Task 2: Developing an Inheritance Hierarchy of Neural Networks

We utilize Tensorflow Keras API to establish a class hierarchy for image classification.
At the base we have the ‘tf.keras.Model’ Extending this we have the superclass ‘NeuralNetwork’ as a blueprint for neural networks, encapsulating common behaviors and properties. that again has two subclasses for different types of neural networks:

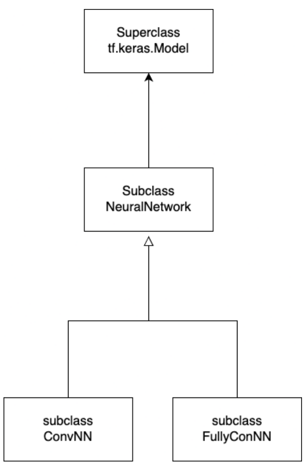
- Convolutional neural networks
- Fully connected neural networks

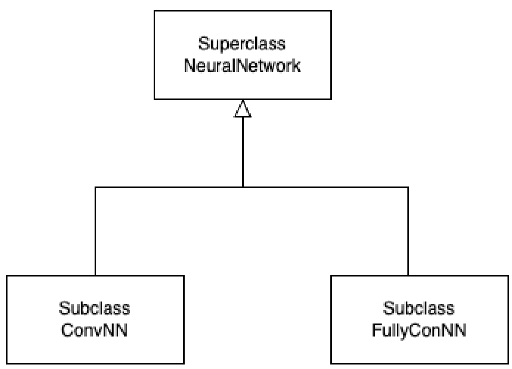
From ‘NeuralNetwork’ we derive the subclasses ‘ConvNN’ and ‘FullyConNN’. The difference between these classes is that in ‘ConvNN’, hidden layers are set up to create a stack of convolutional layers and flatten it to convert the 2D feature maps into a 1D vector. In ‘FullyConNN’ the input is expected to be a flat vector (28*28) here, all the neurons in one layer are connected to the subsequent layer.

The Hierarchy of the classes has tf.keras.Model on top as the Superclass, this class is builtin and is called through the TensorFlow Keras API. This model handels training, prediction, saving, loading and more. NeuralNetwork is a subclass of tf.keras.Model, where you can define common behaviors and properties that are shared across various types of NNs. NeuralNetwork inherits from Keras and therefore gets all the functionality of the keras model, as well as the functionality you add FullyConNN and ConvNN, which are specialized versions of NNs. They are subclasses of the NeuralNetwork class.
ConvNN overrides specific methods to set up the convolutional layers explained earlier. Similarly for FullyConNN, this overrides methods to set up densely connected layers.

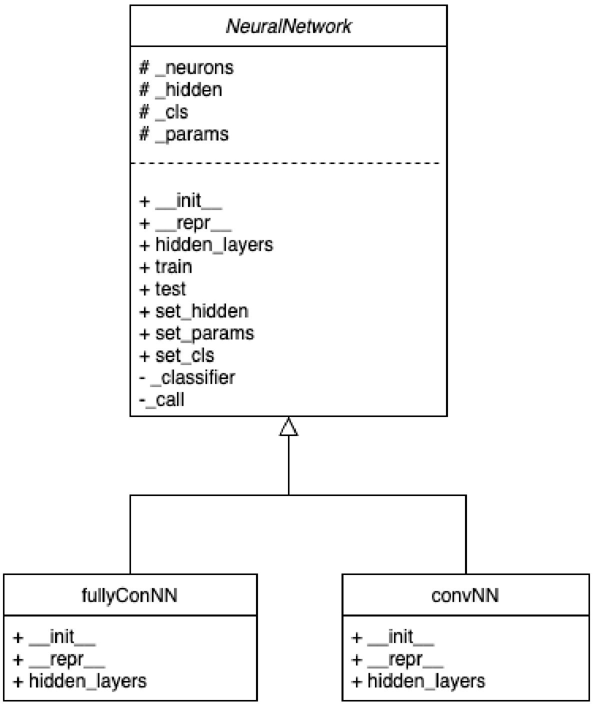
2.1) Inheritance Diagram and Public Interface:

The NeuralNetwork class contains constructor __init__ together with the methods:, __repr__, _hidden_layers, classifier, _call, train, test, set_hidden, set_cls and set_params.
FullyConNN contains the constructor __init__ and method _hidden_layers.
ConvNN contains the same as FullyConv: __init__ and hidden_layers.





The illustration to the left shows us how NeuralNetwork inherits from the tf.keras.Model, and the illustration to the right show us how we imagine our class hierarchy, where the NeuralNetwork is the Superclass of ConvNN and FullyConNN
Public interface:
instance variables
+ Public methods
- Private methods



2.2) Implementation of code:

See python-files.

2.3) Inheritance, Overriding, and Polymorphism:

For full explanation of the parameters taken into methods, and the return of the methods, see docstring. In the code you can write –doc <classname> for docstring of a specific class.

The NeuralNetwork class:
Constructor: In the constructor method we chose to not take in any variables. This is because we assume that the values of the instance variables can be changed and set later. We choose to have instance variables _neurons, _hidden, _cls and _params. All set to None, and then we make them later when we call the hidden_layers method for the subclasses.
hidden_layers: Because of the fact that the hidden_layers method is implemented differently in each subclass we make it abstract in the superclass. Each subclass overrides the method. This is to be able to use the same method call in our programme. The hidden_layers method is called in the subclasses constructors, so we could have chosen to make it a private method, but we chose to keep it public in case a user wants to change the input_shape, neurons and

other parameters, then they could call the method, and change, to fit other needs. In case the format of the pictures are different for example.
__repr__: The method __repr__() is also abstract here, because we do not want our objects that we use to be NeuralNetwork, we want them to be an instance of one of the two subclasses. Each subclass overwrites it to print a representation about what class they are.
_classifier: The _classifier is a private method, because it is only to be called in the hidden_layers method of the subclasses. when the _hidden_layers method is called.
The _classifier method creates an instance of the Sequential class based on the _neurons, y_dim and returns the Sequential object that is our classifier.
_call: The _call method is a private method to be used inside the train method. We do not need to call it outside the class. Therefore we chose to make it private. In our _call method we use the instance variable _hidden, hidden is different depending on what kind of instance it is. But the _call method can calculate the loss function for both instances.
train: The train method is just the same implementation as in the assignment text, we did not need to make any choices of how to implement it here.
test: The test method also uses the _hidden instance variable, so works for both FullyConNN and ConvNN. Here we also just followed the recipe in the text and did not have to make any assumptions.
Own methods: We have also added methods set_hidden(), set_cls, set_params to show how you can access and change the values of the private instance variables. We use these methods in the subclasses to change the initial values from None to the respective values we want, depending on the Subclass.

The FullyConNN class:
Constructor: In our constructor method here, the first thing we do is to change the value of our instance variable _hidden by calling the hidden_layers method. Then the _hidden is referring to a Sequential object. Then we set the value of the instance variable _params. This is to make the object fit to this class's specific needs.
__repr__: The method __repr__ hee is implemented to print out the class name for this class. So overrides the Super class implementation. Usually a __repr__ returns a string representation, but the assignment specifically asked us to print out the string. Therefore we used a print.
hidden_layer: The method hidden_layers is implemented to fit the FullyConNN class. with input_shape = 28*28 as default, and neurons = 50 as default. We then make the hidden object and change the instance variable _neurons to the same value as the neurons parameter in the method, in this way we make sure that the _neurons is equal to the neurons in the hidden_layers. In the end we actually call the method _classifier. When we do this, every time we call the hidden_layers method, the _classifier method also gets called so that we don't have any mismatch between the number of neurons in _hidden and _cls.
The class inherits the instance variables from the NeuralNetwork class, The methods __repr__ and hidden_layers are overridden by this subclass to meet the needs of this class. While it inherits the same implementation of the methods train, test, set_hidden, set_cls and set_params.

The ConvNN class:
Constructor: In this constructor method we also call the hidden_layers() method to set the values of the instance variables. Then we set the value of the instance variable _params.
__repr__: The method __repr__ hee is implemented to print out the class name for this class. Also here we used a print(), even though it is natural to return a string.

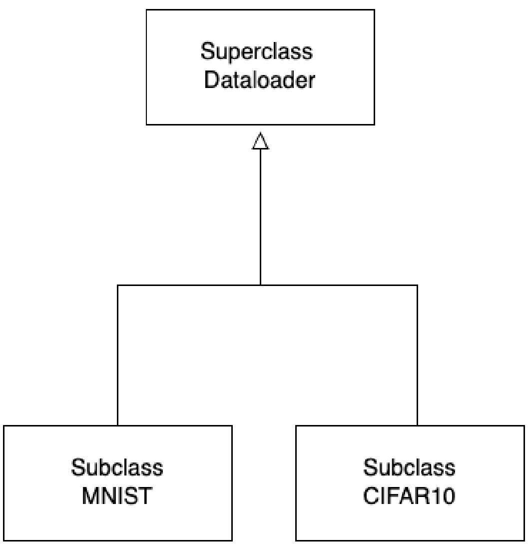
The _hidden_layers method is implemented differently here from the other subclass, made to fit this class needs. Here the default is input_shape = (32, 32, 3), neurons = 50, filters = 32, kernel_size = 3, strides = (2, 2).
The class inherits the instance variables from the NeuralNetwork class, The methods __repr__ and hidden_layers are overridden by this subclass to meets the needs of this class. While it inherits the same implementation of the methods train, test, set_hidden, set_cls and set_params.

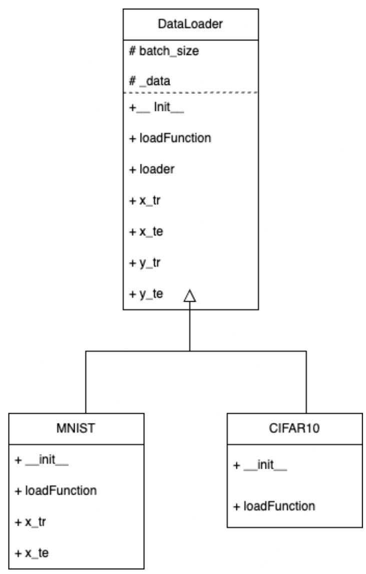
In the designed hierarchy, NeuralNetwork is a subclass of tf.keras.Model. FullyConNN and ConvNN are subclasses of NeuralNetwork.
While tf.keras.Model serves as the base class of our implementation we refer to NeuralNetwork as the superclass, because it is the direct parent class to FullyConNN and ConvNN. Overriding happens when FullyConNN and ConvNN provide their own implementation of a method inherited from the parent class. In this case we see it in the hidden_layers method. Even though NeuralNetwork might declare this method, the subclasses provide their own implementation by setting up layers according to their needs. Polymorphism makes it possible for us to use the same interface (call, train, test) on the different types of neural networks we have. For example, during model training, train will process data according to the specific architecture of the FullyConNN or ConvNN instance.

Task 3 DataLoader Class

3.1) Inheritance Diagram and Public Interface:

Instance variables
+ Public methods/ properties





In the DataLoader superclass we have methods like the constructor __init__, an abstract data loading method 'loadFunction' and a method 'loader'. The properties in this class are x_tr, x_te, y_tr and y_te which is training and test data.

In the MNIST subclass that inherits from DataLoader we have the constructor method and the LoadFunction, the properties are x_tr and x_te. We override the loadFunction method to be able to read the MNIST data set. It overrides the properties x_tr and x_te, to reshape the format. It inherits the loader method , and the properties y_tr and y_te.

In the CIFAR10 subclass the loadFunction method is implemented to read in the CIFAR10 dataset. while all of the properties are inherited from the superclass. The same for the loader method.
Both subclasses call the loadFunction in its constructor, in this way the _data instance variable changes value depending on what instance is created.
In summary, the public interface for these classes provides a consistent way to initialize the data loader, load the data, and access processed features and labels for both training and testing. The MNIST and CIFAR10 classes provide specific behaviors for these actions tailored to their respective datasets.

3.3) Inheritance and Overriding:

The superclass DataLoader is created to be generic enough to work with any dataset. It does not implement dataset-specific loading mechanisms, which is why “loadFunction” is abstract. MNIST and CIFAR10 subclasses inherit from DataLoader and gain access to its methods and properties, meaning they can use the loader method.

The MNIST and CIFAR10 classes override the loadFunction method of DataLoader. This means that both MNIST and CIFAR10 can provide their own implementations of loadFunction. In the MNIST class we also override the x_tr and x_te to reshape the data specifically for the MNIST dataset.

Task 4 Training Your Neural Network

We made a train/test-program where we assert that the input parameters are right. And then create correct objects depending on if the input is ConvNN or FullyConNN. And the same for CIFAR10 and MNIST. We then check that the model can calculate on the data. We assume that the different neural networks subclasses are specified to work on its own data set. And that is why we set default values that match the corresponding data set in our code.
If everything is okay, we just followed the recipe in the task for how to calculate the AUC score.
If the user wants to see the docstring for a specific class the user needs to write python3 train.py –doc <class name>.
To get help, user writes -h or –help.