Computer Science NEA Report

An investigation into how Artificial Neural Networks work, the effects of their hyper-parameters and their applications in Image Recognition.

Max Cotton

Contents

1 Analysis

1.1 About

Artificial Intelligence mimics human cognition in order to perform tasks and learn from them, Machine Learning is a subfield of Artificial Intelligence that uses algorithms trained on data to produce models (trained programs) and Deep Learning is a subfield of Machine Learning that uses Artificial Neural Networks, a process of learning from data inspired by the human brain. Artificial Neural Networks can be trained to learn a vast number of problems, such as Image Recognition, and have uses across multiple fields, such as medical imaging in hospitals. This project is an investigation into how Artificial Neural Networks work, the effects of changing their hyper-parameters and their applications in Image Recognition. To achieve this, I will derive and research all theory behind the project, using sources such as IBM's online research, and develop Neural Networks from first principles without the use of any third-party Machine Learning libraries. I then will implement the Artificial Neural Networks in Image Recognition, by creating trained models and will allow for experimentation of the hyper-parameters of each model to allow for comparisons between each model's performances, via a Graphical User Interface.

1.2 Interview

In order to gain a better foundation for my investigation, I presented my prototype code and interviewed the head of Artificial Intelligence at Cambridge Consultants for input on what they would like to see in my project, these were their responses:

- Q:"Are there any good resources you would recommend for learning the theory behind how Artificial Neural Networks work?"

  A:"There are lots of usefull free resources on the internet to use. I particullarly like the platform 'Medium' which offers many scientific articles as well as more obvious resources such as IBMs'."

- Q:"What do you think would be a good goal for my project?"

  A:"I think it would be great to aim for applying the Neural Networks on Image Recognition for some famous datasets. For you, I would recommend the MNIST dataset as a goal."

- Q:"What features of the Artificial Neural Networks would you like to be able to experiment with?"

  A:"I'd like to be able to experiment with the number of layers and the number of neurons in each layer, and then be able to see how these changes effect the performance of the model. I can see that you've utilised the Sigmoid transfer function and I would recommend having the option to test alternatives such as the ReLu transfer function, which will help stop issues such as a vanishing gradient."

- Q:"What are some practical constraints of AI?"

  A:"Training AI models can require a large amount of computing power, also large datasets are needed for training models to a high accuracy which can be hard to obtain."

- Q:"What would you say increases the computing power required the most?"

  A:"The number of layers and neurons in each layer will have the greatest effect on the computing power required. This is another reason why I recommend adding the ReLu transfer function as it updates the values of the weights and biases faster than the Sigmoid transfer function."

- Q:"Do you think I should explore other computer architectures for training the models?"

  A:"Yes, it would be great to add support for using graphics cards for training models, as this would be a vast improvement in training time compared to using just CPU power."

- Q:"I am also creating a user interface for the program, what hyper-parameters would you like to be able to control through this?"

  A:"It would be nice to control the transfer functions used, as well as the general hyper-parameters of the model. I also think you could add a progress tracker to be displayed during training for the user."

- Q:"How do you think I should measure the performance of models?"

  A:"You should show the accuracy of the model's predictions, as well as example incorrect and correct prediction results for the trained model. Additionally, you could compare how the size of the training dataset effects the performance of the model after training, to see if a larger dataset would seem beneficial."

- Q:"Are there any other features you would like add?"

  A:"Yes, it would be nice to be able to save a model after training and have the option to load in a trained model for testing."

1.3 Project Objectives

- Learn how Artificial Neural Networks work and develop them from first principles

- Implement the Artificial Neural Networks by creating trained models on image datasets

- Allow use of Graphics Cards for faster training
- Allow for the saving of trained models

- Develop a Graphical User Interface
  - Provide controls for hyper-parameters of models
  - Display and compare the results each model's predictions

1.4 Theory behind Artificial Neural Networks

From an abstract perspective, Artificial Neural Networks are inspired by how the human mind works, by consisting of layers of 'neurons' all interconnected via different links, each with their own strength. By adjusting these links, Artificial Neural Networks can be trained to take in an input and give its best prediction as an output.

1.4.1 Structure

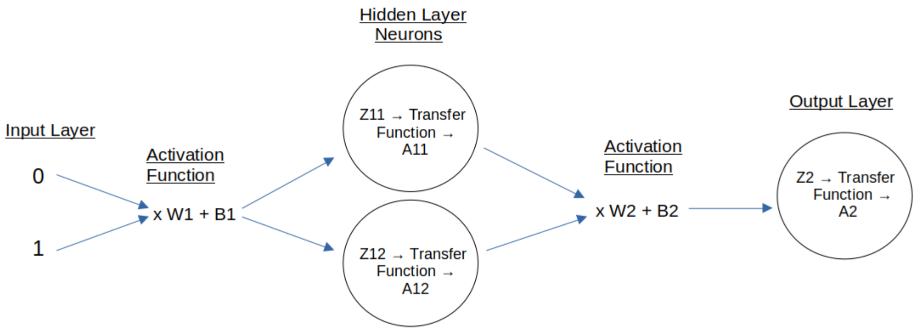


Figure 1: This shows an Artificial Neural Network with one single hidden layer and is known as a Shallow Neural Netwok.

I have focused on Feed-Forward Artificial Neural Networks, where values are entered to the input layer and passed forwards repetitively to the next layer until reaching the output layer. Within this, I have learnt two types of Feed-Forward Artificial Neural Networks: Perceptron Artificial Neural Networks, that contain no hidden layers and are best at learning more linear patterns and Multi-Layer Perceptron Artificial Neural Networks, that contain at least one hidden layer, as a result increasing the non-linearity in the Artificial Neural Network and allowing it to learn more complex / non-linear problems.

Multi-Layer Perceptron Artificial Neural Networks consist of:

- An input layer of input neurons, where the input values are entered.
- Hidden layers of hidden neurons.
- An output layer of output neurons, which outputs the final prediction.

To implement an Artificial Neural Network, matrices are used to represent the layers, where each layer is a matrice of the layer's neuron's values. In order to use matrices for this, the following basic theory must be known about them:

- When Adding two matrices, both matrices must have the same number of rows and columns. Or one of the matrices can have the same number of rows but only one column, then be added by element-wise addition where each element is added to all of the elements of the other matrix in the same row.
- When multiplying two matrices, the number of columns of the 1st matrix must equal the number of rows of the 2nd matrix. And the result will have the same number of rows as the 1st matrix, and the same number of columns as the 2nd matrix. This is important, as the output of one layer must be formatted correctly to be used with the next layer.
- In order to multiply matrices, I take the 'dot product' of the matrices, which multiplies the row of one matrice with the column of the other, by multiplying matching members and then summing up.
- Transposing a matrix will turn all rows of the matrix into columns and all columns into rows.
- A matrix of values can be classified as a rank of Tensors, depending on the number of dimensions of the matrix. (Eg: A 2-dimensional matrix is a Tensor of rank 2)

I have focused on just using Fully-Connected layers, that will take in input values and apply the following calculations to produce an output of the layer:

- An Activation function
  - This calculates the dot product of the input matrix with a weight matrix, then sums the result with a bias matrix
- A Transfer function
  - This takes the result of the Activation function and transfers it to a suitable output value as well as adding more non-linearity to the Neural Network.
  - For example, the Sigmoid Transfer function converts the input to a number between zero and one, making it usefull for logistic regression where the output value can be considered as closer to zero or one allowing for a binary classification of predicting zero or one.

1.4.2 How Artificial Neural Networks learn

To train an Artificial Neural Network, the following processes will be carried out for each of a number of training epochs:

- Forward Propagation:
  - The process of feeding inputs in and getting a prediction (moving forward through the network)
- Back Propagation:
  - The process of calculating the Loss in the prediction and then adjusting the weights and biases accordingly
  - I have used Supervised Learning to train the Artificial Neural Networks, where the output prediction of the Artificial Neural Network is compared to the values it should have predicted. With this, I can calculate the Loss value of the prediction (how wrong the prediction is from the actual value).
  - I then move back through the network and update the weights and biases via Gradient Descent:
    - Gradient Descent aims to reduce the Loss value of the prediction to a minimum, by subtracting the rate of change of Loss with respect to the weights/ biases, multiplied with a learning rate, from the weights/biases.
    - To calculate the rate of change of Loss with respect to the weights/biases, you must use the following calculus methods:
      - Partial Differentiation, in order to differentiate the multivariable functions, by taking respect to one variable and treating the rest as constants.
      - The Chain Rule, where for $y = f(u)$ and $u = g(x)$, $\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} * \frac{\partial u}{\partial x}$
      - For a matrice of $f(x)$ values, the matrice of $\frac{\partial f(x)}{\partial x}$ values is known as the Jacobian matrix
    - This repetitive process will continue to reduce the Loss to a minimum, if the learning rate is set to an appropriate value
    - However, during backpropagation some issues can occur, such as the following:
      - Finding a false local minimum rather than the global minimum of the function
      - Having an 'Exploding Gradient', where the gradient value grows exponentially to the point of overflow errors
      - Having a 'Vanishing Gradient', where the gradient value decreases to a very small value or zero, resulting in a lack of updating values during training.

1.5 Theory Behind Deep Artificial Neural Networks

1.5.1 Setup

- Where a layer takes the previous layer's output as its input X

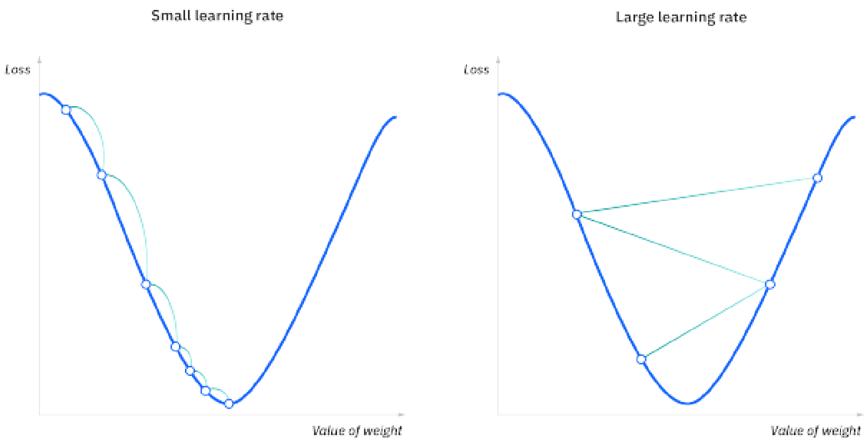


Figure 2: Gradient Descent
sourced from https://www.ibm.com/topics/gradient-descent

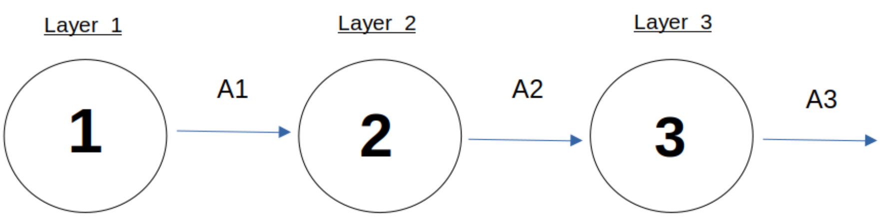


Figure 3: This shows an abstracted view of an Artificial Neural Network with multiple hidden layers and is known as a Deep Neural Netwok.

- Then it applies an Activation function to X to obtain Z, by taking the dot product of X with a weight matrix W, then sums the result with a bias matrix B. At first the weights are intialised to random values and the biases are set to zeros.
  - $Z = W * X + B$
- Then it applies a Transfer function to Z to obtain the layer's output
  - For the output layer, the sigmoid function (explained previously) must be used for either for binary classification via logistic regression, or for multi- class classification where it predicts the output neuron, and the associated class, that has the highest value between zero and one.
    * Where $sigmoid(Z) = \frac{1}{1+e^{-Z}}$

- However, for the input layer and the hidden layers, another transfer function known as ReLu (Rectified Linear Unit) can be better suited as it produces largers values of $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial B}$ for Gradient Descent than Sigmoid, so updates at a quicker rate.
    - Where $relu(Z) = max(0, Z)$

1.5.2 Forward Propagation:

- For each epoch the input layer is given a matrix of input values, which are fed through the network to obtain a final prediction A from the output layer.

1.5.3 Back Propagation:

- First the Loss value L is calculated using the following Log-Loss function, which calculates the average difference between A and the value it should have predicted Y. Then the average is found by summing the result of the Loss function for each value in the matrix A, then dividing by the number of predictions m, resulting in a Loss value to show how well the network is performing.
    - Where $L = -(\frac{1}{m}) * \sum(Y * log(A) + (1 - Y) * log(1 - A))$ and "log()" is the natural logarithm
- I then move back through the network, adjusting the weights and biases via Gradient Descent. For each layer, the weights and biases are updated with the following formulae:
    - $W = W - learningRate * \frac{\partial L}{\partial W}$
    - $B = B - learningRate * \frac{\partial L}{\partial B}$
- The derivation for Layer 2's $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial B}$ can be seen below:
    - Functions used so far:
        1. $Z = W * X + B$
        2. $A_{relu} = max(0, Z)$
        3. $A_{sigmoid} = \frac{1}{1+e^{-Z}}$
        4. $L = -(\frac{1}{m}) * \sum(Y * log(A) + (1 - Y) * log(1 - A))$
    - $\frac{\partial L}{\partial A2} = \frac{\partial L}{\partial A3} * \frac{\partial A3}{\partial Z3} * \frac{\partial Z3}{\partial A2}$
      By using function 1, where A2 is X for the 3rd layer, $\frac{\partial Z3}{\partial A2} = W3$
      => $\frac{\partial L}{\partial A2} = \frac{\partial L}{\partial A3} * \frac{\partial A3}{\partial Z3} * W3$
    - $\frac{\partial L}{\partial W2} = \frac{\partial L}{\partial A2} * \frac{\partial A2}{\partial Z2} * \frac{\partial Z2}{\partial W2}$
      By using function 1, where A1 is X for the 2nd layer, $\frac{\partial Z2}{\partial W2} = A1$
      => $\frac{\partial L}{\partial W2} = \frac{\partial L}{\partial A2} * \frac{\partial A2}{\partial Z2} * A1$
    - $\frac{\partial L}{\partial B2} = \frac{\partial L}{\partial A2} * \frac{\partial A2}{\partial Z2} * \frac{\partial Z2}{\partial B2}$
      By using function 1, $\frac{\partial Z2}{\partial B2} = 1$
      => $\frac{\partial L}{\partial W2} = \frac{\partial L}{\partial A2} * \frac{\partial A2}{\partial Z2} * 1$

- As you can see, when moving back through the network, the $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial B}$ of the layer can be calculated with the rate of change of loss with respect to its output, which is calculated by the previous layer using the above formula; the derviative of the layer's transfer function, and the layers input (which in this case is A1)
  - Where by using function 2, $\frac{\partial A_{relu}}{\partial Z} = 1$ when $Z >= 0$ otherwise $\frac{\partial A_{relu}}{\partial Z} = 0$
  - Where by using function 3, $\frac{\partial A_{sigmoid}}{\partial Z} = A * (1 - A)$
- At the start of backpropagation, the rate of change of loss with respect to the output layer's output has no previous layer's caluculations, so instead it can be found with the derivative of the Log-Loss function, as shown in the following:
  - Using function 4, $\frac{\partial L}{\partial A} = (-\frac{1}{m})(\frac{Y-A}{A*(1-A)})$

1.6 Theory behind training the Artificial Neural Networks

Training an Artificial Neural Network's weights and biases to predict on a dataset, will create a trained model for that dataset, so that it can predict on future images inputted. However, training Artificial Neural Networks can involve some problems such as Overfitting, where the trained model learns the patterns of the training dataset too well, causing worse prediction on a different test dataset. This can occur when the training dataset does not cover enough situations of inputs and the desired outputs (by being too small for example), if the model is trained for too many epochs on the poor dataset and having too many layers in the Neural Network. Another problem is Underfitting, where the model has not learnt the patterns of the training dataset well enough, often when it has been trained for too few epochs, or when the Neural Network is too simple (too linear).

1.6.1 Datasets

- MNIST dataset
  - The MNIST dataset is a famouse dataset of images of handwritten digits from zero to ten and is commonly used to test the performance of an Artificial Neural Network.
  - The dataset consists of 60,000 input images, made up from 28x28 pixels and each pixel has an RGB value from 0 to 255
  - To format the images into a suitable format to be inputted into the Artificial Neural Networks, each image's matrice of RGB values are 'flattened' into a 1 dimensional matrix of values, where each element is also divided by 255 (the max RGB value) to a number between 0 and 1, to standardize the dataset.
  - The output dataset is also loaded, where each output for each image is an array, where the index represents the number of the image, by having a 1 in the index that matches the number represented and zeros for all other indexes.

  - To create a trained Artificial Neural Network model on this dataset, the model will require 10 output neurons (one for each digit), then by using the Sigmoid Transfer function to output a number between one and zero to each neuron, whichever neuron has the highest value is predicted. This is multi-class classification, where the model must predict one of 10 classes (in this case, each class is one of the digits from zero to ten).

- Cat dataset
  - I will also use a dataset of images sourced from https://github.com/marcopeix, where each image is either a cat or not a cat.
  - The dataset consists of 209 input images, made up from 64x64 pixels and each pixel has an RGB value from 0 to 255
  - To format the images into a suitable format to be inputted into the Artificial Neural Networks, each image's matrice of RGB values are 'flattened' into a 1 dimensional array of values, where each element is also divided by 255 (the max RGB value) to a number between 0 and 1, to standardize the dataset.
  - The output dataset is also loaded, and is reshaped into a 1 dimensional array of 1s and 0s, to store the output of each image (1 for cat, 0 for non cat)
  - To create a trained Artificial Neural Network model on this dataset, the model will require only 1 output neuron, then by using the Sigmoid Transfer function to output a number between one and zero for the neuron, if the neuron's value is closer to 1 it predicts cat, otherwise it predicts not a cat. This is binary classification, where the model must use logistic regression to predict whether it is a cat or not a cat.

- XOR dataset
  - For experimenting with Artificial Neural Networks, I solve the XOR gate problem, where the Neural Network is fed input pairs of zeros and ones and learns to predict the output of a XOR gate used in circuits.
  - This takes much less computation time than image datasets, so is usefull for quickly comparing different hyper-parameters of a Network, whilst still not being linearly seperable.

1.6.2 Theory behind using Graphics Cards to train Artificial Neural Networks

Graphics Cards consist of many Tensor cores which are processing units specialiased for matrix operations for calculating the co-ordinates of 3D graphics, however they can be used here for operating on the matrices in the network at a much faster speed compared to CPUs. GPUs also include CUDA cores which act as an API to the GPU's computing to be used for any operations (in this case training the Artificial Neural Networks).

2 Design

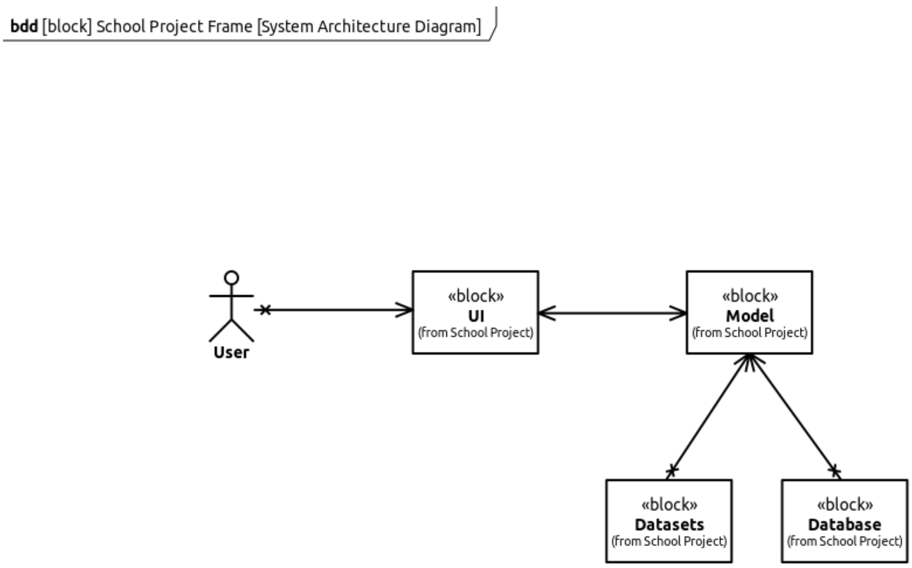
2.1 Introduction

The following design focuses have been made for the project:

- The program will support multiple platforms to run on, including Windows and Linux.
- The program will use python3 as its main programming language.
- I will take an object-orientated approach to the project.
- I will give an option to use either a Graphics Card or a CPU to train and test the Artificial Neural Networks.

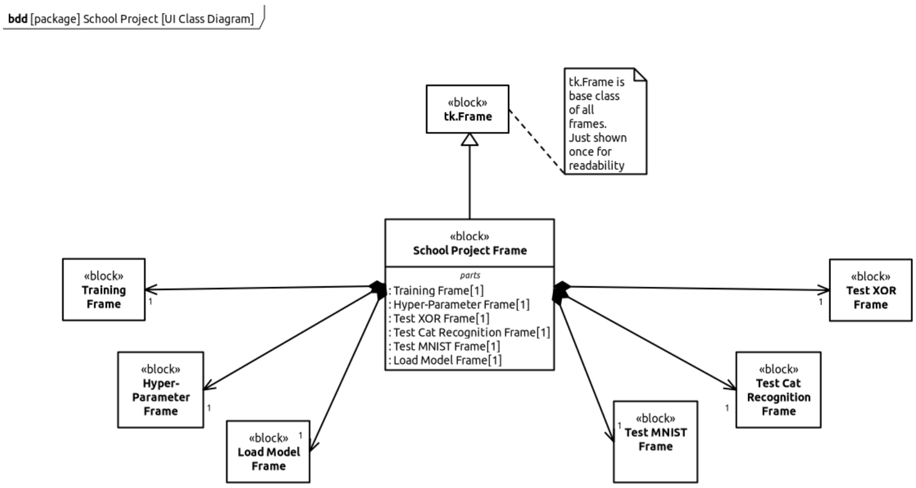
I will also be using SysML for designing the following diagrams.

2.2 System Architecture

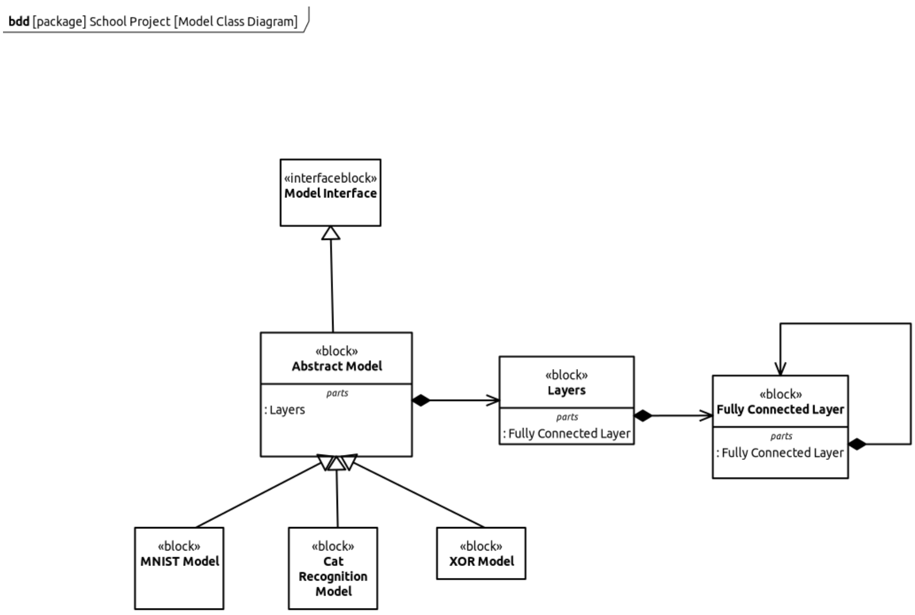


2.3 Class Diagrams

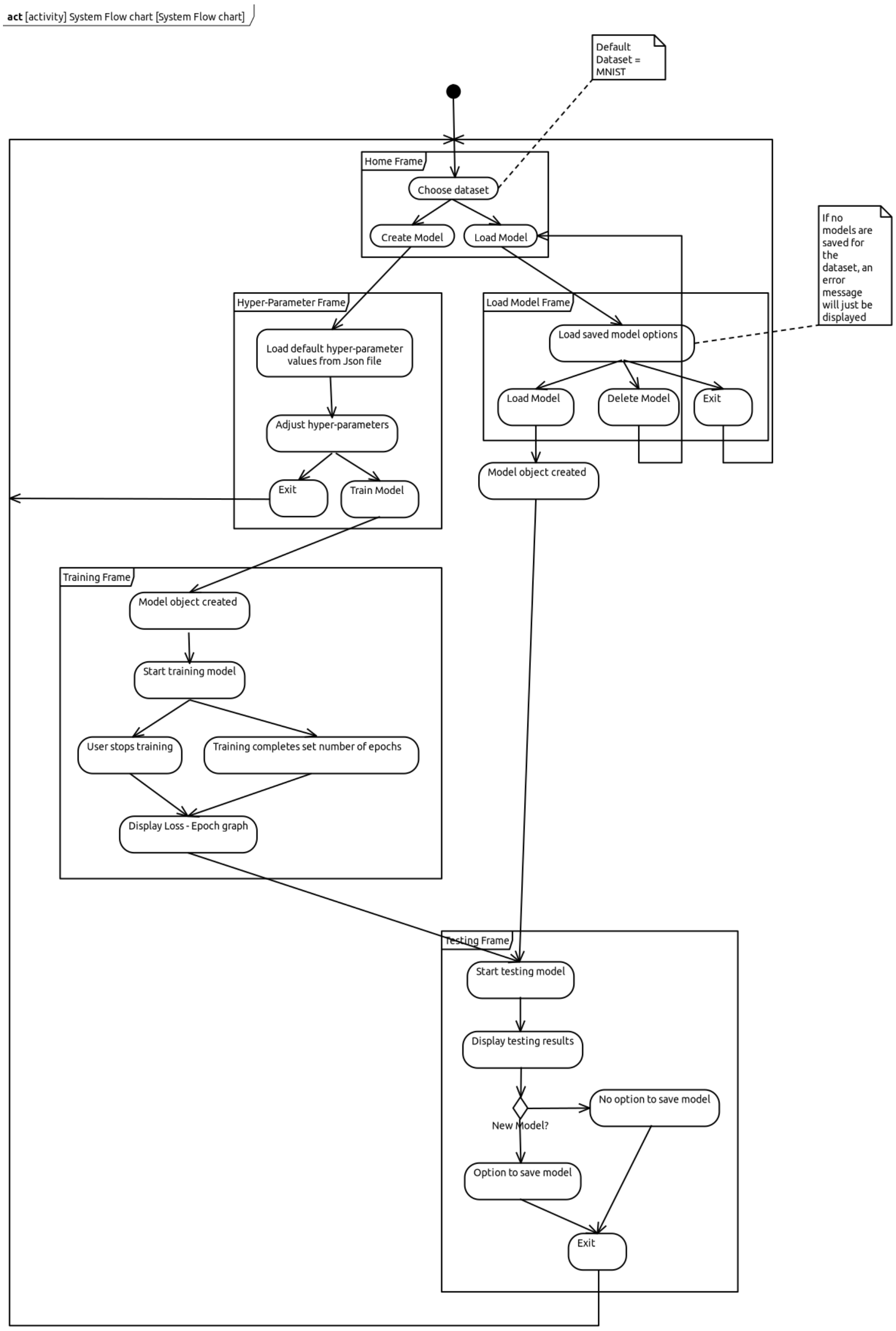
2.3.1 UI Class Diagram



2.3.2 Model Class Diagram



2.4 System Flow chart



2.5 Algorithms

Refer to Analysis for the algorithms behind the Artificial Neural Networks.

2.6 Data Structures

I will use the following data structures in the program:

- Standard arrays for storing data contiguously, for example storing the shape of the Artificial Neural Network's layers.
- Tuples where tuple unpacking is usefull, such as returning multiple values from methods.
- Dictionaries for loading the default hyper-parameter values from a JSON file.
- Matrices to represent the layers and allow for a varied number of neurons in each layer. To represent the Matrices I will use both numpy arrays and cupy arrays.
- A Doubly linked list to represent the Artificial Neural Network, where each node is a layer of the network. This will allow me to traverse both forwards and backwards through the network, as well as storing the first and last layer to start forward and backward propagation respectively.

2.7 File Structure

I will use the following file structures to store necessary data for the program:

- A JSON file for storing the default hyper-parameters for creating a new model for each dataset.
- I will store the image dataset files in a 'datasets' directory. The dataset files will either be a compressed archive file (such as .pkl.gz files) or of the Hierarchical Data Format (such as .h5) for storing large datasets with fast retrieval.
- I will save the weights and biases of saved models as numpy arrays in .npz files (a zipped archive file format) in a 'saved-models' directory, due to their compatibility with the numpy library.

2.8 Database Design

I will use the following Relational database design for saving models, where the dataset, name and features of the saved model (including the location of the saved models' weights and biases and the saved models' hyper-parameters) are saved:

Models	
Model_ID	**integer**
Dataset	text
File_Location	text
Hidden_Layers_Shape	text
Learning_Rate	float
Name	text
Train_Dataset_Size	integer
Use_ReLu	bool

- I will also use the following unique constraint, so that each dataset can not have more than one model with the same name:

```
UNIQUE (Dataset, Name)
```

2.9 Queries

Here are some example queries for interacting with the database:

- I can query the names of all saved models for a dataset with:

```
SELECT Name FROM Models WHERE Dataset=?;
```

- I can query the file location of a saved model with:

```
SELECT File_Location FROM Models WHERE Dataset=? AND Name=?;
```

- I can query the features of a saved model with:

```
SELECT * FROM Models WHERE Dataset=? AND Name=?;
```

2.10 Human-Computer Interaction TODO

- Labeled screenshots of UI

2.11 Hardware Design

To allow for faster training of an Artificial Neural Network, I will give the option to use a Graphics Card to train the Artificial Neural Network if available. I will also give the option to load pretrained weights to run on less computationaly powerfull hardware using just the CPU as standard.

2.12 Workflow and source control

I will use Git along with GitHub to manage my workflow and source control as I develop the project, by utilising the following features:

- Commits and branches for adding features and fixing bugs seperately.
- Using GitHub to back up the project as a repository.
- I will setup automated testing on GitHub after each pushed commit.
- I will also provide the necessary instructions and information for the installation and usage of this project, aswell as creating releases of the project with new patches.

3 Technical Solution TODO

3.1 Setup

3.1.1 File Structure

I used the following file structure to organise the code for the project, where school_project is the main package and is constructed of two main subpackages:

- The models package, which is a self-contained package for creating trained Artificial Neural Network models.
- The frames package, which consists of tkinter frames for the User Interface.

```
.
|-- .github
|   `-- workflows
|       `-- tests.yml
|-- .gitignore
|-- LICENSE
|-- README.md
|-- school_project
|   |-- frames
|   |   |-- create_model.py
|   |   |-- hyper-parameter-defaults.json
|   |   |-- __init__.py
|   |   |-- load_model.py
|   |   `-- test_model.py
|   |-- __init__.py
|   |-- __main__.py
|   |-- models
|   |   |-- cpu
|   |   |   |-- cat_recognition.py
|   |   |   |-- __init__.py
|   |   |   |-- mnist.py
|   |   |   |-- utils
|   |   |   |   |-- __init__.py
|   |   |   |   |-- model.py
|   |   |   |   `-- tools.py
|   |   |   `-- xor.py
|   |   |-- datasets
|   |   |   |-- mnist.pkl.gz
|   |   |   |-- test-cat.h5
|   |   |   `-- train-cat.h5
|   |   |-- gpu
|   |   |   |-- cat_recognition.py
|   |   |   |-- __init__.py
|   |   |   |-- mnist.py
|   |   |   |-- utils
|   |   |   |   |-- __init__.py
|   |   |   |   |-- model.py
|   |   |   |   `-- tools.py
|   |   |   `-- xor.py
|   |   `-- __init__.py
|   |-- saved-models
|   `-- test
|       |-- __init__.py
|       `-- models
|           |-- cpu
|           |   |-- __init__.py
|           |   `-- utils
|           |       |-- __init__.py
|           |       |-- test_model.py
|           |       `-- test_tools.py
|           |-- gpu
|           |   |-- __init__.py
|           |   `-- utils
|           |       |-- __init__.py
|           |       |-- test_model.py
|           |       `-- test_tools.py
|           `-- __init__.py
|-- setup.py
`-- TODO.md

17 directories, 41 files
```

Each package within the school_project package contains a __init__.py file, which allows the school_project package to be installed to a virtual environment so that the modules of the package can be imported from the installed package.

- Here is the contents of the frames package's __init__.py for example, which allows the classes of all modules in the package to be imported at once:

```python
"""Package of tkinter frames for the main window."""

from .create_model import HyperParameterFrame, TrainingFrame
from .load_model import LoadModelFrame
from .test_model import TestMNISTFrame, TestCatRecognitionFrame,
↪  TestXORFrame

__all__ = ['create_model', 'load_model', 'test_model']
```

I have omitted the source code for this report, which included a Makefile for its compilation.

3.1.2 Dependencies

The python dependencies for the project can be installed simply by running the following setup.py file (as described in the README.md in the next section). Instructions on installing external dependencies, such as the CUDA Toolkit for using a GPU, are explained in the README.md in the next section also.

- setup.py code:

```python
from setuptools import setup, find_packages

setup(
    name='school-project',
    version='1.0.0',
    packages=find_packages(),
    url='https://github.com/mcttn22/school-project.git',
    author='Max Cotton',
    author_email='maxcotton22@gmail.com',
    description='Year 13 Computer Science Programming Project',
    install_requires=[
                        'cupy-cuda12x',
                        'h5py',
                        'matplotlib',
                        'numpy',
                        'pympler'
    ],
)
```

3.1.3 Git and Github files

To optimise the use of Git and GitHub, I have used the following files:

- A .gitignore file for specifying which files and directories should be ignored by Git:

```
# Byte compiled files
__pycache__/

# Packaging
*.egg-info

# Database file
school_project/saved_models.db
```

- A README.md markdown file to give installation and usage instructions for the repository on GitHub:
  - Markdown code:

```
<!-- The following lines generate badges showing the current status of the automated testing (Passing or Failing) and a Python3 badge correspondingly.) -->
[![tests](https://github.com/mcttn22/school-project/actions/workflows/tests.yml/badge.svg)](https://
[![python](https://img.shields.io/badge/Python-3-3776AB.svg?style=flat&logo=python&logoColor=white)]

# A-level Computer Science NEA Programming Project

This project is an investigation into how Artificial Neural Networks (ANNs) work and their applications in Image Recognition, by documenting all theory behind the project and developing applications of the theory, that allow for experimentation via a GUI. The ANNs are created without the use of any 3rd party Machine Learning Libraries and I currently have been able to achieve a prediction accuracy of 99.6% on the MNIST dataset. The report for this project is also included in this repository.

## Installation

1. Download the Repository with:

  - ```
    git clone https://github.com/mcttn22/school-project.git
    ```
  - Or by downloading as a ZIP file

</br>

2. Create a virtual environment (venv) with:
  - Windows:
    ```
    python -m venv {venv name}
    ```
  - Linux:
    ```
    python3 -m venv {venv name}
    ```

3. Enter the venv with:
  - Windows:
    ```
    .\{venv name}\Scripts\activate
    ```
  - Linux:
    ```
    source ./{venv name}/bin/activate
    ```
```
```

````

4. Enter the project directory with:
   ```
   cd school-project/
   ```

5. For normal use, install the dependencies and the project to the venv with:
   - Windows:
     ```
     python setup.py install
     ```
   - Linux:
     ```
     python3 setup.py install
     ```

*Note: In order to use an Nvidia GPU for training the networks, the latest Nvdia drivers must be installed and the CUDA Toolkit must be installed from
<a href="https://developer.nvidia.com/cuda-downloads">here</a>.*

## Usage

Run with:
- Windows:
  ```
  python school_project
  ```
- Linux:
  ```
  python3 school_project
  ```

## Development

Install the dependencies and the project to the venv in developing mode with:
- Windows:
  ```
  python setup.py develop
  ```
- Linux:
  ```
  python3 setup.py develop
  ```

Run Tests with:
- Windows:
  ```
  python -m unittest discover .\school_project\test\
  ```
- Linux:
  ```
  python3 -m unittest discover ./school_project/test/
  ```

Compile Project Report PDF with:
```
make all
```
*Note: This requires the Latexmk library*
````

- Which will generate the following:

Tests passing Python 3

A-level Computer Science NEA Programming Project

This project is an investigation into how Artificial Neural Networks (ANNs) work and their applications in Image Recognition, by documenting all theory behind the project and developing applications of the theory, that allow for experimentation via a GUI. The ANNs are created without the use of any 3rd party Machine Learning Libraries and I currently have been able to achieve a prediction accuracy of 99.6% on the MNIST dataset. The report for this project is also included in this repository.

Installation

1. Download the Repository with:
   - ```
     git clone https://github.com/mcttn22/school-project.git
     ```
   - Or by downloading as a ZIP file

2. Create a virtual environment (venv) with:
   - Windows:
     ```
     python -m venv {venv name}
     ```
   - Linux:
     ```
     python3 -m venv {venv name}
     ```
     ```

3. Enter the venv with:

   - Windows:

     ```
     .\{venv name}\Scripts\activate
     ```

   - Linux:

     ```
     source ./{venv name}/bin/activate
     ```

4. Enter the project directory with:

   ```
   cd school-project/
   ```

5. For normal use, install the dependencies and the project to the venv with:

   - Windows:

     ```
     python setup.py install
     ```

   - Linux:

     ```
     python3 setup.py install
     ```

Note: In order to use an Nvidia GPU for training the networks, the latest Nvdia drivers must be installed and the CUDA Toolkit must be installed from here.

Usage

Run with:

- Windows:

  ```
  python school_project
  ```

- Linux:

```
python3 school_project
```

Development

Install the dependencies and the project to the venv in developing mode with:

- Windows:

```
python setup.py develop
```

- Linux:

```
python3 setup.py develop
```

Run Tests with:

- Windows:

```
python -m unittest discover .\school_project\test\
```

- Linux:

```
python3 -m unittest discover ./school_project/test/
```

Compile Project Report PDF with:

```
make all
```

Note: This requires the Latexmk library

- A LICENSE file that describes how others can use my code.

3.1.4 Organisation

I also utilise a TODO.md file for keeping track of what features and/or bugs need to be worked on.

3.2 models package

This package is a self-contained package for creating trained Artificial Neural Networks and can either be used for a CPU or a GPU, as well as containing the test and training data for all three datasets in a datasets directory. Whilst both the cpu and gpu subpackage are similar in functionality, the cpu subpackage uses NumPy for matrices whereas the gpu subpackage utilise NumPy and another library CuPy which requires a GPU to be utilised for operations with the matrices. For that reason it is only worth showing the code for the cpu subpackage.

Both the cpu and gpu subpackage contain a utils subpackage that provides the tools for creating Artificial Neural Networks, and three modules that are the implementation of Artificial Neural Networks for each dataset.

3.2.1 utils subpackage

The utils subpackage consists of a tools.py module that provides a ModelInterface class and helper functions for the model.py module, that contains an AbstractModel class that implements every method from the ModelInterface except for the load_dataset method.

- tools.py module:

```python
"""Helper functions and ModelInterface class for model module."""

from abc import ABC, abstractmethod

import numpy as np

class ModelInterface(ABC):
    """Interface for ANN models."""
    @abstractmethod
    def _setup_layers(setup_values: callable) -> None:
        """Decorator that sets up model layers and sets up values of each layer
           with the method given.

        Args:
            setup_values (callable): the method that sets up the values of each
            layer.
        Raises:
            NotImplementedError: if this method is not implemented.

        """
        raise NotImplementedError

    @abstractmethod
    def create_model_values(self) -> None:
        """Create weights and bias/biases

        Raises:
            NotImplementedError: if this method is not implemented.

        """
        raise NotImplementedError

    @abstractmethod
    def load_model_values(self, file_location: str) -> None:
        """Load weights and bias/biases from .npz file.

        Args:
            file_location (str): the location of the file to load from.
        Raises:
            NotImplementedError: if this method is not implemented.

        """
        raise NotImplementedError

    @abstractmethod
```

```
    def load_datasets(self, train_dataset_size: int) -> tuple[np.ndarray, np.ndarray,
                                                                np.ndarray, np.ndarray]:
        """Load input and output datasets. For the input dataset, each column
            should represent a piece of data and each row should store the values
            of the piece of data.

        Args:
            train_dataset_size (int): the number of train dataset inputs to use.
        Returns:
            tuple of train_inputs, train_outputs,
            test_inputs and test_outputs.
        Raises:
            NotImplementedError: if this method is not implemented.

        """
        raise NotImplementedError

    @abstractmethod
    def back_propagation(self, prediction: np.ndarray) -> None:
        """Adjust the weights and bias/biases via gradient descent.

        Args:
            prediction (numpy.ndarray): the matrice of prediction values
        Raises:
            NotImplementedError: if this method is not implemented.

        """
        raise NotImplementedError

    @abstractmethod
    def forward_propagation(self) -> np.ndarray:
        """Generate a prediction with the weights and bias/biases.

        Returns:
            numpy.ndarray of prediction values.
        Raises:
            NotImplementedError: if this method is not implemented.

        """
        raise NotImplementedError

    @abstractmethod
    def test(self) -> None:
        """Test trained weights and bias/biases.

        Raises:
            NotImplementedError: if this method is not implemented.

        """
        raise NotImplementedError

    @abstractmethod
    def train(self, epochs: int) -> None:
        """Train weights and bias/biases.

        Args:
            epochs (int): the number of forward and back propagations to do.
```

```
        Raises:
            NotImplementedError: if this method is not implemented.

        """
        raise NotImplementedError

    @abstractmethod
    def save_model_values(self, file_location: str) -> None:
        """Save the model by saving the weights then biases of each layer to
           a .npz file with a given file location.

           Args:
               file_location (str): the file location to save the model to.

        """
        raise NotImplementedError

def relu(z: np.ndarray | int | float) -> np.ndarray | float:
    """Transfer function, transform input to max number between 0 and z.

    Args:
        z (numpy.ndarray | int | float):
        the numpy.ndarray | int | float to be transferred.
    Returns:
        numpy.ndarray | float,
        with all values | the value transferred to max number between 0-z.
    Raises:
        TypeError: if z is not of type numpy.ndarray | int | float.

    """
    return np.maximum(0.1*z, 0)  # Divide by 10 to stop overflow errors

def relu_derivative(output: np.ndarray | int | float) -> np.ndarray | float:
    """Calculate derivative of ReLu Transfer function with respect to z.

    Args:
        output (numpy.ndarray | int | float):
        the numpy.ndarray | int | float output of the ReLu transfer function.
    Returns:
        numpy.ndarray | float,
        derivative of the ReLu transfer function with respect to z.
    Raises:
        TypeError: if output is not of type numpy.ndarray | int | float.

    """
    output[output <= 0] = 0
    output[output > 0] = 1

    return output

def sigmoid(z: np.ndarray | int | float) -> np.ndarray | float:
    """Transfer function, transform input to number between 0 and 1.

    Args:
        z (numpy.ndarray | int | float):
        the numpy.ndarray | int | float to be transferred.
    Returns:
        numpy.ndarray | float,
        with all values | the value transferred to a number between 0-1.
```

```
    Raises:
        TypeError: if z is not of type numpy.ndarray | int | float.

    """
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(output: np.ndarray | int | float) -> np.ndarray | float:
    """Calculate derivative of sigmoid Transfer function with respect to z.

    Args:
        output (numpy.ndarray | int | float):
        the numpy.ndarray | int | float output of the sigmoid transfer function.
    Returns:
        numpy.ndarray | float,
        derivative of the sigmoid transfer function with respect to z.
    Raises:
        TypeError: if output is not of type numpy.ndarray | int | float.

    """
    return output * (1 - output)

def calculate_loss(input_count: int,
                   outputs: np.ndarray,
                   prediction: np.ndarray) -> float:
    """Calculate average loss/error of the prediction to the outputs.

    Args:
        input_count (int): the number of inputs.
        outputs (np.ndarray):
        the train/test outputs array to compare with the prediction.
        prediction (np.ndarray): the array of prediction values.
    Returns:
        float loss.
    Raises:
        ValueError:
        if outputs is not a suitable multiplier with the prediction
        (incorrect shapes)

    """
    return np.squeeze(- (1/input_count) * np.sum(outputs * np.log(prediction) + (1 - outputs) * np.log(1 - prediction)))

def calculate_prediction_accuracy(prediction: np.ndarray,
                                  outputs: np.ndarray) -> float:
    """Calculate the percentage accuracy of the predictions.

    Args:
        prediction (np.ndarray): the array of prediction values.
        outputs (np.ndarray):
        the train/test outputs array to compare with the prediction.
    Returns:
        float prediction accuracy

    """
    return 100 - np.mean(np.abs(prediction - outputs)) * 100
```

- model.py module:

```
"""Provides an abstract class for Artificial Neural Network models."""
```

```python

import time

import numpy as np

from .tools import (
                    ModelInterface,
                    relu,
                    relu_derivative,
                    sigmoid,
                    sigmoid_derivative,
                    calculate_loss,
                    calculate_prediction_accuracy
                    )

class _Layers():
    """Manages linked list of layers."""
    def __init__(self) -> None:
        """Initialise linked list."""
        self.head = None
        self.tail = None

    def __iter__(self) -> None:
        """Iterate forward through the network."""
        current_layer = self.head
        while True:
            yield current_layer
            if current_layer.next_layer != None:
                current_layer = current_layer.next_layer
            else:
                break

    def __reversed__(self) -> None:
        """Iterate back through the network."""
        current_layer = self.tail
        while True:
            yield current_layer
            if current_layer.previous_layer != None:
                current_layer = current_layer.previous_layer
            else:
                break

class _FullyConnectedLayer():
    """Fully connected layer for Deep ANNs,
       represented as a node of a Doubly linked list."""
    def __init__(self, learning_rate: float, input_neuron_count: int,
                 output_neuron_count: int, transfer_type: str) -> None:
        """Initialise layer values.

        Args:
            learning_rate (float): the learning rate of the model.
            input_neuron_count (int):
            the number of input neurons into the layer.
            output_neuron_count (int):
            the number of output neurons into the layer.
            transfer_type (str): the transfer function type
            ('sigmoid' or 'relu')

        """
        # Setup layer attributes
        self.previous_layer = None
        self.next_layer = None
```

```python
        self.input_neuron_count = input_neuron_count
        self.output_neuron_count = output_neuron_count
        self.transfer_type = transfer_type
        self.input: np.ndarray
        self.output: np.ndarray

        # Setup weights and biases
        self.weights: np.ndarray
        self.biases: np.ndarray
        self.learning_rate = learning_rate

    def __repr__(self) -> str:
        """Read values of the layer.

        Returns:
            a string description of the layers's
            weights, bias and learning rate values.

        """
        return (f"Weights: {self.weights.tolist()}\n" +
                f"Biases: {self.biases.tolist()}\n")

    def init_layer_values_random(self) -> None:
        """Initialise weights to random values and biases to 0s"""
        np.random.seed(1)  # Sets up pseudo random values for layer weight arrays
        self.weights = np.random.rand(self.output_neuron_count, self.input_neuron_count) - 0.5
        self.biases = np.zeros(shape=(self.output_neuron_count, 1))

    def init_layer_values_zeros(self) -> None:
        """Initialise weights to 0s and biases to 0s"""
        self.weights = np.zeros(shape=(self.output_neuron_count, self.input_neuron_count))
        self.biases = np.zeros(shape=(self.output_neuron_count, 1))

    def back_propagation(self, dloss_doutput) -> np.ndarray:
        """Adjust the weights and biases via gradient descent.

        Args:
            dloss_doutput (numpy.ndarray): the derivative of the loss of the
            layer's output, with respect to the layer's output.
        Returns:
            a numpy.ndarray derivative of the loss of the layer's input,
            with respect to the layer's input.
        Raises:
            ValueError:
            if dloss_doutput
            is not a suitable multiplier with the weights
            (incorrect shape)

        """
        match self.transfer_type:
            case 'sigmoid':
                dloss_dz = dloss_doutput * sigmoid_derivative(output=self.output)
            case 'relu':
                dloss_dz = dloss_doutput * relu_derivative(output=self.output)

        dloss_dweights = np.dot(dloss_dz, self.input.T)
```

```
        dloss_dbiases = np.sum(dloss_dz)

        assert dloss_dweights.shape == self.weights.shape

        dloss_dinput = np.dot(self.weights.T, dloss_dz)

        # Update weights and biases
        self.weights -= self.learning_rate * dloss_dweights
        self.biases -= self.learning_rate * dloss_dbiases

        return dloss_dinput

    def forward_propagation(self, inputs) -> np.ndarray:
        """Generate a layer output with the weights and biases.

        Args:
            inputs (np.ndarray): the input values to the layer.
        Returns:
            a numpy.ndarray of the output values.

        """
        self.input = inputs
        z = np.dot(self.weights, self.input) + self.biases
        if self.transfer_type == 'sigmoid':
            self.output = sigmoid(z)
        elif self.transfer_type == 'relu':
            self.output = relu(z)
        return self.output

class AbstractModel(ModelInterface):
    """ANN model with variable number of hidden layers"""
    def __init__(self,
                 hidden_layers_shape: list[int],
                 train_dataset_size: int,
                 learning_rate: float,
                 use_relu: bool) -> None:
        """Initialise model values.

        Args:
            hidden_layers_shape (list[int]):
            list of the number of neurons in each hidden layer.
            train_dataset_size (int): the number of train dataset inputs to use.
            learning_rate (float): the learning rate of the model.
            use_relu (bool): True or False whether the ReLu Transfer function
            should be used.

        """
        # Setup model data
        self.train_inputs, self.train_outputs,\
        self.test_inputs, self.test_outputs = self.load_datasets(
                                             train_dataset_size=train_dataset_size
                                         )
        self.train_losses: list[float]
        self.test_prediction: np.ndarray
        self.test_prediction_accuracy: float
        self.training_progress = ""
        self.training_time: float

        # Setup model attributes
```

```python
        self.__running = True
        self.input_neuron_count: int = self.train_inputs.shape[0]
        self.input_count = self.train_inputs.shape[1]
        self.hidden_layers_shape = hidden_layers_shape
        self.output_neuron_count = self.train_outputs.shape[0]
        self.layers_shape = [f'{layer}' for layer in (
                            [self.input_neuron_count] +
                            self.hidden_layers_shape +
                            [self.output_neuron_count]
                            )]
        self.use_relu = use_relu

        # Setup model values
        self.layers = _Layers()
        self.learning_rate = learning_rate

    def __repr__(self) -> str:
        """Read current state of model.

        Returns:
            a string description of the model's shape,
            weights, bias and learning rate values.

        """
        return (f"Layers Shape: {','.join(self.layers_shape)}\n" +
                f"Learning Rate: {self.learning_rate}")

    def set_running(self, value: bool) -> None:
        """Set the running attribute to the given value.

        Args:
            value (bool): the value to set the running attribute to.

        """
        self.__running = value

    def _setup_layers(setup_values: callable) -> None:
        """Decorator that sets up model layers and sets up values of each
    ↪   layer
           with the method given.

        Args:
            setup_values (callable): the method that sets up the values of
    ↪   each
            layer.

        """
        def decorator(self, *args, **kwargs) -> None:
            # Check if setting up Deep Network
            if len(self.hidden_layers_shape) > 0:
                if self.use_relu:

                    # Add input layer
                    self.layers.head = _FullyConnectedLayer(

                                            ↪   learning_rate=self.learning_rate,

                                            ↪   input_neuron_count=self.input_neuron_count,

                                            ↪   output_neuron_count=self.hidden_layers_shape[0],
                                            transfer_type='relu'
                                            )
```

```
                current_layer = self.layers.head

                # Add hidden layers
                for layer in range(len(self.hidden_layers_shape) - 1):
                    current_layer.next_layer = _FullyConnectedLayer(
                                learning_rate=self.learning_rate,

                                ↪  input_neuron_count=self.hidden_layers_shape[layer],

                                ↪  output_neuron_count=self.hidden_layers_shape[layer
                                ↪  + 1],
                                transfer_type='relu'
                                )
                    current_layer.next_layer.previous_layer =
                    ↪  current_layer
                    current_layer = current_layer.next_layer
            else:

                # Add input layer
                self.layers.head = _FullyConnectedLayer(

                                        ↪  learning_rate=self.learning_rate,

                                        ↪  input_neuron_count=self.input_neuron_count,

                                        ↪  output_neuron_count=self.hidden_layers_shape[0],
                                        transfer_type='sigmoid'
                                        )
                current_layer = self.layers.head

                # Add hidden layers
                for layer in range(len(self.hidden_layers_shape) - 1):
                    current_layer.next_layer = _FullyConnectedLayer(
                                learning_rate=self.learning_rate,

                                ↪  input_neuron_count=self.hidden_layers_shape[layer],

                                ↪  output_neuron_count=self.hidden_layers_shape[layer
                                ↪  + 1],
                                transfer_type='sigmoid'
                                )
                    current_layer.next_layer.previous_layer =
                    ↪  current_layer
                    current_layer = current_layer.next_layer

            # Add output layer
            current_layer.next_layer = _FullyConnectedLayer(
                                    learning_rate=self.learning_rate,

                                    ↪  input_neuron_count=self.hidden_layers_shape[-1],

                                    ↪  output_neuron_count=self.output_neuron_count,
                                    transfer_type='sigmoid'
                                    )
            current_layer.next_layer.previous_layer = current_layer
            self.layers.tail = current_layer.next_layer

        # Setup Perceptron Network
        else:
            self.layers.head = _FullyConnectedLayer(
                                    learning_rate=self.learning_rate,

                                    ↪  input_neuron_count=self.input_neuron_count,
```

```python
                                        output_neuron_count=self.output_neuron_count,
                                        transfer_type='sigmoid'
                                        )
                self.layers.tail = self.layers.head

            setup_values(self, *args, **kwargs)

        return decorator

    @_setup_layers
    def create_model_values(self) -> None:
        """Create weights and bias/biases"""
        # Check if setting up Deep Network
        if len(self.hidden_layers_shape) > 0:

            # Initialise Layer values to random values
            for layer in self.layers:
                layer.init_layer_values_random()

        # Setup Perceptron Network
        else:

            # Initialise Layer values to zeros
            for layer in self.layers:
                layer.init_layer_values_zeros()

    @_setup_layers
    def load_model_values(self, file_location: str) -> None:
        """Load weights and bias/biases from .npz file.

        Args:
            file_location (str): the location of the file to load from.

        """
        data: dict[str, np.ndarray] = np.load(file=file_location)

        # Initialise Layer values
        i = 0
        keys = list(data.keys())
        for layer in self.layers:
            layer.weights = data[keys[i]]
            layer.biases = data[keys[i + 1]]
            i += 2

    def back_propagation(self, dloss_doutput) -> None:
        """Train each layer's weights and biases.

        Args:
            dloss_doutput (np.ndarray): the derivative of the loss of the
            output layer's output, with respect to the output layer's output.

        """
        for layer in reversed(self.layers):
            dloss_doutput = layer.back_propagation(dloss_doutput=dloss_doutput)

    def forward_propagation(self) -> np.ndarray:
        """Generate a prediction with the layers.

        Returns:
```

```python
            a numpy.ndarray of the prediction values.

        """
        output = self.train_inputs
        for layer in self.layers:
            output = layer.forward_propagation(inputs=output)
        return output

    def test(self) -> None:
        """Test the layers' trained weights and biases."""
        output = self.test_inputs
        for layer in self.layers:
            output = layer.forward_propagation(inputs=output)
        self.test_prediction = output

        # Calculate performance of model
        self.test_prediction_accuracy = calculate_prediction_accuracy(
                                                    prediction=self.test_prediction,
                                                    outputs=self.test_outputs
                                                    )

    def train(self, epoch_count: int) -> None:
        """Train layers' weights and biases.

           Args:
               epoch_count (int): the number of training epochs.

        """
        self.layers_shape = [f'{layer}' for layer in (
                             [self.input_neuron_count] +
                             self.hidden_layers_shape +
                             [self.output_neuron_count]
                             )]
        self.train_losses = []
        training_start_time = time.time()
        for epoch in range(epoch_count):
            if not self.__running:
                break
            self.training_progress = f"Epoch {epoch} / {epoch_count}"
            prediction = self.forward_propagation()
            loss = calculate_loss(input_count=self.input_count,
                                  outputs=self.train_outputs,
                                  prediction=prediction)
            self.train_losses.append(loss)
            if not self.__running:
                break
            dloss_doutput = -(1/self.input_count) * ((self.train_outputs - prediction)/(prediction * (1 - prediction)))
            self.back_propagation(dloss_doutput=dloss_doutput)
        self.training_time = round(number=time.time() - training_start_time,
                                   ndigits=2)

    def save_model_values(self, file_location: str) -> None:
        """Save the model by saving the weights then biases of each layer to
           a .npz file with a given file location.

           Args:
               file_location (str): the file location to save the model to.

```

```python
        """
        saved_model: list[np.ndarray] = []
        for layer in self.layers:
            saved_model.append(layer.weights)
            saved_model.append(layer.biases)
        np.savez(file_location, *saved_model)
```

3.2.2 Artificial Neural Network implementations

The following three modules implement the AbstractModel class from the above model.py module from the utils subpackage, on the three datasets.

- cat_recognition.py module:

```python
"""Implementation of Artificial Neural Network model on Cat Recognition dataset."""

import h5py
import numpy as np

from .utils.model import AbstractModel

class CatRecognitionModel(AbstractModel):
    """ANN model that trains to predict if an image is a cat or not a cat."""
    def __init__(self,
                 hidden_layers_shape: list[int],
                 train_dataset_size: int,
                 learning_rate: float,
                 use_relu: bool) -> None:
        """Initialise Model's Base class.

        Args:
            hidden_layers_shape (list[int]):
            list of the number of neurons in each hidden layer.
            train_dataset_size (int): the number of train dataset inputs to use.
            learning_rate (float): the learning rate of the model.
            use_relu (bool): True or False whether the ReLu Transfer function
            should be used.

        """
        super().__init__(hidden_layers_shape=hidden_layers_shape,
                         train_dataset_size=train_dataset_size,
                         learning_rate=learning_rate,
                         use_relu=use_relu)

    def load_datasets(self, train_dataset_size: int) -> tuple[np.ndarray, np.ndarray,
                                                              np.ndarray, np.ndarray]:
        """Load image input and output datasets.

        Args:
            train_dataset_size (int): the number of train dataset inputs to use.
        Returns:
            tuple of image train_inputs, train_outputs,
            test_inputs and test_outputs numpy.ndarrys.
```

```python

        Raises:
            FileNotFoundError: if file does not exist.

        """
        # Load datasets from h5 files
        # (h5 files stores large amount of data with quick access)
        train_dataset: h5py.File = h5py.File(
             r'school_project/models/datasets/train-cat.h5',
             'r'
             )
        test_dataset: h5py.File = h5py.File(
              r'school_project/models/datasets/test-cat.h5',
              'r'
              )

        # Load input arrays,
        # containing the RGB values for each pixel in each 64x64 pixel
        ↪  image,
        # for 209 images
        train_inputs: np.ndarray =
        ↪  np.array(train_dataset['train_set_x'][:])
        test_inputs: np.ndarray = np.array(test_dataset['test_set_x'][:])

        # Load output arrays of 1s for cat and 0s for not cat
        train_outputs: np.ndarray =
        ↪  np.array(train_dataset['train_set_y'][:])
        test_outputs: np.ndarray = np.array(test_dataset['test_set_y'][:])

        # Reshape input arrays into 1 dimension (flatten),
        # then divide by 255 (RGB)
        # to standardize them to a number between 0 and 1
        train_inputs = train_inputs.reshape((train_inputs.shape[0],
                                             -1)).T / 255
        test_inputs = test_inputs.reshape((test_inputs.shape[0], -1)).T /
        ↪  255

        # Reshape output arrays into a 1 dimensional list of outputs
        train_outputs = train_outputs.reshape((1, train_outputs.shape[0]))
        test_outputs = test_outputs.reshape((1, test_outputs.shape[0]))

        # Reduce train datasets' sizes to train_dataset_size
        train_inputs = (train_inputs.T[:train_dataset_size]).T
        train_outputs = (train_outputs.T[:train_dataset_size]).T

        return train_inputs, train_outputs, test_inputs, test_outputs
```

- mnist.py module:

```python
"""Implementation of Artificial Neural Network model on MNIST dataset."""

import pickle
import gzip

import numpy as np

from .utils.model import AbstractModel

class MNISTModel(AbstractModel):
    """ANN model that trains to predict Numbers from images."""
    def __init__(self, hidden_layers_shape: list[int],
```

```python
                 train_dataset_size: int,
                 learning_rate: float,
                 use_relu: bool) -> None:
        """Initialise Model's Base class.

        Args:
            hidden_layers_shape (list[int]):
            list of the number of neurons in each hidden layer.
            train_dataset_size (int): the number of train dataset inputs to use.
            learning_rate (float): the learning rate of the model.
            use_relu (bool): True or False whether the ReLu Transfer function
            should be used.

        """
        super().__init__(hidden_layers_shape=hidden_layers_shape,
                         train_dataset_size=train_dataset_size,
                         learning_rate=learning_rate,
                         use_relu=use_relu)

    def load_datasets(self, train_dataset_size: int) -> tuple[np.ndarray, np.ndarray,
                                                               np.ndarray, np.ndarray]:
        """Load image input and output datasets.
        Args:
            train_dataset_size (int): the number of dataset inputs to use.
        Returns:
            tuple of image train_inputs, train_outputs,
            test_inputs and test_outputs numpy.ndarrys.

        Raises:
            FileNotFoundError: if file does not exist.

        """
        # Load datasets from pkl.gz file
        with gzip.open(
              'school_project/models/datasets/mnist.pkl.gz',
              'rb'
              ) as mnist:
            (train_inputs, train_outputs),\
            (test_inputs, test_outputs) = pickle.load(mnist, encoding='bytes')

        # Reshape input arrays into 1 dimension (flatten),
        # then divide by 255 (RGB)
        # to standardize them to a number between 0 and 1
        train_inputs = np.array(train_inputs.reshape((train_inputs.shape[0],
                                          -1)).T / 255)
        test_inputs = np.array(test_inputs.reshape(test_inputs.shape[0], -1).T / 255)

        # Represent number values
        # with a one at the matching index of an array of zeros
        train_outputs = np.eye(np.max(train_outputs) + 1)[train_outputs].T
        test_outputs = np.eye(np.max(test_outputs) + 1)[test_outputs].T

        # Reduce train datasets' sizes to train_dataset_size
        train_inputs = (train_inputs.T[:train_dataset_size]).T
        train_outputs = (train_outputs.T[:train_dataset_size]).T
```

```python

        return train_inputs, train_outputs, test_inputs, test_outputs
```

- xor.py module

```python
"""Implementation of Artificial Neural Network model on XOR dataset."""

import numpy as np

from .utils.model import AbstractModel

class XORModel(AbstractModel):
    """ANN model that trains to predict the output of a XOR gate with two
       inputs."""
    def __init__(self,
                 hidden_layers_shape: list[int],
                 train_dataset_size: int,
                 learning_rate: float,
                 use_relu: bool) -> None:
        """Initialise Model's Base class.

        Args:
            hidden_layers_shape (list[int]):
            list of the number of neurons in each hidden layer.
            train_dataset_size (int): the number of train dataset inputs to use.
            learning_rate (float): the learning rate of the model.
            use_relu (bool): True or False whether the ReLu Transfer function
            should be used.

        """
        super().__init__(hidden_layers_shape=hidden_layers_shape,
                         train_dataset_size=train_dataset_size,
                         learning_rate=learning_rate,
                         use_relu=use_relu)

    def load_datasets(self, train_dataset_size: int) -> tuple[np.ndarray, np.ndarray,
                                                                          np.ndarray, np.ndarray]:
        """Load XOR input and output datasets.

        Args:
            train_dataset_size (int): the number of dataset inputs to use.
        Returns:
            tuple of XOR train_inputs, train_outputs,
            test_inputs and test_outputs numpy.ndarrys.

        """
        inputs: np.ndarray = np.array([[0, 0, 1, 1],
                                       [0, 1, 0, 1]])
        outputs: np.ndarray = np.array([[0, 1, 1, 0]])

        # Reduce train datasets' sizes to train_dataset_size
        inputs = (inputs.T[:train_dataset_size]).T
        outputs = (outputs.T[:train_dataset_size]).T

        return inputs, outputs, inputs, outputs
```

3.3 frames package

I decided to use tkinter for the User Interface and the frames package consists of tkinter frames to be loaded onto the main window when needed. The package also includes a hyper-parameter-defaults.json file, which stores optimum default values for the hyper-parameters to be set to.

- hyper-parameter-defaults.json file contents:

```json
{
    "MNIST": {
        "description": "An Image model trained on recognising numbers from images.",
        "epochCount": 150,
        "hiddenLayersShape": [1000, 1000],
        "minTrainDatasetSize": 1,
        "maxTrainDatasetSize": 60000,
        "maxLearningRate": 1
    },
    "Cat Recognition": {
        "description": "An Image model trained on recognising if an image is a cat or not.",
        "epochCount": 3500,
        "hiddenLayersShape": [100, 100],
        "minTrainDatasetSize": 1,
        "maxTrainDatasetSize": 209,
        "maxLearningRate": 0.3
    },
    "XOR": {
        "description": "For experimenting with Artificial Neural Networks, a XOR gate model has been used for its lesser computation time.",
        "epochCount": 4700,
        "hiddenLayersShape": [100, 100],
        "minTrainDatasetSize":  2,
        "maxTrainDatasetSize": 4,
        "maxLearningRate": 1
    }
}
```

- create_model.py module:

```python
"""Tkinter frames for creating an Artificial Neural Network model."""

import json
import threading
import tkinter as tk
import tkinter.font as tkf

from matplotlib.figure import Figure
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import numpy as np

class HyperParameterFrame(tk.Frame):
    """Frame for hyper-parameter page."""
    def __init__(self, root: tk.Tk, width: int,
                 height: int, bg: str, dataset: str) -> None:
        """Initialise hyper-parameter frame widgets.

        Args:
            root (tk.Tk): the widget object that contains this widget.
```

```python
            width (int): the pixel width of the frame.
            height (int): the pixel height of the frame.
            bg (str): the hex value or name of the frame's background colour.
            dataset (str): the name of the dataset to use
            ('MNIST', 'Cat Recognition' or 'XOR')
        Raises:
            TypeError: if root, width or height are not of the correct type.

        """
        super().__init__(master=root, width=width, height=height, bg=bg)
        self.root = root
        self.WIDTH = width
        self.HEIGHT = height
        self.BG = bg

        # Setup hyper-parameter frame variables
        self.dataset = dataset
        self.use_gpu: bool
        self.default_hyper_parameters = self.load_default_hyper_parameters(
                                                                          dataset=dataset
                                                                          )

        # Setup widgets
        self.title_label = tk.Label(master=self,
                                    bg=self.BG,
                                    font=('Arial', 20),
                                    text=dataset)
        self.about_label = tk.Label(
                               master=self,
                               bg=self.BG,
                               font=('Arial', 14),
                               text=self.default_hyper_parameters['description']
                               )
        self.learning_rate_scale = tk.Scale(
                            master=self,
                            bg=self.BG,
                            orient='horizontal',
                            label="Learning Rate",
                            length=185,
                            from_=0,
                            to=self.default_hyper_parameters['maxLearningRate'],
                            resolution=0.01
                            )
        self.learning_rate_scale.set(value=0.1)
        self.epoch_count_scale = tk.Scale(master=self,
                                          bg=self.BG,
                                          orient='horizontal',
                                          label="Epoch Count",
                                          length=185,
                                          from_=0,
                                          to=10_000,
                                          resolution=100)
        self.epoch_count_scale.set(
                               value=self.default_hyper_parameters['epochCount']
                               )
        self.train_dataset_size_scale = tk.Scale(
```

```
                master=self,
                bg=self.BG,
                orient='horizontal',
                label="Train Dataset Size",
                length=185,
                from_=self.default_hyper_parameters['minTrainDatasetSize'],
                to=self.default_hyper_parameters['maxTrainDatasetSize'],
                resolution=1
                )
        self.train_dataset_size_scale.set(
                value=self.default_hyper_parameters['maxTrainDatasetSize']
                )
        self.hidden_layers_shape_label = tk.Label(
                            master=self,
                            bg=self.BG,
                            font=('Arial', 12),
                            text="Enter the number of neurons in each\n" +
                                "hidden layer, separated by commas:"
                            )
        self.hidden_layers_shape_entry = tk.Entry(master=self)
        self.hidden_layers_shape_entry.insert(0, ",".join(
            f"{neuron_count}" for neuron_count in self.default_hyper_parameters['hiddenLayersShape']
            ))
        self.use_relu_check_button_var = tk.BooleanVar(value=True)
        self.use_relu_check_button = tk.Checkbutton(
                                    master=self,
                                    width=13, height=1,
                                    font=tkf.Font(size=12),
                                    text="Use ReLu",
                                    variable=self.use_relu_check_button_var
                                    )
        self.use_gpu_check_button_var = tk.BooleanVar()
        self.use_gpu_check_button = tk.Checkbutton(
                                    master=self,
                                    width=13, height=1,
                                    font=tkf.Font(size=12),
                                    text="Use GPU",
                                    variable=self.use_gpu_check_button_var
                                    )
        self.model_status_label = tk.Label(master=self,
                                        bg=self.BG,
                                        font=('Arial', 15))

        # Pack widgets
        self.title_label.grid(row=0, column=0, columnspan=3)
        self.about_label.grid(row=1, column=0, columnspan=3)
        self.learning_rate_scale.grid(row=2, column=0, pady=(50,0))
        self.epoch_count_scale.grid(row=3, column=0, pady=(30,0))
        self.train_dataset_size_scale.grid(row=4, column=0, pady=(30,0))
        self.hidden_layers_shape_label.grid(row=2, column=1,
                                            padx=30, pady=(50,0))
        self.hidden_layers_shape_entry.grid(row=3, column=1, padx=30)
        self.use_relu_check_button.grid(row=2, column=2, pady=(30, 0))
        self.use_gpu_check_button.grid(row=3, column=2, pady=(30, 0))
        self.model_status_label.grid(row=5, column=0,
```

```python
                                     columnspan=3, pady=50)

    def load_default_hyper_parameters(self, dataset: str) -> dict[
                                                 str,
                                                 str | int | list[int] | float
                                                 ]:
        """Load the dataset's default hyper-parameters from the json file.

           Args:
               dataset (str): the name of the dataset to load hyper-parameters
               for. ('MNIST', 'Cat Recognition' or 'XOR')
            Returns:
                a dictionary of default hyper-parameter values.
        """
        with open('school_project/frames/hyper-parameter-defaults.json') as f:
            return json.load(f)[dataset]

    def create_model(self) -> object:
        """Create and return a Model using the hyper-parameters set.

           Returns:
               a Model object.
        """
        self.use_gpu = self.use_gpu_check_button_var.get()

        # Validate hidden layers shape input
        hidden_layers_shape_input = [layer for layer in self.hidden_layers_shape_entry.get().replace(' ', '').split(',') if layer != '']
        for layer in hidden_layers_shape_input:
            if not layer.isdigit():
                self.model_status_label.configure(
                                        text="Invalid hidden layers shape",
                                        fg='red'
                                        )
                raise ValueError

        # Create Model
        if not self.use_gpu:
            if self.dataset == "MNIST":
                from school_project.models.cpu.mnist import MNISTModel as Model
            elif self.dataset == "Cat Recognition":
                from school_project.models.cpu.cat_recognition import CatRecognitionModel as Model
            elif self.dataset == "XOR":
                from school_project.models.cpu.xor import XORModel as Model
            model = Model(hidden_layers_shape = [int(neuron_count) for neuron_count in hidden_layers_shape_input],
                          train_dataset_size = self.train_dataset_size_scale.get(),
                          learning_rate = self.learning_rate_scale.get(),
                          use_relu = self.use_relu_check_button_var.get())
            model.create_model_values()

        else:
            try:
                if self.dataset == "MNIST":
                    from school_project.models.gpu.mnist import MNISTModel as Model
```

```python
                elif self.dataset == "Cat Recognition":
                    from school_project.models.gpu.cat_recognition import CatRecognitionModel as Model
                elif self.dataset == "XOR":
                    from school_project.models.gpu.xor import XORModel as Model
                model = Model(hidden_layers_shape = [int(neuron_count) for neuron_count in hidden_layers_shape_input],
                              train_dataset_size = self.train_dataset_size_scale.get(),
                              learning_rate = self.learning_rate_scale.get(),
                              use_relu = self.use_relu_check_button_var.get())
                model.create_model_values()
            except ImportError as ie:
                self.model_status_label.configure(
                                        text="Failed to initialise GPU",
                                        fg='red'
                                        )
                raise ImportError
        return model

class TrainingFrame(tk.Frame):
    """Frame for training page."""
    def __init__(self, root: tk.Tk, width: int,
                 height: int, bg: str,
                 model: object, epoch_count: int) -> None:
        """Initialise training frame widgets.

        Args:
            root (tk.Tk): the widget object that contains this widget.
            width (int): the pixel width of the frame.
            height (int): the pixel height of the frame.
            bg (str): the hex value or name of the frame's background colour.
            model (object): the Model object to be trained.
            epoch_count (int): the number of training epochs.
        Raises:
            TypeError: if root, width or height are not of the correct type.

        """
        super().__init__(master=root, width=width, height=height, bg=bg)
        self.root = root
        self.WIDTH = width
        self.HEIGHT = height
        self.BG = bg

        # Setup widgets
        self.model_status_label = tk.Label(master=self,
                                           bg=self.BG,
                                           font=('Arial', 15))
        self.training_progress_label = tk.Label(master=self,
                                                bg=self.BG,
                                                font=('Arial', 15))
        self.loss_figure: Figure = Figure()
        self.loss_canvas: FigureCanvasTkAgg = FigureCanvasTkAgg(
                                                      figure=self.loss_figure,
                                                      master=self
                                                      )
```

```python

        # Pack widgets
        self.model_status_label.pack(pady=(30,0))
        self.training_progress_label.pack(pady=30)

        # Start training thread
        self.model_status_label.configure(
                                        text="Training weights and
                                        ↪  biases...",
                                        fg='red'
                                        )
        self.train_thread: threading.Thread = threading.Thread(

                                                            ↪   target=model.train,

                                                            ↪   args=(epoch_count,)
                                                            )
        self.train_thread.start()

    def plot_losses(self, model: object) -> None:
        """Plot losses of Model training.

           Args:
               model (object): the Model object thats been trained.

        """
        self.model_status_label.configure(
                 text=f"Weights and biases trained in
                 ↪  {model.training_time}s",
                 fg='green'
                 )
        graph: Figure.axes = self.loss_figure.add_subplot(111)
        graph.set_title("Learning rate: " +
                        f"{model.learning_rate}")
        graph.set_xlabel("Epochs")
        graph.set_ylabel("Loss Value")
        graph.plot(np.squeeze(model.train_losses))
        self.loss_canvas.get_tk_widget().pack()
```

- load_model.py module:

```python
"""Tkinter frames for loading a saved Artificial Neural Network Model."""

import sqlite3
import tkinter as tk
import tkinter.font as tkf

class LoadModelFrame(tk.Frame):
    """Frame for load model page."""
    def __init__(self, root: tk.Tk,
                 width: int, height: int,
                 bg: str, connection: sqlite3.Connection,
                 cursor: sqlite3.Cursor, dataset: str) -> None:
        """Initialise load model frame widgets.

        Args:
            root (tk.Tk): the widget object that contains this widget.
            width (int): the pixel width of the frame.
            height (int): the pixel height of the frame.
            bg (str): the hex value or name of the frame's background
↪   colour.
```

```
            connection (sqlite3.Connection): the database connection object.
            cursor (sqlite3.Cursor): the database cursor object.
            dataset (str): the name of the dataset to use
            ('MNIST', 'Cat Recognition' or 'XOR')
        Raises:
            TypeError: if root, width or height are not of the correct type.

        """
        super().__init__(master=root, width=width, height=height, bg=bg)
        self.root = root
        self.WIDTH = width
        self.HEIGHT = height
        self.BG = bg

        # Setup load model frame variables
        self.connection = connection
        self.cursor = cursor
        self.dataset = dataset
        self.use_gpu: bool
        self.model_options = self.load_model_options()

        # Setup widgets
        self.title_label = tk.Label(master=self,
                                    bg=self.BG,
                                    font=('Arial', 20),
                                    text=dataset)
        self.about_label = tk.Label(
                    master=self,
                    bg=self.BG,
                    font=('Arial', 14),
                    text=f"Load a pretrained model for the {dataset} dataset."
                    )
        self.model_status_label = tk.Label(master=self,
                                           bg=self.BG,
                                           font=('Arial', 15))

        # Don't give loaded model options if no models have been saved for the
        # dataset.
        if len(self.model_options) > 0:
            self.model_option_menu_label = tk.Label(
                                                master=self,
                                                bg=self.BG,
                                                font=('Arial', 14),
                                                text="Select a model to load or delete:"
                                                )
            self.model_option_menu_var = tk.StringVar(
                                                   master=self,
                                                   value=self.model_options[0]
                                                   )
            self.model_option_menu = tk.OptionMenu(
                                                    self,
                                                    self.model_option_menu_var,
                                                    *self.model_options
                                                    )
            self.use_gpu_check_button_var = tk.BooleanVar()
```

```
            self.use_gpu_check_button = tk.Checkbutton(
                                        master=self,
                                        width=7, height=1,
                                        font=tkf.Font(size=12),
                                        text="Use GPU",
                                        variable=self.use_gpu_check_button_var
                                        )
        else:
            self.model_status_label.configure(
                                     text='No saved models for this dataset.',
                                     fg='red'
                                     )

        # Pack widgets
        self.title_label.grid(row=0, column=0, columnspan=3)
        self.about_label.grid(row=1, column=0, columnspan=3)
        if len(self.model_options) > 0:  # Check if options should be given
            self.model_option_menu_label.grid(row=2, column=0, padx=(0,30), pady=(30,0))
            self.use_gpu_check_button.grid(row=2, column=2, rowspan=2, pady=(30,0))
            self.model_option_menu.grid(row=3, column=0, padx=(0,30), pady=(10,0))
        self.model_status_label.grid(row=4, column=0,
                                     columnspan=3, pady=50)

    def load_model_options(self) -> list[str]:
        """Load the model options from the database.

           Returns:
                a list of the model options.
        """
        sql = f"""
        SELECT Name FROM Models WHERE Dataset=?
        """
        parameters = (self.dataset.replace(" ", "_"),)
        self.cursor.execute(sql, parameters)

        # Save the string value contained within the tuple of each row
        model_options = []
        for model_option in self.cursor.fetchall():
            model_options.append(model_option[0])

        return model_options

    def load_model(self) -> object:
        """Create model using saved weights and biases.

           Returns:
               a Model object.

        """
        self.use_gpu = self.use_gpu_check_button_var.get()

        # Query data of selected saved model from database
        sql = """
        SELECT * FROM Models WHERE Dataset=? AND Name=?
        """
        parameters = (self.dataset.replace(" ", "_"), self.model_option_menu_var.get())
```

```
        self.cursor.execute(sql, parameters)
        data = self.cursor.fetchone()
        hidden_layers_shape_input = [layer for layer in data[3].replace(' ', '').split(',') if layer != '']

        # Create Model
        if not self.use_gpu:
            if self.dataset == "MNIST":
                from school_project.models.cpu.mnist import MNISTModel as Model
            elif self.dataset == "Cat Recognition":
                from school_project.models.cpu.cat_recognition import CatRecognitionModel as Model
            elif self.dataset == "XOR":
                from school_project.models.cpu.xor import XORModel as Model
            model = Model(
                hidden_layers_shape=[int(neuron_count) for neuron_count in hidden_layers_shape_input],
                train_dataset_size=data[6],
                learning_rate=data[4],
                use_relu=data[7]
                )
            model.load_model_values(file_location=data[2])

        else:
            try:
                if self.dataset == "MNIST":
                    from school_project.models.gpu.mnist import MNISTModel as Model
                elif self.dataset == "Cat Recognition":
                    from school_project.models.gpu.cat_recognition import CatRecognitionModel as Model
                elif self.dataset == "XOR":
                    from school_project.models.gpu.xor import XORModel as Model
                model = Model(
                    hidden_layers_shape=[int(neuron_count) for neuron_count in hidden_layers_shape_input],
                    train_dataset_size=data[6],
                    learning_rate=data[4],
                    use_relu=data[7]
                    )
                model.load_model_values(file_location=data[2])
            except ImportError as ie:
                self.model_status_label.configure(
                                              text="Failed to initialise GPU",
                                              fg='red'
                                              )
                raise ImportError
        return model
```

3.4 __main__.py module

This module is the entrypoint to the project and loads the main window of the User Interface:

```
"""The entrypoint of A-level Computer Science NEA Programming Project."""

import os
```

```python
import sqlite3
import threading
import tkinter as tk
import tkinter.font as tkf
import uuid

import pympler.tracker as tracker

from school_project.frames import (HyperParameterFrame, TrainingFrame,
                                   LoadModelFrame, TestMNISTFrame,
                                   TestCatRecognitionFrame, TestXORFrame)

class SchoolProjectFrame(tk.Frame):
    """Main frame of school project."""
    def __init__(self, root: tk.Tk, width: int, height: int, bg: str) -> None:
        """Initialise school project pages.

        Args:
            root (tk.Tk): the widget object that contains this widget.
            width (int): the pixel width of the frame.
            height (int): the pixel height of the frame.
            bg (str): the hex value or name of the frame's background colour.
        Raises:
            TypeError: if root, width or height are not of the correct type.

        """
        super().__init__(master=root, width=width, height=height, bg=bg)
        self.root = root.title("School Project")
        self.WIDTH = width
        self.HEIGHT = height
        self.BG = bg

        # Setup school project frame variables
        self.hyper_parameter_frame: HyperParameterFrame
        self.training_frame: TrainingFrame
        self.load_model_frame: LoadModelFrame
        self.test_frame: TestMNISTFrame | TestCatRecognitionFrame | TestXORFrame
        self.connection, self.cursor = self.setup_database()
        self.model = None

        # Record if the model should be saved after testing,
        # as only newly created models should be given the option to be saved.
        self.saving_model: bool

        # Setup school project frame widgets
        self.exit_hyper_parameter_frame_button = tk.Button(
                                    master=self,
                                    width=13,
                                    height=1,
                                    font=tkf.Font(size=12),
                                    text="Exit",
                                    command=self.exit_hyper_parameter_frame
                                    )
        self.exit_load_model_frame_button = tk.Button(
                                          master=self,
                                          width=13,
                                          height=1,
                                          font=tkf.Font(size=12),
                                          text="Exit",
                                          command=self.exit_load_model_frame
                                          )
        self.train_button = tk.Button(master=self,
```

```
                                     width=13,
                                     height=1,
                                     font=tkf.Font(size=12),
                                     text="Train Model",
                                     command=self.enter_training_frame)
        self.stop_training_button = tk.Button(
                                       master=self,
                                       width=15, height=1,
                                       font=tkf.Font(size=12),
                                       text="Stop Training Model",
                                       command=lambda: self.model.set_running(
                                                           value=False
                                                           )
                                       )
        self.test_created_model_button = tk.Button(
                                              master=self,
                                              width=13, height=1,
                                              font=tkf.Font(size=12),
                                              text="Test Model",
                                              command=self.test_created_model
                                              )
        self.test_loaded_model_button = tk.Button(
                                               master=self,
                                               width=13, height=1,
                                               font=tkf.Font(size=12),
                                               text="Test Model",
                                               command=self.test_loaded_model
                                               )
        self.delete_loaded_model_button = tk.Button(
                                               master=self,
                                               width=13, height=1,
                                               font=tkf.Font(size=12),
                                               text="Delete Model",
                                               command=self.delete_loaded_model
                                               )
        self.save_model_label = tk.Label(
                                 master=self,
                                 text="Enter a name for your trained model:",
                                 bg=self.BG,
                                 font=('Arial', 15)
                                 )
        self.save_model_name_entry = tk.Entry(master=self, width=13)
        self.save_model_button = tk.Button(master=self,
                                          width=13,
                                          height=1,
                                          font=tkf.Font(size=12),
                                          text="Save Model",
                                          command=self.save_model)
        self.exit_button = tk.Button(master=self,
                                    width=13, height=1,
                                    font=tkf.Font(size=12),
                                    text="Exit",
                                    command=self.enter_home_frame)

        # Setup home frame
        self.home_frame = tk.Frame(master=self,
                                  width=self.WIDTH,
                                  height=self.HEIGHT,
                                  bg=self.BG)
        self.title_label = tk.Label(
                              master=self.home_frame,
                              bg=self.BG,
```

```
                        font=('Arial', 20),
                        text="A-level Computer Science NEA Programming Project"
                        )
        self.about_label = tk.Label(
            master=self.home_frame,
            bg=self.BG,
            font=('Arial', 14),
            text="An investigation into how Artificial Neural Networks work, " +
            "the effects of their hyper-parameters and their applications " +
            "in Image Recognition.\n\n" +
            " - Max Cotton"
            )
        self.model_menu_label = tk.Label(master=self.home_frame,
                                         bg=self.BG,
                                         font=('Arial', 14),
                                         text="Create a new model " +
                                         "or load a pre-trained model "
                                         "for one of the following datasets:")
        self.dataset_option_menu_var = tk.StringVar(master=self.home_frame,
                                                    value="MNIST")
        self.dataset_option_menu = tk.OptionMenu(self.home_frame,
                                                 self.dataset_option_menu_var,
                                                 "MNIST",
                                                 "Cat Recognition",
                                                 "XOR")
        self.create_model_button = tk.Button(
                                      master=self.home_frame,
                                      width=13, height=1,
                                      font=tkf.Font(size=12),
                                      text="Create Model",
                                      command=self.enter_hyper_parameter_frame
                                      )
        self.load_model_button = tk.Button(master=self.home_frame,
                                           width=13, height=1,
                                           font=tkf.Font(size=12),
                                           text="Load Model",
                                           command=self.enter_load_model_frame)

        # Grid home frame widgets
        self.title_label.grid(row=0, column=0, columnspan=4, pady=(10,0))
        self.about_label.grid(row=1, column=0, columnspan=4, pady=(10,50))
        self.model_menu_label.grid(row=2, column=0, columnspan=4)
        self.dataset_option_menu.grid(row=3, column=0, columnspan=4, pady=30)
        self.create_model_button.grid(row=4, column=1)
        self.load_model_button.grid(row=4, column=2)

        self.home_frame.pack()

        # Setup frame attributes
        self.grid_propagate(flag=False)
        self.pack_propagate(flag=False)

    @staticmethod
    def setup_database() -> tuple[sqlite3.Connection, sqlite3.Cursor]:
        """Create a connection to the pretrained_models database file and
           setup base table if needed.

           Returns:
               a tuple of the database connection and the cursor for it.

        """
        connection = sqlite3.connect(
```

```
                                database='school_project/saved_models.db'
                                )
        cursor = connection.cursor()
        cursor.execute("""
        CREATE TABLE IF NOT EXISTS Models
        (Model_ID INTEGER PRIMARY KEY,
        Dataset TEXT,
        File_Location TEXT,
        Hidden_Layers_Shape TEXT,
        Learning_Rate FLOAT,
        Name TEXT,
        Train_Dataset_Size INTEGER,
        Use_ReLu INTEGER,
        UNIQUE (Dataset, Name))
        """)
        return (connection, cursor)

    def enter_hyper_parameter_frame(self) -> None:
        """Unpack home frame and pack hyper-parameter frame."""
        self.home_frame.pack_forget()
        self.hyper_parameter_frame = HyperParameterFrame(
                                    root=self,
                                    width=self.WIDTH,
                                    height=self.HEIGHT,
                                    bg=self.BG,
                                    dataset=self.dataset_option_menu_var.get()
                                    )
        self.hyper_parameter_frame.pack()
        self.train_button.pack()
        self.exit_hyper_parameter_frame_button.pack(pady=(10,0))

    def enter_load_model_frame(self) -> None:
        """Unpack home frame and pack load model frame."""
        self.home_frame.pack_forget()
        self.load_model_frame = LoadModelFrame(
                                    root=self,
                                    width=self.WIDTH,
                                    height=self.HEIGHT,
                                    bg=self.BG,
                                    connection=self.connection,
                                    cursor=self.cursor,
                                    dataset=self.dataset_option_menu_var.get()
                                    )
        self.load_model_frame.pack()

        # Don't give option to test loaded model if no models have been saved
        # for the dataset.
        if len(self.load_model_frame.model_options) > 0:
            self.test_loaded_model_button.pack()
            self.delete_loaded_model_button.pack(pady=(5,0))

        self.exit_load_model_frame_button.pack(pady=(5,0))

    def exit_hyper_parameter_frame(self) -> None:
        """Unpack hyper-parameter frame and pack home frame."""
        self.hyper_parameter_frame.pack_forget()
        self.train_button.pack_forget()
        self.exit_hyper_parameter_frame_button.pack_forget()
        self.home_frame.pack()

    def exit_load_model_frame(self) -> None:
        """Unpack load model frame and pack home frame."""
```

```
        self.load_model_frame.pack_forget()
        self.test_loaded_model_button.pack_forget()
        self.delete_loaded_model_button.pack_forget()
        self.exit_load_model_frame_button.pack_forget()
        self.home_frame.pack()

    def enter_training_frame(self) -> None:
        """Load untrained model from hyper parameter frame,
           unpack hyper-parameter frame, pack training frame
           and begin managing the training thread.
        """
        try:
            self.model = self.hyper_parameter_frame.create_model()
        except (ValueError, ImportError) as e:
            return
        self.hyper_parameter_frame.pack_forget()
        self.train_button.pack_forget()
        self.exit_hyper_parameter_frame_button.pack_forget()
        self.training_frame = TrainingFrame(
                root=self,
                width=self.WIDTH,
                height=self.HEIGHT,
                bg=self.BG,
                model=self.model,
                epoch_count=self.hyper_parameter_frame.epoch_count_scale.get()
                )
        self.training_frame.pack()
        self.stop_training_button.pack()
        self.manage_training(train_thread=self.training_frame.train_thread)

    def manage_training(self, train_thread: threading.Thread) -> None:
        """Wait for model training thread to finish,
           then plot training losses on training frame.

        Args:
            train_thread (threading.Thread):
            the thread running the model's train() method.
        Raises:
            TypeError: if train_thread is not of type threading.Thread.

        """
        if not train_thread.is_alive():
            self.training_frame.training_progress_label.pack_forget()
            self.training_frame.plot_losses(model=self.model)
            self.stop_training_button.pack_forget()
            self.test_created_model_button.pack(pady=(30,0))
        else:
            self.training_frame.training_progress_label.configure(
                                              text=self.model.training_progress
                                              )
            self.after(100, self.manage_training, train_thread)

    def test_created_model(self) -> None:
        """Unpack training frame, pack test frame for the dataset
           and begin managing the test thread."""
        self.saving_model = True
        self.training_frame.pack_forget()
        self.test_created_model_button.pack_forget()
        if self.hyper_parameter_frame.dataset == "MNIST":
            self.test_frame = TestMNISTFrame(
                                   root=self,
                                   width=self.WIDTH,
```

```python
                                height=self.HEIGHT,
                                bg=self.BG,
                                use_gpu=self.hyper_parameter_frame.use_gpu,
                                model=self.model
                                )
        elif self.hyper_parameter_frame.dataset == "Cat Recognition":
            self.test_frame = TestCatRecognitionFrame(
                                root=self,
                                width=self.WIDTH,
                                height=self.HEIGHT,
                                bg=self.BG,
                                use_gpu=self.hyper_parameter_frame.use_gpu,
                                model=self.model
                                )
        elif self.hyper_parameter_frame.dataset == "XOR":
            self.test_frame = TestXORFrame(root=self,
                                        width=self.WIDTH,
                                        height=self.HEIGHT,
                                        bg=self.BG,
                                        model=self.model)
        self.test_frame.pack()
        self.manage_testing(test_thread=self.test_frame.test_thread)

    def test_loaded_model(self) -> None:
        """Load saved model from load model frame, unpack load model frame,
           pack test frame for the dataset and begin managing the test thread."""
        self.saving_model = False
        try:
            self.model = self.load_model_frame.load_model()
        except (ValueError, ImportError) as e:
            return
        self.load_model_frame.pack_forget()
        self.test_loaded_model_button.pack_forget()
        self.delete_loaded_model_button.pack_forget()
        self.exit_load_model_frame_button.pack_forget()
        if self.load_model_frame.dataset == "MNIST":
            self.test_frame = TestMNISTFrame(
                                     root=self,
                                     width=self.WIDTH,
                                     height=self.HEIGHT,
                                     bg=self.BG,
                                     use_gpu=self.load_model_frame.use_gpu,
                                     model=self.model
                                     )
        elif self.load_model_frame.dataset == "Cat Recognition":
            self.test_frame = TestCatRecognitionFrame(
                                     root=self,
                                     width=self.WIDTH,
                                     height=self.HEIGHT,
                                     bg=self.BG,
                                     use_gpu=self.load_model_frame.use_gpu,
                                     model=self.model
                                     )
        elif self.load_model_frame.dataset == "XOR":
            self.test_frame = TestXORFrame(root=self,
                                        width=self.WIDTH,
                                        height=self.HEIGHT,
                                        bg=self.BG,
                                        model=self.model)
        self.test_frame.pack()
        self.manage_testing(test_thread=self.test_frame.test_thread)

```

```
    def manage_testing(self, test_thread: threading.Thread) -> None:
        """Wait for model test thread to finish,
           then plot results on test frame.

        Args:
            test_thread (threading.Thread):
            the thread running the model's predict() method.
        Raises:
            TypeError: if test_thread is not of type threading.Thread.

        """
        if not test_thread.is_alive():
            self.test_frame.plot_results(model=self.model)
            if self.saving_model:
                self.save_model_label.pack(pady=(30,0))
                self.save_model_name_entry.pack(pady=10)
                self.save_model_button.pack()
            self.exit_button.pack(pady=(20,0))
        else:
            self.after(1_000, self.manage_testing, test_thread)

    def save_model(self) -> None:
        """Save the model, save the model information to the database, then
           enter the home frame."""
        model_name = self.save_model_name_entry.get()

        # Check if model name is empty
        if model_name == '':
            self.test_frame.model_status_label.configure(
                                            text="Model name can not be blank",
                                            fg='red'
                                            )
            return

        # Check if model name has already been taken
        dataset = self.dataset_option_menu_var.get().replace(" ", "_")
        sql = """
        SELECT Name FROM Models WHERE Dataset=?
        """
        parameters = (dataset,)
        self.cursor.execute(sql, parameters)
        for saved_model_name in self.cursor.fetchall():
            if saved_model_name[0] == model_name:
                self.test_frame.model_status_label.configure(
                                                        text="Model name taken",
                                                        fg='red'
                                                        )
                return

        # Save model to random hex file name
        file_location = f"school_project/saved-models/{uuid.uuid4().hex}.npz"
        self.model.save_model_values(file_location=file_location)

        # Save the model information to the database
        sql = """
        INSERT INTO Models
        (Dataset, File_Location, Hidden_Layers_Shape, Learning_Rate, Name,
    ↪   Train_Dataset_Size, Use_ReLu)
        VALUES (?, ?, ?, ?, ?, ?, ?)
        """
        parameters = (
                      dataset,
```

```
                    file_location,
                    self.hyper_parameter_frame.hidden_layers_shape_entry.get(),
                    self.hyper_parameter_frame.learning_rate_scale.get(),
                    model_name,
                    self.hyper_parameter_frame.train_dataset_size_scale.get(),
                    self.hyper_parameter_frame.use_relu_check_button_var.get()
                    )
        self.cursor.execute(sql, parameters)
        self.connection.commit()

        self.enter_home_frame()

    def delete_loaded_model(self) -> None:
        """Delete saved model file and model data from the database."""
        dataset = self.dataset_option_menu_var.get().replace(" ", "_")
        model_name = self.load_model_frame.model_option_menu_var.get()

        # Delete saved model
        sql = f"""SELECT File_Location FROM Models WHERE Dataset=? AND Name=?"""
        parameters = (dataset, model_name)
        self.cursor.execute(sql, parameters)
        os.remove(self.cursor.fetchone()[0])

        # Remove model data from database
        sql = """DELETE FROM Models WHERE Dataset=? AND Name=?"""
        parameters = (dataset, model_name)
        self.cursor.execute(sql, parameters)
        self.connection.commit()

        # Reload load model frame with new options
        self.exit_load_model_frame()
        self.enter_load_model_frame()

    def enter_home_frame(self) -> None:
        """Unpack test frame and pack home frame."""
        self.model = None  # Free up trained Model from memory
        self.test_frame.pack_forget()
        if self.saving_model:
            self.save_model_label.pack_forget()
            self.save_model_name_entry.delete(0, tk.END)  # Clear entry's text
            self.save_model_name_entry.pack_forget()
            self.save_model_button.pack_forget()
        self.exit_button.pack_forget()
        self.home_frame.pack()
        summary_tracker.create_summary()  # BUG: Object summary seems to reduce
                                          # memory leak greatly

def main() -> None:
    """Entrypoint of project."""
    root = tk.Tk()
    school_project_frame = SchoolProjectFrame(root=root, width=1280,
                                              height=835, bg='white')
    school_project_frame.pack(side='top', fill='both', expand=True)
    root.mainloop()

    # Stop model training when GUI closes
    if school_project_frame.model != None:
        school_project_frame.model.set_running(value=False)

if __name__ == "__main__":
    summary_tracker = tracker.SummaryTracker()  # Setup object tracker
    main()
```