Problem Set 1

CMSC 828L

Due: 2/22/22, 12:30 pm (start of class)

The purpose of this problem set is for you to implement backpropagation from scratch. You will use your results to experiment with some simple networks on simple, synthetic data sets, to get a feeling for what it is like to tune your networks. A useful guide to the techniques needed can be found in *Neural Networks and Deep Learning*, Chapters 2 and 3. It is easy to find code for backpropagation on line, but I strongly urge you to write your own code, without referring to others.

The first two problems are warm-up exercises. Then you will be asked to implement more general, fully connected networks. You could skip right to implementing general networks, and use these to solve the first two problems. However, solving the first two problems in a more limited way will help build your intuitions.

For each problem, you will implement the ability to train and apply the network. Training adjusts the weights and bias terms with gradient descent. You do not need to implement stochastic gradient descent except for the MNIST problem (that is, at each step, compute the full gradient using all the training examples). During training, you can use a single parameter, η, to control the step size. That is, the new weights and bias equal the old weights and biases minus the gradient times the step size.

Python is required for this assignment. You must submit a separate Python script for each subproblem. The name of the file should have the form: `sol_<QUESTION_NUMBER>{a,b}.py`, where part a concerns scalar input and part b concerns high-dimensional input. For example, problem 1 with scalar input is `sol_1a.py`. If you want to submit zip file, please make sure `sol_*.py` is in the top-level of the zip file (i.e. not in any directory). This format is required by the autograder. Failure to adhere to it will delay grading. All linear algebra operations, or any related operations should be performed using `numpy`. Libraries like `PyTorch` and `TensorFlow` are not allowed. Only `numpy`, `matplotlib`, and Python built-in libraries are allowed.

In addition to the code, you will also need to submit a written description of your work, including figures. A PDF will be required, in addition to whatever files are used to generate it, for example Jupyter notebooks.

Your write-up should be self-contained, so that reading it allows us to understand your approach and experiments. In many cases, the problems below require you to design your own test data. Explain how you did this and why, illustrating what you did with figures where appropriate. If we can't figure out what you did from reading the write-up, you will not get a good grade; we shouldn't have to look at your code to know what you did.

Your networks will be graded on correctness and numerical accuracy. You should train your networks until the training loss converges in some suitable sense. We will provide some public unit-tests to help your develop your code. You will also be graded on private tests which will not be distributed.

1. This network contains an input layer and an output layer, with no nonlinearities. The output is just a linear combination of the input. Specifically, We can denote

the input values by :

$$a_1^0, a_2^0, ..., a_d^0$$

Then the output will be given by:

$$a_1^1 = \sum_{k=1}^{d} w_{1k}^0 a_k^0 + b_1^1$$

Use a regression loss to train the network. That is, the loss would be:

$$\sum_{i=1}^{n} \frac{1}{2}(y_i - a_1^1(x_i))^2$$

where $a_1^1(x_i)$ denotes the activation of the output unit when the input is x_i. Write code to train this network and to apply it to data.

Tests: Perform the following tests to validate your code. The main hyperparameters for your code are the step size and number of iterations of gradient descent that you perform. You may have to play with these to get a good result. There are also implicit hyperparameters that determine how you initialize the weights and bias of the network.

(a) Generate some random 1D test data according to a simple linear function, with Gaussian noise added. For example, your data might be generated with: $y = 7x + 3 + \xi$, where ξ is a Gaussian random variable. Include a plot showing the training data and the function that your network computes. (You can plot the function by evaluating it on a range of different inputs). This is all 1D, so easy to visualize.

(b) Perform a similar experiment using higher-dimensional input. Demonstrate that your trained network is computing a reasonable function by evaluating it on some held-out test data, and comparing the computed values to ground truth values.

(c) How difficult was it to find an appropriate set of hyperparameters to solve this problem?

Your write-up should include plots and tables as needed to visualize your results. It should include a clear description of how you generated the training data, and explicit answers to all of the questions listed above.

2. A Shallow Network

Now implement a fully connected neural network with a single hidden layer, and a ReLU nonlinearity. This should work for any number of units in the hidden layer and any sized input (but still just one output unit). This means:

$$z_j^1 = \sum_{k=1}^{d} w_{jk}^1 a_k^0 + b_j^1$$

and

$$a_j^1 = \max(0, z_j^1)$$

The output unit is:

$$a_1^2 = \sum_{k=1}^{d} w_{1k}^2 a_k^1 + b_1^2$$

as there is no non-linearity at the output layer. You can continue to use a regression loss in training the network.

Tests: Perform the following tests to validate your code. You now have another hyperparameter, the number of hidden units.

(a) 1D. Generate training data from a simple 1D function, such as a sine wave. You do not need to add noise to the function. Train your network to fit this data. Turn in a plot that shows the training data, along with a curve showing the function that your network computes.

(b) Perform a similar experiment using higher-dimensional input. Demonstrate that your trained network is computing a reasonable function by evaluating it on some held-out test data, and comparing the computed values to ground truth values.

(c) How difficult was it to find an appropriate set of hyperparamters to solve this problem? Do you notice any difference in the difficulty of solving the 1D problem and solving the higher dimensional problem?

Again, your write-up should include explicit answers to each question, along with graphs and tables needed to demonstrate your results.

3. General, Deep Networks

Now, write more general code to handle a fully-connected network of arbitrary depth. This will be just like the network in Problem 2, but with more layers. Each layer still has a ReLU activation function, except for the final layer.

(a) Test your network with the same training data that you used in Problem 2, using both 1D and higher dimensional data. Experiment with using 3 and 5 hidden layers. Evaluate the accuracy of your solutions in the same way as Problem 2.

(b) Play around with different choices of hyperparameters. Based on your experience, do you think it is easier or harder to choose effective hyperparameters for a deeper network than for the shallow network? Explain?

(c) Do some experiments to determine whether the depth of a network has any significant effect on how quickly your network can converge to a good solution. Include at least one plot to justify your conclusions.

4. Cross-Entropy Loss

Now modify your network for classification instead of regression. You will use a cross-entropy loss, with a logistic activation. Previously, there was no activation

function in the output layer, so $a_1^L = z_1^L$ for a network with L layers. Now, instead you will have:

$$a_1^L = \frac{1}{1 + e^{-z_1^L}}$$

where a_1^L denotes the (only) unit in the output layer. This converts the output into a value between 0 and 1, which can be interpreted as the probability that the input belongs to class 1. Then, use a loss of:

$$-\sum_{i=1}^{n} y_i \log(a_1^L(x_i)) + (1 - y_i) \log(1 - a_1^L(x_i))$$

Here we are assuming that the label, y_i, is a binary value (0 or 1), so either y_i or $1 - y_i$ is equal to 0, and the other value is 1.

Note: On numerical stability: Exponentiating positive number is always risky. In sigmoid, you are exponentiating $e^{-z_1^L}$, which can potentially overflow if z_1^L is negative. Hint: you can avoid this by multiplying numerator and denominator of a_1^L by $e^{z_1^L}$ when z_1^L is negative. By using `np.maximum` and `np.minimum`, you can have a concise implementation of this idea.

Tests:

(a) Test your network with two 1D problems; that is, your input is just a scalar. First, choose two sets of points that are linearly separable. Vary the margin between the points and the number of layers in the network. Is it more difficult to find hyperparameters that solve a problem with a smaller margin? Does the speed with which the network converges to a good solution depend on the margin? Include a plot to support your answer to the second question.

Second, test your network with some 1D data that is not linearly separable. What differences do you observe?

(b) Repeat these experiments, using higher dimensional data. Again, select data that is linearly separable, and then select data that is not linearly separable. Include data to support your conclusions.

5. Train an MLP on MNIST. Use an MLPs with 3-5 hidden layers as in previous problems. MNIST is a multi-class classification problem. You should implement a numerically stable combined softmax and cross-entropy layer for this problem. Definition of softmax when N output where z_i is the ith unit in the output layer:

$$a_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

Due to the large size of this dataset, you have to implement stochastic gradient descent with mini-batches. Unlike previous problems, you will be changing the

data of the `data_layer` and labels of the `loss_layer` in each training iteration. Feel free to try out different batch sizes and pick the one that works the best for you.

Submit a checkpoint for this problem named `mnist_weight.npz`, with this example snippet:

```
np.savez("mnist_weight.npz", weight=trainer.network.state_dict())}.
```

Your code should be named `sol_mnist.py`.

Acknowledgements

We gratefully acknowledge use of Prof. Tom G's layer and autograd code in writing the framework for this assignment.