**Introduction to Deep Neural Network – HomeWork #1**

**[Implementing Fully-connected Neural Network]**

2021313549 정성수

**Introduction**

In this assignment there are three tasks given. According to those, two fully-connected neural networks are built, one with Numpy and the other with PyTorch. Then the two networks are compared in terms of implementation itself and their results.

**Task #1**

Given neural network architecture and a pair of inputs with weights, implement neural networks using Numpy and PyTorch. The architecture of the neural networks is as follows.

* Input layer with 3 nodes
* Hideen layer with 4 nodes and ReLU activation function
* Output layer with 2 nodes and softmax activation function

Use the following weights for the inputs:

x1 = [1.0, 2.0, 3.0]

x2 = [4.0, 5.0, 6.0]

The weights for the neural network are:

w1 = [[0.1, 0.2, 0.3, 0.4], [0.5, 0.6, 0.7, 0.8], [0.9, 1.0, 1.1, 1.2]]

w2 = [[0.2, 0.1], [0.4, 0.5], [0.6, 0.2], [0.8, 0.7]]

Implement the neural networks using both Numpy and PyTorch and print the outputs of the neural networks for the given inputs. We note that we do not use bias terms (i.e., w0 = [0, … , 0].)

**Task #2**

Print out gradients of loss with respect to a specific weight from two neural networks. The neural networks are the ones you implemented in Task 1. The loss function is cross-entropy loss. The target values for the output layer are:

y1 = [0, 1] y2 = [1, 0]

Print the gradients of the loss function with respect to weight w1 for both neural networks, which are from both NumPy and PyTorch. There is no learning or update for the task 2.

**Task 1 Implementation**

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Input and weight values are implemented for both numpy and PyTorch to use array for numpy network and tensor for torch network.

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Two neural network forwarding functions are implemented in this image. There are two steps, input layer to hidden layer, and hidden layer to output layer. In PyTorch, we simply use built-in relu and softmax method for computing, but in Numpy we must implement the activation functions manually using numpy computing methods such as dot, exp, sum, and maximum. Note that when printing the tensor, gradient must be detached in order to print only the numbers.

**Task 2 Implementation**

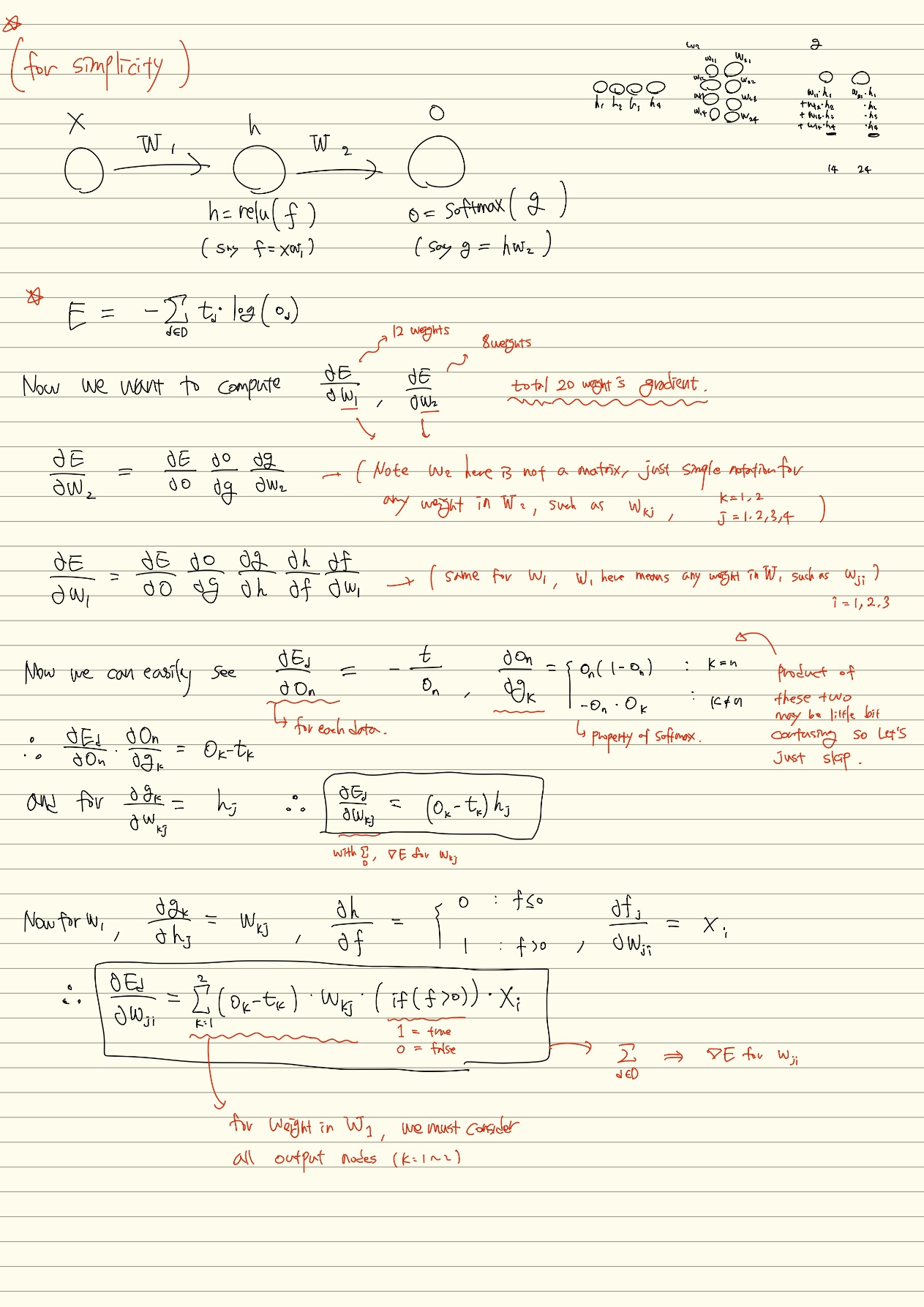
텍스트, 스크린샷, 문서, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

For task 2, cross-entropy loss function must be implemented first. Therefore, two cross-entropy loss functions were implemented easily using summation method and log method each for Numpy and PyTorch.

Now gradients must be computed for both neural networks. While PyTorch tensor can automatically accumulate gradients of every computation by setting .requires\_grad\_(True), Numpy doesn’t have any automatic gradient computing methods. So the function backward\_numpy() was created, which manually computes gradient by partial derivatives.

Fortunately, unlike the lecture slide’s complex formulas using MSE loss function, Cross-entropy loss and ReLU’s derivatives are relatively simpler for coding (but softmax’s derivative was quite difficult). The next image will explain how I derived the code’s partial derivative logic very well. This took almost an hour for me to calculate in hands.



**Task #3**

Please repeat the processes from Task 2 and update the w1 and w2 FOR 100 times (i.e., 100 epochs). In your report, provide and compare the updated w1 and w2 for both NumPy and PyTorch. Use the following learning rate for gradient descent optimization: learning\_rate = 0.01

**Task 3 implementation**

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Minor details are written in the code as comments. The key point of task 3 is to apply general gradient descent algorithms with two hyper-parameters: 100 epochs(iterations) and 0.01 learning rate. The two neural networks have exactly same structure. For each epoch, they do forward computation with given weights and inputs, then calculate the loss and compute the gradient of loss in respect of each weight. Using the gradients, update the weight for the next epoch. Unlike Numpy network, PyTorch network must initialize weights’ accumulated gradients so that the gradient of previous epoch can’t mess up later epoch’s gradient calculation. So actually there is no need for with torch\_no\_grad() block since we initialize the accumulated gradients very after updating the weights. But it seemed better to leave that for consistency and flexibility of code.

Updated results of w1 and w2 will be provided and analyzed(comparing two networks) in the next section.

**Code Result**

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트, 흑백이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**Conclusion (for task 3)**

**Interpretation of the results**

Values of weights are updated throughout the iteration. They look good since there are no outlying values which may indicate overfitting problem.

**Comparing two networks**

The results of each network are exactly the same. Even for me it’s quite surprising that the values are exactly same while every computation they have had is from two distinct libraries. I guess PyTorch’s built-in methods are almost the same for my Numpy implementation.