Cristopher McIntyre Garcia

In this assignment, we explore back propagation and how to compute the gradients of weights in a simple neural network. We also explore different architectures and and their performances as image classifiers.

Question 01:

Because we were not using any libraries other then math and random, the matrix multiplication and vector element wise addition functions had to be made from scratch. We are using KxK matrices and all the vectors are of size K, therefor the implementation of these functions was simple. A sigmoid function also had to be made, along with a derived version of the sigmoid function. The derivative of the sigmoid function is seen bellow in figure 1.

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Figure 1. Derivative of the sigmoid function

The loss of the function was the squared norm of the output. A forward-propagation of the model would entail computing the values of the functions provided to us in the assignment, and saving all the values in a dictionary. The dictionary includes all the variables and weights of the network. Back-propagation entails computing all the gradients of the weights that can then be used to update them. To compute the gradients of the weights, we use a simplified version of the chain derivative rule.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial \mathrm{net}_j}\frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$

Figure 2. Formula for computing the gradient of a weight

Essentially what we want to do is calculate the contribution to the loss of every weight. Finding the partial derivative of the network with respect to the weight is just the input value that is multiplied with the weight. If we use an activation function (e.g. sigmoid) the partial derivative of the output of the node with respect to the network is just the derivative of the activation function with the input value the input to the function. The final term depends on the current layer in the network where the weights are found. In our case, when computing the gradient for C, we simply calculate the partial derivative of the loss function with respect to the weights of the layer.

Therefore,

Gradient of C_{ij} = (2 * w_j) * (1) * z_i

When it comes to the other matrices, we must compute the impact of the weights of the layers that proceed them. Because both A and B connect to the same final layer without any hidden layers between, they share the same loss function partial derivative with respect to their outputs. It is essentially the sum of all the nodes' impact on the loss. Each one of these can be calculated the same way, by calculating the gradients of the node. We do this by calculating the partial derivative of the network with respect to the input.

Therefore,

Gradient of A_{ij} = x_i* derived_sig(x_j) * sum_over_j((2 * w_j) * (1) * C_{ij})

Gradient of B_{ij} = x_i* (1) * sum_over_j((2 * w_j) * (1) * C_{ij})

Note we multiply by (1) when there is not activation function.

Here are some results:

Let,

```
x =
    [0.8311474366942035,     0.8439379477805831,      0.8388421326877166]

A =
[
    [0.25129816473730215,    0.45937691785148405,     0.8846707302565932],
    [0.6881257902175721,     0.4289543556188269,      0.1266690829131678],
    [0.7904021199234799,     0.2987045256966764,      0.931388707459593]
]

B =
[
    [0.16394946490626883,    0.6448089772297525,      0.21880857546460808],
    [0.6799780955590629,     0.11349175428867442,     0.5273784554376459],
    [0.8979181883772658,     0.6031537907940281,      0.41638580170599393]
]

C =
[
    [0.6999395243144166,     0.6208118228563083,      0.8931884731417817],
    [0.8199629213367873,     0.891376816624735,       0.45207900447394345],
    [0.0552324942671899,     0.5511150479258549,      0.693654765988074]
]
```

Resulting gradients are,

```
A =
[
    [2.164841106297983,      2.6681140650967543,      1.2098228357464476],
    [2.1981558022807874,     2.7091736184596087,      1.2284407748869652],
    [2.1848830308131784,     2.69281525010984,        1.2210232780700203]
]

B =
[
    [14.089372084700397,     13.535386460481702,      8.792896243733097],
    [14.306193146636476,     13.743682248855672,      8.928209946118045],
    [14.21981035611282,      13.660695977626299,      8.87430016860969]
]

C =
[
    [14.65610661598256,      18.529402674179796,      18.78898960873385],
    [12.038309499435838,     15.219777671936383,      15.432998545805004],
    [11.676555948682925,     14.762420381467074,      14.969233926447846]
]

Loss = 44.061213140343526
```

Here is what I obtain using pyTorch’s torch.backward() function

```
A =
tensor(
[
        [2.1648,     2.6681,     1.2098],
        [2.1982,     2.7092,     1.2284],
        [2.1849,     2.6928,     1.2210]
])

B =
tensor(
[
        [14.0894,    13.5354,    8.7929],
        [14.3062,    13.7437,    8.9282],
        [14.2198,    13.6607,    8.8743]
])

C =
tensor(
[
        [14.6561,    18.5294,    18.7890],
        [12.0383,    15.2198,    15.4330],
        [11.6766,    14.7624,    14.9692]
])

Loss = tensor(44.0612, grad_fn=<SumBackward0>)
```

The results are the same, and if my results are converted to tensors, they would be identical.

With these gradients, we are able to perform gradient descent. To do so, we chose a batch size (N) to generate more input training data, and chose a learning rate (lr). We then compute the gradients with the data and updated the weights using the formula in figure 3. Doing this once would consist of one epoch, which is all we need to do to see the results.

$$\theta^{\text{new}} \quad := \quad \theta^{\text{old}} - \lambda \frac{d\mathcal{L}}{d\theta}(\theta^{\text{old}})$$

Figure 3. Gradient descent

To compute the new weights, we simply multiply our gradient by a learning rate and subtract that from the old weight. We do this over all the data points, and divide the whole by the batch size. Here are some results:

```
lr=0.01, N=1000

Loss After GD = 18.81529994150537

Weights after GD,

A =
[
        [0.24017412919517878,     0.4473755425332635,       0.8785168346064472],
        [0.6774545258293653,      0.41711389454463954,      0.12024556582964653],
        [0.7795125738251214,      0.2864881870607056,       0.9251390029462334]
]

B =
[
        [0.10792901426014781,     0.5904502701472999,       0.18448234620161366],
        [0.6244072920496712,      0.059853681113064404,     0.4932458587976601],
        [0.8411513355136166,      0.5482576731145452,       0.3815936895123655]]

C =
[
        [0.6330969007925663,      0.5357020211930207,       0.8067636563389196],
        [0.7640008862200083,      0.8200711638409246,       0.38008086520718337],
        [-0.00020800936939068394, 0.4804732043217004,       0.6220844252118314]
]
```

As can be seen, the loss has gone down drastically, from around 44 to around 18.

Results from another run are found in the Question 01/results.txt file where I used 100 epochs instead of 1 to see how low it would get.