GRA4152 - OOP Project

November 2023

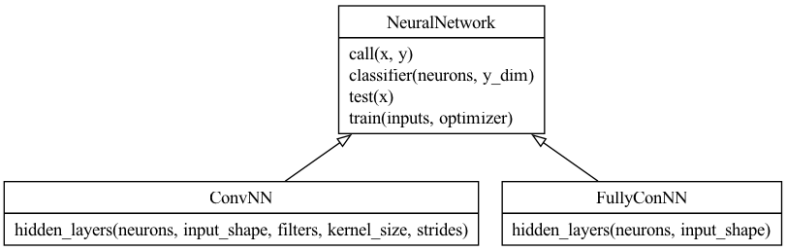

Figure 1: Inheritance diagram for Neural Networks

1 Neural Networks

NeuralNetwork. Inheritance diagram is provided in Figure 1. The constructor of NeuralNetwork requires a user to specify the number of neurons in the last hidden layer (**neurons**) and the number of classes the model is intended to identify (**y_dim**). This is done intentionally, because in this project we work only with classification problems. Hence, it makes sense to design our superclass in such a way so that it is expected to be used for classification tasks.

NeuralNetwork class has only one instance variable which is protected and is called _cls. It is supposed to contain the classification layer of the neural network and this variable is defined by calling a public method classifier(**neurons**, **y _dim**) within the constructor. The classifier method is intentionally designed as a public method so as to allow users to change the last layer of the neural network depending on their needs. We decided to make only one of the variables to be instance variable in order to minimize unproductive memory usage.

FullyConNN. The constructor method of this subclass expects three input parameters. Apart from **neurons** and **y_dim**, it also requires **input_shape** which captures the size of a vector of features for each observation. We expect that this model will be used only for MNIST dataset as CIFAR10 dataset contains images in RGB format. That is the reason why the default value of **input_shape** is (28 * 28) as MNIST dataset contains images with resolution 28 * 28. As it can be seen, **input_shape** is one-dimensional, so a matrix with pixels for each image in MNIST dataset will be flattened into a vector. It is also important to note, that the interpretations of **neurons** is different in this subclass. Here it is extended also to be the number of neurons in dense layers.

Furthermore, FullyConNN has three protected instance variables: 1) _cls; 2) _hidden, 3) _params. _cls is defined after calling the constructor method of NeuralNetwork superclass, _hidden contains the hidden layers and is defined by calling the public method hidden_layers(**neurons**, **input_shape**), and _params contains trainable variables of hidden and classification layers.

ConvNN. The constructor method of this subclass expects six input parameters. Apart from **neurons** and **y_dim**, which were described in the design of NeuralNetwork superclass, it also requires: **input_shape** - tuple, the dimensions of input observations (since convolutional neural networks are often used when dealing with RGB images, in this project we expect that this subclass will be used for CIFAR10 dataset, which is why the default dimensions here are (32, 32, 3) which corresponds to images from CIFAR10); **filters** - int, the number of filters to be used in the first convolutional layer, and half the number of filters for the second convolutional layer (by default equals to 32); **kernel_size** - int, kernel size to be used in all three convolutional layers (by default is equal to 3); **strides** - tuple, strides to be used in the first two convolutional layers (by default is (2, 2)).

Similarly to FullyConNN, ConvNN has three protected instance variables(_cls; _hidden, _params) that are created in a similar way to FullyConNN.

Public interface of NeuralNetwork

classifier(neurons, y_dim) This method sets the instance variable _cls.
1) **neurons**: int, the number of neurons in the last layer before the classification layer;
2) **y_dim**: int, the number of classes the model is intended to identify.

call(x, y) Given the features and true class labels, uses the neural network to generate (pseudo)probabilities based on observed features and calculates a loss function given true labels and (pseudo)probabilities.
1) **x**: an array of features (e.g. images);
2) **y**: an array of true class labels, one-hot encoded;
Its returns an array which contains results of application of the loss function to the true labels and (pseudo)probabilities.

train(inputs, optimizer) Updates model parameters using the provided optimizer.
1) **inputs**: a tuple (x, y) which contains an array of features and an array of labels;
2) **optimizer**: optimizer object from keras.
Its returns an array which contains results of application of the loss function to the true labels and (pseudo)probabilities.

test(x) Generates (pseudo)probabilities for provided observations without true labels.
1) **x**: array, features of observations (e.g. images) for which (pseudo)probabilities should be calculated.
It returns an array of predicted (pseudo)probabilities. It is important to mention that we are not converting (pseudo)probabilities into class labels, because in this assignment we are asked only for AUC metric.

Public interface of FullyConNN

hidden_layers(neurons, input_shape) Defines two hidden layers for a fully connected neural network. A setter method for instance variable _hidden.
1) **neurons**: int, the number of neurons to be used in both hidden layers;
2) **input_shape**: the size of a vector of features for each observation.

Public interface of ConvNN

hidden_layers(neurons, input_shape, filters, kernel_size, strides) Defines three convolutional layers in two dimensions and flattens the output of the last one. A setter method for instance variable _hidden.
neurons: int, the number of filters to be used in the last convolutional layer;
input_shape: a tuple, the dimensions of input observations;
filters: int, the number of filters to be used in the first convolutional layer, and half the number of filters for the second convolutional layer;
kernel_size: int, kernel size to be used in all three convolutional layers;
strides: tuple, strides to be used in the first two convolutional layers.

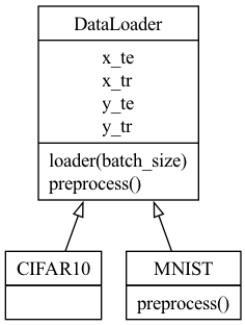

Figure 2: Inheritance diagram for Dataloaders

Inheritance. We are using inheritance when we develop our superclass NeuralNetwork since it inherits from *tensorflow.keras.Model*. The following four methods are inherited by FullyConNN and ConvNN: classifier, call, train, test. We are also inheriting _cls instance variable in subclasses.

Overriding. We are overriding __repr__method in the superclass and in all subclasses. The constructor method is also overridden in both subclasses, since the parameters of the hidden_layers method are different across our subclasses.

Polymorphism. Inside the call method of NeuralNetwork superclass we are using the instance variable _hidden which is not defined inside the superclass, but is defined inside our subclasses. This enables us to use call method on all subclasses of NeuralNetwork. Since hidden layers are different, the behaviour will be different, but all objects will be handled in a uniform way. The same is true about test method. Similarly, in the train method of NeuralNetwork superclass we are using the instance variable _params, which is not defined inside the superclass, but is defined inside our subclasses. Our NeuralNetwork superclass should be treated as an abstract class, even though we are not specifying any abstract methods.

2 Dataloaders

Figure 2 shows our inheritance diagram. Again, we intentionally put most of the public methods and accessor methods in our superclass so as not to repeat ourselves in the subclasses. The constructor method for DataLoader is intentionally left blank so that it only constructs an object. It was used for testing purposes. DataLoader superclass should be treated as an abstract class since all its methods require some instance variables to work, but none of them is defined in the constructor of the superclass. All of them should be defined in a subclass constructor.

In both MNIST and CIFAR10 the first line is intended for loading the corresponding data and assigning (_x_tr, _y_tr), (_x_te, and _y_te) to contain (training features and labels), and (testing features, and labels). In the constructors we are testing whether our loaded arrays have the expected shapes using assertions. And we are calling preprocess method in both constructors. It would be safer to make preprocess a protected method, but since in this project we have quite specific objectives for using these classes it doesn't affect our results.

Public interface of DataLoader

Accessor methods. The following accessor methods are developed using property decorator for respective instance variables: **x_tr**, **x_te**, **y_tr**, **y_te**.

preprocess() Applies preprocessing trasformations on dataset. The transformations include image scaling and one-hot-encoding of labels.

loader(batch_size) Enables a user to load dataset in batches.
batch_size: int, the number of images in one batch.
It returns a tensorflow data loader object.

Public interface for MNIST

preprocess() Applies preprocessing trasformations for MNIST dataset. Adds images reshaping from 2D to 1D to the preprocessing steps from DataLoader.

Public interface for CIFAR10

All instance variables and public methods are the same as in the superclass DataLoader.

Inheritance, Overriding, Polymorphism

We are inheriting the accessor methods from the superclass and public method loader in both subclasses. In CIFAR10 subclass we are also inheriting preprocess method.

We are overriding preprocess method in MNIST method to include reshaping of images from matrices with dimensions (28, 28) to vectors of (784) elements.

We provided the opportunity for using polymorphism by placing most of our public methods and accessor methods in the superclass. We have used polymorphism for testing our subclasses in the tester function that is provided at the end of the file dataloader.py.

3 Training Neural Networks

By using argparse, we've ensured that the code will run only if a user provides the program with either "mnist" or "cifar10" for our dset argument, and either "fully_con" or "conv" for our nn_type argument. We've also ensured that epochs, neurons, and batch_size are all of type integer and have appropriate default values. We've used assertions for testing that epochs and neurons provided are greater than zero. Additionally, we raise ValueError if batch_size is less than 1 or higher than the number of images in the specified dataset.

We are using polymorphism: 1) to create tf data loader object for the provided dataset; 2) in the training routine, where we are calling train method on model variable, where model can be an instance of either FullyConNN or ConvNN; 3) for calculating (pseudo)probabilities for the test subset.

Overall, we were able to achieve ≈ 0.9920 AUC score for MNIST dataset using fully connected neural network, and ≈ 0.7336 AUC score for CIFAR10 dataset using convolutional neural network.