2023

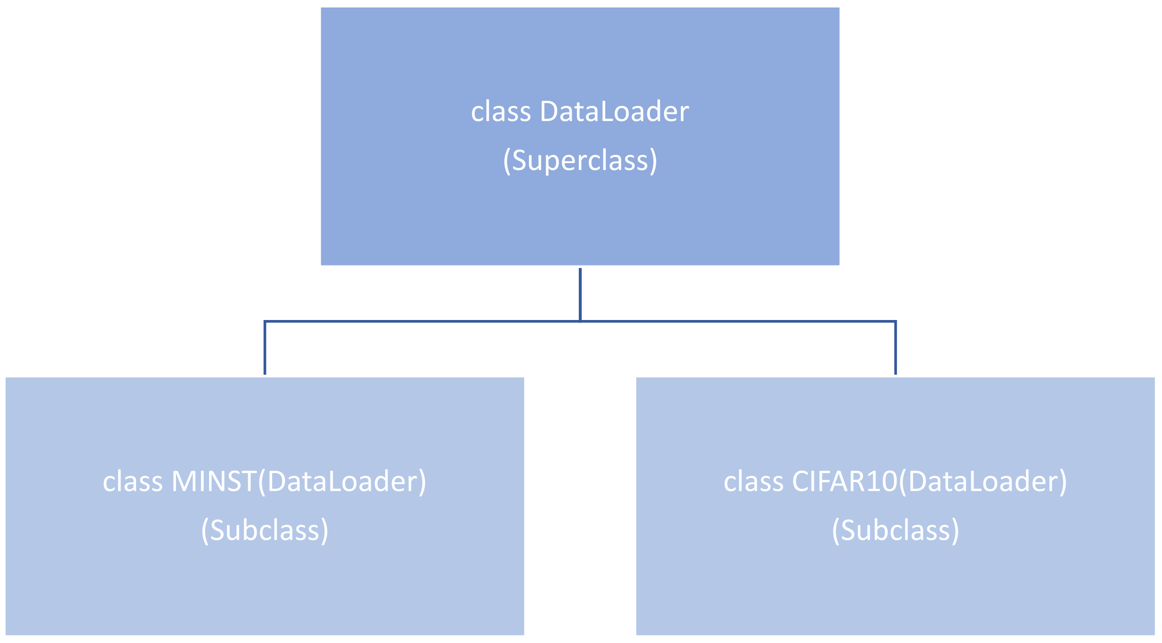
Group Project – GRA4152 OOP with Python

STUDENT ID'S: 1042033 & 1032519

Table of Contents

Dataloader

Class hierarchy



Common responsibilities

```
For each data set:
    If data set is MNIST:
        Load MNIST data set from Tensorflow.keras.datasets
        Reshape x_tr and x_te
    Else if data set is CIFAR10:
        Load CIFAR10 data set from Tensorflow.keras.datasets
```

Subclass responsibilities

```
    Preprocess data set:
        Convert x_tr and x_te to [0,1] by dividing by 255 and set type to Float32
        Convert y_tr and y_te to categorical with tf.keras.utils.to_categorical()
    Access x_tr, x_te, y_tr, and y_te.
    Load data set in batches ready to be used for training
```

Superclass responsibilities

Methods, overriding, and public interface

List of methods that needs to be created:

- __init__()
- x_tr() -> as a property method
- x_te() -> as a property method
- y_tr() -> as a property method
- x_te() -> as a property method
- loader()
- _preprocess_data()

All of these methods can be shared by both subclasses (except that they need their own constructor method), hence they can all be declared in the superclass DataLoader. The subclasses MNIST and CIFAR10 does not need to override any of the superclass' methods, they can all be inherited. Since both subclasses have a common need to preprocess their data (as shown in the previous section), we have chosen to declare this as a private method in the superclass. It is private, because it is not intended to be apart of the public interface of these classes. It simply needs to be called in the subclasses' constructors after loading the specific data sets. The loader method is not unique for the specific data sets and can, therefore, also be declared in the superclass and inherited. The public interface for the subclasses is, therefore, the accessor methods and the loader method, all inherited from the DataLoader superclass.

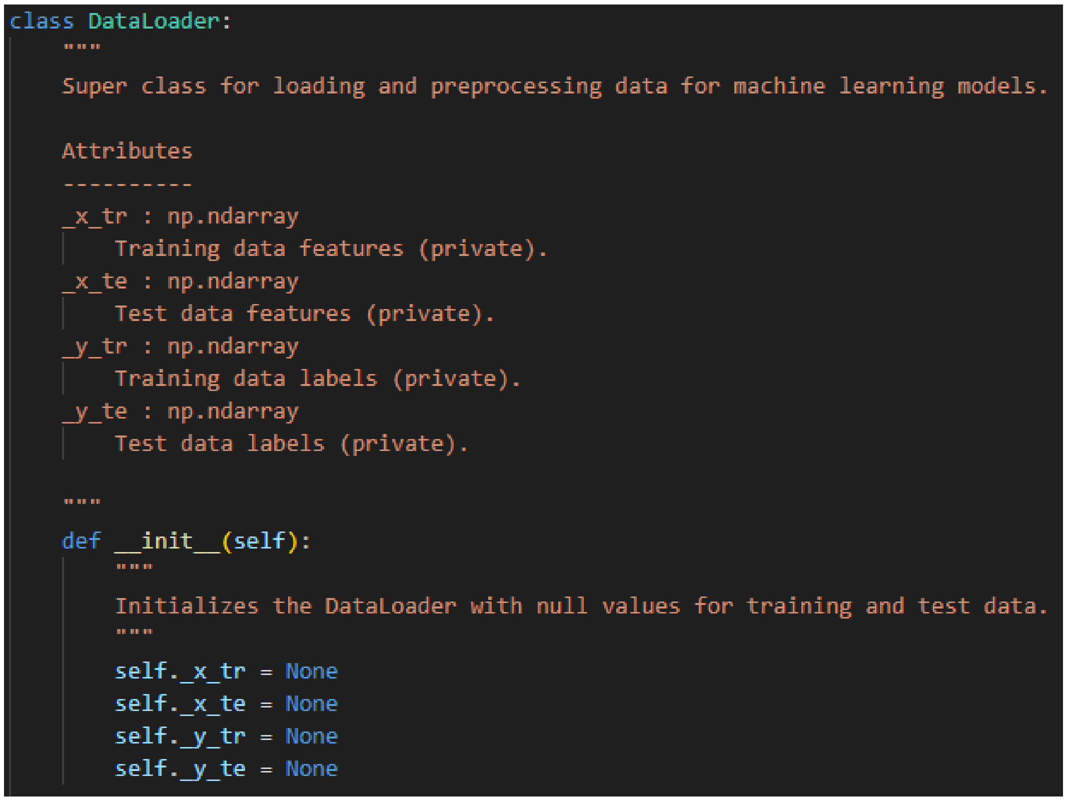
Instance variables

Since the subclasses are created to represent two data sets, the instance variables that are needed are the sets of data in these data sets. That includes a training set and a test set, each containing a set of pictures (x – the independent variable) and a corresponding set of labels (y – the dependent variable). Therefore, we have the instance variables _x_tr, _y_tr, _x_te, and _y_te. In line with the methodology of the book and this course, all the instance variables are private to uphold the principle of encapsulation. In the constructor of the DataLoader class, we have chosen to initialize these variables to "None". We do this so that the code editor does not complain that the methods in the superclass use undeclared variables.

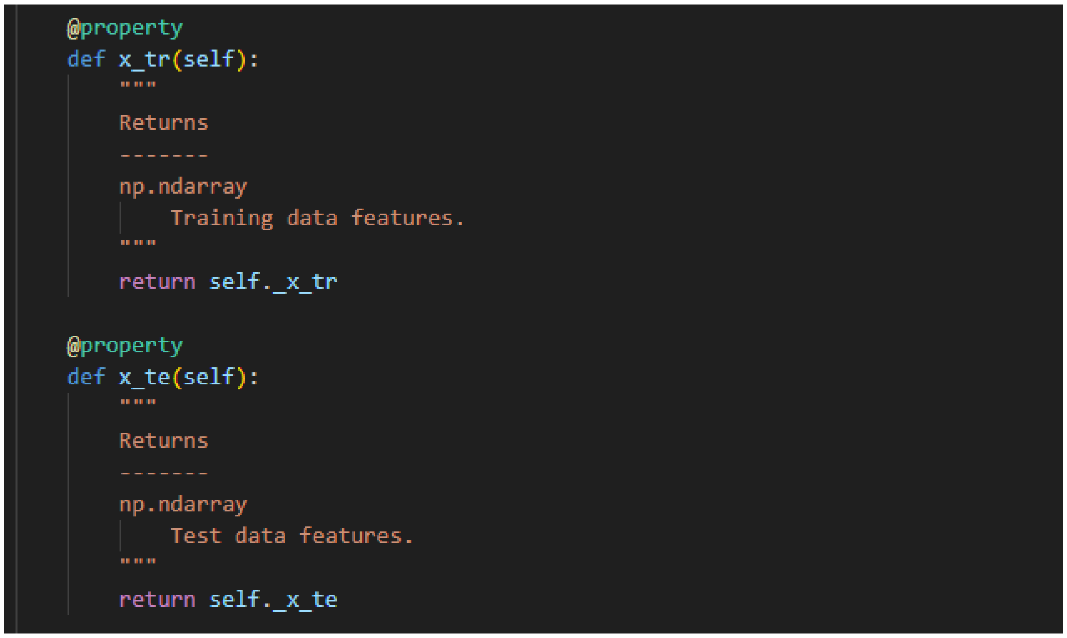
Implementation

We have chosen to use a docstring style that is more native to Python than the style that is used in the book (which is more native to Java). This is simply to stick to best practices and for our own benefit for writing Python code in the future. This is the style you, e.g., will find in the NumPy library.

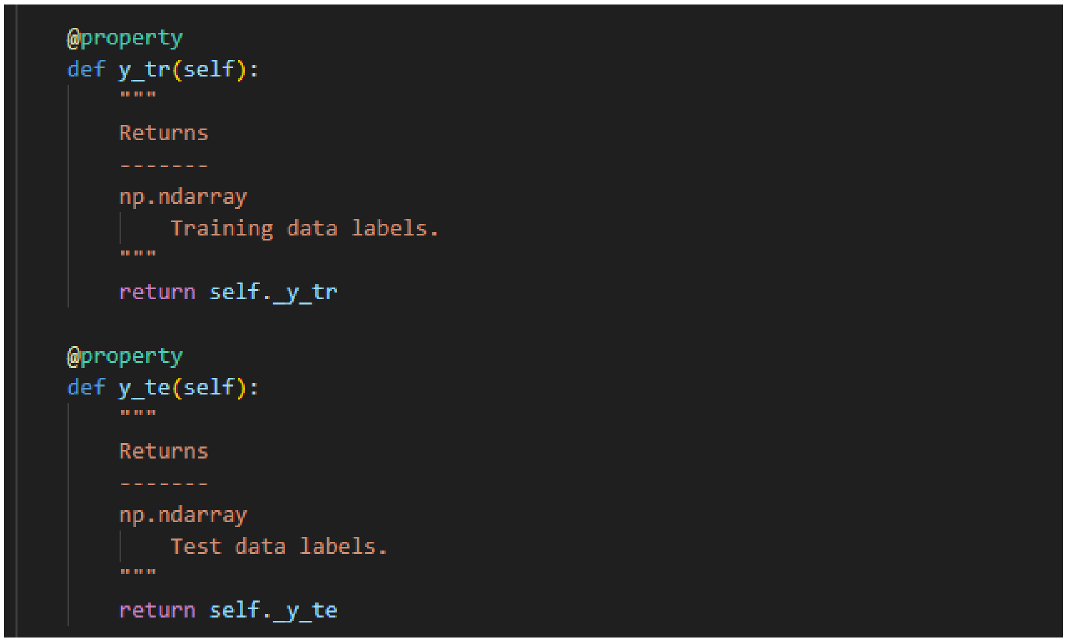
Starting with the DataLoader class:



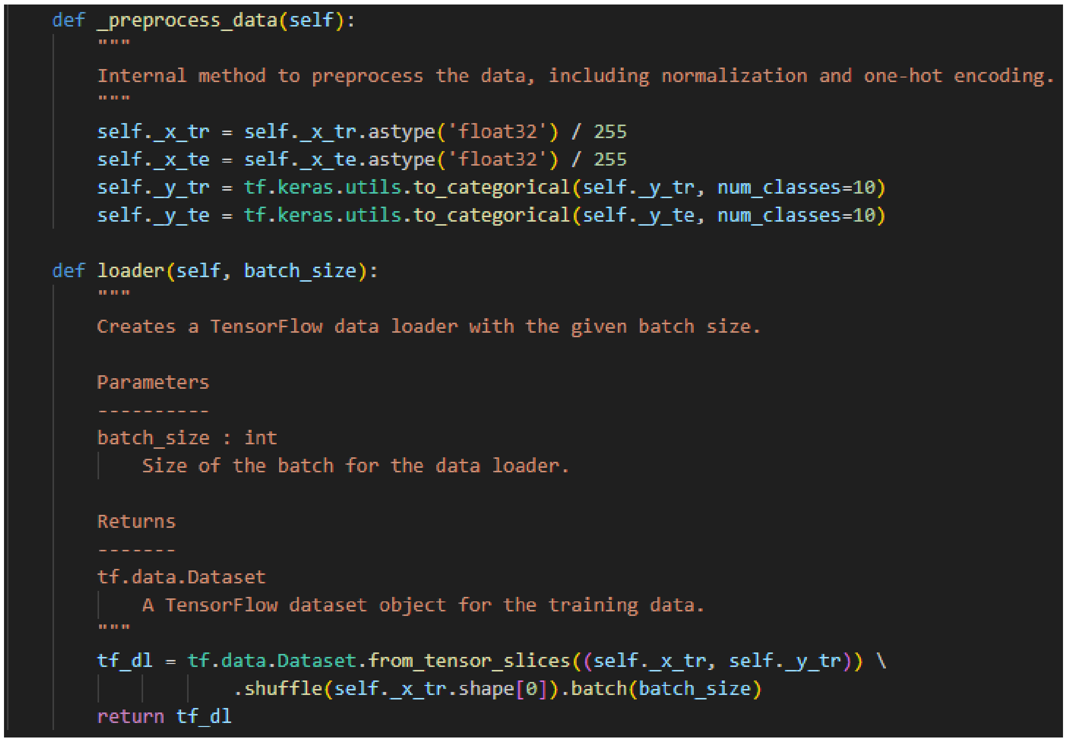
```python
class DataLoader:
    """
    Super class for loading and preprocessing data for machine learning models.

    Attributes
    ----------
    _x_tr : np.ndarray
        Training data features (private).
    _x_te : np.ndarray
        Test data features (private).
    _y_tr : np.ndarray
        Training data labels (private).
    _y_te : np.ndarray
        Test data labels (private).

    """
    def __init__(self):
        """
        Initializes the DataLoader with null values for training and test data.
        """
        self._x_tr = None
        self._x_te = None
        self._y_tr = None
        self._y_te = None
```



```python
    @property
    def x_tr(self):
        """
        Returns
        -------
        np.ndarray
            Training data features.
        """
        return self._x_tr

    @property
    def x_te(self):
        """
        Returns
        -------
        np.ndarray
            Test data features.
        """
        return self._x_te
```

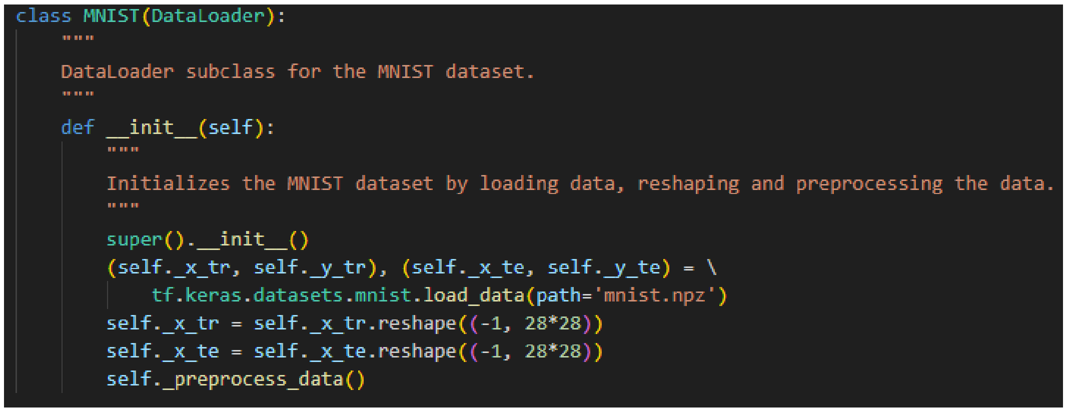


```python
    @property
    def y_tr(self):
        """
        Returns
        -------
        np.ndarray
            Training data labels.
        """
        return self._y_tr

    @property
    def y_te(self):
        """
        Returns
        -------
        np.ndarray
            Test data labels.
        """
        return self._y_te
```

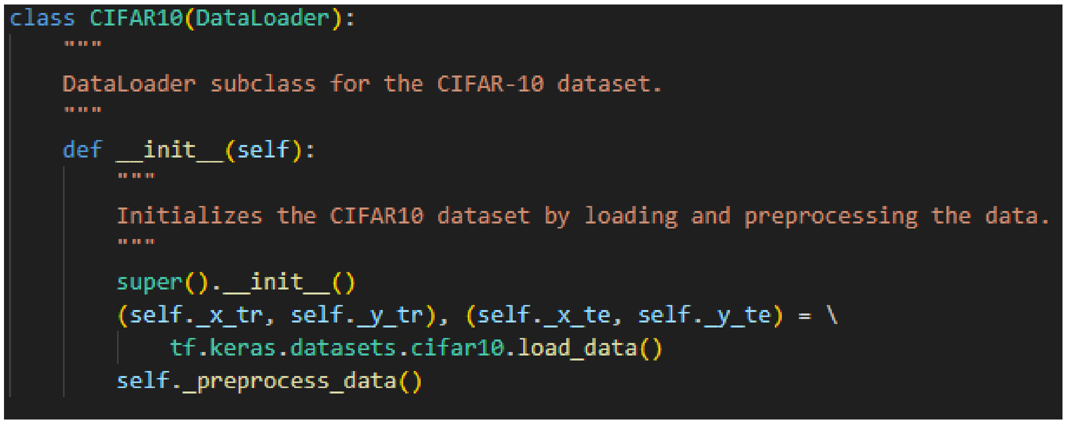


```python
    def _preprocess_data(self):
        """
        Internal method to preprocess the data, including normalization and one-hot encoding.
        """
        self._x_tr = self._x_tr.astype('float32') / 255
        self._x_te = self._x_te.astype('float32') / 255
        self._y_tr = tf.keras.utils.to_categorical(self._y_tr, num_classes=10)
        self._y_te = tf.keras.utils.to_categorical(self._y_te, num_classes=10)

    def loader(self, batch_size):
        """
        Creates a TensorFlow data loader with the given batch size.

        Parameters
        ----------
        batch_size : int
            Size of the batch for the data loader.

        Returns
        -------
        tf.data.Dataset
            A TensorFlow dataset object for the training data.
        """
        tf_dl = tf.data.Dataset.from_tensor_slices((self._x_tr, self._y_tr)) \
                    .shuffle(self._x_tr.shape[0]).batch(batch_size)
        return tf_dl
```

Then the MNIST class, which inherits from the DataLoader class. Its constructor will call the constructor of the DataLoader class, load the data from the Tensorflow.keras.datasets library, reshape the x-variables, and then call the preprocessing method inherited from the DataLoader class.

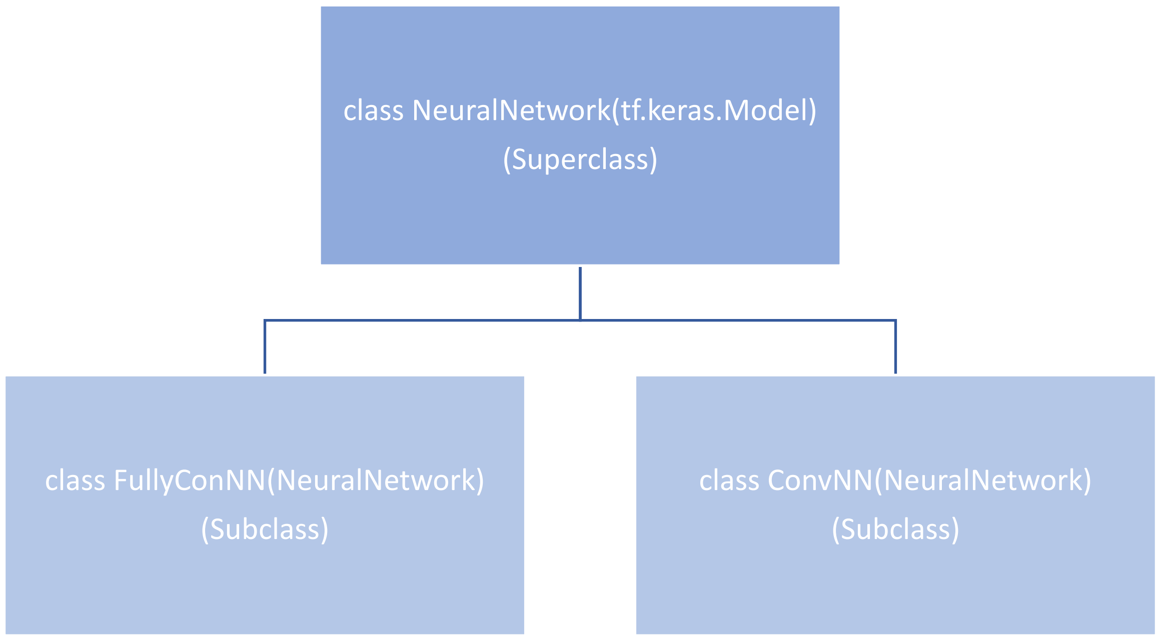

```python
class MNIST(DataLoader):
    """
    DataLoader subclass for the MNIST dataset.
    """
    def __init__(self):
        """
        Initializes the MNIST dataset by loading data, reshaping and preprocessing the data.
        """
        super().__init__()
        (self._x_tr, self._y_tr), (self._x_te, self._y_te) = \
            tf.keras.datasets.mnist.load_data(path='mnist.npz')
        self._x_tr = self._x_tr.reshape((-1, 28*28))
        self._x_te = self._x_te.reshape((-1, 28*28))
        self._preprocess_data()
```

The CIFAR10 class is almost identical to the MNIST class, except that it loads a different data set and do not need to reshape its x-variables.


```python
class CIFAR10(DataLoader):
    """
    DataLoader subclass for the CIFAR-10 dataset.
    """
    def __init__(self):
        """
        Initializes the CIFAR10 dataset by loading and preprocessing the data.
        """
        super().__init__()
        (self._x_tr, self._y_tr), (self._x_te, self._y_te) = \
            tf.keras.datasets.cifar10.load_data()
        self._preprocess_data()
```

Neural Network

Class hierarchy


class NeuralNetwork(tf.keras.Model)
(Superclass)
class FullyConNN(NeuralNetwork)
(Subclass)
class ConvNN(NeuralNetwork)
(Subclass)

Common responsibilities

```
For each neural network:
    If neural network is FullyConNN:
        Construct hidden layers with the parameters: number of neurons and
        input shape.

        Print the model specification for a fully connected neural network.

    Else if neural network is ConvNN:
        Construct hidden layers with the parameters: number of neurons,
        input shape, number of filters, kernel size, and number of strides.

        Print the model specifications for a convolutional neural network.
```

$\Big\}$ Subclass responsibilities

Modelling, training, and testing

Create classification layer.

Access the classification layer.

Forward pass method that takes the input data and computes the loss.

Test the model and return both (pseudo)probabilities and predicted labels

Train the model.

Superclass responsibilities

Methods, overriding, and public interface

The list of methods that needs to be created in the superclass:

- _classifier()
- _get_cls()
- call()
- test()
- train()

These methods are defined in the superclass as they are common methods for all subclasses. We saw this in the common responsibilities in the previous chapter. These methods are inherited by the subclasses. The call, test, and train methods are all part of the public interface. However, even though the call method is not private it is only intended to be used, in this scenario, in the train method. This is the only method that has a protected name, meaning it needs to be named "call" because of the Tensorflow.keras.Model library. The classifier method is a private method, only intended to declare the _cls instance variable. Hence, this method is not a part of the classes' public interface. This is also the case for the getter method _get_cls(), which is created to uphold the principles of encapsulation for the _cls instance variable. We will comment on the instance variables in the next chapter.

List of methods that need to be created in the subclasses:

- _hidden_layers()
- __repr__()

These methods are defined in the subclasses because they have different assignments depending on whether we use the fully connected neural network or the convolutional neural network. The first method is private and not part of the public interface to signal that it should not be called directly by the user. It is only intended to be called in the subclasses' constructors to declare the _hidden instance variable. We override Python's built-in __repr__ method to specify our model specifications and what type of neural network it is. Because of how we have used inheritance, there is not a clear-cut example of polymorphism in the public interface of FullyConNN and ConNN, except if you count the override of __repr__() as polymorphism. _hidden_layers() is unique to the subclasses, but has different sets of parameters and is only meant to be called from the constructor. Hence, it is not a great example of polymorphism either, if we define polymorphism as the ability of different classes to respond to the same function call in different, class-specific ways.

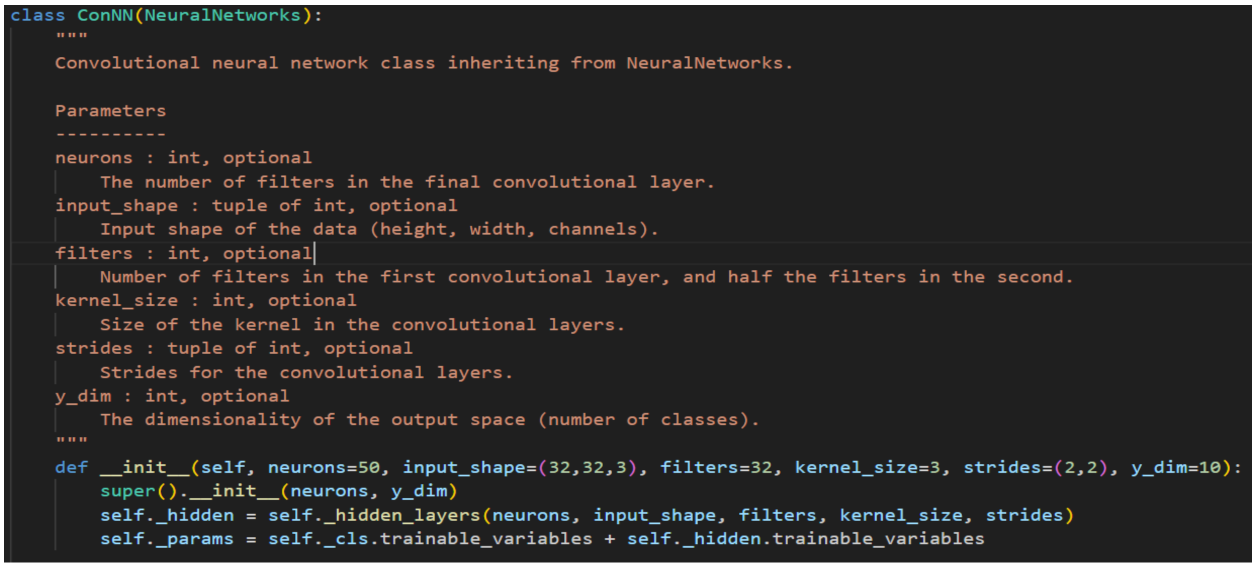
Instance variables

The instance variables in the neural network model are the variables that are in between the user input (data) and the output of the model (predictions). The instance variables are _hidden, _cls and _params. They are private to uphold the principles of encapsulation. We have chosen these to be instance variables as they are used by the methods declared in the superclass, and inherited by the subclasses, making it natural to have them as instance variables, and not as parameters. The _cls instance variable is common to both subclasses, hence, it is declared in the constructor of the superclass. However, since the ”trainable_variables” attribute of the _cls variable is needed to declare the _params variable, we needed to create the _get_cls() getter method mentioned in the previous chapter to uphold the principles of encapsulation. The _hidden and _params variables are declared to “None” in the constructor of the superclass for the same reasons mentioned for the DataLoader class, but these are unique to the subclasses.

Implementation

We refer to the above discussion on choice of docstring type, given in the implementation of the DataLoader class.

We do not think it is productive to provide screenshots of the over 240 lines of code related to our implementation of the NeuralNetworks, FullyConnNN, and ConNN classes. The biggest choices we have yet to comment on is how we have chosen to implement the parameters that are given in the exercise. We believe the best practice is to make these parameters keyword arguments in the constructor of the subclasses and pass these as parameters to the private methods we have declared, _hidden_layers() and _classifier(). Alternatively, we could have either hard-coded most of the parameters or made them all into instance variables. While the former generally is not a good practice in any programming setting, we believe passing them as parameters is a better choice than the latter. This is because all of them are only used by private methods that generally only serve to make the code in the constructors more concise. Also, by making them keyword arguments, we have the option to include these as input parameters in our training code if we wish to do so in the future. We believe this optionality is a good coding practice. As an illustration of these choices, here is the docstring and constructor of the convolutional neural network class:



```
class ConNN(NeuralNetworks):
    """
    Convolutional neural network class inheriting from NeuralNetworks.

    Parameters
    ----------
    neurons : int, optional
        The number of filters in the final convolutional layer.
    input_shape : tuple of int, optional
        Input shape of the data (height, width, channels).
    filters : int, optional
        Number of filters in the first convolutional layer, and half the filters in the second.
    kernel_size : int, optional
        Size of the kernel in the convolutional layers.
    strides : tuple of int, optional
        Strides for the convolutional layers.
    y_dim : int, optional
        The dimensionality of the output space (number of classes).
    """
    def __init__(self, neurons=50, input_shape=(32,32,3), filters=32, kernel_size=3, strides=(2,2), y_dim=10):
        super().__init__(neurons, y_dim)
        self._hidden = self._hidden_layers(neurons, input_shape, filters, kernel_size, strides)
        self._params = self._cls.trainable_variables + self._hidden.trainable_variables
```

Test

We use argparse to provide a description of the program with the arguments that the user can specify, as well as an epilog with two examples of terminal commands. We have also made assertions that checks the input of the user. If the input is not correctly specified an assertion error will be raised with a description of what is expected. The arguments that the user can specify are:

- nn_type (either “fully_con” or “conv”)
- epochs (integer number, default is 10)
- neurons (integer number, default is 10)
- batch_size (integer number, default is 256)
- dset (either “mnist” or “cifar10”)

The neural network models cannot be used interchangeably with the data sets that can be chosen. Hence, we have assertions that check whether the correct model is used with the correct data set. We also have assertions that make sure the arguments for epochs, neurons, and batch size are positive. We do not need to make sure these are integer values, as argparse will raise an error if they are not.

We have tested that the two following lines work:

- python3 train.py --dset cifar10 --nn_type conv --epochs 10
- python3 train.py --dset mnist --nn_type fully_con --epochs 10

where the user can expect around the following results:

- AUC of 0.7376 for the convolutional neural network.
- AUC of 0.9924 for the fully connected neural network.