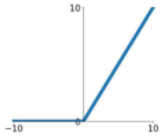
Homework 4 zl9901

Question 1

ReLU

$$\max(0, x)$$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

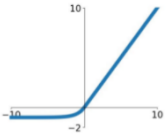


Fig. 1

As the figure 1 shown above, Relu has vanishing gradient problem especially when x<0. I choose to use ELU instead. When x<0, we use y=$\alpha(e^x - 1)$ instead of using y=0. The problem will be solved in this way. We need to update both forward and backward activation function.

For loss function, I noticed if we use cross entropy, the program will fail frequently. If I use MSE (mean square error), the result is much better.

So I choose to use both ELU and MSE loss function to avoid vanishing gradient problem. The results are shown below.

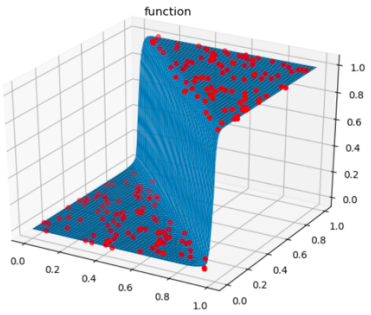


Fig. 2 my_nn_1

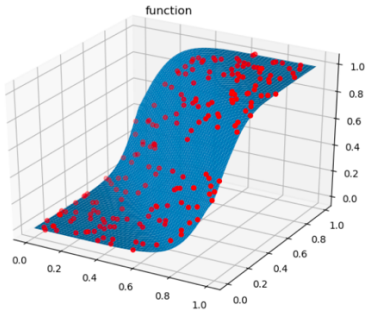


Fig. 3 my_nn_2

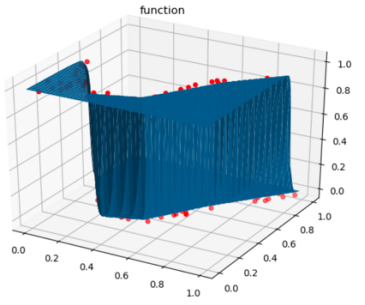


Fig. 4 my_nn_3

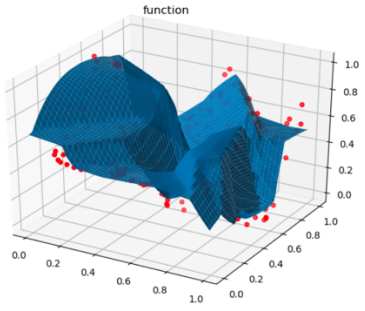


Fig. 5 my_nn_4

Question 2

The idea of this question is based on the formula:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \theta^{-1} - S + \Gamma$$

$$\frac{\partial \mathcal{L}}{\partial \gamma_{gr}} = \theta_{gr}$$

g and r represent nodes and they are independent in the given graph.

We initialize the matrix Γ which size is 7*7 and then we update this matrix based on gradient descent, we should keep in mind we only update parameters in Γ that are independent in the given graph.

In conclusion, there are two gradient descent formulas for this question. And we should use appropriate learning rate to achieve a better result.

Question 3

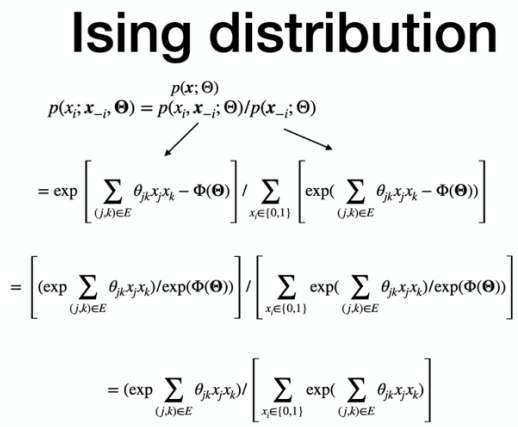


Fig. 5

The pseudo code of question 3 is shown below.

We initialize $x = x_1, x_2, \ldots, x_n$ uniformly at random

Repeat many iterations (we can define number of iterations by ourselves):

For i in range(1, n):

Randomly choose and update $x_i \sim p(x_i; x_{-i}, \theta)$ (we can find this in figure 2)

For sufficient many repeats, independent of intial conditions

$p(process\ stops\ at\ x) \approx p(x; \theta)$

We estimate \phi to be 43.50

Question 4
The basic process of PCA using numpy and scipy is shown below.

1 we get our original data matrix named ‘data’
2 data - = np.mean(data, axis=0)
we change the center of the data
3 np / = np.std(data, axis=0)
we normalize the data
4 matrix = np.cov(data, rowvar=False)
we calculate the covariance matrix of the data, rowvar is false means each row represents a sample
5 values, vectors = LA.eigh(matrix)
we calculate eigenvectors and eigenvalues of covariance matrix
6 key = np.argsort(values)[::-1][:num_components]
This function returns array of indices that sort values in descending order. If we define num_components which is equal to 2, then we get first 2 elements of the array returned.
7 eigen_value=values[key]
We get the eigen value according to indices we get from last step
8 eigen_value=vectors[:,key]
We get the eigen vectors according to indices
9 res=np.dot(data, eigen_vector)
According to the eigen vectors we get, we calculate the matrix product and get our final result

The distribution of data doesn’t change a lot. We use PCA mainly to reduce computational complexity. Please see the images below which are results of PCA.

Fig. 6