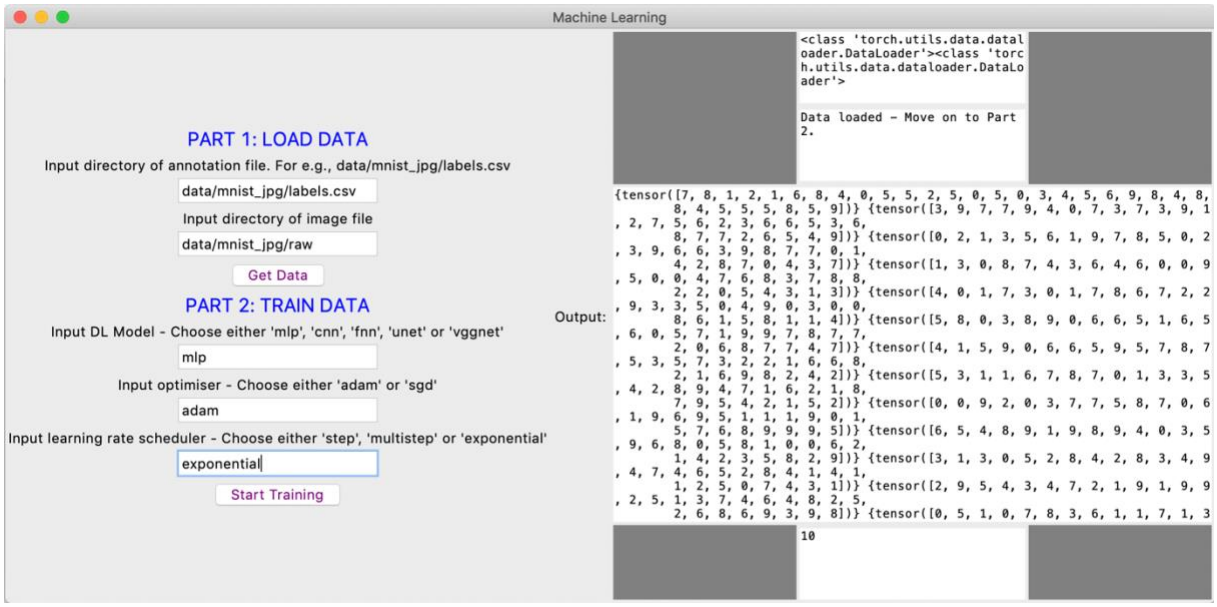

Figure 4. GUI with user input

As you can see, the first box in the “Output” field (right side of the window) shows the type of the train and test dataloader. I chose to input this to give an indication to the user that an event has occurred, that the program has run and fetched the necessary data. Should the grey box remain as in Figure 3, there is probably an error. Just to give the user a clearer prompt that they can move on, the program will generate a “Data loaded – Move on to Part 2” message in the “Output” field.

This is another area for improvement – inputting the exact error message as can be observed from the command line into the “Output” field for the user’s view.

For Part 2, I have opted to print out the output of the 'predict' and 'count_classes' methods of the Trainer class in core.py to indicate to the user that the program has successfully run. Furthermore, these constitute as the desired output that the user wants.

Section 6: Test Suites

To test and verify the code for correctness, several test cases were designed and carried out as described below.

Code for the unit tests can be found within unit_test.py and can be accessed via this link https://github.com/kavitakaur/tarzan/blob/master/test_suite.py.

6.1 Unit Tests

For the unit tests, I used the unittest module for constructing and running the tests. Three testcases were designed, one each for the Data, Classifier and Trainer Classes. These were created by subclassing unnittest.TestCase.

6.1.1 Unit Tests for Data Class

Within the TestData class, three individual tests were designed for the load, split and dataloader methods of the Data class.

The setup method runs before each test method is executed and could also serve to test that the images are loaded without error.

'test_load' method subsets the first image, label pair and check that they are torch tensors of the appropriate size.

'test_split' method checks if the dataset is split correctly according to the stated proportion.

'test_both_dataloader' method checks that the first image, label pair of both the train and test datasets are torch tensors of the appropriate size.

6.1.2 Unit Tests for Classifier Class

Within the TestClassifier class, three individual tests were designed for the training_step, predict_proba_step and predict_step methods of the Classifier class.

The setup method runs before each test method is executed and could also serve to test that the Classifier instantiates.

'test_training_step' method checks for the accuracy of the size of the torch tensor for loss within a batch.

'test_ predict_proba_step' method checks for the accuracy of the size of the torch tensor for y_hat within a batch.

'test_ predict_step' method checks for the accuracy of the size of the torch tensor for predicted_labels within a batch.

6.1.3 Unit Tests for Trainer Class

Within the TestTrainer class, three individual tests were designed for the predict, count_classes and test methods of the Trainer class.

The setup method runs before each test method is executed and could also serve to test that the Trainer instantiates.

'test_predict' method checks for the accuracy of the size of the torch tensor for the first element of list_pred.

'test_ count_classes' method checks that the output for the number of characters is less than or equal to the total number of different characters, 10 in this case.

'test_test' method checks that the output for accuracy (in per cent) is less than or equal to 100.

6.2 System Test

For the system test to test and validate the complete and fully integrated software product, several types of testing could be done. For example, a usability test which would mainly focus on the user's ease in using the application and handling the controls. This could be done with a small sample size just before full scale deployment of the software.

Another test that could be done is a load test to know if this software could perform under real-life load. As I have run the various files in this report like test.py and gui.py and obtained the desired output, we know that this software at least runs for the MNIST dataset.

REFERENCES

Dependency in UML
Retrieved from: https://www.uml-diagrams.org/dependency.html

Introduction to GUI with Tkinter in Python
Retrieved from: https://www.datacamp.com/community/tutorials/gui-tkinter-python

Python's Unit testing framework.
Retrieved from: https://docs.python.org/3/library/unittest.html

What is System Testing? Types and Definition with Example.
Retrieved from: https://www.guru99.com/system-testing.html

APPENDICES

Appendix A: Tutorial - Predicting images (with existing example data)

Import dependencies.

```
import tarzan.core as tz
import torch
from torchvision import transforms, datasets
```

User provides annotation file of images in .csv format (with header). First column of .csv file is image name, second column is labels. User to provide file path of directory containing all images. As an example, you could refer to the MNIST_jpg data which is provided in this repository.

Transformation of the data in terms of normalisation can also be done in this step.

```
annot_path = 'data/mnist_jpg/labels.csv'
mnist_path = 'data/mnist_jpg/raw'
save_path = 'data/mnist_jpg/processed'

transformations = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

dataset = tz.Data()
dataset.load_image(
    annot_path,
    mnist_path,
    transformations
)
```

The data can then be split into training and test sets with default proportion of 80% train & 20% test but user is allowed to change this by inputting a value between 0 and 1 - with 0.5 to mean 50%, etc.

This data could be saved and loaded to and from a selected path. This step is optional.

The next steps convert train and test dataset into dataloader format so that it can be run in the Trainer Class.

```
dataset.split()
# dataset.save(save_path)
# dataset.load(save_path)

train_dataloader = dataset.train_dataloader()
test_dataloader = dataset.test_dataloader()
```

User needs to instantiate a network under the Classifier Class with a string, either "mlp" or "cnn" (these are the 2 choices for now. More can be easily added).

Under the Classifier Class is also where user selects the option for Optimiser (either "adam" or "sgd") and Learning Rate Scheduler (either "exponential" or "multistep", more options could also be added).

Both the set_optimiser and set_scheduler methods require inputs to be in string form and a ValueError with the input options is raised should the user provide an incorrect input.

The trainer is then instantiated and set to run. User can change the epochs by inputting an int value, otherwise the default value is set at 5 at the moment.

```
# instantiate the neural network
clf = tz.Classifier("mlp")
clf.set_optimiser("adam")
clf.set_scheduler("exponential")

trainer = tz.Trainer()
trainer.fit(clf, train_dataloader, epochs=1)
# acc = trainer.test(test_dataloader)
```

Here we can run the machine learning tasks. What has been provided are for the character recognition and counting the number of different characters.

```
predictions = trainer.predict(test_dataloader)
trainer.count_classes(test_dataloader)
```

Appendix B: Tutorial - To save parameters of trained model so that it can be loaded again

User provides the file path of directory for where to save the trained model. This option is available so that the user can go back to the trained model again to use it in testing for example, without having to carry out the whole training steps again.

Without this step, user will have to retrain the data again if they wish to refer to it again after shutting down the program.

```
# save parameters of trained model
PATH = "net_state/mlp.pth"
trainer.save_net(PATH)
```

To load the trained parameters, user has to instantiate the corresponding neural network first. Ensure that the neural network instantiated has the same architecture as the one used in the training.

```
# Instantiate the nn used to train data
clf = tz.Classifier("mlp")
# clf.set_optimiser("adam")
# clf.set_scheduler("exponential")

trainer = tz.Trainer()
```

```
PATH = "net_state/mlp.pth"
trainer.load_net(clf, PATH)
```