CSE474/574: Introduction to Machine Learning(Fall 2016)

Instructor: Sargur N. Srihari
Teaching Assistants: Jun Chu, Kyung Won Lee, Junfei Wang

Project 3: Classification

Due Date: Code, Report (Softcopy, Hardcopy) Wednesday, December 7, 2016 (11:59pm)

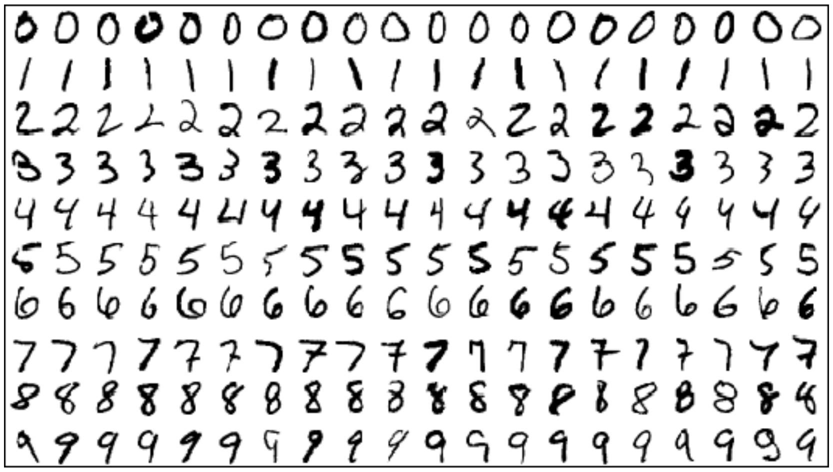
1 Overview

This project is to implement and evaluate classification algorithms. The classification task will be to recognize a 28×28 grayscale handwritten digit image and identify it as a digit among 0, 1, 2, ... , 9. You are required to do the following three tasks.

1. Implement logistic regression, train it on the MNIST digit images and tune hyperparameters (Appendix 1).
2. Implement single hidden layer neural network, train it on the MNIST digit images and tune hyperparameters such as the number of units in the hidden layer (Appendix 2).
3. Use a publicly available convolutional neural network package, train it on the MNIST digit images and tune hyperparameters (Appendix 3).
4. Test your MNIST trained models on USPS test data and compare the performance with that of the MNIST data. Does your finding support the "No Free Lunch" theorem?

1.1 MNIST Data

For both training and testing of our classifiers, we will use the MNIST dataset. The MNIST database is a large database of handwritten digits that is commonly used for training various image processing systems.The database is also widely used for training and testing in the field of machine learning.



The database contains 60,000 training images and 10,000 testing images. The dataset could be downloaded from here:

http://yann.lecun.com/exdb/mnist/

The original black and white (bilevel) images from MNIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. the images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field.

1.2 USPS Data

We use USPS handwritten digit as another testing data for this project to test whether your models could be generalize to a new population of data. Examples of each of the digits are given below. The dataset will be available on UBLearns.

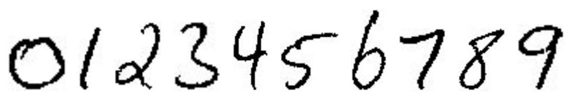


Figure 1: Examples of each of the digits

Each digit has 2000 samples available for testing. These are segmented images scanned at a resolution of 100ppi and cropped. Resize or fill the images to 28x28 like MNIST digits and feed this into your trained model and compare the result on USPS data and MNIST data.

1.3 Evaluation

Evaluate your solution on a seperate validation dataset using classification error rate

$$E = \frac{N_{\text{wrong}}}{N_{\text{V}}},$$

where N_{wrong} is the number of misclassification and N_{V} is the size of the validation dataset. Under the 1-of-K coding scheme, the input will be classified as

$$C = \arg\max_i y_i.$$

2 Plan of Work

1. **Extract feature values and labels from the data**: Download the MNIST dataset from the Internet and process the original data file into a Numpy array that contains the feature vectors and a Numpy array that contains the labels.
2. **Data Partition**: The MNIST dataset is originally partitioned into a training set and a testing set. You will use this partition and train your model on the training set.
3. **Train model parameter**: For a given group of hyper-parameters such as the number of layers and the number of nodes in each layer, train the model parameters on the training set.
4. **Tune hyper-parameters**: Validate the classfication performance of your model on the validation set. Change your hyper-parameters and repeat step 3. Try to find what values those hyper-parameters should take so as to give better performance on the testing set.

3 Deliverables

There are two parts in your submission:

1. Report

   The report describes your implementations and results using graphs, tables, etc. Write a concise project report, which includes a description of how you implement the models and tune the parameters. Your report should be edited in PDF format. Additional grading considerations will include the performance of the training, creativity in paramter tuning and the clarity and flow of your report. Highlight the innovative parts and do not include what is already in the project description. You should also include the printed out results from your code in your report.

   Submission:

   Submit the PDF on a CSE student server with the following script:

   `submit_cse474 proj3.pdf` for undergraduates

   `submit_cse574 proj3.pdf` for graduates

   In addition to the PDF version of the report, you also need to hand in the hard copy version by the due date or else your project **will not be graded**.

2. Code

   The code for your implementations. Code in Python is the only accepted one for this project. You can submit multiple files, but the name of the entrance file should be main.py. All Python code files should be packed in a ZIP file named `proj3code.zip`. After extracting the ZIP file and executing command `python main.py` in the first level directory, it should be able to generate all the results and plots you used in your report and print them out in a clear manner.

Submission:

Submit the Python code on a CSE student server with the following script:

`submit_cse474 proj3code.zip` for undergraduates

`submit_cse574 proj3code.zip` for graduates

4 Due Date and Time

The due date is **11:59PM, Dec 7** for both online submission and hardcopy submission

Appendix 1 Logistic Regression

Suppose we use 1-of-K coding scheme $\mathbf{t} = [t_1, ..., t_K]$. for our multiclass classfication task. Our multiclass logistic regression model could be represented in the form:

$$p(\mathcal{C}_k|\mathbf{x}) = y_k(\mathbf{x}) = \frac{\exp(a_k)}{\sum_j \exp(a_j)} \tag{1}$$

where the activation a_k are given by $a_k = \mathbf{w}_k^\top \mathbf{x} + b_k$. The cross-entropy error function for multiclass classification problem seeing a training sample $\mathbf{x}$ would be

$$E(\mathbf{x}) = -\sum_{k=1}^{K} t_k \ln y_k \tag{2}$$

where $y_k = y_k(\mathbf{x})$. The gradient of the error function would be

$$\nabla_{\mathbf{w}_j} E(\mathbf{x}) = (y_j - t_j)\,\mathbf{x}$$

You can then use stochastic gradient descent which uses first order derivatives to update

$$\mathbf{w}_j^{t+1} = \mathbf{w}_j^t - \eta \nabla_{\mathbf{w}_j} E(\mathbf{x})$$

to find the optimum of the error function and find the solution for $\mathbf{w}_j$.

Appendix 2 Single Layer Neural Network

Suppose we are using a neural network with one hidden layer. Suppose the input layers is denoted by x_i and the output is y_k. The feed forward propagation is as follows:

$$z_j = h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + b_j^{(1)}\right)$$

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + b_k^{(2)}$$

$$y_k = \frac{\exp(a_k)}{\sum_j \exp(a_j)}$$

where z_j are the activation of the hidden layer and $h(\cdot)$ is the activation function for the hidden layer. You have three choices for the activation function: logistic sigmoid, hyperbolic tangent or rectified linear unit.

We use cross-entropy error function

$$E(\mathbf{x}) = -\sum_{k=1}^{K} t_k \ln y_k$$

where $y_k = y_k(\mathbf{x})$. The backpropagation is done as follows,

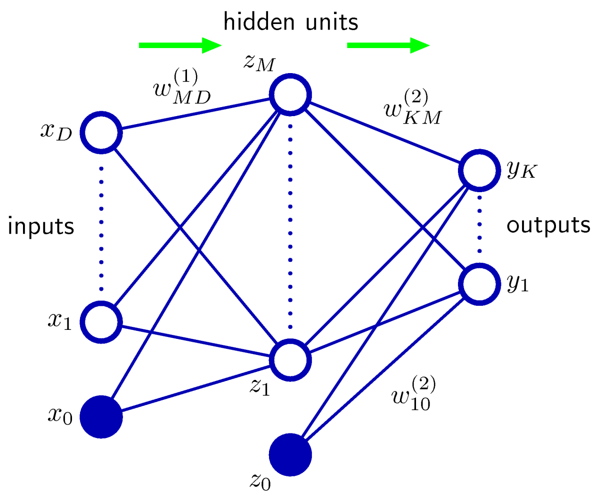


Figure 2: Network diagram for the two- layer neural network

$$\delta_k = y_k - t_k$$

$$\delta_j = h'(z_j) \sum_{k=1}^{K} w_{kj} \delta_k$$

The gradient of the error function would be

$$\frac{\partial E}{\partial w^{(1)}_{ji}} = \delta_j x_i, \qquad \frac{\partial E}{\partial w^{(2)}_{kj}} = \delta_k z_j$$

Having the gradients, we will be able to use stochastic gradient descent to train the neural network.

$$\mathbf{w}^{t+1} = \mathbf{w}^{t} - \eta \nabla_{\mathbf{w}} E(\mathbf{x})$$

where $\mathbf{w}$ is all parameters of the neural network.

Appendix 3 Convolutional Neural Network

For the convolutional neural network, you can use packages from online. Therefore, in this project description, we will not go into detail of the implementation. This could be done using a newly open-sourced package Tensorflow. Refer to www.tensorflow.org/tutorials.

Appendix 4 Mini-batch stochastic gradient descent

Mini-batch stochastic gradient descent is something between batch gradient descent and stochastic gradient descent. In each iteration of the mini-batch SGD, it samples a small chunk of samples

$\mathbf{z}_1, \mathbf{z}_2, ..., \mathbf{z}_m$ from the training data and uses this chunk to update the parameters $\mathbf{w}$:

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \sum_{i=1}^{m} \nabla_{\mathbf{w}} E\left(\mathbf{z}_i\right)$$

The strength of mini-batch SGD compared to SGD is that the computation of $\sum_{i=1}^{m} \nabla_{\mathbf{w}} E\left(\mathbf{z}_i\right)$ can usually be performed using matrix operation and thus largely out-performs the speed of computing $\nabla_{\mathbf{w}} E\left(\mathbf{z}_i\right)$ individually and updating $\mathbf{w}$ sequentially. However, within same computing time, mini-batch SGD updates the weights much more often than batch gradient descent, which gives mini-batch SGD faster converging speed. The choice of mini-batch size m is the tradeoff of the two effects.

Instead of randomly sampling $\mathbf{z}_1, \mathbf{z}_2, ..., \mathbf{z}_m$ from the training data each time, the normal practice is we randomly shuffle the training set $\mathbf{x}_1, ..., \mathbf{x}_N$, partition it into mini-batches of size m and feed the chunks sequentially to the mini-batch SGD. We loop over all training mini-batches until the training converges.