backprop-solution

September 29, 2024

1 Backprop for linear regression and beyond (solution)

In this assignment you'll write backprop code for a few simple neural nets. The first one implements linear regression, but we also work with neural nets having a hidden layer of neurons.

You'll also tweak the learning rate and number of training steps to try to get your nets to perform well.

The understanding you get of the how neural nets work will help a lot in the coming weeks.

Instructions:

- All homework is to be done completely on your own. Don't refer to the web for solutions and don't even discuss the homework with others.
- Start by reading the notebook.
- There are 7 problems. For each problem, add code to the indicated notebook cell.
- Do not modify any other cells, and do not add imports.
- It is not enough to duplicate my output – your code must solve the problem as stated.
- Be sure to "restart and run all" before submitting your notebook.

```
<IPython.core.display.HTML object>
```

Feel free to use the functions below in your code.

Heart disease data

1.1 Linear regression, 2 inputs

The figure below shows linear regression written as a tiny neural net. There are two inputs, x1 and x2, along with a bias input. The true target value is y, and the predicted target value is v. The value of the loss function is z.

Writing the network as equations:

1.1.1 Problem 1

This problem is like a lab problem we worked on in class.

Add your code at the marked locations. You can use whatever variable names you like, but you will need to use b0, b1, b2 for the weights.

I recommend using the names in the diagram above. For the partial derivatives, I used names like dv_b0 for the partial derivative of v with respect to b0 and dz_b2 for the partial derivatives of z with respect to b2.

1.1.2 Problem 2

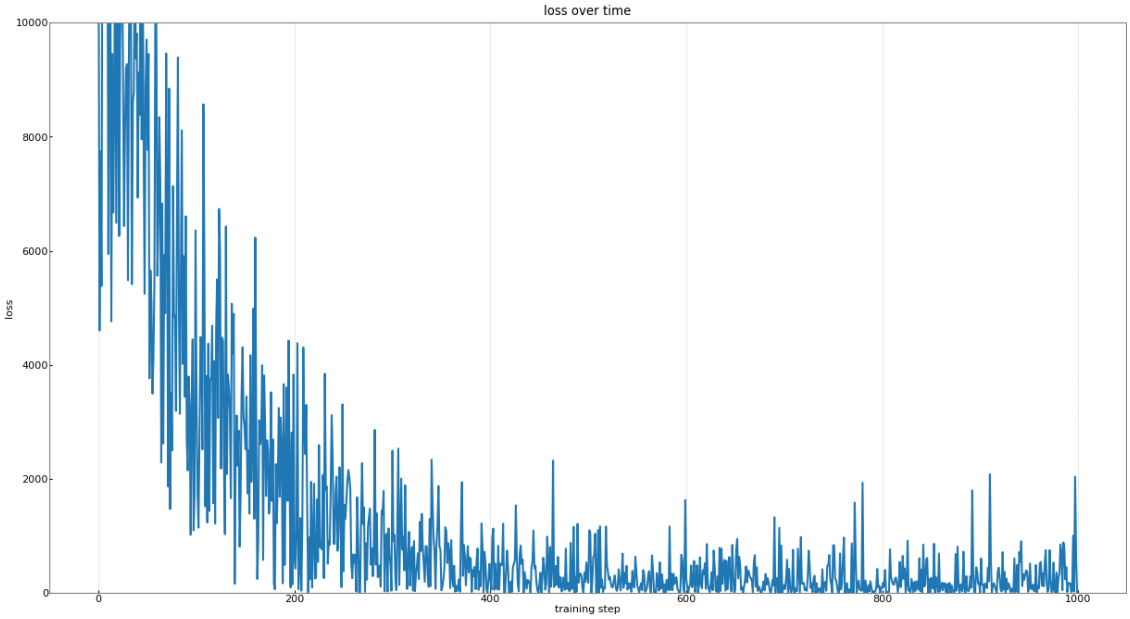
Set the values of alpha (learning rate) and num_iterations below such that you get a good result when training. You can see what a good result is by comparing to LinearRegression below.

Because stochastic gradient descent is being used, there is a lot of noise, and the result you get will depend on the run. Don't waste time "chasing the noise".

Print the results

```
training result: [148.936   1.012 -10.704]
```

Plot loss over time Looking at the change in loss over time is helpful when setting up learning rate and number of iterations.



Compute the training MSE

```
Training MSE for neural net: 447.627
```

Compare to the training MSE obtained with linear regression

```
Training MSE for linear regression: 445.280
```

Coefficients from linear regression.

```
coefficients: [149.678   1.766  -9.783]
```

1.2 Linear regression, any number of inputs

It's pretty easy to tweak your result for problem 1 so that it will work with any number of inputs.

Add your code in the cell below at the marked locations.

Note that the number of inputs can be determined from input 2D array X.

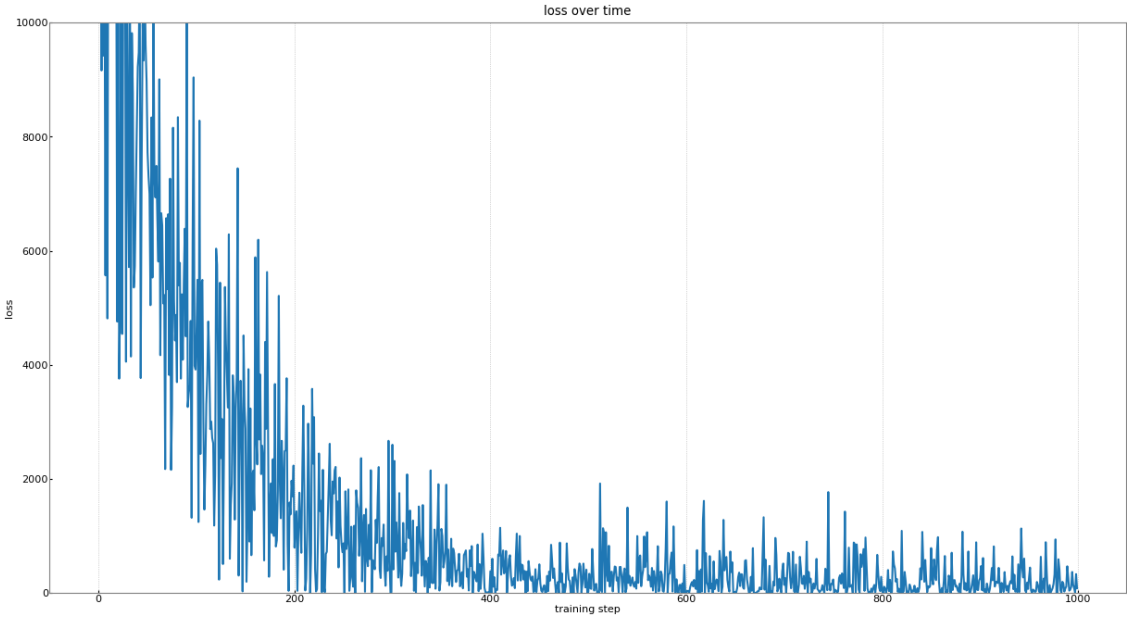
1.2.1 Problem 3

```
(array([-0.17635309, -0.08455398, -0.43493889]), [0.16038898533778548])
```

1.2.2 Problem 4

Set the values of alpha (learning rate) and num_iterations below to get a good result when training.

```
training result: [148.228   1.899  -8.158]
```




```
Training MSE for neural net: 450.158
```

1.3 Linear regression plus a hidden layer of 2 neurons, each with sigmoid activation

The figure below is a tiny neural net for regression. It has a single hidden layer with two neurons. The neurons use a sigmoid activation function (which isn't a common choice for modern neural nets).

In writing backprop code it's important to have a clear scheme for naming the weights and partial derivatives in the network.

The figure shows the names I used. Array x1 is used for the inputs to neuron 1 (the top neuron), and array x2 is used for the inputs to neuron 2. Notice that the value of array x1 is equal to the

value of array x2. Also, note that x1[0], the first input to neuron 1, is a bias input – it always has value 1.

Similarly, xs is the array of inputs to the summation node.

For the partial derivatives, I use px1[0] for the partial derivative of xs[1] with respect to x1[0]. In other words, I just use a p in front of the input name. We can associate the names of the partial derivatives with the names of the inputs because every node has exactly one output, so we don't need to name it.

1.3.1 Problem 5

Add your code in the cell below at the marked locations.

What makes backprop different here is that both the summation node and the neurons need to have their weights updated. You should update the weights of the summation node right after you compute the partial derivatives for that node.

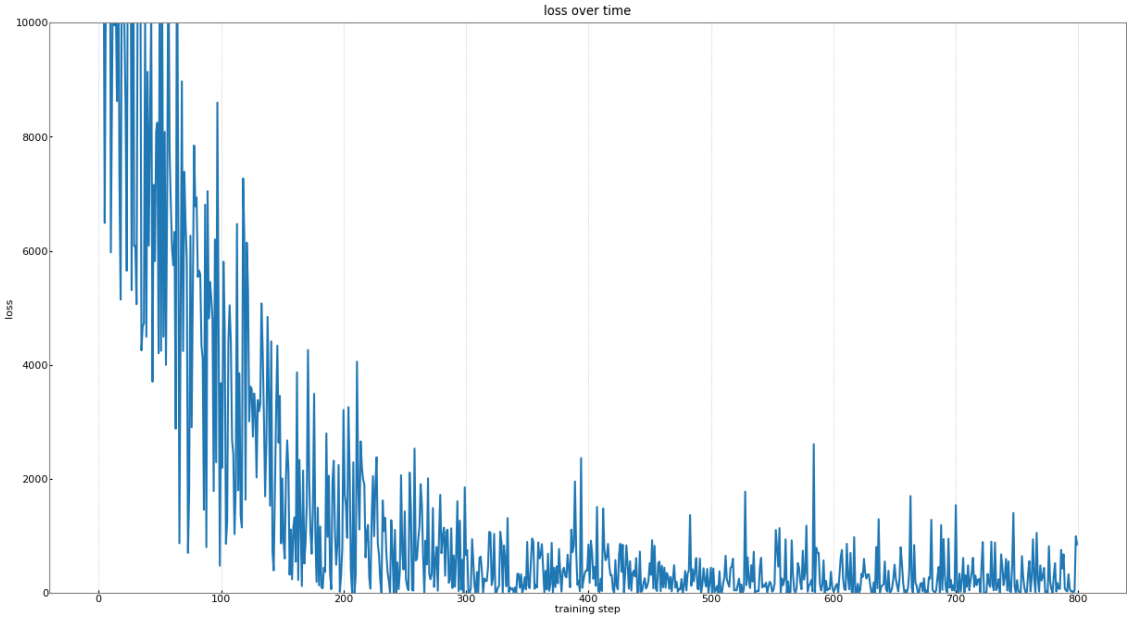
1.3.2 Problem 6

Making predications is a little more complicated now that multiple nodes have weights. Therefore, we will create a prediction function that will take all the network weights, plus input array X, and produce predictions.

Add your code in the cell below at the marked locations.

1.3.3 Problem 7

Set the values of alpha (learning rate) and num_iterations below such that you get a good result when training.



Compute MSE on the training set Don't use the mse() function here, as it only works on pair of vectors, and uses the 0.5 factor.

```
Training MSE with neural net: 534.287
```

```
1 [ 0.4114 -0.2582 -0.3531]
1 [-0.3556 -0.3642  0.473 ]

3 [0.3177 0.4477 0.2253 0.1133]
3 [ 0.2145  0.2787 -0.3563 -0.2902  0.2181]

5 [ 0.4486  0.2636 -0.3599]
5 [-0.1085 -0.3757  0.4564]

6 [1.451 1.257 1.378 1.364 1.25 ]
6 [1.078 1.09  1.102 1.094 1.071]
```