Inheritance Hierarchy of Neural Networks
Object Oriented Programming
June 2024

Neural Networks

Modern artificial intelligence uses, among other things, neural networks. There are several types of neural network architectures, but fully connected and convolutional neural networks are some of the most popular architectures. Convolutional neural networks are widely used in image analysis, whereas fully connected neural networks are used in a wide range of applications where data can be represented as a vector, including black-and-white images. There are several `Python` libraries that are designed for the development and training of neural networks, including fully implemented layers, data loaders for efficient memory consumption and data preprocessing, optimizers for training neural networks, etc. A popular library of neural networks is `Tensorflow`.

1 Neural Network Layers

A neural network is composed of at least a layer of neurons. If multiple layers, which are commonly referred to as hidden layers, are stacked together, we call such an architecture a deep neural network. Using `Tensorflow` is very easy to build fully connected and convolutional layers. See the following code where we specify some fully connected hidden layers as well as convolutional hidden layers:

```
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential

# fully connected
hidden = Sequential([layers.InputLayer(input_shape=input_shape),
                     layers.Dense(neurons),
                     layers.Dense(neurons)])

# Or convolutional
hidden = Sequential([layers.InputLayer(input_shape=input_shape),
layers.Conv2D(filters=filters,kernel_size=kernel_size,strides=strides),
layers.Conv2D(filters=2*filters,kernel_size=kernel_size,strides=strides
    ),
layers.Conv2D(filters=neurons,kernel_size=kernel_size,strides=(5,5)),
layers.Flatten()])
```

For this exerccise, you can think of `hidden` as an instance of the `Sequential` class. This instance can be called by passing a data set `x` as an argument, i.e., `out = hidden(x)`.

2 Developing an Inheritance Hierarchy of Neural Networks

In this exercise, you will develop an inheritance hierarchy of neural networks for image classification using fully connected and convolutional neural networks. To that end you need to develop the following classes: `NeuralNetwork`, `FullyConNN`, and `ConvNN`. Make sure to explain all the choices you make, e.g. instance variables, decorators, input parameters, etc. You can also add more methods if needed.

2.1) Inheritance Diagram and Public Interface: Start by creating an inheritance diagram, showing the hierarchy that you select for your code. Furthermore, describe the public interface of the three classes.

2.2) Methods: You should at least develop the following methods (see the Implementation section for details):

- `__init__`
- `__repr__`
- `hidden_layers`
- `classifier`
- `call`
- `train`
- `test`

Depending on how you use inheritance, overriding, and polymorphism, it is not necessary to code all methods in all classes. Note: make sure that your superclass inherits from `tf.keras.Model`.

2.3) Inheritance, Overriding, and Polymorphism: Explain how you used inheritance, overriding, and polymorphism in the `NeuralNetwork`, `FullyConNN`, and `ConvNN` classes.

3 DataLoader Class

For this exercise, you will use two popular data sets: `MNIST` (link) and `CIFAR10` (link). Even though it is easy to get these data sets from `Keras` (see the examples in the link), `MNIST` and `CIFAR10` require some preprocessing before they can be loaded into `tf.data.Dataset.from_tensor_slices()`, which is the in-build `Tensorflow` data loader that is used for model training. Therefore, you will develop the following classes: `DataLoader`, `MNIST`, `CIFAR10`. In the following explanations, I am intentionally not specifying if variables need to be instance variables or if they need to be accessed from other methods, as this is a choice that you have to make depending on the approach you take to develop your classes.

3.1) Inheritance Diagram and Public Interface: Start by creating an inheritance diagram, showing the hierarchy that you select for your code. Furthermore, describe the public interface of the three classes.

3.2 Methods:

- Use decorators to access `x_tr`, `x_te`, `y_tr`, and `y_te`. You are free to develop more accesor methods if needed.
- `x_tr` and `x_te` in both data sets need to be scaled between [0 1] by dividing by 255. Furthermore, `x_tr` and `x_te` must be numbers `float32`.
- `y_tr` in both data sets needs to be transformed with the function `tf.keras.utils.to_categorical()` to convert it into a one-hot-encoder.
- In addition to the above common preprocessing steps, the `MNIST` data set must be reshaped from $(60000, 28, 28)$ to $(60000, 784)$ for the training data set and from $(10000, 28, 28)$ to $(10000, 784)$ for the test data set.
- Add the following method in your superclass:

```
###
# @x: numpy array
# @y: numpy array
# @return: tf data loader object
def loder(self, batch_size):
    tf_dl = tf.data.Dataset.from_tensor_slices((x_tr,y_tr)).shuffle
   (x_tr.shape[0]).batch(batch_size)
    return tf_dl
```

to load a tuple, i.e. `(x_tr,y_tr)`, into the in-build `Tensorflow` data loader.

3.3) Inheritance and Overriding: Explain how you used inheritance and overriding in the `DataLoader`, `MNIST`, and `CIFAR10` classes.

4 Training your Neural Networks

Create a file called `train.py` and use `argparse` to define the following variables that you need to train your neural networks:

- `nn_type`: Specifies whether to use `FullyConNN` or `ConvNN`
- `epochs`: Number of epochs to train your neural network
- `neurons` Number of neurons to use in your neural networks
- `batch_size` Size of the batch used in the optimization routine
- `dset` Specifies whether to use `MNIST` or `CIFAR10`

Use 10, 50, and 256 as default values for `epochs`, `neurons`, and `batch_size`, respectively. You must use `assertions` and/or `exceptions` to ensure that the inputs are as expected.

Use the following optimizer

```
optimizer = tf.keras.optimizers.Adam(learning_rate=5e-4)
```

in the following training routine

```
step=0
while step < args.epochs:
    for i, data_batch in enumerate(tr_data):
        losses = model.train(data_batch, optimizer)
    step+=1
```

where model is an instance of the class `FullyConNN` or `ConvNN`, and `tr_data` is the output of the method `loader` that you obtain from an instance of the class `MNIST` or `CIFAR10`.

The area under the ROC curve, or `AUC`, is a common measure of classification performance. You can calculate the `AUC` for your classifier using

```
from sklearn.metrics import roc_auc_score

auc = roc_auc_score(data_loader.yte, pi_hat)
print ('final auc %0.4f' % (auc))
```

where pi_hat are the (pseudo)probabilities that you can obtain from the test method in your neural network object (see the Implementation section for details).

Your code must run by using the following commands from the terminal

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

and, at least, the AUC should be printed at the end of your file train.py.

Implementation

Below you find detailed explanations for all methods that you need to implement in the neural network classes. Note that intentionally I am not specifying if variables need to be instance variables or if they need to be accessed from other methods, as this is a choice that you have to make depending on the approach you take to develop your classes.

def __init__: You must define all the variables you need for the rest of the methods. The variables that you define in the constructor must be available for the rest of the methods without needing to be passed as arguments. In addition, you can invoke method calls that need to be initialized. In the constructor for FullyConNN and ConvNN you must include this line:

```
params = cls.trainable_variables + hidden.trainable_variables
```

to specify all trainable parameters in the neural network that will be optimized in the train method.

def __repr__: Prints a short description of the class.

def hidde_layers: Defines one of the layers in Section 1. You should use the following input parameters for fully connected layers

- input_shape = 28*28
- neurons = 50

and these parameters for convolutional layers

- input_shape=(32,32,3)
- neurons = 50
- filters=32
- kernel_size=3
- strides=(2,2)

def classifier: Adds the following classification layer:

```
cls = Sequential([layers.InputLayer(input_shape=neurons),
                  layers.Dense(y_dim, activation='softmax')])
```

where neurons=50 and y_dim=10.

def call: Takes x,y as input parameters. Then you use the instance hidden and pass the data set x as an argument. Assign a name to this output, for example, out. Then, pass out as an argument to the instance cls and assign a name to that output, e.g. out again. Finally, use the following line

```
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, out))
```

to calculate a cross entropy loss function.

`def test`: Takes `x` as input parameter. Then you use the instance `hidden` and pass the data set `x` as an argument. Assign a name to this output, for example, `out`. Then, pass `out` as an argument to the instance `cls` and assign a name to that output, e.g. `out` again. The output of the `cls` layer can be interpreted as (pseudo)probabilities, which sum up to 1. The following line

```
y_hat  = tf.math.argmax(out,1)
```

converts (pseudo)probabilities into a class label corresponding to the category with highest probability.

`def train`: Takes `inputs` and `optimizer` as input parameters, where `inputs=(x,y)` and `optimizer` is defined in *4 Training your Neural Networks*. Then you can paste the following code in your method:

```
with tf.GradientTape() as tape:
    loss = self.call(inputs)
gradients = tape.gradient(loss, params)
optimizer.apply_gradients(zip(gradients, params))

return loss
```

Bonus Tips

If you are struggling to train your neural network, start by debugging the neural network classes. If properly coded, you should be able to create objects and call some of their methods before training. Although it is not a good practice, you can try to access private variables of the objects to see if they are as they are supposed to be. Recall that your superclass must inherit from `tf.keras.Model`.

After you are sure that your neural networks classes are correct, start debugging the data loaders classes. Follow the same approach, that is, define an object and check that the methods are working as expected.

At this point, you are sure that the neural network and data loader classes are correct. Then you can focus on the last step, which is training your neural network. You can think of this last step as a *tester* file to verify that you did everything correctly. The `train.py` file is straightforward and does not require many lines of code. You need to i) load the data set, initialize your model, set the optimizer, get the `Tensorflow` loader, train the model and calculate the `AUC`. You should obtain higher `AUC` values for `MNIST` than for the `CIFAR10` data set.