1 Explaining Code

Front Propagation이란 입력을 받고 신경망에서의 가중치를 거쳐 결과를 내는 것이다. 지금은 파일에 값을 넣고 정보들을 읽어 가중치를 미리 저장하고 연산을 거쳐 결과를 내는 방식을 사용할 것이다.

```c
int main()
{
    int nLayer, *NodeInfo;
    Layer *Layers;
    char filename[100];
    scanf("%s", filename);
    FILE *file = fopen(filename, "r");

    if (file == NULL)
    {
        perror("Error opening file");
        return 1;
    }

    if (ReadLayerInfo(file, &nLayer, &NodeInfo, &Layers) != 0)
    {
        fclose(file);
        return 1;
    }

    AllocateWeights(file, nLayer, NodeInfo, Layers);
    ReadInputValues(file, NodeInfo, Layers);
    fclose(file);

    // Perform forward propagation
    ForwardPropagation(nLayer, Layers);

    // Print the final layer's input values (output of the networ
    PrintOutputLayer(nLayer, Layers);

    // Free allocated memory
    FreeMemory(nLayer, Layers, NodeInfo);

    return 0;
}
```

```c
typedef struct Layer
{
    int currentSize; // currentSize is number of nodes in current layer
    int nextSize;    // nextSize is number of nodes in next layer
    float **weights; // weights matrix [next_size x current_size]
    float *input;    // store the input values (from previous layer or init
} Layer;
```

먼저, Layer를 구조체로 선언한다. 구조체는 현재 노드를 몇개 가지고 있는지, 다음 레이어는 몇개인지, 가중치 행렬을 동적할당하기 위한 이중 포인터, 노드들의 값은 열벡터이므로 일차원 배열로 사용한다.

이제 main에서 레이어의 개수를 담을 nLayer, 각 레이어의 노드 개수를 가지는 일차원 배열을 위한 int* NodeInfo를 선언한다. Layer도 일종의 일차원 배열로 넣어둘 예정이므로 포인터로 선언해준다. 파일을 읽어야 하기 때문에 filename[100] 정도로 넉넉하게 선언해주고 지금의 경우 info1.txt, info2.txt를 입력해주면 프로그램이 돌아가게 된다.

먼저 file이 없어서 NULL을 반납하게 된다면 파일 열기 실패로 에러를 출력시켜주고 그 다음은 ReadLayerInfo 함수를 사용하게 된다.

```c
// Read nLayer, NodeInfo array, Layers array
int ReadLayerInfo(FILE *file, int *nLayer, int **NodeInfo, Layer **Layers)
{
    if (file == NULL)
    {
        perror("Reading file fail");
        return 1;
    }
    // Reading the number of layers
    fscanf(file, "%d", nLayer);
    *NodeInfo = (int *)malloc(sizeof(int) * (*nLayer));

    // Reading the number of nodes in each layer
    for (int i = 0; i < *nLayer; i++)
    {
        fscanf(file, "%d", &(*NodeInfo)[i]);
    }

    // Dynamic allocation of layer array
    *Layers = (Layer *)malloc((*nLayer) * sizeof(Layer));
    return 0;
}
```

이 함수는 param으로 File*, int*, int**, Layer**을 선언하는데 main에 있는 각 변수들을 접근하기 위해서이다. file이 NULL인지 확인하는 부분은 제거해도 용이하다.

```
4
2 3 3 2
0.5 0.6 0.1 0.2 0.3 0.4
0.5 0.6 0.7 0.8 0.9 0.1 0.2 0.3 0.4
0.1 0.2 0.3 0.4 0.5 0.6
1.0 0.5
```

텍스트 파일을 보면 첫 줄은 레어어의 개수, 둘째 줄은 레이어마다의 노드 각각의 개수, 가중치가 필요한 곳은 3군데이므로 세줄이 존재하는데, c언어에서 배열을 사용할 때 cache 관점에서 사용할 것이므로 가중치 배열의 크기는 이 예시에서 3*2, 3*3, 2*3을 사용한다. 즉 다음 레이어의 크기 * 현재 레이어의 크기 형태의 행렬을 사용하고 현재 노드들의 값이 뒤에 붙어 곱해지면 다음 레이어의 노드들의 크기와 맞기 때문이다. 순서대로 첫 노드에서 다음 레이어의 첫번째 노드의 합에 이용되는 가중치, 두번째 노드에서 다음 레이어의 첫번째 노드로 가는 가중치 이렇게 배열되어 있다.

다시 코드로 돌아가보면 nLayer의 주소를 가져왔기 때문에 그 곳에 4를 넣어주고 NodeInfo는 *nLayer의 크기만큼 int 일차원 배열을 만들어 낸다.

이제 두번째 줄을 읽어서 NodeInfo의 각 index에 넣어주는 과정을 거치고 Layer 구조체의 배열도 *nLayer만큼 만들어 주도록 한다.

```
AllocateWeights(file, nLayer, NodeInfo, Layers);
ReadInputValues(file, NodeInfo, Layers);
fclose(file);

// Perform forward propagation
ForwardPropagation(nLayer, Layers);

// Print the final layer's input values (output of the network)
PrintOutputLayer(nLayer, Layers);

// Free allocated memory
FreeMemory(nLayer, Layers, NodeInfo);
```

```
// Allocate weights implementation
void AllocateWeights(FILE *file, int nLayer, int *NodeInfo, Layer *layers)
{
    for (int i = 0; i < nLayer - 1; i++)
    {
        layers[i].currentSize = NodeInfo[i];
        layers[i].nextSize = NodeInfo[i + 1];

        int sRow = layers[i].nextSize;
        int sCol = layers[i].currentSize;

        layers[i].weights = (float **)malloc(sRow * sizeof(float *));
        for (int j = 0; j < sRow; j++)
        {
            layers[i].weights[j] = (float *)malloc(sCol * sizeof(float));
            // Read the weights
            for (int k = 0; k < sCol; k++)
            {
                fscanf(file, "%f", &layers[i].weights[j][k]);
            }
        }

        layers[i + 1].input = (float *)malloc(sRow * sizeof(float)); // Alloca
    }
    int OutPutLayerIndex = nLayer - 1;
    layers[OutPutLayerIndex].currentSize = NodeInfo[OutPutLayerIndex];
    layers[OutPutLayerIndex].nextSize = 0;
    layers[OutPutLayerIndex].weights = NULL;
}
```

이제 AllocateWeighs와 ReadInputValues를 설명하면, file에서 아까 있던 세 줄을 통해 각 레이어의 가중치 배열을 할당해주기를 먼저 시작하자. 배열의 크기는 다음 레이어의 노드 길이 * 현재 레이어의 노드 길이가 되고 행렬이기 때문에 행이 다음 레이어의 노드, 열이 현재 레이어의 노드 길이가 된다. 현재 크기는 NodeInfo의 인덱스의 값을 가져오고, 다음 크기는 다음 인덱스의 값을 가져온다.

여기서 반복문이 index nLayer - 2까지만 도는 이유는 output Layer는 다음 곳을 참조할 필요가 없기 때문이다. 또한 output Layer는 가중치 행렬이 없고 업데이트된 input을 출력해주면 그만이기 때문이다.

그래서 sRow, sCol을 보면 위에 행과 열의 정의에 의해서 변수로 지정되고 먼저 현재 Layer의 weight를 동적할당 해준다. 이차원 동적할당은 **로 *들을 행만큼 만들어주고 *로 열만큼의 float 크기의 배열을 만들어주면 된다. i는 레이어를 가리키고, j는 각 행마다, k는 각 열마다기 때문에 layers[i]의 weight[j][k]에 파일을 각각 읽어서 저장해 준다. j가 행의 크기, k가 열의 크기만큼 돌기 때문에 2 3 3 2 0.5 0.6 0.1 0.2 0.3 0.4 의 경우 3*2의 크기이므로 0.5 0.6;0.1 0.2;0.3 0.4 순서대로 저장되며 두 개의 input이 각 행과 행렬연산을 하며 다음

레이어의 세개의 노드에 저장되게 되는 구조다. 또한 i의 for문을 돌면서 다음 레이어의 input은 sRow 즉 다음 레이어의 사이즈만큼 할당을 해주게 된다.

이 함수를 통해 단계적으로 0번 레이어(첫번째 레이어)부터 가중치를 할당시키고, 1번 레이어의 input도 할당해준다. 이것을 따로 빼준 이유는 output layer는 input이 필요하고 여기에 값을 받은 것을 따로 함수를 정의해서 출력시켜주면 되기 때문이다.

output layer의 인덱스를 따로 저장해주었다. 원래는 int i를 지역변수로 선언하고 for문을 돌고 나서 사용하려 했으나 변수 이름을 지정해 주는 것이 더 보기 좋아서 이렇게 저장하고 현재 사이즈와 nextSize, weight들을 초기화를 따로 해주었다.

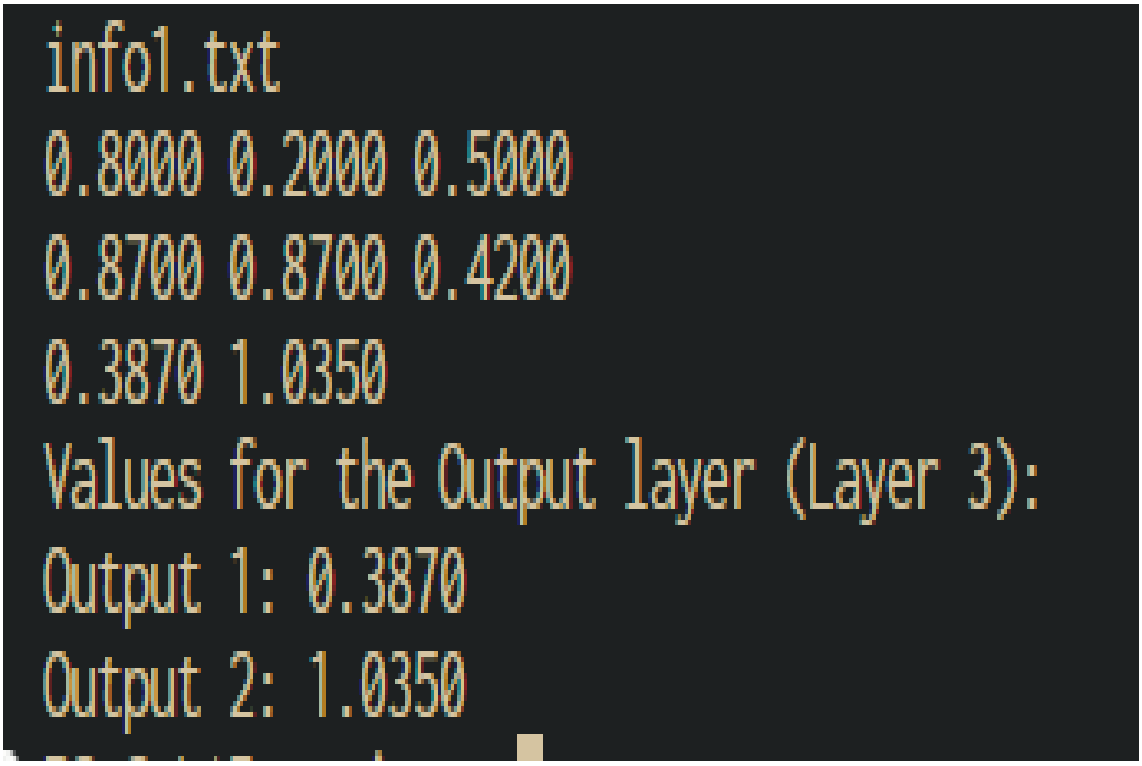
```
// Read input values into the first layer's input
void ReadInputValues(FILE *file, int *NodeInfo, Layer *layers)
{
    int sInput = NodeInfo[0];
    layers[0].input = (float *)malloc(sInput * sizeof(float));
    for (int i = 0; i < sInput; i++)
    {
        fscanf(file, "%f", &layers[0].input[i]);
    }
}
```

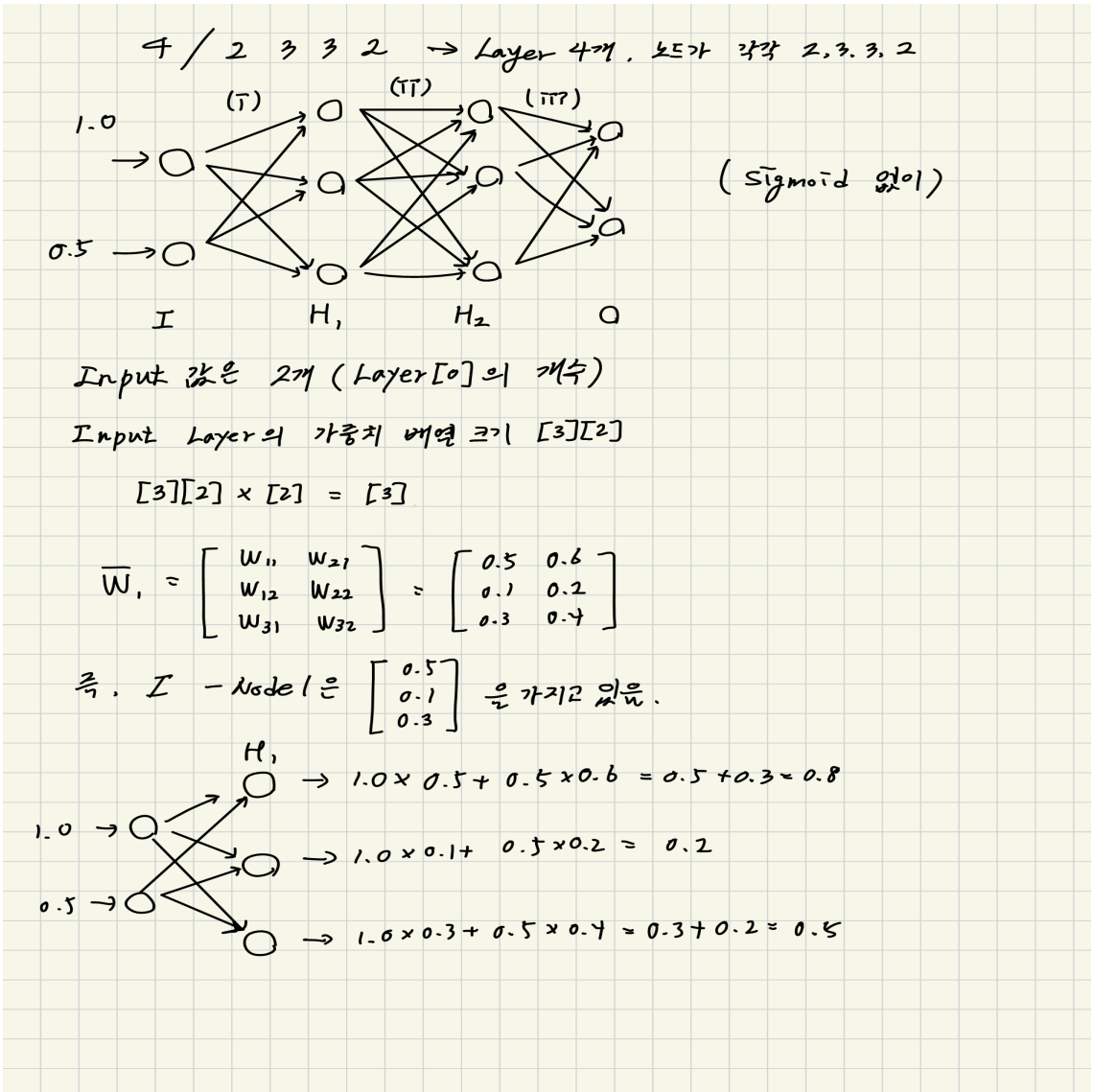
이것은 입력값들을 초기화해주는 함수로 볼 수 있다. sInput 변수는 NodeInfo[0], 즉 첫 레이어의 노드 개수를 저장한다. 아까는 다음 레이어의 input을 할당해줬을 뿐 첫번째 레이어는 할당을 받지 못하였다. 그렇기 때문에 input 배열을 할당 시켜주고 sInput만큼 input 입력값을 받아주는 역할을 한다.

```
// Forward propagation
void ForwardPropagation(int nLayer, Layer *layers)
{
    for (int i = 0; i < nLayer - 1; i++)
    {
        for (int j = 0; j < layers[i].nextSize; j++)
        {
            float sum = 0.0;
            for (int k = 0; k < layers[i].currentSize; k++)
            {
                sum += layers[i].input[k] * layers[i].weights[j][k];
            }
            layers[i + 1].input[j] = roundToDecimals(sum); // Next layer's
            printf("%.4f ", layers[i + 1].input[j]);
        }
        printf("\n");      You, 18 minutes ago • Uncommitted changes
    }
}
```

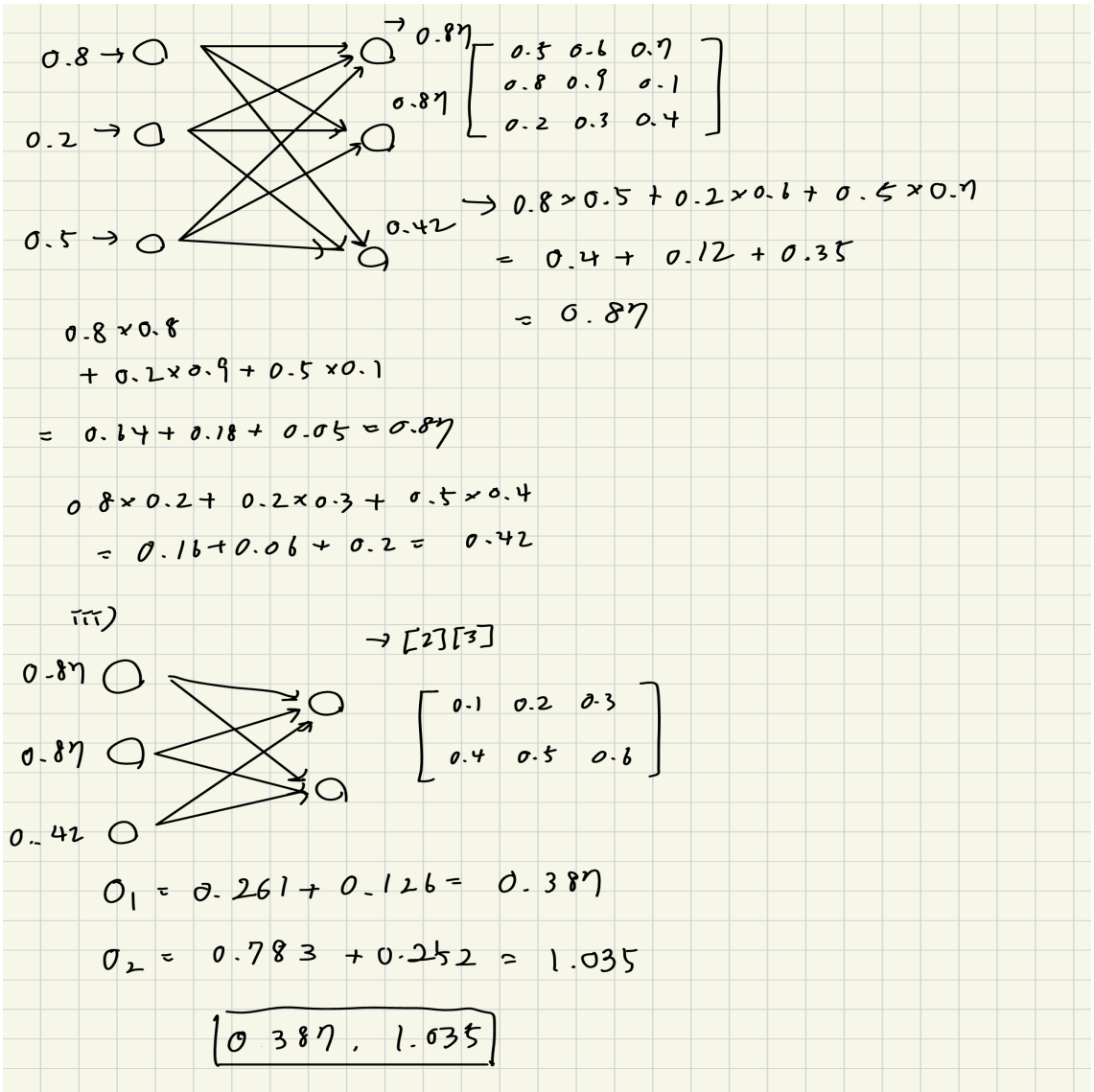
이제 순전파를 시작하는 함수가 등장하였다. 원래 식은 이것이 아닌데 info1.txt를 계산하는 방식을 먼저 보여주고, 미리 선언해둔 반올림 함수와 sigmoid를 활용해볼 것이다. 일단 계산이 맞다면 sigmoid된 값들을 반올림해서 사용하는 과정도 들어맞기 때문에 복잡한 계산을 해보는 것보다 간단한 산수부터 해본 이후 코드를 보여주는 것이 좋을 것 같다.

순전파를 할때 input부터 output전까지의 레이어는 모두 가중치 행렬을 가지므로 i는 0부터 nLayer - 2까지 돌고 j는 0부터 다음 레이어의 노드수만큼 돌면서 먼저 sum은 0.0으로 초기화 해준 후 k는 현재 레이어의 노드 수만큼 반복문을 도는데 이유는 현재 레이어가 노드 2개, 다음 레이어가 노드 3개를 가진다면 3*2의 가중치 행렬을 가지며, input은 [2]의 size를 가지므로 넘겨줄 땐 [3]의 크기를 넘겨줘야 하기 때문에 [j][k]와 [k]가 곱해져 sum에 계속 저장되는 구조를 가진다. j가 다음 레이어의 노드수를 가리키므로 layer[i+1].input[j]는 다음 레이어의 노드에 값을 넣어주는 과정이고 roundToDecimals는 4자리 반올림을 해주는 함수이다.


info1.txt
0.8000 0.2000 0.5000
0.8700 0.8700 0.4200
0.3870 1.0350
Values for the Output layer (Layer 3):
Output 1: 0.3870
Output 2: 1.0350



이 그림은 손으로 그린 것인데 2 3 3 2 의 노드 개수를 sigmoid 없이 우선 계산해보았다. 1.0을 받은 노드는 가중치를 0.5, 0.1, 0.3을 가지고 0.5를 입력받은 노드는 가중치를 0.6, 0.2, 0.4를 가지고 있는데 H1 즉 hidden layer1은 노드 세 개가 있고 첫번째 노드는 1.0*0.5 + 0.5*0.6을 해서 0.8이 되는 과정 같은 것들을 적어 두었다. 프로그램과 수식이 맞는 것을 보여주는 일종의 과정이다.



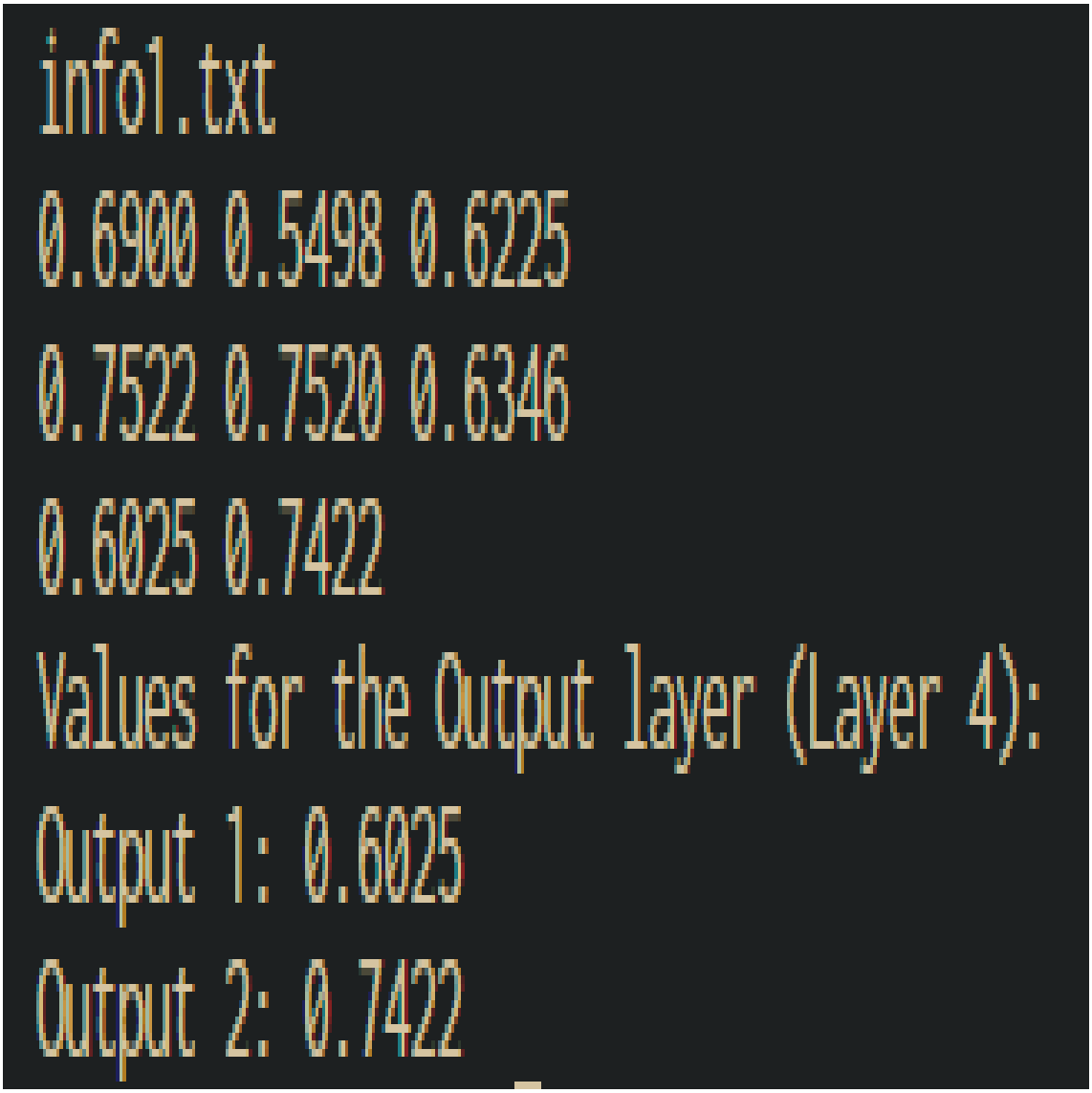
H1의 input들을 활용하여 H2의 input을 계산하는 과정이 이번 그림이다. 그렇게 해서 나온 H2의 input들로 가중치 행렬을 사용하면 마지막 output 레이어의 값들이 저장되고, 그것이 그대로 나오면 프로그램과 수식이 완전히 일치하게 된다.

```
float roundToDecimals(float value)
{
    return roundf(value * 10000) / 10000;
}

float sigmoid(float sum)
{
    return roundToDecimals(1.0 / (1.0 + exp(-sum)));
}

// Forward propagation
void ForwardPropagation(int nLayer, Layer *layers)
{
    for (int i = 0; i < nLayer - 1; i++)
    {
        for (int j = 0; j < layers[i].nextSize; j++)
        {
            float sum = 0.0;
            for (int k = 0; k < layers[i].currentSize; k++)
            {
                sum += layers[i].input[k] * layers[i].weights[j][k];
            }
            layers[i + 1].input[j] = sigmoid(sum); // Next layer's input is
            printf("%.4f ", layers[i + 1].input[j]);
        }
        printf("\n");
    }
}
```

실제로는 sigmoid를 넣어서 저장하므로 아까와 다르게 sigmoid(sum)을 사용한다 위에 sigmoid함수가 있고 roundToDecimals는 sigmoid값을 우선 10000을 곱하여 소수점 네자리들을 끌어내주고 반올림 시켜주는 역할을 한다. 그리고 다시 10000으로 나누어 소숫점 네자리까지 반올림을 하는 효과를 보여주게 된다.



이번에는 sigmoid를 적용시켜 출력한 결과이다. 아까도 위에 써놨지만 일반 계산이 잘 들어맞기 때문에 sigmoid한 결과도 sigmoid 식이 정의대로이므로 맞다고 볼 수 있을 것이다.

```c
// Print the final layer's input
void PrintOutputLayer(int nLayer, Layer *layers)
{
    int OutputLayerIndex = nLayer - 1;
    printf("Values for the Output layer (Layer %d):\n", OutputLayerIndex +

    for (int i = 0; i < layers[OutputLayerIndex - 1].nextSize; i++)
    {
        printf("Output %d: %.4f\n", i + 1, layers[OutputLayerIndex].input[i
    }
}

void FreeMemory(int nLayer, Layer *layers, int *NodeInfo)
{
    for (int i = 0; i < nLayer; i++)
    {
        free(layers[i].input);
        if (i < nLayer - 1)
        {
            for (int j = 0; j < layers[i].nextSize; j++)
            {
                free(layers[i].weights[j]);
            }
            free(layers[i].weights);
        }
    }
    free(NodeInfo);
    free(layers);       You, 17 minutes ago • Uncommitted changes
}
```

이 두 함수들은 첫번째로는 결과를 출력해주는 함수이다 아웃풋 레이어의 인덱스를 먼저 받고 +1 한 레이어의 개수(실제로 4개면 4번째로 출력)를 출력해주고 layers[OutputLayerIndex - 1].nextSize는 아까 가중치와 input할당 시에 안했을 수도 있으므로 혹여나 그래서 마지막 레이어의 더 전의 레이어에서 nextSize를 가져오면 아웃풋 레이어의 노드 수를 알아내고 반복문을 돌면서 결과 값들을 출력하는 역할을 한다.

동적 할당은 free가 생명이다. 그렇기 때문에 메인에서 동적할당한 것들을 불러와서 먼저 input을 해제, weights는 이중포인터이므로 먼저 행 사이즈만큼 free해주고 그 행들을 가리켰던 이중포인터 weights를 free시켜준다 그 밑으로는 NodeInfo, layers를 해제시켜주면 한 레이어가 사라진다. if문이 달린 이유는 마지막 레이어 아웃풋은 가중치 배열이 없기 때문에 그 전까지의 레이어의 가

중치 배열들을 삭제시켜주고 아웃풋 레이어는 가중치 배열을 제외한 input만 삭제시켜주면 되겠다.

큰 틀을 살펴보면 파일에 레이어 개수, 각 개수만큼의 노드 개수, 레이어 개수만큼의 가중치들을 미리 줄마다 선언하고 입력값을 주면 순전파를 하는 과정입니다.

조금 고려해줘야 할 점이 있다면 c언어가 컴퓨터의 캐시를 사용하기 때문에 다른 언어도 마찬가지겠지만 우선 최대 레이어 16개, 각 레이어마다 256개의 노드가 들어갈 수 있다고 가정한다면 cache를 사용하여 배열 계산을 하기 위해 각 노드의 첫번째부터 n번째 노드들을 각 열에 배치하기 때문에 값을 입력하는 데에 사용자가 헷갈릴 수 있으나 열벡터들의 집합이라고 보면 계산과 편리함을 다 잡는 코드라고 생각합니다.