12410106 Assignment1

Zhang Sihua

DeepLearning 25 fall

Abstract. This is the report about assignment1 of DeepLearning course, perceptron&MLP . In this report, we illustrate the implementation details and results analtsis of the assignment. The report contains three part: the perceptron, the mutli-layer perceptron and stochastic gradient descent. To run our code, just run the jupyter notebook in directory Part_x (x=1,2,3) or execute bash commands provided in the notebook to run python scripts.

1 Part1 perceptron

In this part we implement and test a simple artificial neuron: a perceptron.

1.1 Task1

First we generate a dataset of points in R^2 from two Gaussian distributions and sample 100 points from each. We keep 80 points per distribution as training data(160 in total), 20 for testing (40 in total).

We specify the mean and covariance between two features of each Gaussian distribution, and use `numpy.random.multivate_normal()` to sample from each two-dimensional Gaussian distribution.Then we use `numpy.ones()`to generate labels. Two Distributions serve as two class, denoted as -1 and 1 respectively. Finally we stack them together as X and y, and visualize them using matplotlib. The visualization result is Fig.1.

Then, we `use train_test_split()` in sickit-learn to split the data into training and testing sets.The spliting ratio is 4:1 for training and testing.

1.2 Task2

We implement the perceptron following the specs in perceptron.py and the pseudocode in perceptronslides.pdf. Note that I add a batch_size parameter to support mini-batch gradient descent because it is both efficient and robust, and a test function for testing convenience. Also I concatenate W, b together as shown in perceptron_tutorial.pdf. Implementation details can be found in source code.

1.3 Task3

We train the perceptron on the training data (160 points) and test on the remaining 40 test points. The test accuracy is 0.97. We further visualize the decision boundary to inspect classification outcome. The visualization result is Fig.2.

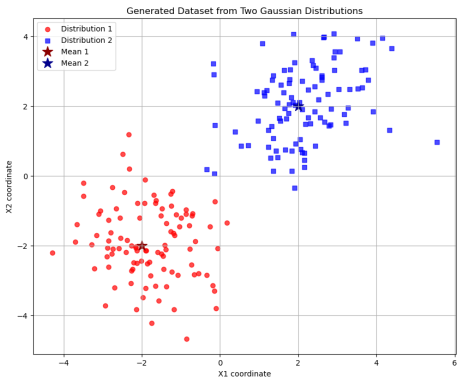

Fig. 1. 2Gaussian

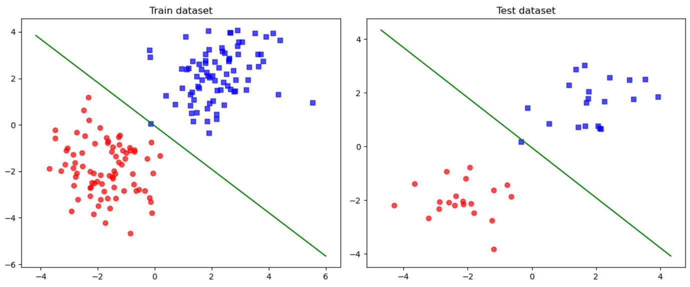

Fig. 2. DecisoinBoundary

1.4 Task4

We do more experiments on different means and variances of gaussian distributions and analyse the result. Intuitivily, if the two Gaussians' means are two close or their vaiance is too large, they may not be linearly separable since the overlapping of probability density become larger.

Fig.3 and Fig.4 shows the result where the means of two distribution are closer to each other, variances remain the same. Fig.5 and Fig.6 shows the result where two variances both increase from 1 to 3, and means remains the same. We can see in both cases, the decision boundary fails to perfectly classify all data points correctly. This is because perceptron is a linear model, when the dataset is no longer linearly separatable, is is not able to find a separating hyperplane for all data points. Divergence of loss during training is also a strong indicator to this.

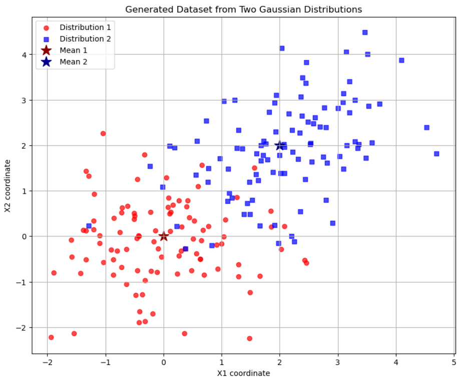

Fig. 3. CloserMeanDistributions

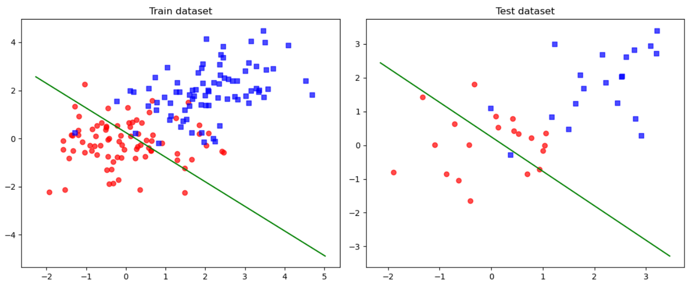

Fig. 4. CloserMeanDicision

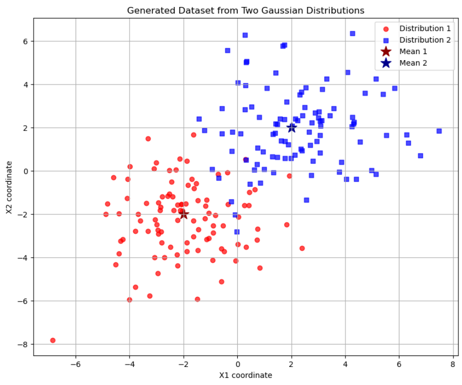

Fig. 5. LargerVarDistributions

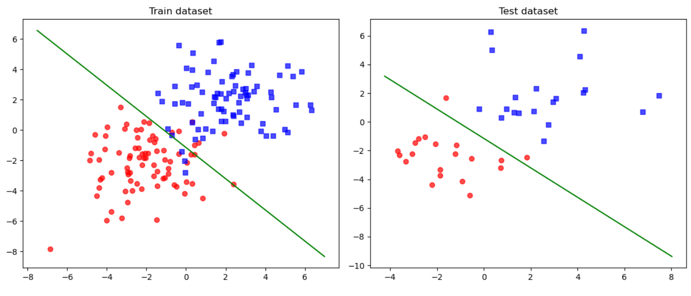

Fig. 6. LargervarDicision

2 Part2 multi-layer perceptron

2.1 Task1

We implement Linear, ReLU, Softmax and CrossEntropy in modules.py using numpy. Keypoints are:

1. We should initialize a cache to store intermediate values. When we need them to calculate gradients, we can directly fetch them from cahce to avoid recomputation.
2. Be careful with matrix calculas where errors can be easily introduced. For example, the derivative of $XW + b$ w.r.t W is X^T, not X.

Then we implement a simple MLP using these modules. Although our MLP isn't going to be extremly deep, I still use He initialization for the weights in Linear layers, which can prevent gradient from exploding or vanishing to some extent. It is known that He initialization does better than Xavier initialization for ReLU activation function. However, we still encounter the problem of gradient explosion, probably due to the lack of normalization or large learning rate (in our experiment is 0.01). But it dose not degrade the whole model.

Besides, I add update_params() function to update the parameters of the network during training.

2.2 Task2

After our MLP is ready to use, we use scikit-learn's `makemoons()`, `train_test_split()` ,`OneHotEncoder` to generate and preprocess our datasets. The visualization result is Fig.7.

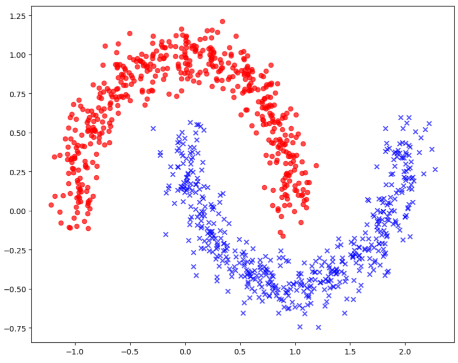

Fig. 7. Moons

Then we implement training and testing script in train mlp numpy.py. I prefer mini-batch gradient descent (MBGD) and didn't get the meaning that we ought to implement batch gradient descent (BGD) in this part, hence I add a dataloader class to assist mini-batch data loading. However, you can find BGD, MBGD, SGD in Part3.

2.3 Task3

We test the MLP on test dataset with default parameters. If you unset the random seed and test it multiple times you should find that on average the accuracy is about 0.5, which is not better than randomly guessing since we only have 2 classes. Then we train it on train dataset using BGD for 15 epochs. Batch size is set to 32. We record the running losses and accuracies duing training, and test it every 1 epoch (this hyperparameter can be larger than 1). Finally we visualize the loss and accuracy curve for trian and test dataset in Fig.8.

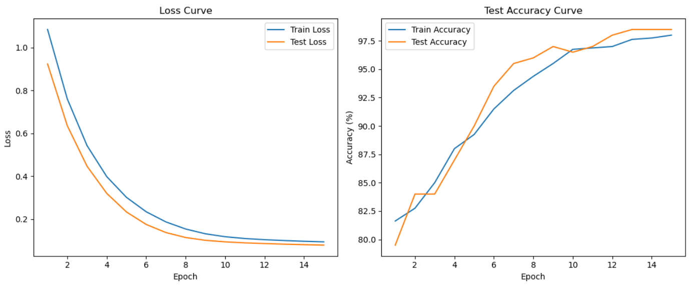

Fig. 8. LossAccCurves

The figure shows that at the beginning of training, train_loss is high and train_acc is low (80% this time, thanks to good random seed) . As the training progresses, train_loss decreases and train_acc increases. Our MLP gradually learns to classify the training data.

Note that sometimes the test_loss is even lower than the train_loss. This can happen due to random data splits, or the test set being easier than the training set by chance. As training progresses, train_loss usually becomes lower than test_loss. This does not indicate overfitting or underfitting, but rather reflects the randomness in data and training process.

3 Part3 stochastic gradient descent

3.1 Task1

I misunderstood the instruction and implemented minibatch gradient descent in Part2. As a result, I implement batch gradient descent, minibatch gradient descent and stochastic gradient descent all three kinds of gradient descent algorithms in Part3.

The only changes from Part2 is that I add two new command line argument to specify the gradient descent mode, –gd_mode, along with some changes in the train function, and –batch_size, which is the batch size for mini-batch gradient descent.

–gd_mode can take three int values: 0 stands for ”batch”, 1 stands for ”mini-batch”, and 2 stands for ”stochastic”. The default value is 0.

–batch_size only work if –gd_mode is 1. The default value is 32.

Model and training configuration is same as Part2.

3.2 Task2

Again we test our MLP with default parameters to show that our model is initialized randomly. Then we use three gradient descent algorithms to train our model and plot losses and accuracies results. The results of BGD, MBGD, SDG is Fig.9, Fig.10, Fig.11. The training hyperparameters are

1. BGD: epochs = 200, learning_rate = 1e-2
2. MBGD: epochs = 15, Batch_size = 32, learning_rate = 1e-2
3. SGD: epochs = 15, Batch_size = 1, learning_rate = 1e-3

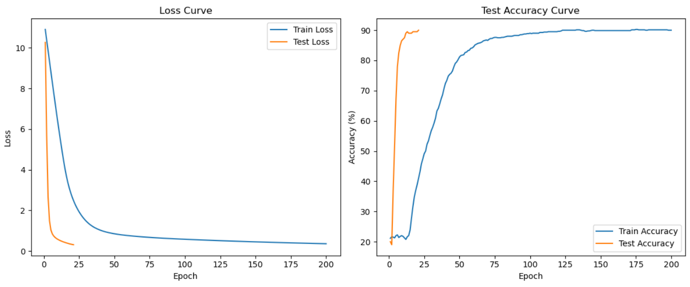

Fig. 9. BGD

Batch Gradient Descent (BGD) use all training examples as a batch, while mini-batch gradient descent (MBGD) uses a subset of training examples as a batch and Stochastic Gradient Descent (SGD) uses only one training example as a batch.When we decrease the batch size, our experiments show that the model will converge faster, but probably more unstable. If you run the notebook multiple times, you will see that BGD leads to similar loss and acc curves, while the curves of MBGD and SGD can be vary dramatically each time. This is because BGD use the whole dataset to learn, but mini-batch gradient descent randomly samples a subset to approximate the whole data distribution, and SGD iterates over a single example which add even more noice. Adding noice to training process may result in difficulty in converging, but it can also help to avoid overfitting and improve the model's generalization ability.

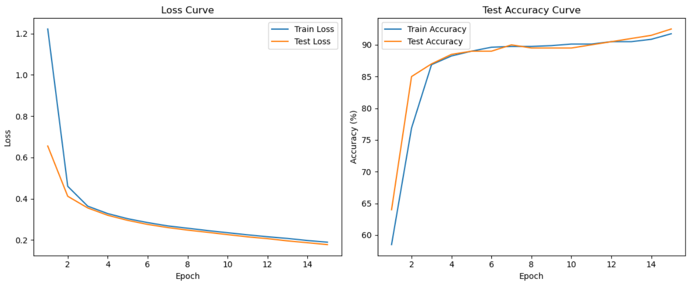

Fig. 10. MBGD

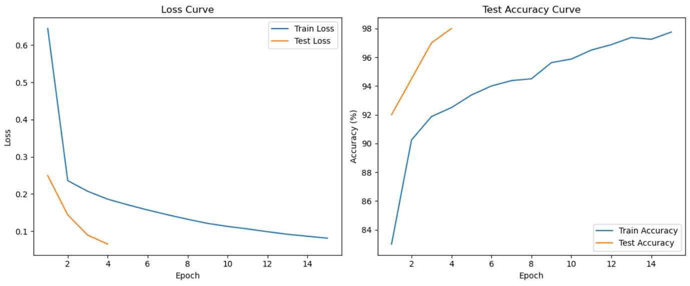

Fig. 11. SGD