Lab1: Perceptron & MLP

Zhang Yifan 11711335

Contents

1 Introduction

The aim of this lab assignment is for us to get to know the perceptron. The perceptron is an algorithm for supervised learning of binary classifiers. A binary classifier is a function which can decide whether or not an input, represented by a vector of numbers, belongs to some specific class. In this assignment, first, we were asked to implement a perceptron with only two layers, then train it and use it to classify which normal distribution the point comes from. Then we were asked to implement a multi-layer perceptron which can take in many parameters such as the number of hidden layers and the number of neurons for each hidden layer, the maximum epoch number, the learning rate and so on. And then use the Jupyter notebook to display the result. I finished all the parts and here is my report for assignment 1.

2 Part I: the percrptron

2.1 Task 1

The first thing to do is to generate the dataset of points in for this part. We need to generate two Gaussian distributions and sample from each of them. The two distributions should have different mean and covariance, which means they would scatter from different centers and after we train the perceptron, by training I mean to adjust the weight and bias of the perceptron, it would be able to classify which distribution some point is from by the output of the point going through this perceptron. As for implementation details, I used NumPy in python to generate the Gaussian distributions. First I asked the user to input the desired mean and cov of the distribution, and then generate and repeat this for another normal distribution. The most important line is $x, y = np.random.multivariate\_normal(mean, cov, 100).T$. Then I sample 100 points out of each distribution and use 80 points as training data and the rest 20 points as test data.

2.2 Task 2

Then we need to implement the perceptron. It consists of two steps: first is to initiate the perceptron, second is to implement the methods to train this perceptron. I implemented the perceptron.py according to perceptronslides.pdf. In init I defined the number of input, max_epochs, learning rate and so on, and I initiated the weight vector and bias for the perceptron. As for the values to initialize weight and bias, I took the strategy of using all zeros. I found that this strategy works for the linear classifier requested in this part because, with slightly distant distributions, my classifier can reach the desired accuracy of 100%. However, if things get complicated, I do believe that I should avoid using all zeros but instead random ways to initiate the weight vector, for example, using a normal distribution of the mean of 0 and a pretty small std.
In the training method, I implemented the training process of the perceptron. The main idea of the training process is to forward the point with weight and bias and check the result with its label. I found since I set the labels to be 1 and -1, I can simply update them by checking the sign of . If we get a mistake on positive, we should adjust the W by adding , otherwise subtract. So for convenience, we can include the label into this process, and if $np.any(labels[i] * (np.dot(self.weights, training_inputs[i]) + self.bias) <= 0)$, then it means the weight and bias needs to be updated.

2.3 Task 3

To finish this part, we need to first generate the data and shuffle them. To test more distributions, I modified my file to accept mean and cov for both distributions. After using these parameters to generate 200 points in total, I labeled all the points from the first Gaussian distribution as 1 and all the points from the second Gussain distribution as -1. Then I trained the 160 training points and save the rest 40 for the test. The perceptron's accuracy is computed by the $\frac{accurate\ time}{all\ predictions} * 100\%$. And gladly, when I use mean=[5,5], cov=[[2,0],[0,2]] for the first distribution and mean=[-5,-5], cov=[[1,0],[0,1]] for the second distribution, I found it gladly reached the desired accuracy of 100%.

2.4 Task 4

To analyze the question given in task 4, I thought the best way would be to first do the experiment and then analyze the experiment result, so I experimented with many sets of points from different distributions:

Mean1	**Cov1**	**Mean2**	**Cov2**	**Accuracy**
[5,5]	[[2,0],[0,2]]	[-5,-5]	[[1,0],[0,1]]	100%
[2,2]	[[1,0],[0,1]]	[-2, -2]	[[1,0],[0,1]]	97.5%
[1,1]	[[1,0],[0,1]]	[-1, -1]	[[1,0],[0,1]]	62.5%
[1,0]	[[1,0],[0,1]]	[0, -1]	[[1,0],[0,1]]	50%
[2,2]	[[2,2],[2,2]]	[-2, -2]	[[2,2],[2,2]]	60%
[2,2]	[[5,5],[5,5]]	[-2, -2]	[[5,5],[5,5]]	65%
[2,2]	[[10,0],[0,10]]	[-2, -2]	[[10,0],[0,10]]	52.5%

I did the above experiment with a lot of different sets of different Gaussian distributions. I found that is the means of two Gaussians are too close and/or if their variance is too high, the accuracy would dramatically decrease from at most 100% to nearly 50%. For example, of the same cov, if one distribution is of mean $[2, 2]$ and another one of $[-2, -2]$, my classifier can reach an accuracy of 97.5%, which is acceptable, but this accuracy is lower than of a larger mean difference such as $[5, 5]$ and $[-5, -5]$, as the classifier could make some mistakes. It's because that the perceptron would be confused by the two categories and cannot predict accurately. This observation fits with my anticipation because if the mean of the two distributions is two close, it would be like they scatter around the same center point and it would be hard for us and the perceptron to categorize them. If the means are very distant, for example, one is $[10, 10]$ and one is $[-10, -10]$, with appropriate cov, the points from two distributions are very likely to never get near the another, so classification would be easier. And if the cov is very low, they closely scatter around their center and are easy to classify. But if the cov is high, it means they could be everywhere around the center and may be confused with the other distribution.

2.5 README

Run perceptron.py.

Notice:
1. Mean should be an array of 1*2, so input one integer + space + another and then press enter.
2. Cov should be array of 2*2, so after input a 1*2 array, press enter and then input another 1*2 array.

3 Part II: the multi-layer perceptron

3.1 Task 1

A multi-layer perceptron consists of an input layer, a list of hidden layers, and an output layer. First I analyzed the forward process of the MLP. The input layer inputs the target data. Then for each hidden layer, it gets the input from the last layer and do the affine mapping first and then go through the activation function, and the result is the output of this layer and the input for the next layer. The affine mapping is the same as in part I, equals to WX+b, and the activation function for hidden layers is ReLU. The output layer also first do the adding mapping, however, it has a different activation function called the softmax, which is used to compute the valid probability mass function. And then to computes the cross-entropy loss L, which measures the loss between the predicted result and data's original label.
Then I analyzed the backward process of MLP. Simply to say, the backward process is the inverse of the forward process. The forward process is to forward the data through the MLP and get the result, while the backward process is to backward the loss or to say derivatives of input, back the MLP and adjust the parameters such as weight and bias of every layer of the MLP. So this process would be first to compute the derivative with respect to the input of the cross-entropy loss layer as out and pass this dout back to softmax, which is the activation function of the output layer. And then from the output layer to the last

hidden layer, consequently pass the dout through outlayer's weight and bias, last hidden layer's activation function, and then the last hidden layer's weight and bias and to the next hidden layer. First I implemented modules.py, I would respectively introduce the implemented modules, which contain Linear, ReLU, SoftMax, and CrossEntropy four modules.

3.1.1 Modules.py

1. Linear
Linear is a simulation of a linear layer. It has three functions, init, forward and backward. I first initialized it with params: weight and bias. I initialized bias with zeros, and the size is $out\_feature$. And when initializing weights, I encountered some problems: when I set std to 0.0001 as the comments say, it's just too small and the weight doesn't seem to be working. So I change to 0.5 and it seems to be working fine. And also for calculation convenience, I recorded other variables such as input value, output value, dw, db and so on in self.params and self.grads. The forward function is to apply the affine mapping, which is computed by Wx+b, and this value, as well as the input, needs to be recorded for backward computation. The most important things for the backward process for the linear module are two: 1. Compute its own grads['weight'] and grads['bias'] for its parameter updating, 2. Forward(in the reverse order) the gradient to the next layer. I calculated the gradients from the gradients of the previous module dout as:

$$\begin{aligned} G_w &= x.T * dout \\ G_b &= dout \\ dx &= dout * Weight.T \end{aligned} \tag{1}$$

and this dx should be the parameter dout in the backward process for the next module.

2. ReLU
ReLu, aka rectifier, is the activation function and is used to forward the positive part of its argument . Its forward method should be $relu_forward = np.maximum(x, 0)$ which only forward the positive part and sets the rest to zero. However, only forward this is not enough. In order to calculate its derivative, we also need to record this information, so I used a list of boolean variables to record it. I set it to Ture where x is larger than zero. And its backward method should be $dx = np.where(self.mask, dout, 0)$.

3. SoftMax
The softmax module is the activation function for the output layer. It is also known as a normalized exponential function, and its forward function is to take in input as a vector of K real numbers, and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers. After forward, the result should be consisting of numbers which are all between (0,1) and sum up to 1. To implement the forward process, I used the Max Trick from https://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/.

$$\begin{aligned} y &= np.exp(x - np.max(x)) \\ out &= y/y.sum() \end{aligned} \tag{2}$$

4. Cross-Entropy
The cross-entropy between two probability distributions p and q over the same underlying set of events measures the average number of bits needed to identify an event drawn from the set if a coding scheme used for the set is optimized for an estimated probability distribution q, rather than the true distribution p. In this assignment, I figured it would be suitable to use this CE Loss to evaluate the convergence of the MLP as well as the training efforts. I use y to denote the labels of input, and x to denote the forward result of my MLP, so CE loss should be $-np.sum(y * np.log(x))$.

5. SoftMax+CE loss
In the above two modules, I didn't mention the backward process of SoftMax and Cross Entropy, that is

because I used a trick here and combine their backward process to one single equation.

$$\begin{aligned} Softmax:\ p_i &= \frac{e^{a_i}}{\sum e^{a_k}} \\ Cross\ Entropy\ Loss\ &= -\sum y_k log(p_k) \\ \frac{\alpha L}{\alpha a_i} &= -\sum y_k \frac{1}{p_k} \frac{\alpha p_k}{\alpha a_i} \end{aligned} \tag{3}$$

So Cross Entropy Loss with Softmax:

$$\frac{\alpha L}{\alpha a_i} = p_i - y_i$$

3.1.2 mlp_numpy.py

Then I implemented mlp.numpy.py to construct a multi-layer percrptron with Numpy.

1. Init
Since we have already implemented all the modules, what we have to do here is to construct the whole structure of the MLP with modules from modules.py. First I initialized it with linear layers: hidden layers and the output layer. We need to pay extra attention to the parameter n_hidden, it is not an integer, but a list of integers, which represent the number of neurons in each hidden layer, so the number of hidden layers should be len(n_hidden). All the linear layers each have different dimensions but are all connected with the neighbor layers, so for example, the out_feature of hidden layer l should be the input feature of hidden layer l+1(if there is l+1, or should be output layer). So after initiation, one MLP object has a list of hidden layers, one output layer, and sizes stored within it.

2. Forward
The forward function of an MLP should be the whole process of getting an input x, forward it from the input layer to hidden layers, and at last to the output layer and return this result. Suppose we have n hidden layers numbered from 0 to n-1, this whole process should be represented as follows: First, the input layer takes the x and hidden layer 0 forward this x by its forward function, now the get u0 as the value before activation, then hidden layer 0's relu module would forward this u0 and turn it into a0, , and this a is the output of hidden layer 0, as well as the input of hidden layer 1. Then hidden layer 1 takes this a and forward it, and this process continues as it reaches the hidden layer n-1. Now the hidden layer has the output value of an-1, and the output layer should forward this value to get un. Now the activation function for the output layer is Softmax, the output layer's softmax module takes this value and forwards it to output z. Now we get the predictable result of the whole MLP, but this value, as well as the label of input x, should be computed by the Cross-Entropy Loss module and get the loss of this prediction.

3. Backward
The backward process should proceed when the forward function is finished and the loss is computed. Now we have the prediction p and the correct label L, also the values of the input, before_activation, after_activation are all stored in the corresponding module, we can start backward. This is in the reverse order of forward and the values passing on should be the gradients, which here is denoted by dout. First, we compute the gradient of cross-entropy loss with softmax, which is easied calculated as (predict-L), then the output layer does the backward calculation, compute its gradients of weight and gradient of bias, and comput its dx and backward this value to hidden layer n-1. Hidden layer n-1 takes this value and first backward it with its relu function, and then backward the output of its relu. Also, hidden layer n-1 computes its gradients of weight and gradient of bias, and comput its dx and backward this value to hidden layer n-2. This process goes on until hidden layer 0 computes its gradients of weight and gradient of bias.

3.2 Task 2

The first thing to train my MLP would be to generate train and test data. As the assignment requires, I used the python package sklearn.datasets.make_moons to generate this data set. I generated 1000 data,

shuffle them, record their respective label, and used 80% as train data and 20% as test data. To use their label and include it in the process of training, I used the trick called the one-hot label to encode labels. After I know what this encoding and decoding process means, I wrote two functions: encode and decode to transform labels. The encoding process would change one integer into a 1-D array with length 2: if this integer is 1, then the one hot label should [1, 0], or it should be [0, 1]. And decode function would transform one array back to an integer: is the first value is larger than 0.5, then the answer should be 1, or should be 0. Then I try to use the given argument parser. I found that this could take into four possible parameters including the maximum number of epoch, the list of neurons in hidden layers, the learning rate, and the evaluation frequency, which is used to compute and record accuracy and plot them at the end. So I finished the accuracy evaluation function and prepare to begin training. In this part, I used batch gradient descent as the optimizer. BGD means that this MLP would only update its weight and bias after one epoch, which means all the data has forward and backward once. The detailed implementation is that, during each epoch, first we forward the data one by one. After each data is forwarded, the gradient of weight and gradients of bias computed should be added to a structure which is designed to save all these gradients of each layer, so after all the data is calculated once, we use the sum of the gradient of weight of every linear module, divide by the batch size (in this case is the length of training data), multiplies with learning rate and update each layer's weight and bias.

$$\begin{aligned} w &= w - learning\ rate * \frac{1}{batch\ size} \sum gradients\ of\ weights \\ b &= b - learning\ rate * \frac{1}{batch\ size} \sum gradients\ of\ bias \end{aligned} \tag{4}$$

Also, for convenience, I add argument –task to argument parser, so in part2.ipynb, we can simply use $--task2$ to run this part.

3.3 Task 3

3.3.1 README

part2.ipynb is used to display the result of applying all the default parameters to train this MLP. To apply other parameters, there are two ways: one is to simply run train_mlp_numpy.py with desired arguments or to modify the script in part2.ipynb because for convenience I used *%runtrain_mlp_numpy.py* –task 2 in part2.ipynb. NOTE: –task is a new argument option that takes in an integer and we can specify tasks 2 or 3(later) as asked. It would print WORKING to show that you've successfully run this program and it's doing the calculation.

3.3.2 Result & Analysis

I used matplotlib.pyplot to draw the accuracy curves and loss curves. Here I draw two sub figures, the first subplot is for the accuracy of training data with respect to training epochs and test data with respect to training epochs, where accuracy is defined by the correctly predicted numbers divide by the number of predicted data. The second subplot is the loss curve, the cross entropy loss of the training data and test data, with respect to training epochs.

The default parameters are: learning rate = 1e-2, max steps = 1500, evaluation frequency = 10 epochs, hidden units = "20". The following figure is for MLP using default parameters for task 2, where x-axis is the number of epochs, and the y-axis are accuracy in percentage and CE loss respectively:

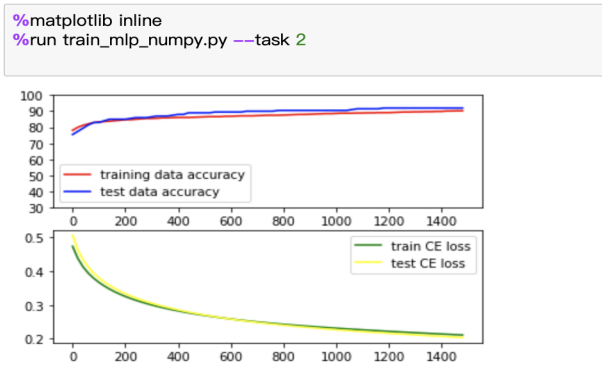

Figure 1: Task 2 Default parameters

As we can see, the default values can only achieve an accuracy of around 90%, which is not the most desired result. I think this might due to that batch gradient descent only update weight and bias every epoch, which is relatively slow. But this also causes that the BGD curves are very smooth, which is the big characteristic of batch gradient descent, and we can see the accuracy is ascending with more epochs while loss is descending. So this fugire shows that my MLP is training just fine and is a valid and efficient classifier.

Before, we used the default parameters. However, 90% may not be our desired result, so I did more experiment, and adjusted the parameters. I found that, with a larger learning rate, it might get a better result, but it could cause problems such as oscillate more rapidly and the model could seem to be less convergent.

For example, I used a larger learning rate 1e-1 and increase the max_steps to 5000, and reached 100% accuracy for both training and testing data by BGD:

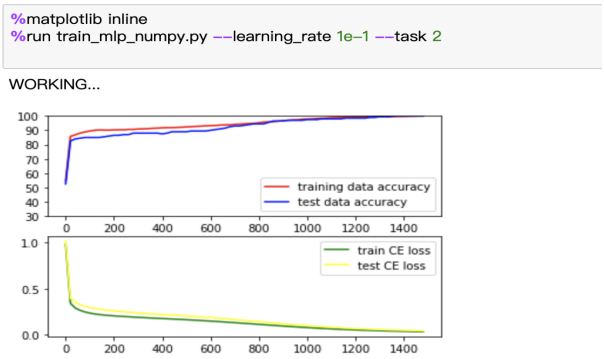

Figure 2: Task 2 Adjusted parameters

From this figure we can see that, with a larger learning_rate, the model oscillate more, however, it can reaches 100% accuracy around 1400 epoches. So I adjusted the parameters and allowed BGD to reach a 100% accuracy.

4 Part III: Stochastic gradient descent

4.1 Task 1

I modified train_mlp_numpy.py to accept an input that will decide the type of optimizer. The input should be an integer. As for the detailed implementation of stochastic gradient descent, it's only slightly different

than batch gradient descent. That is, we forward each one point in the training data, and immediately do the backpropagation and update the parameters. After one epoch, we should have updated the parameters of each layer of the length of training data times.

To understand the differences, I wrote the pseudocode of these two optimizers, the first one is SGD and the second one is BGD.

```
For each epoch:
   For each data:
        mlp.forward(data)
        mlp.backpropagation
        UPDATE w and b
   On evaluation:
        calculate accuracy
```

```
For each epoch:
    For each data:
        mlp.forward(data)
        mlp.backpropagation
        record dw and db
    UPDATE w and b
    On evaluation:
        calculate accuracy
```

The characterisctics of SGD is that it converges more rapidly than BGD. This is because SGD is noisy. It responds to the effects of each and every sample, and the samples themselves will no doubt contain an element of noisiness. While this can be a benefit in that it can act to "kick" the gradient descent out of local minimum values of the cost function, it can also hinder it settling down into a good minimum. There is tradeoff between SGD and BGD, so that why there is also MINI-batch, Adam and so on. They each have their own advantages and can b emore suitable in different occasions.

4.2 Task 2

4.2.1 README

part3.ipynb is used to display the result of applying all the default parameters to train this MLP. In order to apply other parameters, there are two ways: one is to simply run train_mlp_numpy.py with desired arguments, or to modify the script in part3.ipynb because for convinience I used %run train_mlp_numpy.py –task 3 in part3.ipynb. NOTE: –task is a new argument option which takes in an integer and we can specify task 2 or 3(now) as asked. It would print WORKING to show that you've successfully run this program and it's doing the calculation.

4.2.2 Result & Analysis

I used matplotlib.pyplot to draw the accuracy curves. Here I draw two sub figures, one is for the accuracy of training data and test data, where accuracy is defined by correctly predicted numbers divide by number of all data. Another one is the loss curve which is the cross entropy loss of the training data. As we can see in the picture, we first input the optimizer type and then begin training.

The default parameters are: learning rate = 1e-2, max steps = 1500, evaluation frequency = 10 epochs, hidden units = "20". The following figure is for MLP using default parameters and SGD for task 3:

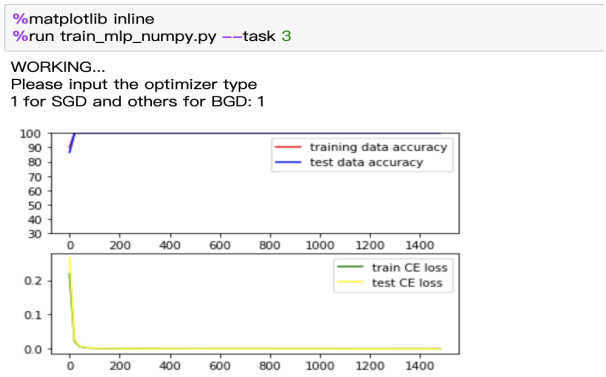

Figure 3: Task 3 Default parameters

As we can see, the MLP using the default parameter and SGD can quickly achieve an accuracy of 100% in only about 200 epochs, however, the curve is not as smooth as BGD, but we can see that in this case, it has a better result than BGD. I believe the reason is that SGD update parameters way more often than BGD and it's very likely it reaches good results during one update and maintains that.

This result is basically in line with expectations because SGD responds to the effects of each and every sample. This can be a benefit in that it can act to "kick" the gradient descent out of local minimum values of the cost function, it can also hinder it settling down into a good minimum. And in fact, SGD can reach an accuracy of 100% after only about 20 epochs, which is something BGD cannot achieve. But BGD can ascends more gradually and smoothly, which guarantees the loss is always descending, while SGD cannot guarantees this.

5 Appendix

The structure of the uploded zip file 11711335_assignment1.zip : In the root directory, there are a report in .pdf and three folders, one for each task. In each folder there is python code, .ipynb file and screenshots used in the report.

6 References

References

[1] https://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/

[2] https://deepnotes.io/softmax-crossentropy

[3] https://blog.csdn.net/JiaJunLee/article/details/79665062

[4] https://machinelearningmastery.com/how-to-one-hot-encode-sequence-data-in-python/

[5] https://adventuresinmachinelearning.com/stochastic-gradient-descent/