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

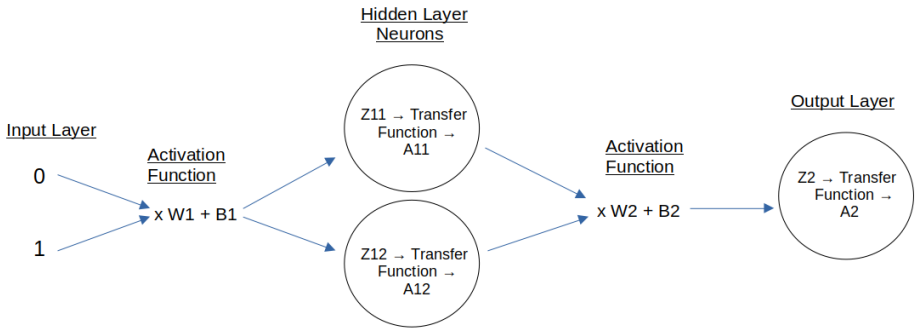


Figure 1: This shows an Artificial Neural Network with one single hidden layer and is known as a Shallow Neural Netwok.

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

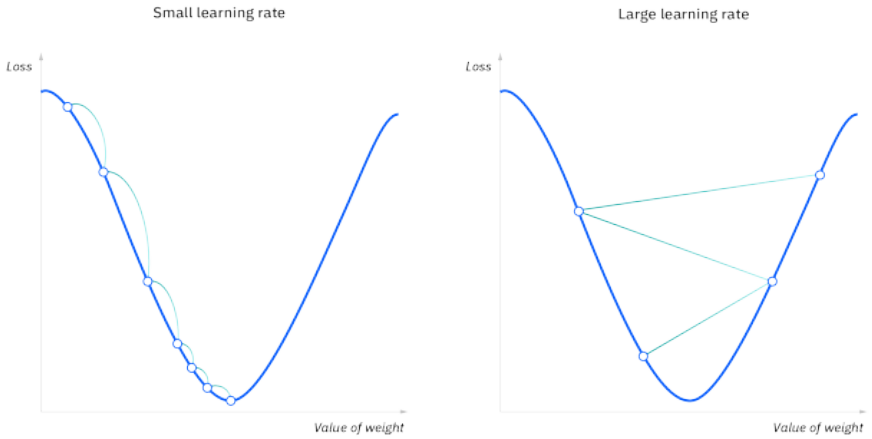


Figure 2: Gradient Descent
sourced from https://www.ibm.com/topics/gradient-descent

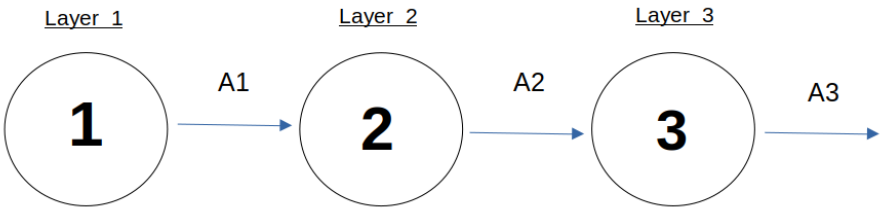


Figure 3: This shows an abstracted view of an Artificial Neural Network with multiple hidden layers and is known as a Deep Neural Netwok.

- 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.

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

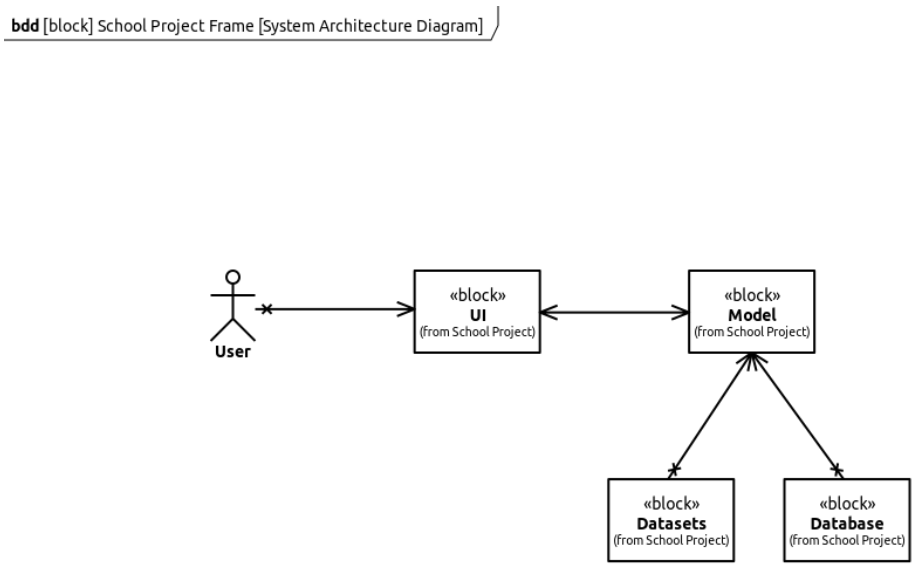
2.1 Introduction

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.

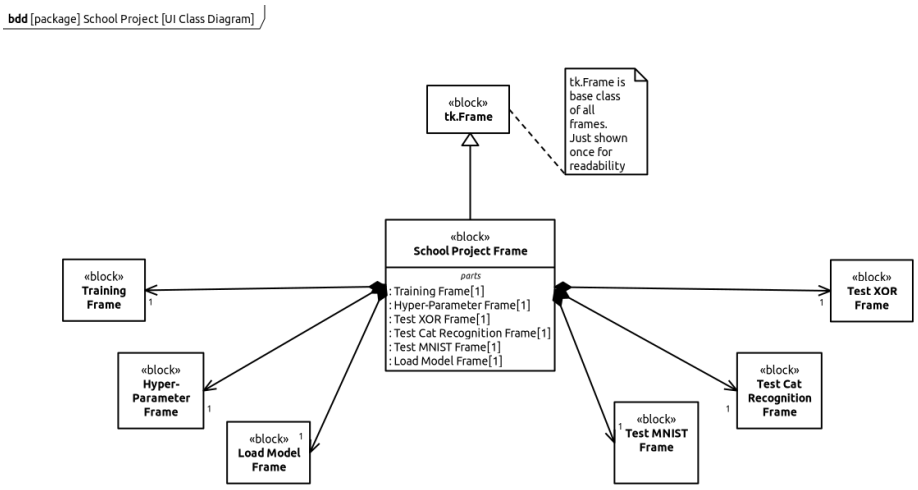
I will also be using SysML for designing the following diagrams.

2.2 System Architecture

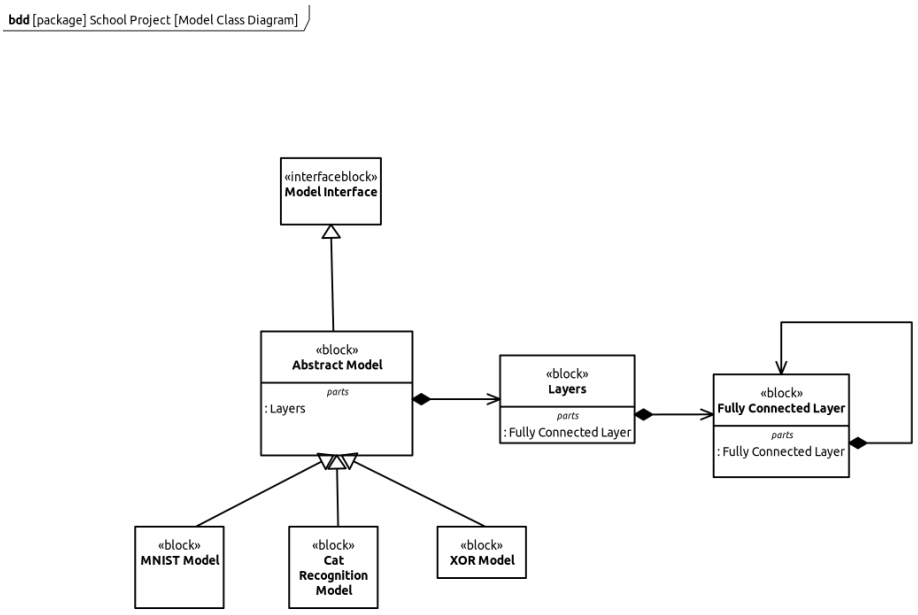


2.3 Class Diagrams

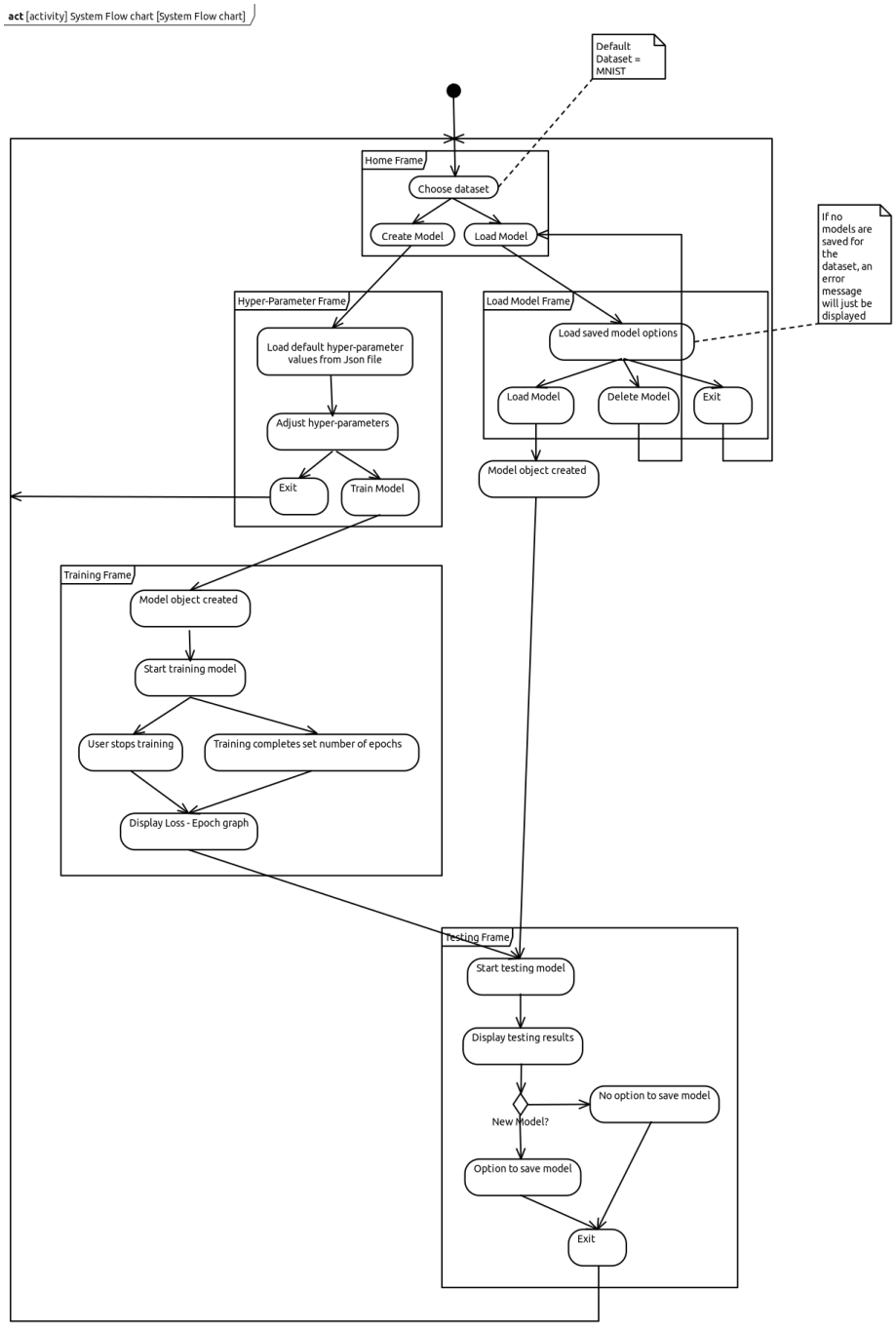
2.3.1 UI Class Diagram



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.

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

```
.
|-- .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
```

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:

```
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
here.*

Usage
```
```

````
59
60  Run with:
61  - Windows:
62    ```
63    python school_project
64    ```
65  - Linux:
66    ```
67    python3 school_project
68    ```
69
70  ## Development
71
72  Install the dependencies and the project to the venv in developing
    ↪  mode with:
73  - Windows:
74    ```
75    python setup.py develop
76    ```
77  - Linux:
78    ```
79    python3 setup.py develop
80    ```
81
82  Run Tests with:
83  - Windows:
84    ```
85    python -m unittest discover .\school_project\test\
86    ```
87  - Linux:
88    ```
89    python3 -m unittest discover ./school_project/test/
90    ```
91
92  Compile Project Report PDF with:
93  ```
94  make all
95  ```
96  *Note: This requires the Latexmk library*
````

– Which will generate the following:

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:

```
"""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:
        """Setup model layers"""
        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."""

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):
        """Initialise linked list."""
        self.head = None
        self.tail = None

    def __iter__(self):
        """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):
        """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
        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
```

```
74
75      def __repr__(self) -> str:
76          """Read values of the layer.
77
78          Returns:
79              a string description of the layers's
80              weights, bias and learning rate values.
81
82          """
83          return (f"Weights: {self.weights.tolist()}\n" +
84                  f"Biases: {self.biases.tolist()}\n")
85
86      def init_layer_values_random(self) -> None:
87          """Initialise weights to random values and biases to 0s"""
88          np.random.seed(1)  # Sets up pseudo random values for layer weight
            ↪  arrays
89          self.weights = np.random.rand(self.output_neuron_count,
            ↪  self.input_neuron_count) - 0.5
90          self.biases = np.zeros(shape=(self.output_neuron_count, 1))
91
92      def init_layer_values_zeros(self) -> None:
93          """Initialise weights to 0s and biases to 0s"""
94          self.weights = np.zeros(shape=(self.output_neuron_count,
            ↪  self.input_neuron_count))
95          self.biases = np.zeros(shape=(self.output_neuron_count, 1))
96
97      def back_propagation(self, dloss_doutput) -> np.ndarray:
98          """Adjust the weights and biases via gradient descent.
99
100         Args:
101             dloss_doutput (numpy.ndarray): the derivative of the loss of the
    ↪
102             layer's output, with respect to the layer's output.
103         Returns:
104             a numpy.ndarray derivative of the loss of the layer's input,
105             with respect to the layer's input.
106         Raises:
107             ValueError:
108             if dloss_doutput
109             is not a suitable multiplier with the weights
110             (incorrect shape)
111
112         """
113         match self.transfer_type:
114             case 'sigmoid':
115                 dloss_dz = dloss_doutput *
                    ↪  sigmoid_derivative(output=self.output)
116             case 'relu':
117                 dloss_dz = dloss_doutput *
                    ↪  relu_derivative(output=self.output)
118
119         dloss_dweights = np.dot(dloss_dz, self.input.T)
120         dloss_dbiases = np.sum(dloss_dz)
121
122         assert dloss_dweights.shape == self.weights.shape
123
124         dloss_dinput = np.dot(self.weights.T, dloss_dz)
125
126         # Update weights and biases
127         self.weights -= self.learning_rate * dloss_dweights
128         self.biases -= self.learning_rate * dloss_dbiases
129
```

```
130             return dloss_dinput
131
132         def forward_propagation(self, inputs) -> np.ndarray:
133             """Generate a layer output with the weights and biases.
134
135             Args:
136                 inputs (np.ndarray): the input values to the layer.
137             Returns:
138                 a numpy.ndarray of the output values.
139
140             """
141             self.input = inputs
142             z = np.dot(self.weights, self.input) + self.biases
143             if self.transfer_type == 'sigmoid':
144                 self.output = sigmoid(z)
145             elif self.transfer_type == 'relu':
146                 self.output = relu(z)
147             return self.output
148
149     class AbstractModel(ModelInterface):
150         """ANN model with variable number of hidden layers"""
151         def __init__(self,
152                      hidden_layers_shape: list[int],
153                      train_dataset_size: int,
154                      learning_rate: float,
155                      use_relu: bool) -> None:
156             """Initialise model values.
157
158             Args:
159                 hidden_layers_shape (list[int]):
160                 list of the number of neurons in each hidden layer.
161                 train_dataset_size (int): the number of train dataset inputs to
    ↪    use.
162                 learning_rate (float): the learning rate of the model.
163                 use_relu (bool): True or False whether the ReLu Transfer
    ↪    function
164                 should be used.
165
166             """
167             # Setup model data
168             self.train_inputs, self.train_outputs,\
169             self.test_inputs, self.test_outputs = self.load_datasets(
170
                                                    ↪  train_dataset_size=train_dataset_size
171                                                    )
172             self.train_losses: list[float]
173             self.test_prediction: np.ndarray
174             self.test_prediction_accuracy: float
175             self.training_progress = ""
176             self.training_time: float
177
178             # Setup model attributes
179             self.__running = True
180             self.input_neuron_count: int = self.train_inputs.shape[0]
181             self.input_count = self.train_inputs.shape[1]
182             self.hidden_layers_shape = hidden_layers_shape
183             self.output_neuron_count = self.train_outputs.shape[0]
184             self.layers_shape = [f'{layer}' for layer in (
185                                  [self.input_neuron_count] +
186                                  self.hidden_layers_shape +
187                                  [self.output_neuron_count]
188                                  )]
```

```
189             self.use_relu = use_relu
190
191             # Setup model values
192             self.layers = _Layers()
193             self.learning_rate = learning_rate
194
195         def __repr__(self) -> str:
196             """Read current state of model.
197
198             Returns:
199                 a string description of the model's shape,
200                 weights, bias and learning rate values.
201
202             """
203             return (f"Layers Shape: {','.join(self.layers_shape)}\n" +
204                     f"Learning Rate: {self.learning_rate}")
205
206         def set_running(self, value:bool):
207             self.__running = value
208
209         def _setup_layers(setup_values: callable) -> None:
210             """Setup model layers"""
211             def decorator(self, *args, **kwargs):
212                 # Check if setting up Deep Network
213                 if len(self.hidden_layers_shape) > 0:
214                     if self.use_relu:
215
216                         # Add input layer
217                         self.layers.head = _FullyConnectedLayer(
218
                                                 ↪  learning_rate=self.learning_rate,
219
                                                 ↪  input_neuron_count=self.input_neuron_count,
220
                                                 ↪  output_neuron_count=self.hidden_layers_shape[0],
221                                              transfer_type='relu'
222                                              )
223                         current_layer = self.layers.head
224
225                         # Add hidden layers
226                         for layer in range(len(self.hidden_layers_shape) - 1):
227                             current_layer.next_layer = _FullyConnectedLayer(
228                                         learning_rate=self.learning_rate,
229
                                         ↪  input_neuron_count=self.hidden_layers_shape[layer],
230
                                         ↪  output_neuron_count=self.hidden_layers_shape[layer
                                         ↪  + 1],
231                                      transfer_type='relu'
232                                      )
233                             current_layer.next_layer.previous_layer =
                             ↪  current_layer
234                             current_layer = current_layer.next_layer
235                     else:
236
237                         # Add input layer
238                         self.layers.head = _FullyConnectedLayer(
239
                                                 ↪  learning_rate=self.learning_rate,
240
                                                 ↪  input_neuron_count=self.input_neuron_count,
241
                                                 ↪  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,
                                    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:
```

```
296                 layer.init_layer_values_zeros()
297
298     @_setup_layers
299     def load_model_values(self, file_location: str) -> None:
300         """Load weights and bias/biases from .npz file.
301
302         Args:
303             file_location (str): the location of the file to load from.
304
305         """
306         data: dict[str, np.ndarray] = np.load(file=file_location)
307
308         # Initialise Layer values
309         i = 0
310         keys = list(data.keys())
311         for layer in self.layers:
312             layer.weights = data[keys[i]]
313             layer.biases = data[keys[i + 1]]
314             i += 2
315
316     def back_propagation(self, dloss_doutput) -> None:
317         """Train each layer's weights and biases.
318
319         Args:
320             dloss_doutput (np.ndarray): the derivative of the loss of the
321             output layer's output, with respect to the output layer's
    ↪  output.
322
323         """
324         for layer in reversed(self.layers):
325             dloss_doutput =
                ↪  layer.back_propagation(dloss_doutput=dloss_doutput)
326
327     def forward_propagation(self) -> np.ndarray:
328         """Generate a prediction with the layers.
329
330         Returns:
331             a numpy.ndarray of the prediction values.
332
333         """
334         output = self.train_inputs
335         for layer in self.layers:
336             output = layer.forward_propagation(inputs=output)
337         return output
338
339     def test(self) -> None:
340         """Test the layers' trained weights and biases."""
341         output = self.test_inputs
342         for layer in self.layers:
343             output = layer.forward_propagation(inputs=output)
344         self.test_prediction = output
345
346         # Calculate performance of model
347         self.test_prediction_accuracy = calculate_prediction_accuracy(
348
                                                ↪  prediction=self.test_prediction,
349                                             outputs=self.test_outputs
350                                             )
351
352     def train(self, epoch_count: int) -> None:
353         """Train layers' weights and biases.
354
```

```
            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.

        """
        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:

```
"""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.

        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:

```
"""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],
                 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

        return train_inputs, train_outputs, test_inputs, test_outputs
```

- xor.py module

```
"""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:

```
{
    "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:

```
"""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.
            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
                                                                )
```

```
41
42          # Setup widgets
43          self.title_label = tk.Label(master=self,
44                                      bg=self.BG,
45                                      font=('Arial', 20),
46                                      text=dataset)
47          self.about_label = tk.Label(
48                                master=self,
49                                bg=self.BG,
50                                font=('Arial', 14),
51
                                  ↪  text=self.default_hyper_parameters['description']
52                                )
53          self.learning_rate_scale = tk.Scale(
54                              master=self,
55                              bg=self.BG,
56                              orient='horizontal',
57                              label="Learning Rate",
58                              length=185,
59                              from_=0,
60
                                ↪  to=self.default_hyper_parameters['maxLearningRate'],
61                              resolution=0.01
62                              )
63          self.learning_rate_scale.set(value=0.1)
64          self.epoch_count_scale = tk.Scale(master=self,
65                                            bg=self.BG,
66                                            orient='horizontal',
67                                            label="Epoch Count",
68                                            length=185,
69                                            from_=0,
70                                            to=10_000,
71                                            resolution=100)
72          self.epoch_count_scale.set(
73
                                  ↪  value=self.default_hyper_parameters['epochCount']
74                                )
75          self.train_dataset_size_scale = tk.Scale(
76                      master=self,
77                      bg=self.BG,
78                      orient='horizontal',
79                      label="Train Dataset Size",
80                      length=185,
81
                        ↪  from_=self.default_hyper_parameters['minTrainDatasetSize'],
82                      to=self.default_hyper_parameters['maxTrainDatasetSize'],
83                      resolution=1
84                      )
85          self.train_dataset_size_scale.set(
86
                         ↪  value=self.default_hyper_parameters['maxTrainDatasetSize']
87                       )
88          self.hidden_layers_shape_label = tk.Label(
89                                    master=self,
90                                    bg=self.BG,
91                                    font=('Arial', 12),
92                                    text="Enter the number of neurons in
                                      ↪  each\n" +
93                                           "hidden layer, separated by
                                             ↪  commas:"
94                                    )
95          self.hidden_layers_shape_entry = tk.Entry(master=self)
```

```
96          self.hidden_layers_shape_entry.insert(0, ",".join(
97              f"{neuron_count}" for neuron_count in
                ↪  self.default_hyper_parameters['hiddenLayersShape']
98              ))
99          self.use_relu_check_button_var = tk.BooleanVar(value=True)
100         self.use_relu_check_button = tk.Checkbutton(
101                                         master=self,
102                                         width=13, height=1,
103                                         font=tkf.Font(size=12),
104                                         text="Use ReLu",
105
                                            ↪  variable=self.use_relu_check_button_var
106                                             )
107         self.use_gpu_check_button_var = tk.BooleanVar()
108         self.use_gpu_check_button = tk.Checkbutton(
109                                         master=self,
110                                         width=13, height=1,
111                                         font=tkf.Font(size=12),
112                                         text="Use GPU",
113
                                            ↪  variable=self.use_gpu_check_button_var
114                                             )
115         self.model_status_label = tk.Label(master=self,
116                                            bg=self.BG,
117                                            font=('Arial', 15))
118
119         # Pack widgets
120         self.title_label.grid(row=0, column=0, columnspan=3)
121         self.about_label.grid(row=1, column=0, columnspan=3)
122         self.learning_rate_scale.grid(row=2, column=0, pady=(50,0))
123         self.epoch_count_scale.grid(row=3, column=0, pady=(30,0))
124         self.train_dataset_size_scale.grid(row=4, column=0, pady=(30,0))
125         self.hidden_layers_shape_label.grid(row=2, column=1,
126                                             padx=30, pady=(50,0))
127         self.hidden_layers_shape_entry.grid(row=3, column=1, padx=30)
128         self.use_relu_check_button.grid(row=2, column=2, pady=(30, 0))
129         self.use_gpu_check_button.grid(row=3, column=2, pady=(30, 0))
130         self.model_status_label.grid(row=5, column=0,
131                                      columnspan=3, pady=50)
132
133     def load_default_hyper_parameters(self, dataset: str) -> dict[
134                                                 str,
135                                                 str | int | list[int] |
                                                    ↪  float
136                                                 ]:
137         """Load the dataset's default hyper-parameters from the json file.
138
139            Args:
140                dataset (str): the name of the dataset to load
    ↪  hyper-parameters
141                for. ('MNIST', 'Cat Recognition' or 'XOR')
142             Returns:
143                 a dictionary of default hyper-parameter values.
144         """
145         with open('school_project/frames/hyper-parameter-defaults.json') as
            ↪  f:
146             return json.load(f)[dataset]
147
148     def create_model(self) -> object:
149         """Create and return a Model using the hyper-parameters set.
150
151            Returns:
```

```
152                 a Model object.
153             """
154             self.use_gpu = self.use_gpu_check_button_var.get()
155 
156             # Validate hidden layers shape input
157             hidden_layers_shape_input = [layer for layer in
                ↪  self.hidden_layers_shape_entry.get().replace(' ',
                ↪  '').split(',') if layer != '']
158             for layer in hidden_layers_shape_input:
159                 if not layer.isdigit():
160                     self.model_status_label.configure(
161                                             text="Invalid hidden layers shape",
162                                             fg='red'
163                                             )
164                     raise ValueError
165 
166             # Create Model
167             if not self.use_gpu:
168                 if self.dataset == "MNIST":
169                     from school_project.models.cpu.mnist import MNISTModel as
                        ↪  Model
170                 elif self.dataset == "Cat Recognition":
171                     from school_project.models.cpu.cat_recognition import
                        ↪  CatRecognitionModel as Model
172                 elif self.dataset == "XOR":
173                     from school_project.models.cpu.xor import XORModel as Model
174                 model = Model(hidden_layers_shape = [int(neuron_count) for
                    ↪  neuron_count in hidden_layers_shape_input],
175                               train_dataset_size =
                                  ↪  self.train_dataset_size_scale.get(),
176                               learning_rate = self.learning_rate_scale.get(),
177                               use_relu = self.use_relu_check_button_var.get())
178                 model.create_model_values()
179 
180             else:
181                 try:
182                     if self.dataset == "MNIST":
183                         from school_project.models.gpu.mnist import MNISTModel
                            ↪  as Model
184                     elif self.dataset == "Cat Recognition":
185                         from school_project.models.gpu.cat_recognition import
                            ↪  CatRecognitionModel as Model
186                     elif self.dataset == "XOR":
187                         from school_project.models.gpu.xor import XORModel as
                            ↪  Model
188                     model = Model(hidden_layers_shape = [int(neuron_count) for
                        ↪  neuron_count in hidden_layers_shape_input],
189                                   train_dataset_size =
                                      ↪  self.train_dataset_size_scale.get(),
190                                   learning_rate =
                                      ↪  self.learning_rate_scale.get(),
191                                   use_relu =
                                      ↪  self.use_relu_check_button_var.get())
192                     model.create_model_values()
193                 except ImportError as ie:
194                     self.model_status_label.configure(
195                                             text="Failed to initialise GPU",
196                                             fg='red'
197                                             )
198                     raise ImportError
199             return model
200 
```

```
201 class TrainingFrame(tk.Frame):
202     """Frame for training page."""
203     def __init__(self, root: tk.Tk, width: int,
204                  height: int, bg: str,
205                  model: object, epoch_count: int) -> None:
206         """Initialise training frame widgets.
207 
208         Args:
209             root (tk.Tk): the widget object that contains this widget.
210             width (int): the pixel width of the frame.
211             height (int): the pixel height of the frame.
212             bg (str): the hex value or name of the frame's background
    ↪  colour.
213             model (object): the Model object to be trained.
214             epoch_count (int): the number of training epochs.
215         Raises:
216             TypeError: if root, width or height are not of the correct
    ↪  type.
217 
218         """
219         super().__init__(master=root, width=width, height=height, bg=bg)
220         self.root = root
221         self.WIDTH = width
222         self.HEIGHT = height
223         self.BG = bg
224 
225         # Setup widgets
226         self.model_status_label = tk.Label(master=self,
227                                            bg=self.BG,
228                                            font=('Arial', 15))
229         self.training_progress_label = tk.Label(master=self,
230                                                 bg=self.BG,
231                                                 font=('Arial', 15))
232         self.loss_figure: Figure = Figure()
233         self.loss_canvas: FigureCanvasTkAgg = FigureCanvasTkAgg(
234 
                                                    ↪  figure=self.loss_figure,
235                                                 master=self
236                                                 )
237 
238         # Pack widgets
239         self.model_status_label.pack(pady=(30,0))
240         self.training_progress_label.pack(pady=30)
241 
242         # Start training thread
243         self.model_status_label.configure(
244                                           text="Training weights and
                                                ↪  biases...",
245                                           fg='red'
246                                           )
247         self.train_thread: threading.Thread = threading.Thread(
248 
                                                               ↪  target=model.train,
249 
                                                               ↪  args=(epoch_count,)
250                                                               )
251         self.train_thread.start()
252 
253     def plot_losses(self, model: object) -> None:
254         """Plot losses of Model training.
255 
256             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:

```
"""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))
```

```
94              self.model_option_menu.grid(row=3, column=0, padx=(0,30),
                ↪  pady=(10,0))
95          self.model_status_label.grid(row=4, column=0,
96                                       columnspan=3, pady=50)
97
98      def load_model_options(self) -> list[str]:
99          """Load the model options from the database.
100
101             Returns:
102                 a list of the model options.
103         """
104         sql = f"""
105         SELECT Name FROM Models WHERE Dataset=?
106         """
107         parameters = (self.dataset.replace(" ", "_"),)
108         self.cursor.execute(sql, parameters)
109
110         # Save the string value contained within the tuple of each row
111         model_options = []
112         for model_option in self.cursor.fetchall():
113             model_options.append(model_option[0])
114
115         return model_options
116
117     def load_model(self) -> object:
118         """Create model using saved weights and biases.
119
120             Returns:
121                 a Model object.
122
123         """
124         self.use_gpu = self.use_gpu_check_button_var.get()
125
126         # Query data of selected saved model from database
127         sql = """
128         SELECT * FROM Models WHERE Dataset=? AND Name=?
129         """
130         parameters = (self.dataset.replace(" ", "_"),
                ↪  self.model_option_menu_var.get())
131         self.cursor.execute(sql, parameters)
132         data = self.cursor.fetchone()
133         hidden_layers_shape_input = [layer for layer in data[3].replace('
                ↪  ', '').split(',') if layer != '']
134
135         # Create Model
136         if not self.use_gpu:
137             if self.dataset == "MNIST":
138                 from school_project.models.cpu.mnist import MNISTModel as
                        ↪  Model
139             elif self.dataset == "Cat Recognition":
140                 from school_project.models.cpu.cat_recognition import
                        ↪  CatRecognitionModel as Model
141             elif self.dataset == "XOR":
142                 from school_project.models.cpu.xor import XORModel as Model
143             model = Model(
144                 hidden_layers_shape=[int(neuron_count) for neuron_count in
                        ↪  hidden_layers_shape_input],
145                 train_dataset_size=data[6],
146                 learning_rate=data[4],
147                 use_relu=data[7]
148                 )
149             model.load_model_values(file_location=data[2])
```

```
150
151             else:
152                 try:
153                     if self.dataset == "MNIST":
154                         from school_project.models.gpu.mnist import MNISTModel
                            ↪  as Model
155                     elif self.dataset == "Cat Recognition":
156                         from school_project.models.gpu.cat_recognition import
                            ↪  CatRecognitionModel as Model
157                     elif self.dataset == "XOR":
158                         from school_project.models.gpu.xor import XORModel as
                            ↪  Model
159                     model = Model(
160                         hidden_layers_shape=[int(neuron_count) for neuron_count
                            ↪  in hidden_layers_shape_input],
161                         train_dataset_size=data[6],
162                         learning_rate=data[4],
163                         use_relu=data[7]
164                         )
165                     model.load_model_values(file_location=data[2])
166                 except ImportError as ie:
167                     self.model_status_label.configure(
168                                                       text="Failed to initialise
                                                          ↪  GPU",
169                                                       fg='red'
170                                                       )
171                     raise ImportError
172             return model
```

3.4 __main__.py module

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

```
1   """The entrypoint of A-level Computer Science NEA Programming Project."""
2
3   import os
4   import sqlite3
5   import threading
6   import tkinter as tk
7   import tkinter.font as tkf
8   import uuid
9
10  import pympler.tracker as tracker
11
12  from school_project.frames.create_model import (HyperParameterFrame,
13                                                  TrainingFrame)
14  from school_project.frames.load_model import LoadModelFrame
15  from school_project.frames.test_model import (TestMNISTFrame,
16                                                TestCatRecognitionFrame,
17                                                TestXORFrame)
18
19  class SchoolProjectFrame(tk.Frame):
20      """Main frame of school project."""
21      def __init__(self, root: tk.Tk, width: int, height: int, bg: str) -> None:
22          """Initialise school project pages.
23
24          Args:
25              root (tk.Tk): the widget object that contains this widget.
26              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")
```

```
151         self.dataset_option_menu = tk.OptionMenu(self.home_frame,
152                                                  self.dataset_option_menu_var,
153                                                  "MNIST",
154                                                  "Cat Recognition",
155                                                  "XOR")
156         self.create_model_button = tk.Button(
157                                       master=self.home_frame,
158                                       width=13, height=1,
159                                       font=tkf.Font(size=12),
160                                       text="Create Model",
161                                       command=self.enter_hyper_parameter_frame
162                                       )
163         self.load_model_button = tk.Button(master=self.home_frame,
164                                            width=13, height=1,
165                                            font=tkf.Font(size=12),
166                                            text="Load Model",
167                                            command=self.enter_load_model_frame)
168 
169         # Grid home frame widgets
170         self.title_label.grid(row=0, column=0, columnspan=4, pady=(10,0))
171         self.about_label.grid(row=1, column=0, columnspan=4, pady=(10,50))
172         self.model_menu_label.grid(row=2, column=0, columnspan=4)
173         self.dataset_option_menu.grid(row=3, column=0, columnspan=4, pady=30)
174         self.create_model_button.grid(row=4, column=1)
175         self.load_model_button.grid(row=4, column=2)
176 
177         self.home_frame.pack()
178 
179         # Setup frame attributes
180         self.grid_propagate(flag=False)
181         self.pack_propagate(flag=False)
182 
183     @staticmethod
184     def setup_database() -> tuple[sqlite3.Connection, sqlite3.Cursor]:
185         """Create a connection to the pretrained_models database file and
186            setup base table if needed.
187 
188            Returns:
189                a tuple of the database connection and the cursor for it.
190 
191         """
192         connection = sqlite3.connect(
193                                   database='school_project/saved_models.db'
194                                   )
195         cursor = connection.cursor()
196         cursor.execute("""
197         CREATE TABLE IF NOT EXISTS Models
198         (Model_ID INTEGER PRIMARY KEY,
199         Dataset TEXT,
200         File_Location TEXT,
201         Hidden_Layers_Shape TEXT,
202         Learning_Rate FLOAT,
203         Name TEXT,
204         Train_Dataset_Size INTEGER,
205         Use_ReLu INTEGER,
206         UNIQUE (Dataset, Name))
207         """)
208         return (connection, cursor)
209 
210     def enter_hyper_parameter_frame(self) -> None:
211         """Unpack home frame and pack hyper-parameter frame."""
212         self.home_frame.pack_forget()
```

```
213             self.hyper_parameter_frame = HyperParameterFrame(
214                                          root=self,
215                                          width=self.WIDTH,
216                                          height=self.HEIGHT,
217                                          bg=self.BG,
218                                          dataset=self.dataset_option_menu_var.get()
219                                          )
220             self.hyper_parameter_frame.pack()
221             self.train_button.pack()
222             self.exit_hyper_parameter_frame_button.pack(pady=(10,0))
223 
224         def enter_load_model_frame(self) -> None:
225             """Unpack home frame and pack load model frame."""
226             self.home_frame.pack_forget()
227             self.load_model_frame = LoadModelFrame(
228                                          root=self,
229                                          width=self.WIDTH,
230                                          height=self.HEIGHT,
231                                          bg=self.BG,
232                                          connection=self.connection,
233                                          cursor=self.cursor,
234                                          dataset=self.dataset_option_menu_var.get()
235                                          )
236             self.load_model_frame.pack()
237 
238             # Don't give option to test loaded model if no models have been saved
239             # for the dataset.
240             if len(self.load_model_frame.model_options) > 0:
241                 self.test_loaded_model_button.pack()
242                 self.delete_loaded_model_button.pack(pady=(5,0))
243 
244             self.exit_load_model_frame_button.pack(pady=(5,0))
245 
246         def exit_hyper_parameter_frame(self) -> None:
247             """Unpack hyper-parameter frame and pack home frame."""
248             self.hyper_parameter_frame.pack_forget()
249             self.train_button.pack_forget()
250             self.exit_hyper_parameter_frame_button.pack_forget()
251             self.home_frame.pack()
252 
253         def exit_load_model_frame(self) -> None:
254             """Unpack load model frame and pack home frame."""
255             self.load_model_frame.pack_forget()
256             self.test_loaded_model_button.pack_forget()
257             self.delete_loaded_model_button.pack_forget()
258             self.exit_load_model_frame_button.pack_forget()
259             self.home_frame.pack()
260 
261         def enter_training_frame(self) -> None:
262             """Load untrained model from hyper parameter frame,
263                unpack hyper-parameter frame, pack training frame
264                and begin managing the training thread.
265             """
266             try:
267                 self.model = self.hyper_parameter_frame.create_model()
268             except (ValueError, ImportError) as e:
269                 return
270             self.hyper_parameter_frame.pack_forget()
271             self.train_button.pack_forget()
272             self.exit_hyper_parameter_frame_button.pack_forget()
273             self.training_frame = TrainingFrame(
274                     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,
                                    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)
```

```
337             self.test_frame.pack()
338             self.manage_testing(test_thread=self.test_frame.test_thread)
339 
340         def test_loaded_model(self) -> None:
341             """Load saved model from load model frame, unpack load model frame,
342                pack test frame for the dataset and begin managing the test thread."""
343             self.saving_model = False
344             try:
345                 self.model = self.load_model_frame.load_model()
346             except (ValueError, ImportError) as e:
347                 return
348             self.load_model_frame.pack_forget()
349             self.test_loaded_model_button.pack_forget()
350             self.delete_loaded_model_button.pack_forget()
351             self.exit_load_model_frame_button.pack_forget()
352             if self.load_model_frame.dataset == "MNIST":
353                 self.test_frame = TestMNISTFrame(
354                                                  root=self,
355                                                  width=self.WIDTH,
356                                                  height=self.HEIGHT,
357                                                  bg=self.BG,
358                                                  use_gpu=self.load_model_frame.use_gpu,
359                                                  model=self.model
360                                                  )
361             elif self.load_model_frame.dataset == "Cat Recognition":
362                 self.test_frame = TestCatRecognitionFrame(
363                                                  root=self,
364                                                  width=self.WIDTH,
365                                                  height=self.HEIGHT,
366                                                  bg=self.BG,
367                                                  use_gpu=self.load_model_frame.use_gpu,
368                                                  model=self.model
369                                                  )
370             elif self.load_model_frame.dataset == "XOR":
371                 self.test_frame = TestXORFrame(root=self,
372                                                    width=self.WIDTH,
373                                                    height=self.HEIGHT,
374                                                    bg=self.BG,
375                                                    model=self.model)
376             self.test_frame.pack()
377             self.manage_testing(test_thread=self.test_frame.test_thread)
378 
379         def manage_testing(self, test_thread: threading.Thread) -> None:
380             """Wait for model test thread to finish,
381                then plot results on test frame.
382 
383             Args:
384                 test_thread (threading.Thread):
385                 the thread running the model's predict() method.
386             Raises:
387                 TypeError: if test_thread is not of type threading.Thread.
388 
389             """
390             if not test_thread.is_alive():
391                 self.test_frame.plot_results(model=self.model)
392                 if self.saving_model:
393                     self.save_model_label.pack(pady=(30,0))
394                     self.save_model_name_entry.pack(pady=10)
395                     self.save_model_button.pack()
396                 self.exit_button.pack(pady=(20,0))
397             else:
398                 self.after(1_000, self.manage_testing, test_thread)
```

```
399
400     def save_model(self) -> None:
401         """Save the model, save the model information to the database, then
402            enter the home frame."""
403         model_name = self.save_model_name_entry.get()
404 
405         # Check if model name is empty
406         if model_name == '':
407             self.test_frame.model_status_label.configure(
408                                         text="Model name can not be blank",
409                                         fg='red'
410                                         )
411             return
412 
413         # Check if model name has already been taken
414         dataset = self.dataset_option_menu_var.get().replace(" ", "_")
415         sql = """
416         SELECT Name FROM Models WHERE Dataset=?
417         """
418         parameters = (dataset,)
419         self.cursor.execute(sql, parameters)
420         for saved_model_name in self.cursor.fetchall():
421             if saved_model_name[0] == model_name:
422                 self.test_frame.model_status_label.configure(
423                                                     text="Model name taken",
424                                                     fg='red'
425                                                     )
426                 return
427 
428         # Save model to random hex file name
429         file_location = f"school_project/saved-models/{uuid.uuid4().hex}.npz"
430         self.model.save_model_values(file_location=file_location)
431 
432         # Save the model information to the database
433         sql = """
434         INSERT INTO Models
435         (Dataset, File_Location, Hidden_Layers_Shape, Learning_Rate, Name,
    ↪ Train_Dataset_Size, Use_ReLu)
436         VALUES (?, ?, ?, ?, ?, ?, ?)
437         """
438         parameters = (
439                     dataset,
440                     file_location,
441                     self.hyper_parameter_frame.hidden_layers_shape_entry.get(),
442                     self.hyper_parameter_frame.learning_rate_scale.get(),
443                     model_name,
444                     self.hyper_parameter_frame.train_dataset_size_scale.get(),
445                     self.hyper_parameter_frame.use_relu_check_button_var.get()
446                     )
447         self.cursor.execute(sql, parameters)
448         self.connection.commit()
449 
450         self.enter_home_frame()
451 
452     def delete_loaded_model(self) -> None:
453         """Delete saved model file and model data from the database."""
454         dataset = self.dataset_option_menu_var.get().replace(" ", "_")
455         model_name = self.load_model_frame.model_option_menu_var.get()
456 
457         # Delete saved model
458         sql = f"""SELECT File_Location FROM Models WHERE Dataset=? AND Name=?"""
459         parameters = (dataset, model_name)
```

```
460             self.cursor.execute(sql, parameters)
461             os.remove(self.cursor.fetchone()[0])
462 
463             # Remove model data from database
464             sql = """DELETE FROM Models WHERE Dataset=? AND Name=?"""
465             parameters = (dataset, model_name)
466             self.cursor.execute(sql, parameters)
467             self.connection.commit()
468 
469             # Reload load model frame with new options
470             self.exit_load_model_frame()
471             self.enter_load_model_frame()
472 
473         def enter_home_frame(self) -> None:
474             """Unpack test frame and pack home frame."""
475             self.model = None  # Free up trained Model from memory
476             self.test_frame.pack_forget()
477             if self.saving_model:
478                 self.save_model_label.pack_forget()
479                 self.save_model_name_entry.delete(0, tk.END)  # Clear entry's text
480                 self.save_model_name_entry.pack_forget()
481                 self.save_model_button.pack_forget()
482             self.exit_button.pack_forget()
483             self.home_frame.pack()
484             summary_tracker.create_summary()  # BUG: Object summary seems to reduce
485                                               # memory leak greatly
486 
487     def main() -> None:
488         """Entrypoint of project."""
489         root = tk.Tk()
490         school_project_frame = SchoolProjectFrame(root=root, width=1280,
491                                                   height=835, bg='white')
492         school_project_frame.pack(side='top', fill='both', expand=True)
493         root.mainloop()
494 
495         # Stop model training when GUI closes
496         if school_project_frame.model != None:
497             school_project_frame.model.set_running(value=False)
498 
499     if __name__ == "__main__":
500         summary_tracker = tracker.SummaryTracker()  # Setup object tracker
501         main()
```