Report. Lab 5.

1. Description of the network (e. g. number of layers, number of neurons per layer, etc.).

A neural network is composed of three layers: an input layer, a hidden layer, and an output layer. The input layer receives pixel values of digit images, the image size is 20x20. Thus, there are 400 input layer units (excluding the extra bias unit which always outputs +1). The data is loaded into the variables X and y.

2. Impact of specific parameters such as λ, number of iterations, weight initialization, etc.

Lambda: the tuning parameter that decides how much we want to penalize the flexibility of our model. When λ = 0, the term has no effect. However, as lambda grows, the term starts having more influence on estimated parameters.

Number of iterations: number of times we do backpropagation. The bigger the number of iterations is, the better the accuracy of the classifier becomes. However, is the number of iterations being too large, there is a chance to overfit the model so it would perform worse on new data. To avoid overfitting, it is important to use techniques such as regularization and early stopping.

Weight initialization: When creating neural networks, it is crucial to randomly initialize the weights and other parameters if relevant, for symmetry breaking. To randomly initialize weights, I randomly uniformly selected values from range [-0.12, 0.12]. The justification for such values I found in lab description.

3. How does the regularization affect the training of your ANN?

Regularization helps prevent overfitting of the model. Overfitting happens if the model was trained too precisely on the training data set. It means that the performed classification boundaries are too complex and will not work well on new data sets. Regularization helps to prevent overfitting by adding constraints to the network. The classification then becomes simpler and more general.

4. Did you manage to improve the initial results (using values in debugweights.mat)? Which was your best result? How did you configure the system? How could you improve them even more?

I'm not sure I got the question.

5. Imagine that you want to use a similar solution to classify 50x50 pixel grayscale images containing letters (consider an alphabet with 26 letters). Which changes would you need in the current code to implement this classification task?

If the image is 50x50 layers that means that there are 2500 input layer units. There are 26 letters (classes in our jargon), so the number of labels should be equal to 26. Also, we should consider how many letters are there in our dataset and change the sizes of variables related to it, such as theta 1 and theta 2.

6. Change the value of the variable show_examples (in the python version, run the relevant block in the Jupyter one) in ex_nn, which information is provided? Did you get the expected information? Is anything unexpected there?

Which information is provided?

```
Training Set Accuracy:  97.52 – training accuracy
```

```
0 – the index of the image printed
Displaying Example Image – displays the image of a digit below
Neural Network Prediction: [6] (digit [6]) – the actual value of y and
the value of y predicted
```

Did you get the expected information? Is anything unexpected there? I got everything as I expected. The part that might confuse the reader is that the picture and the true and predicted number are not equal. The number on the picture is always bigger than the predicted and expected value by 1. It happens because of the difference in indexing in Matlab (first element in array has index 1) and Python (first element in array has index 0). Thus, label value 2 corresponds to picture of digit 3, label value 0 corresponds to picture of digit 1, label value 9 corresponds to picture of digit 0, etc. So that is the expected result, nothing goes wrong.

7. How does your sigmoidGradient function work? Which is the return value for different values of z? How does it work with the input is a vector and with it is a matrix?

My sigmoidGradient works amazingly! :) For different values of z, it returns

g = sigmoid(z)*(1-sigmoid(z))

which is the derivative of g with respect to z. g here is sigmoid activation function.

I have conducted a few experiments, and these are the results:

Input: [-15 -1 -0.5 0 0.5 1 15] – Output: [3.05902133e-07 1.96611933e-01 2.35003712e-01 2.50000000e-01 2.35003712e-01 1.96611933e-01 3.05902133e-07]

Input: 100000 – Output: 0.0

Input: 0 – Output: 0.25

Input: [[1 4 5] [-5 8 9]] – Output: [[1.96611933e-01 1.76627062e-02 6.64805667e-03] [6.64805667e-03 3.35237671e-04 1.23379350e-04]]

For vectors and matrices in the input my sigmoidGradient function returns vectors and matrices of the relevant sizes.