인공지능 5주차 과제

임베디드시스템공학과 3학년 202201673 이진성

Artificial Intelligence week5 Assignment

Forward Propagation Algorithm in C

내 용

- C언어로 Forward Propagation Algorithm 구현
- L개의 레이어를 갖는 인공 신경망
- L은 2에서 16의 값을 가짐
- 각 레이어는 N개의 노드를 가짐
- N은 2에서 256 사잉의 값을 가짐
- 각 레이어마다 노드의 수는 상이할 수 있음
- 최대한 General 한 코드로 구현할 것
- C로 구현 후 Python으로도 구현해볼 것
- CUDA를 사용한 하드웨어 가속을 통한 런타임 실험 해볼 것

Ⅰ. 기본 코드 구현

1.1. 기본 코드

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
```

기본 IO를 위한 stdio.h와 동적할당을 위한 stdlib.h, Sigmoid 연산에 필요한 Exponential을 사용하기 위한 math.h를 include 하였다.

```c
double* layerCreation(int n){
    double* layer = (double*)malloc(sizeof(double) * n);
    return layer;
}
```

layerCreation(int n)은 L개의 레이어가 각각 몇 개의 노드를 가지는지에 대한 값을 입력받고, 이에 맞춰 노드를 위한 공간을 할당받는다. 이때 할당받은 공간은 x를 연산하고 sigmoid 연산 후 덮어써진 다음 출력값으로 사용된다.

```c
double** weightTableCreation(int numberOfOutputNode, int numberOfInputNode){
    double** weightTable = (double**)malloc(sizeof(double*) * numberOfOutputNode);
    for(int i=0;i<numberOfOutputNode;i++){
        weightTable[i] = (double*)malloc(sizeof(double) * numberOfInputNode);
        printf("Enter w_%d,1 to w_%d,%d\n",i+1,i+1,numberOfInputNode);
        for(int j=0;j<numberOfInputNode;j++){
            scanf("%lf",&weightTable[i][j]);
        }
    }
    return weightTable;
}
```

double** weightTableCreation(int numberOfOutputNode, int numberOfInputNode)은 두 레이어 간의 복수개의 weight를 table 형태로 저장하기 위한 함수이다. 입력인수로 값이 출력되는 레이어의 노드의 수인 numberOfOutputNode와 값을 입력받는 레이어의 노드의 수인 numberOfInputNode를 받는다.
weightTable은 {numberOfOutputNode} x {numberOfInputNode} 사이즈의 공간을 할당받으며, weightTable은 row vector로 input을 받아 $I \times X$을 연산하여 row vector인 output을 내놓게 된다. 이를 위해 double type의 double pointer인 weightTable은 numberOfOutputNode 만큼의 메모리를 할당받은 후 각 weightTable[]은 numberOfInputNode 만큼의 메모리를 할당받아 {numberOfOutputNode} x {numberOfInputNode}의 공간을 할당받는다.
이후 double type의 double pointer 자체를 반환한다.

```
22 void matrixMultiplication(double* input,
23                            int inputSize,
24                            double** weight,
25                            double* output,
26                            int outputSize){
27     for(int i=0;i<outputSize;i++){
28         double add=0.0;
29         for(int j=0;j<inputSize;j++){
30             add = add + input[j] * weight[j][i];
31         }
32         output[i] = add;
33     }
34 }
```

matrixMultiplication()은 행렬곱을 구현하는 코드 중 두 개의 입력 중 하나를 vector로 취급하여 연산할 수 있도록 수정한 함수이다. 입력으로 input과 output vector의 pointer, 각 vector에 대한 사이즈, weight의 경우 두 레이어 간의 weight table은 2차원 배열 형태이므로 double type의 double pointer를 입력으로 받는다. 이때 행렬곱 연산의 출력인 output의 pointer를 사용하므로 별도의 return을 사용하는 것이 아닌 연산 결과를 바로 저장한다. 또한 weight의 size는 inputSize와 outputSize를 활용한다.

```
36 double sigmoid(double x){
37     return 1.0 / (1.0 + exp(-x));
38 }
```

sigmoid(double x)는 sigmoid 연산을 위해

$$\frac{1}{1+e^{-x}}$$

를 연산하고 반환한다. 이를 통해 network는 weight와의 연산 후 1차적으로 x가 저장되었다가 sigmoid 출력값으로 덮어써진다.

```
40 int main(){
41     int L;
42     printf("How many layers? Maximum 16\n");
43     scanf("%d",&L);
44     double** network = (double**)malloc(sizeof(double*) * L);
45 
46     int* N = (int*)malloc(sizeof(int)*L);
47     for(int i=0;i<L;i++){
48         printf("How many nodes in layer %d? Maximum 256\n",i+1);
49         scanf("%d",&N[i]);
50         network[i] = layerCreation(N[i]);
51     }
```

main 함수에서는 레이어의 개수인 L을 입력받고, 이를 바탕으로 각 레이어의 노드의 수를 저장하기 위한 N과 모든 노드의 x와 sigmoid 값을 저장하기 위한 network를 할당받는다. 이때 network[]는 layerCreation 함수를 사용하여 할당받는다.

```
53      double*** weight = (double***)malloc(sizeof(double**) * (L-1));
54      for(int i=0;i<(L-1);i++){
55          printf("Create weight table for layer %d to %d\n",i+1,i+2);
56          weight[i] = weightTableCreation(N[i],N[i+1]);
57      }
```

각 레이어 간의 가중치의 경우 두 레이어 내에 있는 노드들의 개수가 m, n이라고 할 때 $m \times n$사이즈를 갖는 배열 형태로 나타내게 된다. 이때 두 레이어 간에 배열이 나타나며, 총 레이어의 수는 L개 이므로 총 $L-1$개의 배열이 필요하다. 이 모든 가중치들을 하나의 포인터를 사용하여 나타내기 위해 double type의 triple pointer를 사용하였으며, L개의 double pointer를 위한 메모리를 할당받는다. weight[]는 weightTableCreation 함수를 사용하여 weight table을 할당받고 가중치를 입력받아 weight table을 구성할 수 있도록 하였다.

```
62      printf("Enter %d input\n",N[0]);
63      for(int i=0;i<N[0];i++)scanf("%lf",&network[0][i]);
```

Weight 입력 후 신경망의 입력값을 받으며, 이는 바로 network[0][]에 저장된다.

```
67      for(int i=0;i<(L-1);i++){
68          matrixMultiplication(network[i],N[i],weight[i],network[i+1],N[i+1]);
69          for(int j=0;j<N[i+1];j++)network[i+1][j] = sigmoid(network[i+1][j]);
70      }

74      printf("print output\n");
75      for(int i=0;i<N[L-1];i++)printf("%f ",network[L-1][i]);
```

입력 완료 후 matrixMultiplication 함수와 sigmoid 함수를 연속적으로 호출해서 최종 출력값에 대한 연산을 끝낸다. 직후에 출력값을 순차적으로 출력한다.

```
79      {
80          for(int i=0;i<(L-1);i++){
81              for(int j=0;j<N[i];j++)free(weight[i][j]);
82              free(weight[i]);
83          }
84          free(weight);
85
86          free(N);
87
88          for(int i=0;i<L;i++)free(network[i]);
89          free(network);
90          return 0;
91      }
92  }
```

모든 연산을 종료한 후 할당된 모든 변수를 순차적으로 할당해제한 후 프로세스를 종료한다.

1.2. 동작 결과

프로세스를 실행하면 레이어의 수를 입력하며, 이에 따라 각 레이어의 노드의 수를 입력받는다. 입력받은 노드의 수를 이용하여 각 레이어 간의 가중치 테이블을 입력받은 후 신경망의 입력값을 받아 출력값을 출력하고 종료한다.

```
nabi@ijinseong-ui-MacBookPro 20240930 % ./main
How many layers? Maximum 16
4
How many nodes in layer 1? Maximum 256
4
How many nodes in layer 2? Maximum 256
3
How many nodes in layer 3? Maximum 256
4
How many nodes in layer 4? Maximum 256
2
Create weight table for layer 1 to 2
Enter w_1,1 to w_1,3
0.1 0.3 0.5
Enter w_2,1 to w_2,3
0.2 0.4 0.6
Enter w_3,1 to w_3,3
0.1 0.3 0.5
Enter w_4,1 to w_4,3
0.2 0.4 0.6
Create weight table for layer 2 to 3
Enter w_1,1 to w_1,4
0.1 0.2 0.5 0.7
Enter w_2,1 to w_2,4
0.3 0.4 0.5 0.6
Enter w_3,1 to w_3,4
0.7 0.2 0.3 0.1
Create weight table for layer 3 to 4
Enter w_1,1 to w_1,2
0.7 0.8
Enter w_2,1 to w_2,2
0.5 0.6
Enter w_3,1 to w_3,2
0.1 0.2
Enter w_4,1 to w_4,2
0.3 0.4
Enter 4 input
1.0 2.0 1.0 3.0
print output
0.761399 0.810649
```

그림 1. 기본 코드 동작 결과

그림 1에서 사용한 신경망 모델은 2023년도 2학년 1학기 전공 강의인 선형 시스템의 과제 04에서 제시한 모델을 사용하였다. 이는 그림 2를 통해 확인할 수 있다. 그림 2에서 제시한 입력값과 가중치에 따른 출력값이 일치하는 것을 확인할 수 있다.

그림 1의 경우 검은 배경에 흰 글자였으나 문서의 전체적인 가시성을 위하여 색상을 반전하였다.

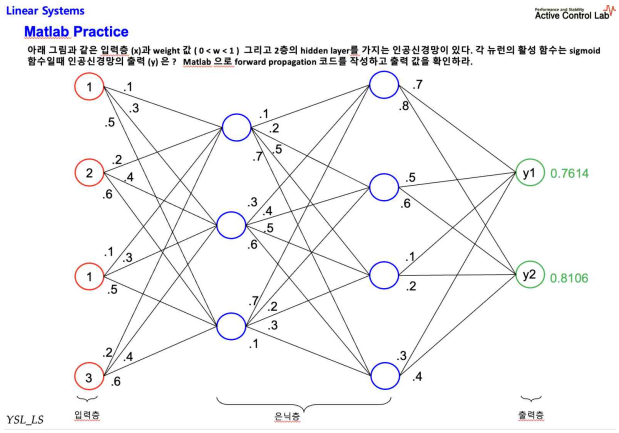

그림 2. 그림 1에 적용한 신경망

II. 랜덤 가중치 및 입력값과 16개 레이어 256개 노드

2.1. 코드 구현

```
4 #include<time.h>
```

시간 측정 및 랜덤값 생성을 위한 time.h를 include 하였다.

```
11 double** weightTableCreation(int numberOfOutputNode, int numberOfInputNode){
12     double** weightTable = (double**)malloc(sizeof(double*) * numberOfOutputNode);
13     for(int i=0;i<numberOfOutputNode;i++){
14         weightTable[i] = (double*)malloc(sizeof(double) * numberOfInputNode);
15         // printf("Enter w_%d,1 to w_%d,%d\n",i+1,i+1,numberOfInputNode);
16         for(int j=0;j<numberOfInputNode;j++){
17             // scanf("%lf",&weightTable[i][j]);
18             weightTable[i][j]=(double)rand()/RAND_MAX - 0.5;
19         }
20     }
21     return weightTable;
22 }
```

출력값 단순화(line15)와 가중치 입력이 아닌(line17) 랜덤값을 통한 가중치 생성을 위해 rand()를 사용하였다(line18). 이때 -0.5를 하지 않았을 때 값이 대부분 1.0이 나옴에 따라 sigmoid함수를 통하기 전의 값이 지나치게 커짐으로 판단하였으며 값을 인위적으로 0.5만큼 감소시킨 값을 가중치로 사용하였다.

```
45     int L;
46     // printf("How many layers? Maximum 16\n");
47     // scanf("%d",&L);
48     L=16;
49     double** network = (double**)malloc(sizeof(double*) * L);
50 
51     int* N = (int*)malloc(sizeof(int)*L);
52     for(int i=0;i<L;i++){
53         // printf("How many nodes in layer %d? Maximum 256\n",i+1);
54         // scanf("%d",&N[i]);
55         N[i]=256;
56         network[i] = layerCreation(N[i]);
57     }
58 
59     double*** weight = (double***)malloc(sizeof(double**) * (L-1));
60     for(int i=0;i<(L-1);i++){
61         // printf("Create weight table for layer %d to %d\n",i+1,i+2);
62         weight[i] = weightTableCreation(N[i],N[i+1]);
63     }
```

L=16, N=256으로 고정하였으며(line48, line55), 모든 출력과 입력은 삭제하였다.

```
67      // printf("Enter %d input\n",N[0]);
68      for(int i=0;i<N[0];i++){
69          // scanf("%lf",&network[0][i]);
70          network[0][i]=(double)rand()/RAND_MAX - 0.5;
71      }
```

입력값 또한 랜덤으로 생성하도록 하였다(line70).

```
73      clock_t start = clock();
74
75      for(int i=0;i<(L-1);i++){
76          matrixMultiplication(network[i],N[i],weight[i],network[i+1],N[i+1]);
77          for(int j=0;j<N[i+1];j++)network[i+1][j] = sigmoid(network[i+1][j]);
78      }
79
80      clock_t end = clock();
```

연산시간 측정을 위해 clock_t start, end를 사용하였으며, 이때 연산시간을 측정하는 영역은 for 문 내에서 행렬곱과 sigmoid 연산을 하는 영역으로 지정하였다.

```
82      // printf("print output\n");
83      // for(int i=0;i<N[L-1];i++)printf("%f ",network[L-1][i]);
84
85      double duration = (double)(end - start) / CLOCKS_PER_SEC;
86
87      printf("Execution time: %lf seconds\n", duration);
```

연산시간 측정을 제외한 출력구문을 주석처리하였으며, 연산시간을 출력할 수 있도록 하였다.
후술할 CUDA 연산과의 시간차이를 계산하기 위해 동일한 컴파일러로 컴파일하고자 하였으며, 이를 위해 파일 확장자를 NVIDIA의 CUDA 프로그래밍 확장자인 .cu로 통일하였으며, nvcc 컴파일러를 사용하였다.

III. CUDA 가속의 구현

3.1. 활용 하드웨어

I5-11300H CPU와 RTX 3050 Laptop GPU를 사용하는 노트북을 사용하였으며 운영체제는 윈도우 OS를 사용하였다. CUDA 활용을 위해 NVIDIA CUDA Toolkit을 설치하여 활용하였다.

3.2. GPT 활용

CUDA를 활용한 하드웨어 가속을 사용하기 위한 자료를 서치하였으며, 해당 부분을 모두 이해하고 코드를 작성하기에는 제출 기한 내에 코드 작성 및 실험을 진행하기 어렵다는 판단을 내렸다. 이에 따라 2.1.에서 구현한 코드를 바탕으로 CUDA 가속을 활용하는 코드를 작성할 수 있도록 GPT를 활용하였다.

3.3. 코드 구현

```
5  #include <cuda_runtime.h>
```

CUDA 라이브러리 사용을 위해 cuda_runtime.h를 include 하였다.

```
9  __global__ void matrixMultiplyKernel(double *d_input,
10                                       double *d_weight,
11                                       double *d_output,
12                                       int inputSize,
13                                       int outputSize) {
14     int row = blockIdx.x * blockDim.x + threadIdx.x;
15
16     if (row < outputSize) {
17         double sum = 0.0;
18         for (int j = 0; j < inputSize; j++) {
19             sum += d_input[j] * d_weight[j * outputSize + row];
20         }
21         d_output[row] = sum;
22     }
23 }
24
25 __global__ void sigmoidKernel(double *d_data, int size) {
26     int idx = blockIdx.x * blockDim.x + threadIdx.x;
27     if (idx < size) {
28         d_data[idx] = 1.0 / (1.0 + exp(-d_data[idx]));
29     }
30 }
```

2.1.에서 연산시간 측정 기준으로 삼은 행렬곱과 sigmoid 연산을 위한 두 함수를 global로 선언하고 GPU가 연산할 수 있도록 하였다.

```
    // CUDA 관련 변수 선언
    double *d_input, *d_output, **d_weightList;
    CUDA_CALL(cudaMalloc((void**)&d_input, sizeof(double) * N[0]));
    CUDA_CALL(cudaMalloc((void**)&d_output, sizeof(double) * N[1]));

    // 가중치 배열을 위한 포인터 배열 생성 및 데이터 복사
    d_weightList = (double**)malloc(sizeof(double*) * (L - 1));
    for (int layer = 0; layer < (L - 1); layer++) {
        int inputSize = N[layer];
        int outputSize = N[layer + 1];

        // 가중치 2차원 배열을 1차원 배열로 변환 후 디바이스로 복사
        double *weightFlat = (double*)malloc(sizeof(double) * inputSize * outputSize);
        for (int i = 0; i < inputSize; i++) {
            for (int j = 0; j < outputSize; j++) {
                weightFlat[i * outputSize + j] = weight[layer][j][i];
            }
        }
        CUDA_CALL(cudaMalloc((void**)&d_weightList[layer], sizeof(double) * inputSize *
outputSize));
        CUDA_CALL(cudaMemcpy(d_weightList[layer], weightFlat, sizeof(double) * inputSize *
outputSize, cudaMemcpyHostToDevice));
        free(weightFlat);
    }
```

기존에 기본 malloc으로 할당받은 자원을 cudaMalloc으로 할당받음으로서 GPU의 메모리에 할당받을 수 있도록 하였다.

```
92      clock_t start = clock();
93
94      for (int layer = 0; layer < (L - 1); layer++) {
95          int inputSize = N[layer];
96          int outputSize = N[layer + 1];
97
98          // 입력 레이어를 디바이스로 복사
99          CUDA_CALL(cudaMemcpy(d_input, network[layer], sizeof(double) * inputSize,
    cudaMemcpyHostToDevice));
100
101         // 행렬 곱셈을 위한 CUDA 커널 호출
102         int blockSize = 256;
103         int gridSize = (outputSize + blockSize - 1) / blockSize;
104         matrixMultiplyKernel<<<gridSize, blockSize>>>(d_input, d_weightList[layer], d_output,
    inputSize, outputSize);
105         CUDA_CALL(cudaDeviceSynchronize());
106
107         // 시그모이드 활성화 함수를 위한 CUDA 커널 호출
108         sigmoidKernel<<<gridSize, blockSize>>>(d_output, outputSize);
109         CUDA_CALL(cudaDeviceSynchronize());
110
111         // 결과를 호스트로 복사
112         CUDA_CALL(cudaMemcpy(network[layer + 1], d_output, sizeof(double) * outputSize,
    cudaMemcpyDeviceToHost));
113     }
114
115     clock_t end = clock();
```

clock_t start, end 내에서는 행렬곱과 sigmoid 연산만을 할 수 있도록 프롬프트를 작성하였으며, CUDA를 통해서 동작하도록 하였다.

```
92      for (int layer = 0; layer < (L - 1); layer++) {
93          CUDA_CALL(cudaFree(d_weightList[layer]));
94      }
95      free(d_weightList);
96      CUDA_CALL(cudaFree(d_input));
97      CUDA_CALL(cudaFree(d_output));
```

할당 해제에 있어 cudaMalloc으로 할당받은 메모리는 cudaFree를 통해 할당해제 되었다.

Ⅳ. CPU연산과 CUDA 가속의 비교

4.1. 연산 시간 출력 결과

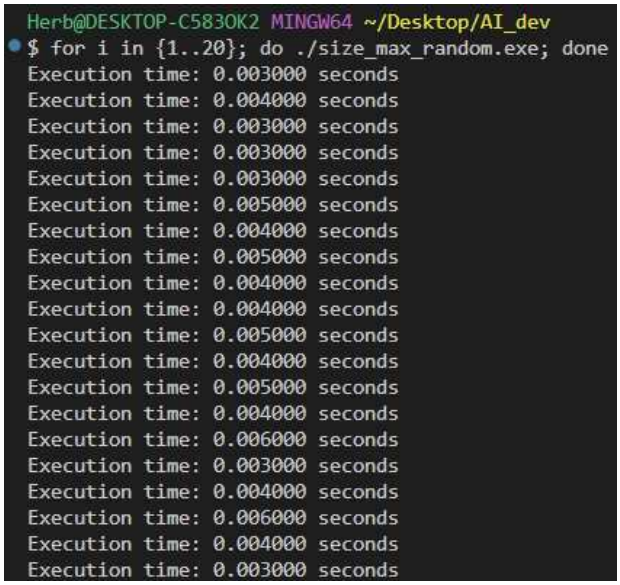

그림 3. 윈도우 OS의 CPU 연산 시간 출력 결과

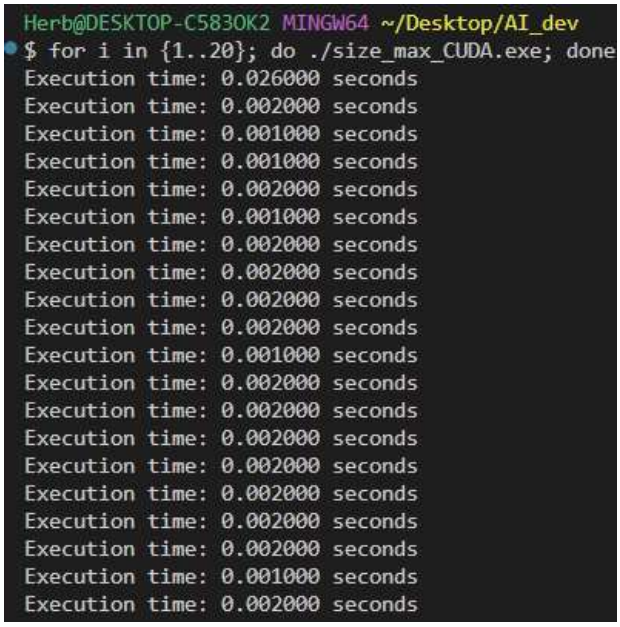

그림 4. 윈도우 OS의 CUDA 가속 연산 시간 출력 결과

최초 코드 작성시 GPU의 메모리에 복제하는 코드를 시간 측정에 포함하였으나 해당 시간에 따라 CPU 연산 대비 100배 가까운 연산시간을 보여 비교대상을 잘못 판단함을 인지하였다.

이에 따라 행렬곱과 Sigmoid 연산만 측정시간에 포함하기로 하였으며, 이에 따라 그림 3과 그림 4와 같은 결과값을 얻을 수 있었다.

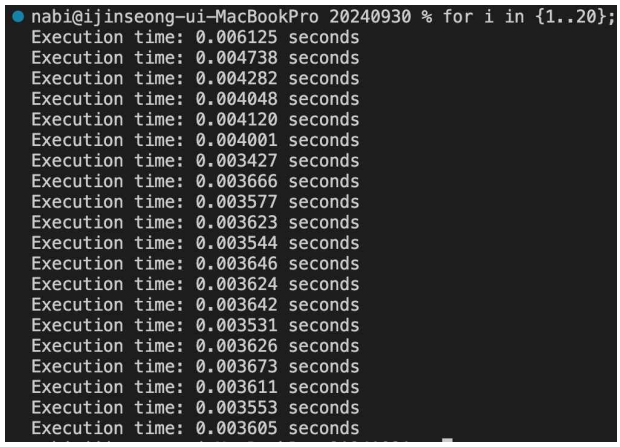

그림 5. Mac OS의 GCC 컴파일 동작 연산 시간 출력 결과

Mac OS에서 CPU 연산 코드를 GCC로 컴파일하여 동작한 경우 시간이 소숫점 단위로 정상적으로 나오는 것을 확인할 수 있었으나, 윈도우 OS에서 동작할 때 0.001초 단위로만 출력되는 점에 따라 정확한 연산시간의 비율을 측정하는 것은 무리가 있어 보인다.

그럼에도 불구하고 CPU 연산과 CUDA 가속의 연산시간을 비교하였을 때 0.003~0.005초의 연산시간을 보이는 CPU 연산 대비 CUDA 가속 프로세스에서 0.001~0.002초의 연산시간을 보이는 것으로 하드웨어 가속이 정상적으로 동작함을 확인할 수 있다. 다만 그림 4의 첫 번째 실행결과에서 0.026초대가 나온 부분에 대한 분석이 필요하다.