Project 3: Neural Networks

Deadline: 2018-11-29

Abstract

The goal of this project is to help you better understand Artificial Neural Networks and learn how to apply them to solve machine learning tasks.

Introduction

The project has two parts. In the first part, you need to implement artificial neural networks (ANNs) and some of the methods covered in class. Then in the second part, you will use your implementation to train an ANN to recognize handwritten digits.

Part 1: Neural Network

In this part, you will code a Python class for representing an ANN and its methods for inference, training and evaluation. In your implementation, you will use the `numpy` package, which is a library supporting large multi-dimensional arrays (in particular matrices) and providing fast operations on them. To learn more about this library, you can check `http://www.numpy.org`.

The file named `networks.py`, which we provide you, contains the partial definition of the class `Network` that will represent an ANN. Note that we assume that the ANN is a multi-layer perceptron for simplicity. To help you in this project, we provide you the constructor of `Network`, which we recall below:

```
def __init__(self, sizes):
    self.num_layers = len(sizes)
    self.sizes = sizes
    self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
    self.weights = [np.random.randn(y, x)
                    for x, y in zip(sizes[:-1], sizes[1:])]
```

where the parameter `sizes` is a list containing the number of neurons in the respective layers of the network. For example, a 3-layer network with 10 neurons in the first layer, 5 neurons in the second layer, and 1 neuron in the last layer can be constructed with the following call: `Network([10, 5, 1])`.

Recall that in Python, the parameter `self` is not used during the invocation of a method. It provides a reference to the instance on which a method is called. Moreover, if `a` and `b` are two iterables (e.g., lists), `zip(a, b)` returns a sequence of pairs, where the first pair contains the first elements of `a` and `b`, the second pair the second elements of `a` and `b` and so on. The number of returned pairs depends on the length of the shorter iterable.

In this part, you will start by implementing the following methods:

- `inference(self, x)` which returns the output of the ANN for input `x`, which is assumed to be an 1-D array.
- `training(self, trainData, T, n, alpha)` which trains the ANN with training dataset `trainData` using stochastic gradient descent with mini-batches of size `n`, a total number of iteration `T` and learning rate `alpha`. This method calls the next one `updateWeights` at each iteration.
- `updateWeights(self, batch, alpha)`, which updates the weights and biases of the ANN using the gradient (computed by the next method `backprop`) with mini-batch `batch` and learning rate `alpha`. The mini-batch is represented as a list of pair `(x, y)`.
- `backprop(self, x, y)` which returns a tuple `(nablaW, nablaB)` representing the gradient of the empirical risk for an instance `x, y`. The gradient `nablaW` and `nablaB` are layer-by-layer lists of arrays, following the same structure as `self.weights` and `self.biases`.
- `evaluate(self, data)` which returns the number of correct predictions of the current ANN on the (e.g., validation or test) dataset `data`, which is a list of pair `(x, y)`. The prediction of the ANN is taken as the argmax of its output.

For the training part, you will first assume that the loss is the squared error and all the activation functions are sigmoids. To help you, we provide you the following functions:

- `dSquaredLoss(a, y)` returns the vector of partial derivatives of the squared loss with respect to the output activations for prediction `a` with correct label y, i.e. $a - y$.

- `sigmoid(z)` implements the sigmoid function.
- `dSigmoid(z)` implements the derivative of the sigmoid function.

When you have finished implementing the previous methods of `Network`, you should extend it so that an ANN could be trained with different activation functions in each layer, L2 regularization and/or early stopping:

- To allow different activation functions in each layer, a new parameter `activationFcns`, which is a layer-by-layer list of activation functions, can be added to the constructor. For instance, the previous 3-layer ANN would be now constructed by the following call:
  `Network([10, 5, 1], [None, sigmoid, sigmoid])`.
  Note that as the input layer does not use any activation function, we provide the value `None`. You will make sure that the correct activation functions are used for inference and training.
- To enable L2 regularization, you can simply add a new parameter `lmbda` (note `lambda` is a reserved word in Python), which corresponds to the λ hyperparameter of the regularization term, to the following methods `training` and `updateWeights`, and you should modify `updateWeights` accordingly to add the regularization term.
- To implement early-stopping, you can modify the method `training` and add a new parameter `validationData` to it. This new parameter represents a validation dataset that is used to track the performance of the ANN in order to decide when to stop the training. Note parameter T is still used, therefore the training can also be stopped if the number of iterations reaches T.

All new parameters should be added as last parameters of a method.

Coding Tasks

In `networks.py`, the interface for the previously mentioned functions are given. You need to provide the code for the methods with the missing implementation.

Important:

- You should implement the methods **without changing the names and signatures**.
- Frameworks including but not limited to `pytorch`, `tensorFlow`, `theano` are **not allowed in this part**.

Figure 1: Sample images from MNIST test dataset.

Part 2: MNIST

The MNIST (Modified National Institute of Standards and Technology database) database is a data set of images of handwritten digits. The images have been size-normalized and centered in a fixed-size image, see Figure 1.

The goal of this part is to train an ANN to learn to recognize these handwritten digits. This MNIST data set consists of many 28 by 28 pixel images of scanned handwritten digits. Therefore, the input layer of the ANN should contain $784 = 28 \times 28$ nodes. The input pixels are greyscale, with a value of 0.0 representing white, a value of 1.0 representing black, and in-between values representing gradually darkening shades of grey.

The output layer of the ANN should contain 10 neurons to tell us which digit (0, 1, 2, 3, ..., 9) corresponds to the input image. Intuitively, the i-th output node could be interpreted as how much the ANN believes that the input is digit i. The prediction of the ANN is assumed to be given by the argmax of its output.

Coding Tasks

For this part, we provide you the MNIST dataset (`mnist.pkl.gz`) and two helper functions (in `database_loader.py`), one for loading the dataset and the other for converting a digit (i.e., value from 0 to 9) to its one-hot representation. Note that the function that loads the data returns three data sets: training data, validation data and test data. The first data set is used for training (i.e., learning the weights and biases), the second may be used

for choosing the ANN architecture or other hyperparameters and the last one is only used to evaluate your final trained model.

You will use your implementation of ANN realized in Part 1 to complete this task. In order to achieve the best performance possible, you will test different approaches (e.g., different activation functions, L2 regularization, early stopping) to train the ANN. In particular, following the examples of `sigmoid` and `dsigmoid`, you should implement at least these activation functions: `tanh`, and `ReLU`.

All the code necessary to reproduce your experiments should be saved in a file named `experiments.py`. For evaluation, you should output your training time, training accuracy and testing accuracy.

Important:

- You are free to choose the architecture of your ANN. We will provide bonus points if you compare different architectures. These experimental results should be explained in your report.
- You should explain how you chose the hyperparameters of your model (e.g., λ, n...).
- You should only provide the code for and report the results of the experiments where the trained ANN has a testing accuracy **above 90**%
- Frameworks including but not limited to `pytorch`, `tensorFlow`, `theano` are **not allowed in this part** as well.

Submission and Due Date

You need to submit a zipped file named `P3-Firstname-Lastname.zip` where you replace `Firstname` and `Lastname` by your first name and last name respectively. This compressed file should contain all the source code files mentioned above and a **short** report in pdf format containing:

- a README text file describing how to reproduce your experiments,
- for part 2, a summary explaining the main part of your code,
- for part 2, a comparison of the different methods with respect to training time, training accuracy and testing accuracy, with different activation functions,

- the presentation of any extra work you did (e.g., different architectures, other activation functions, dropout, L1 regularization..), which may receive bonus points.

Please submit your file on Canvas. The due date is 11:59 pm on Nov. 29th, 2018. There will be a penalty of 20% per day for late submission. Note that submissions will be checked for plagiarism.

Grading

Your project will be graded along three criteria:

1. Functional Correctness
2. Implementation Constraints
3. Report

An example of Functional Correctness is whether or not your algorithms produce the correct output. The accuracy of your training will also be considered. Implementation Constraints checks whether you stick to the implementation requirements. The length of the report has not much importance: we prefer clear and concise comments/explanations.