시스템프로그램 과제 5 보고서

2015****** 경영학과 김주형, 2018****** **********과 ******

1. 개요

본 프로젝트에서 선택한 프로그램은 최근 많은 인기를 끌고 있는 딥러닝 모델 중 하나인 Convolutional Neural Network계열 신경망 알고리즘을 간단한 필기체 인식 문제에서 테스트하는 코드를 구현한 것이다. 다양한 Convolutional Neural Network 모델 중에서도 사물 인식 문제에 주로 적용되는 고전적인 모델 구조가 아닌, 최근에 인위적인 이미지 생성, 고해상도화, 영역 검출 등의 복잡한 문제를 해결하는데 많이 사용되고 있는 Fully Convolutional Network라는 모델 구조를 구현해, 분석해 보았다. 간단히 설명하자면, 이미지를 입력 받았을 때 이미지의 차원을 축소하며 다음 층에 넘겨주어, 마지막 층에서 출력 된 여러 개의 행렬들을 가지고 각각 전체 픽셀 값을 평균 내어 요약한 후, 문제를 해결하는데 사용하는 특징을 계산해내는 구조라고 할 수 있다.

본 프로젝트에서는 이처럼 최근에 많이 사용되고 있는 CNN계열 신경망 코드의 C 구현체를 최적화 해보고 어떤 연산 과정이 주로 병목을 만드는지 분석해보며, 파이썬과 C++같은 다른 언어에서도 일반화하여 적용해볼 수 있는 지식과 경험을 쌓는데 의의를 두었다. 먼저 프로젝트 진행에 앞서 몇 가지 고려 사항이 있었다.

1). 본 프로젝트에서 분석한 구현체는 가장 Naive한 수준의 구현으로써, 신경망 훈련 및 추론에 가장 핵심적이라고 할 수 있는 행렬 연산, 역전파 알고리즘도 기본적인 함수를 사용해 구현해보았다. 따라서 해당 코드는 시스템 수준의 최적화 뿐만 아니라 알고리즘 수준의 최적화가 성능을 획기적으로 개선할 여지가 있다. 엄밀히 말하자면, 현재 중첩 반복문으로 구현한 각종 핵심 연산들을 벡터화시킨 행렬 곱셈으로 변환하고, 행렬 곱셈을 빠르게 처리 해주는 알고리즘을 적용하는 것이 성능 개선이 가장 뚜렷할 것이다.

2). 본 프로젝트를 진행한 팀원들(컴퓨터공학 복수전공 2학기, **********과 2학년)은 기존에 프로젝트에서 사용할 수 있는 조건을 충족시키는 프로그램을 짜본 적이 없어, 시스템 프로그램 과제5의 공지가 올라온 후 개발을 시작하였다. 따라서, 수업에서 학습한 C 프로그래밍 지식들이 이미 개발 과정에서 암묵적으로 반영되어, 최적화를 수행할 여지가 적어졌다는 것을 체감할 수 있었다.

3). 현재는 신경망을 훈련시키는 데에 GPU를 사용하는 것이 일반적이고, 추론 과정에만 CPU를 종종 사용하는 관행과는 달리, 본 프로젝트에는 훈련 과정부터 CPU를 사용하였으므로 실제 application과는 실행 시간 프로파일에 다소 차이가 있을 수 있음을 염두에 두었다.

4). 신경망 훈련 코드의 실행 시간은, 신경망 모델의 크기와 구조, 사용하는 데이터 량, 반복 학습 횟수 등의 요인에 의존한다. 현실적으로는, 이러한 요인들 모두 정확도와 같은 작업 성능에 크게 영향을 주므로, 결국 전체 코드 실행 시간은 수행하고자 하는 작업과 목표로 하는 성능 수준에 따라 크게 달라질 수 있다. 산업적으로 사용되는 신경망 모델을 훈련할 때는 1000단위의 GPU 일(days)이 요구되기도 하지만, 예제 데이터 셋을 학습하는 예제 모델은 몇 시간 안에도 더 이상 개선하기 어려운 State of the art 성능을 도달하기도 한다. 따라서 성능에 대한 고려를 하기보다는, 코드 실행 시간이 실행 시도마다 큰 변동을 갖지 않는 수준에서, 적절한 모델 크기와 구조 그리고 데이터 규모를 선택해 고정한 후 최적화를 진행하였다.

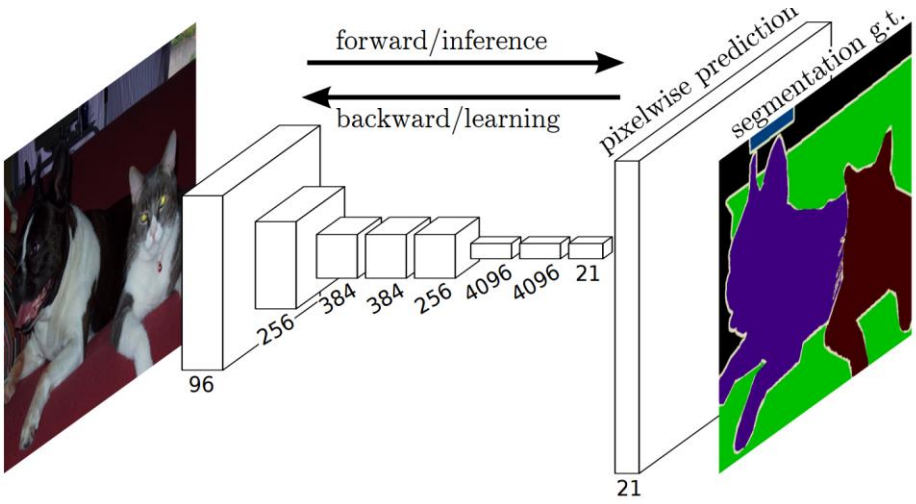
테스트한 신경망의 구조는 총 3개의 Convolution 층으로 이루어져 있었으며, 제일 첫 층에서 이미지를 입력받으면, 각각의 층을 거치며 크기가 더 작은 여러 개의 이미지들로 축소된다. 한 층을 거칠 때마다, leaky ReLU라는 비선형적인 함수(if(x > 0), then x, else 0.01 * x, for input x)를 거쳐 다음 층으로 입력된다. 최종 층에서는 0 ~ 9 각 숫자의 점수를 예측하도록 Softmax 층을 추가하였다.

Model Structure:
Layer Index Input Output Width Height

Layer 0	1 x 28 x 28	16 x 14 x 14	3	3
Layer 1	16 x 14 x 14	32 x 7 x 7	3	3
Layer 2	32 x 7 x 7	64 x 3 x 3	3	3
Softmax	64	10		

Total number of parameters: 23946

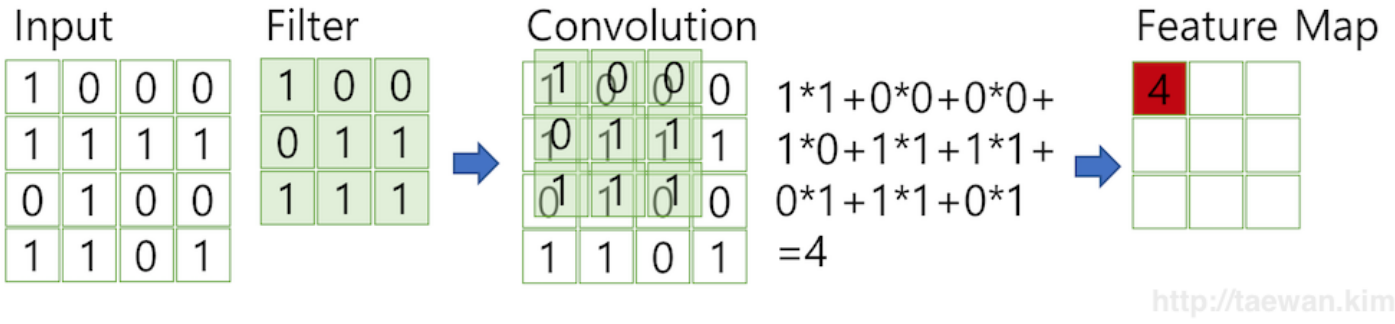
<테스트한 신경망의 구조>



<영역 검출 문제에 적용된 Fully Convolutional Network 구조 예시. 출처: Fully Convolutional Network for sementic segmantation 논문>

신경망 훈련에 사용한 데이터는 28 * 28의 픽셀을 가진 0 ~ 9 숫자를 grayscale 이미지로 저장한 것으로써, 본 프로젝트에서는 1000개의 이미지를 한번 학습하는데 걸리는 시간을 기준으로 최적화를 진행하였다.
위 사진은 본 프로젝트에서 사용된 신경망 구조와 유사한 형태를 가진 대표적인 신경망 모형을 도식화한 것이다.

아래는 본 코드에서 가장 빈번하게 사용될 convolution이라는 연산의 계산 과정이다. convolution연산은 이미지 처럼 격자 구조 또는 배열 형태를 가진 데이터를 입력 받았을 때, 크기가 학습할 수 있는 가중치들로 구성된 필터를 이용해, 입력 이미지의 특정 지점에서의 가중합 픽셀을 계산해 내는 연산이다.



<convolution 연산 도식, 출처: http://taewan.kim/post/cnn/>

CNN 계열의 신경망은 위와 같은 convolution 연산을 매 입력마다, 층마다, 입력이미지의 영역마다 반복적으로 적용하게 되고, 따라서 코드에서 nested된 loop가 많이 등장하며, loop와 관련된 최적화가 일부 가능할 것으로 기대할 수 있다.

1. 컴파일 명령어

GCC 컴파일 명령으로는 다음과 같은 명령을 사용하였다.

gcc-8 [입력파일명] -Og -g -pg -o [출력파일명] -lm -Wall -Wextra -pedantic

-g 는 디버그를 위한 명령어이다.
-pg 옵션은 프로파일링을 위한 명령어이다.

-Og 옵션은 컴파일러가 최적화를 하지 않도록 제한하기 위해 사용하였다.

-lm은 본 프로젝트 코드를 실험한 우분투 OS에서는 기본적으로 링킹해주지 않는 <math.h> 헤더를 링킹시키기 위한 옵션이다.

-Wall -Wextra -pedantic 옵션들은 잠재적 오류가 발생할 수 있는 부분에 대한 경고를 보여주는 명령어이다.

그 후 다음 명령을 실행하여 프로파일링 결과를 출력하고 텍스트 파일(.txt)로 저장하였다.

gprof [입력파일명] gmon.out > [출력파일명]

2. Source code(Version0)

모든 소스코드는 각각의 최적화 시도를 할 때 마다 새로운 버전으로 표시하여 저장하였다.

원본 소스코드는 800줄 정도이고, 다음의 함수들이 정의되어 있다:

```
img *load_img(int width, int height, FILE *img_file, FILE *label_file, int ind);
void print_img(img *data);
void conv_init(conv *layer, int *prev_dim, int output_channel, int width, int height, int stride_width, int stride_height);
void softmax_init(softmax *layer, conv *prev_layer, int output_dim);
double convolution(conv *layer, double ***input, int current_ch, int w_offset, int h_offset);
double ***conv_forward(conv *layer, double ***input);
double *output(conv *prev_layer, softmax *out, double ***input, short backward);
double leaky_relu(double val);
double loss(double *pred, int label, softmax *out, short backward);
void update(conv **params, softmax *out, double ****forwards, int depth, double learning_rate, double l2_penalty);
double train(conv **params, softmax *out, int depth, img *input, double learning_rate, double l2_penalty);
int inference(conv **params, softmax *out, int depth, img *input, short print);
```

Original 버전에 대한 프로파일링 결과는 아래와 같았다.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
72.40	16.33	16.33	5000	3.27	3.27	update
26.97	22.41	6.08	26452800	0.00	0.00	convolution
0.27	22.47	0.06	15030	0.00	0.41	conv_forward
0.18	22.51	0.04	5010	0.01	0.01	load_img
0.11	22.53	0.03	26452800	0.00	0.00	leaky_relu
0.09	22.55	0.02	5010	0.00	0.00	output
0.02	22.56	0.01	1	5.00	5.00	softmax_init
0.00	22.56	0.00	5000	0.00	0.00	loss
0.00	22.56	0.00	5000	0.00	4.50	train
0.00	22.56	0.00	10	0.00	1.24	inference
0.00	22.56	0.00	10	0.00	0.00	print_img
0.00	22.56	0.00	3	0.00	0.00	conv_init

프로파일링 결과를 보면, update 함수가 가장 많은 실행시간을 소요한다는 것을 확인할 수 있었다. 해당 함수는 신경망의 역전파 알고리즘과 역전파 후 파라미터 조정을 담당하는 부분으로써, 이미지 데이터를 통과시킨 신경망을 한단계씩 학습시키는 부분이다.신경망에서 파라미터는 모델의 기능을 조정하는 역할을 하는 최소 단위이고, 신경망의 구조는 여러 개의 층으로 이루어져 있으며, 각각의 층은 다수의 파라미터를 포함하고 있다. 역전파 알고리즘은 결국 각각의 파라미터를 얼마만큼 조정해야 할지 계산하는 과정이라고 볼 수 있다. update함수에서는 역전파를 한 후에, Gradient Descent라는 가장 기본적인 최적화 알고리즘을 이용해 파라미터를 조정하는 과정이 포함되어 있는데, 이 부분도 모든 파라미터를 각각 조정하는 과정이라 많은 중첩 for문이 사용된다.

Update 부분의 함수 코드는 다음과 같았다.

```
//Backpropagation and gradient descent
void update(conv **params, softmax *out, double ****forwards, int depth, double learning_rate, double l2_penalty) {
    int layer, in_ch, in_w, in_h, out_ch, out_w, out_h, fil_w, fil_h, pos_w, pos_h;
    double ***temp_input_grad = NULL;
    double ****temp_filter_grad = NULL;
    double **temp_weight_grad = NULL;
    double *temp_bias_grad = NULL;
```

```
//Applying Chain rule for softmax layer--------------------------------------------------------------------------
//Making temporal storage
temp_weight_grad = (double **)calloc(out->input_dim, sizeof(double *));
for (in_ch = 0; in_ch < out->input_dim; in_ch++) {
    temp_weight_grad[in_ch] = (double *)calloc(out->output_dim, sizeof(double));
}

temp_input_grad = (double ***)calloc(params[depth - 1]->output_shape[0], sizeof(double **));
for (in_ch = 0; in_ch < params[depth - 1]->output_shape[0]; in_ch++) {
    temp_input_grad[in_ch] = (double **)calloc(params[depth - 1]->output_shape[1], sizeof(double *));
    for (in_w = 0; in_w < params[depth - 1]->output_shape[1]; in_w++) {
        temp_input_grad[in_ch][in_w] = (double *)calloc(params[depth - 1]->output_shape[2], sizeof(double));
    }
}

//Calculating backpropagation
for (in_ch = 0; in_ch < out->input_dim; in_ch++) {
    for (out_ch = 0; out_ch < out->output_dim; out_ch++) {
        for (in_w = 0; in_w < params[depth - 1]->output_shape[1]; in_w++) {
            for (in_h = 0; in_h < params[depth - 1]->output_shape[2]; in_h++) {
                temp_weight_grad[in_ch][out_ch] +=
                    forwards[depth][in_ch][in_w][in_h] *
                    out->output_grad[out_ch];
                temp_input_grad[in_ch][in_w][in_h] +=
                    out->weight[in_ch][out_ch] *
                    out->output_grad[out_ch];
                temp_input_grad[in_ch][in_w][in_h] /=
                    (double)(params[depth - 1]->output_shape[1] * params[depth - 1]->output_shape[2]);
            }
        }
    }
}

//Replacing output gradient with loss gradient
for (in_ch = 0; in_ch < out->input_dim; in_ch++) {
    for (out_ch = 0; out_ch < out->output_dim; out_ch++) {
        out->weight_grad[in_ch][out_ch] *= temp_weight_grad[in_ch][out_ch];
    }
    free(temp_weight_grad[in_ch]);
}
free(temp_weight_grad);

for (in_ch = 0; in_ch < params[depth - 1]->output_shape[0]; in_ch++) {
    for (in_w = 0; in_w < params[depth - 1]->output_shape[1]; in_w++) {
        for (in_h = 0; in_h < params[depth - 1]->output_shape[2]; in_h++) {
            params[depth - 1]->output_grad[in_ch][in_w][in_h] *=
                temp_input_grad[in_ch][in_w][in_h];
        }
        free(temp_input_grad[in_ch][in_w]);
    }
    free(temp_input_grad[in_ch]);
}

free(temp_input_grad);
for (out_ch = 0; out_ch < out->output_dim; out_ch++) {
    out->bias_grad[out_ch] *= out->output_grad[out_ch];
    out->output_grad[out_ch] = 0.0;
}

//Applying Chain rule from output at each convolution layer-----------------------------------------
for (layer = depth - 1; layer > 0; layer--) {
    //Making temporal storage
    temp_filter_grad = (double ****)calloc(params[layer]->filter_size[0], sizeof(double ***));
    for (in_ch = 0; in_ch < params[layer]->filter_size[0]; in_ch++) {
        temp_filter_grad[in_ch] = (double ***)calloc(params[layer]->filter_size[1], sizeof(double **));
```

```
    for (out_ch = 0; out_ch < params[layer]->filter_size[1]; out_ch++) {
        temp_filter_grad[in_ch][out_ch] = (double **)calloc(params[layer]->filter_size[2], sizeof(double *));
        for (fil_w = 0; fil_w < params[layer]->filter_size[2]; fil_w++) {
            temp_filter_grad[in_ch][out_ch][fil_w] = (double *)calloc(params[layer]->filter_size[3], sizeof(double));
        }
    }
}

temp_input_grad = (double ***)calloc(params[layer]->input_shape[0], sizeof(double **));
for (in_ch = 0; in_ch < params[layer]->input_shape[0]; in_ch++) {
    temp_input_grad[in_ch] = (double **)calloc(params[layer]->input_shape[1], sizeof(double *));
    for (in_w = 0; in_w < params[layer]->input_shape[1]; in_w++) {
        temp_input_grad[in_ch][in_w] = (double *)calloc(params[layer]->input_shape[2], sizeof(double));
    }
}

temp_bias_grad = (double *)calloc(params[layer]->output_shape[0], sizeof(double));

//Calculating per-layer backpropagation
for (in_ch = 0; in_ch < params[layer]->filter_size[0]; in_ch++) {
    for (out_ch = 0; out_ch < params[layer]->output_shape[0]; out_ch++) {
        for (out_w = 0; out_w < params[layer]->output_shape[1]; out_w++) {
            for (out_h = 0; out_h < params[layer]->output_shape[2]; out_h++) {
                for (fil_w = 0; fil_w < params[layer]->filter_size[2]; fil_w++) {
                    for (fil_h = 0; fil_h < params[layer]->filter_size[3]; fil_h++) {
                        pos_w = out_w * (1 + params[layer]->stride_width) + fil_w - (params[layer]->filter_size[2] / 2);
                        pos_h = out_h * (1 + params[layer]->stride_height) + fil_h - (params[layer]->filter_size[3] / 2);
                        if ((0 <= pos_w) && (pos_w < params[layer]->input_shape[1]) &&
                            (0 <= pos_h) && (pos_h < params[layer]->input_shape[2])) {
                                temp_input_grad[in_ch][pos_w][pos_h] +=
                                params[layer]->filter[in_ch][out_ch][fil_w][fil_h] *
                                params[layer]->output_grad[out_ch][out_w][out_h];
                            temp_filter_grad[in_ch][out_ch][fil_w][fil_h] +=
                                forwards[layer][in_ch][pos_w][pos_h] *
                                params[layer]->output_grad[out_ch][out_w][out_h];
                            temp_bias_grad[out_ch] +=
                                params[layer]->output_grad[out_ch][out_w][out_h];
                        }
                    }
                }
                params[layer]->output_grad[out_ch][out_w][out_h] = 0.0;
            }
        }
    }
}

//Replacing per-layer output gradients with loss gradients
for (in_ch = 0; in_ch < params[layer]->filter_size[0]; in_ch++) {
    for (out_ch = 0; out_ch < params[layer]->filter_size[1]; out_ch++) {
        for (fil_w = 0; fil_w < params[layer]->filter_size[2]; fil_w++) {
            for (fil_h = 0; fil_h < params[layer]->filter_size[3]; fil_h++) {
                params[layer]->filter_grad[in_ch][out_ch][fil_w][fil_h] =
                    temp_filter_grad[in_ch][out_ch][fil_w][fil_h];
            }
            free(temp_filter_grad[in_ch][out_ch][fil_w]);
        }
        free(temp_filter_grad[in_ch][out_ch]);
    }
    free(temp_filter_grad[in_ch]);
}
free(temp_filter_grad);

for (in_ch = 0; in_ch < params[layer]->input_shape[0]; in_ch++) {
    for (in_w = 0; in_w < params[layer]->input_shape[1]; in_w++) {
        for (in_h = 0; in_h < params[layer]->input_shape[2]; in_h++) {
```

```
                params[layer - 1]->output_grad[in_ch][in_w][in_h] *=
                    temp_input_grad[in_ch][in_w][in_h];
            }
            free(temp_input_grad[in_ch][in_w]);
        }
        free(temp_input_grad[in_ch]);
    }
    free(temp_input_grad);

    for (out_ch = 0; out_ch < params[layer]->output_shape[0]; out_ch++) {
        params[layer]->bias_grad[out_ch] *=
            temp_bias_grad[out_ch];
    }
    free(temp_bias_grad);
}

//Calculate Backpropagtion of first layer
temp_filter_grad = (double ****)calloc(params[0]->filter_size[0], sizeof(double ***));
for (in_ch = 0; in_ch < params[0]->filter_size[0]; in_ch++) {
    temp_filter_grad[in_ch] = (double ***)calloc(params[0]->filter_size[1], sizeof(double **));
    for (out_ch = 0; out_ch < params[0]->filter_size[1]; out_ch++) {
        temp_filter_grad[in_ch][out_ch] = (double **)calloc(params[0]->filter_size[2], sizeof(double *));
        for (fil_w = 0; fil_w < params[0]->filter_size[2]; fil_w++) {
            temp_filter_grad[in_ch][out_ch][fil_w] = (double *)calloc(params[0]->filter_size[3], sizeof(double));
        }
    }
}

temp_bias_grad = (double *)calloc(params[0]->output_shape[0], sizeof(double));

//Calculating per-layer backpropagation
for (in_ch = 0; in_ch < params[0]->filter_size[0]; in_ch++) {
    for (out_ch = 0; out_ch < params[0]->output_shape[0]; out_ch++) {
        for (out_w = 0; out_w < params[0]->output_shape[1]; out_w++) {
            for (out_h = 0; out_h < params[0]->output_shape[2]; out_h++) {
                for (fil_w = 0; fil_w < params[0]->filter_size[2]; fil_w++) {
                    for (fil_h = 0; fil_h < params[0]->filter_size[3]; fil_h++) {
                        pos_w = out_w * (1 + params[0]->stride_width) + fil_w - (params[0]->filter_size[2] / 2);
                        pos_h = out_h * (1 + params[0]->stride_height) + fil_h - (params[0]->filter_size[3] / 2);
                        if ((0 <= pos_w) && (pos_w < params[0]->input_shape[1]) &&
                            (0 <= pos_h) && (pos_h < params[0]->input_shape[2])) {
                            temp_filter_grad[in_ch][out_ch][fil_w][fil_h] +=
                                forwards[0][in_ch][pos_w][pos_h] *
                                params[0]->output_grad[out_ch][out_w][out_h];
                            temp_bias_grad[out_ch] +=
                                params[0]->output_grad[out_ch][out_w][out_h];
                        }
                    }
                }
                params[0]->output_grad[out_ch][out_w][out_h] = 0.0;
            }
        }
    }
}

//Replacing per-layer output gradients with loss gradients at first layer
for (in_ch = 0; in_ch < params[0]->filter_size[0]; in_ch++) {
    for (out_ch = 0; out_ch < params[0]->filter_size[1]; out_ch++) {
        for (fil_w = 0; fil_w < params[0]->filter_size[2]; fil_w++) {
            for (fil_h = 0; fil_h < params[0]->filter_size[3]; fil_h++) {
                params[0]->filter_grad[in_ch][out_ch][fil_w][fil_h] =
                    temp_filter_grad[in_ch][out_ch][fil_w][fil_h];
            }
            free(temp_filter_grad[in_ch][out_ch][fil_w]);
        }
```

```
            free(temp_filter_grad[in_ch][out_ch]);
        }
        free(temp_filter_grad[in_ch]);
    }
    free(temp_filter_grad);

    for (out_ch = 0; out_ch < params[0]->output_shape[0]; out_ch++)
    {
        params[0]->bias_grad[out_ch] *=
            temp_bias_grad[out_ch];
    }
    free(temp_bias_grad);

    //Update parameters-------------------------------------------------------------------------------------
    for (layer = 0; layer < depth; layer++) {
        for (in_ch = 0; in_ch < params[layer]->filter_size[0]; in_ch++) {
            for (out_ch = 0; out_ch < params[layer]->filter_size[1]; out_ch++) {
                for (fil_w = 0; fil_w < params[layer]->filter_size[2]; fil_w++){
                    for (fil_h = 0; fil_h < params[layer]->filter_size[3]; fil_h++) {
                        params[layer]->filter[in_ch][out_ch][fil_w][fil_h] *= (1.0 - learning_rate * l2_penalty);
                        params[layer]->filter[in_ch][out_ch][fil_w][fil_h] -=
                            params[layer]->filter_grad[in_ch][out_ch][fil_w][fil_h] * learning_rate;
                        params[layer]->filter_grad[in_ch][out_ch][fil_w][fil_h] = 0.0;
                    }
                }
            }
        }

        for (out_ch = 0; out_ch < params[layer]->output_shape[0]; out_ch++) {
            params[layer]->bias[out_ch] -= params[layer]->bias_grad[out_ch] * learning_rate;
            params[layer]->bias_grad[out_ch] = 0.0;
        }
    }

    for (in_ch = 0; in_ch < out->input_dim; in_ch++) {
        for (out_ch = 0; out_ch < out->output_dim; out_ch++) {
            out->weight[in_ch][out_ch] *= (1.0 - learning_rate * l2_penalty);
            out->weight[in_ch][out_ch] -= out->weight_grad[in_ch][out_ch] * learning_rate;
            out->weight_grad[in_ch][out_ch] = 0.0;
        }
    }

    for (out_ch = 0; out_ch < out->output_dim; out_ch++) {
        out->bias[out_ch] -= out->bias_grad[out_ch] * learning_rate;
        out->bias_grad[out_ch] = 0.0;
    }
}
```

함수 코드를 대략적으로 확인해 보면, 앞서 말했듯이 for문이 많이 중첩되어 있는 것을 확인 할 수 있다. 중첩된 for문으로 인해, 안쪽 루프의 실행시간과 관련된 최적화, 반복 조건과 관련된 최적화 등이 유용할 것을 기대할 수 있었다. Convolution 함수의 코드는 아래와 같다.

```
//Calculate Convolution
double convolution(conv *layer, double ***input, int current_ch, int w_offset, int h_offset) {
    int in_ch, fil_w, fil_h, pos_w, pos_h;
    double result = 0.0;

    //calculation convolution
    for (in_ch = 0; in_ch < layer->input_shape[0]; in_ch++) {
        for (fil_w = -layer->filter_size[2] / 2; fil_w <= layer->filter_size[2] / 2; fil_w++) {
            for (fil_h = -layer->filter_size[3] / 2; fil_h <= layer->filter_size[3] / 2; fil_h++) {
                //convolution considering padding
                pos_w = w_offset * (1 + layer->stride_width) + fil_w;
                pos_h = h_offset * (1 + layer->stride_height) + fil_h;
                if ((0 <= pos_w) && (pos_w < layer->input_shape[1]) &&
```

```
                (0 <= pos_h) && (pos_h < layer->input_shape[2])) {
                result += input[in_ch][pos_w][pos_h]
                        * layer->filter[in_ch][current_ch][fil_w + layer->filter_size[2] / 2][fil_h + layer->filter_size[3] / 2];
            }
        }
    }
  }

  result += layer->bias[current_ch];
  result = leaky_relu(result);

  //calculating gradients
  if (result > 0) {
    layer->output_grad[current_ch][w_offset][h_offset] = 1.0;
  } else {
    layer->output_grad[current_ch][w_offset][h_offset] = 0.01;
  }

  return result;
}
```

convolution 함수 부분도 for문이 많이 중첩되어 있는데, Convolution함수는 코드에서 제일 많이 호출되는 함수임에 따라, 작은 최적화도 전체 코드 실행 시간에 많은 영향을 줄 수 있음을 기대할 수 있었다. update와 convolution 두 함수를 최적화 하는 것만으로도, 전체 코드 실행 시간의 99.45%에 해당하는 부분을 최적화하는 것이므로, 본 프로젝트에서는 이 부분에 집중하도록 하였다. 먼저, 각각의 for문의 반복 조건에 있는 구조체 참조 부분을 새 변수를 만들어 그 변수에 반복 조건에서 비교할 값을 저장하도록 코드를 수정해 보고자 하였다. 불행히도, update 함수에서는 이런 과정을 거치면 사용하는 정수 변수와 포인터 변수의 개수가 I7 processor의 정수 레지스터의 총 개수인 16개를 넘어, general purpose register들 만으로는 모든 변수를 저장할 수 없다. 따라서 새로운 변수 선언으로 인한 추가적인 메모리 참조(스택 참조)가 발생할 수 있다는 것을 알 수 있었다.

2.2 Version1

먼저 원본 소스 코드에서, for문의 반복 조건에서 구조체를 참조하는 부분을 일반 변수로 선언하고 그 변수를 for문의 반복 조건에 대입하는 Code Motion을 적용하였다. 또한 update와 convolution 함수에 있는 중첩 for문에서, 가장 안쪽의 loop에서 이루어지던 덧셈 연산을 중첩 for문 밖으로 빼서 loop inefficiency를 감소시켰다. 해당 과정은 함수 부분의 코드가 전체적으로 수정되어서, 변경된 부분이 너무 많아 보고서에서는 생략하였다. 첨부된 version0와 version1 부분의 코드에 전체적인 차이가 있음을 쉽게 확인할 수 있다.

그러나, 결과적으로는 오히려 코드 실행시간이 증가하였다. 앞서 설명하였듯이, Code Motion을 적용하는 과정에서 사용하는 정수 변수와 포인터 변수의 개수가 프로세서의 총 정수 레지스터 개수를 넘어서 메모리를 사용하게 되는 것이 아닐까 추측할 수 있었다.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
75.33	17.54	17.54	5000	3.51	3.51	update
24.32	23.20	5.66	26452800	0.00	0.00	convolution
0.13	23.23	0.03	26452800	0.00	0.00	leaky_relu
0.09	23.25	0.02	15030	0.00	0.38	conv_forward
0.09	23.27	0.02	5010	0.00	0.00	load_img
0.09	23.29	0.02	5010	0.00	0.00	output
0.00	23.29	0.00	5000	0.00	0.00	loss
0.00	23.29	0.00	5000	0.00	4.65	train
0.00	23.29	0.00	10	0.00	1.14	inference
0.00	23.29	0.00	10	0.00	0.00	print_img
0.00	23.29	0.00	3	0.00	0.00	conv_init
0.00	23.29	0.00	1	0.00	0.00	softmax_init

gprof를 이용해 프로파일링을 해보니, update함수의 실행시간은 16.33초에서 17.54초로 7.4% 오히려 증가하였고, convolution함수의 실행시간은 6.08초에서 5.66초로 22.56초로 6.9% 개선되었으나, 결과적으로는 전체 코드 실행 시간은 22.56초에서 23.29초로 3% 정도 증가하였다.

2.3 Version2

2번째 최적화 과정에서는, 첫번째 최적화 과정을 수행하며 생긴 불필요한 변수들을 제거하고, loop inefficiency를 더 감소시키기 위해 update함수에서 가장 내부에 있던 곱셈 연산을 최대한 바깥쪽 루프로 빼서 반복적인 연산을 줄여보고자 하였다. 예를 들어, 다음과 같은 변경을 하였다. 붉게 표시된 부분이 수정한 부분이고, 변경 전 코드는 아래와 같다.

```
//Calculating per-layer backpropagation
    for (in_ch = 0; in_ch < lim_in_ch; in_ch++) {
      for (out_ch = 0; out_ch < lim_out_ch; out_ch++) {
        for (out_w = 0; out_w < lim_out_w; out_w++) {
          for (out_h = 0; out_h < lim_out_h; out_h++) {
            for (fil_w = 0; fil_w < lim_fil_w; fil_w++) {
              for (fil_h = 0; fil_h < lim_fil_h; fil_h++) {
                pos_w = out_w * receptive_field_w + fil_w - (lim_fil_w / 2);
                pos_h = out_h * receptive_field_h + fil_h - (lim_fil_h / 2);
                if ((0 <= pos_w) && (pos_w < lim_in_w) &&
                  (0 <= pos_h) && (pos_h < lim_in_h)) {
                    temp_input_grad[in_ch][pos_w][pos_h] +=
                    params[layer]->filter[in_ch][out_ch][fil_w][fil_h] *
                    params[layer]->output_grad[out_ch][out_w][out_h];
                  temp_filter_grad[in_ch][out_ch][fil_w][fil_h] +=
                    forwards[layer][in_ch][pos_w][pos_h] *
                    params[layer]->output_grad[out_ch][out_w][out_h];
                  temp_bias_grad[out_ch] +=
                    params[layer]->output_grad[out_ch][out_w][out_h];
                }
              }
            }
            params[layer]->output_grad[out_ch][out_w][out_h] = 0.0;
          }
        }
      }
    }
```

변경 후 코드는 아래와 같다.

```
//Calculating per-layer backpropagation
    for (in_ch = 0; in_ch < lim_in_ch; in_ch++) {
      for (out_ch = 0; out_ch < lim_out_ch; out_ch++) {
        for (out_w = 0; out_w < lim_out_w; out_w++) {
          offset_w = out_w * receptive_field_w - (lim_fil_w / 2);
          for (out_h = 0; out_h < lim_out_h; out_h++) {
            offset_h = out_h * receptive_field_h - (lim_fil_h / 2);
            for (fil_w = 0; fil_w < lim_fil_w; fil_w++) {
              for (fil_h = 0; fil_h < lim_fil_h; fil_h++) {
                pos_w = offset_w + fil_w;
                pos_h = offset_h + fil_h;
                if ((0 <= pos_w) && (pos_w < lim_in_w) &&
                  (0 <= pos_h) && (pos_h < lim_in_h)) {
                  temp_input_grad[in_ch][pos_w][pos_h] +=
                    params[layer]->filter[in_ch][out_ch][fil_w][fil_h] *
                    params[layer]->output_grad[out_ch][out_w][out_h];
                  temp_filter_grad[in_ch][out_ch][fil_w][fil_h] +=
                    forwards[layer][in_ch][pos_w][pos_h] *
                    params[layer]->output_grad[out_ch][out_w][out_h];
                  temp_bias_grad[out_ch] +=
                    params[layer]->output_grad[out_ch][out_w][out_h];
                }
              }
            }
```

```
                params[layer]->output_grad[out_ch][out_w][out_h] = 0.0;
            }
        }
    }
}
```

결과적으로, integer 자료형 offset_w와 offset_h변수를 새로 선언하여, 곱셈 연산을 신경망의 각 층마다 lim_fil_w * lim_fil_h 배 줄일 수 있었다. 수업시간에 곱셈 연산이 꽤 많은 cpu cycle을 소모한다고 배운 만큼, 해당 최적화가 준수한 성능 개선을 보일 것을 기대할 수 있었다. 또한 convolution 함수도 중복된 연산을 최대한 루프 밖으로 빼내는 비슷한 방식으로, 더 최적화 하였다. convolution부분의 변경 전 코드는 아래와 같았다.

```
//Calculate Convolution
double convolution(conv *layer, double ***input, int current_ch, int w_offset, int h_offset) {
    int in_ch, fil_w, fil_h, pos_w, pos_h;

    int lim_in_ch = layer->input_shape[0];
    int lim_in_w = layer->input_shape[1];
    int lim_in_h = layer->input_shape[2];

    int lim_fil_half_w = layer->filter_size[2]/2;
    int lim_fil_half_h = layer->filter_size[3]/2;

    int receptive_field_w = layer->stride_width + 1;
    int receptive_field_h = layer->stride_height + 1;

    double result = 0.0;

    //calculating convolution
    for (in_ch = 0; in_ch < lim_in_ch; in_ch++) {
        for (fil_w = -lim_fil_half_w; fil_w <= lim_fil_half_w; fil_w++) {
            for (fil_h = -lim_fil_half_h; fil_h <= lim_fil_half_h; fil_h++) {
                //convolution considering padding
                pos_w = w_offset * receptive_field_w + fil_w;
                pos_h = h_offset * receptive_field_h + fil_h;
                if ((0 <= pos_w) && (pos_w < lim_in_w) &&
                    (0 <= pos_h) && (pos_h < lim_in_h)) {
                    result += input[in_ch][pos_w][pos_h]
                            * layer->filter[in_ch][current_ch][fil_w + lim_fil_half_w][fil_h + lim_fil_half_h];
                }
            }
        }
    }

    result += layer->bias[current_ch];
    result = leaky_relu(result);

    //calculating gradients
    if (result > 0) {
        layer->output_grad[current_ch][w_offset][h_offset] = 1.0;
    } else {
        layer->output_grad[current_ch][w_offset][h_offset] = 0.01;
    }

    return result;
}
```

변경 후 코드는 아래와 같았다. 함수의 아규먼트명은 함수의 기능을 더 명확히 하기 위해 변경하였고, 최적화와는 관계가 없다.

```
//Calculate Convolution
double convolution(conv *layer, double ***input, int current_ch, int in_w, int in_h) {
    int in_ch, fil_w, fil_h, pos_w, pos_h;

    int lim_in_ch = layer->input_shape[0];
```

```
    int lim_in_w = layer->input_shape[1];
    int lim_in_h = layer->input_shape[2];

    int lim_fil_w = layer->filter_size[2];
    int lim_fil_h = layer->filter_size[3];

    int receptive_field_w = layer->stride_width + 1;
    int receptive_field_h = layer->stride_height + 1;

    int w_offset = in_w * receptive_field_w - (lim_fil_w / 2);
    int h_offset = in_h * receptive_field_h - (lim_fil_h / 2);

    double result = 0.0;

    //calculating convolution
    for (in_ch = 0; in_ch < lim_in_ch; in_ch++) {
        for (fil_w = 0; fil_w < lim_fil_w; fil_w++) {
            for (fil_h = 0; fil_h < lim_fil_h; fil_h++) {
                //convolution considering padding
                pos_w = w_offset + fil_w;
                pos_h = h_offset + fil_h;
                if ((0 <= pos_w) && (pos_w < lim_in_w) &&
                    (0 <= pos_h) && (pos_h < lim_in_h)) {
                    result += input[in_ch][pos_w][pos_h]
                            * layer->filter[in_ch][current_ch][fil_w][fil_h];
                }
            }
        }
    }

    result += layer->bias[current_ch];
    result = leaky_relu(result);

    //calculating gradients
    if (result > 0) {
        layer->output_grad[current_ch][in_w][in_h] = 1.0;
    } else {
        layer->output_grad[current_ch][in_w][in_h] = 0.01;
    }

    return result;
}
```

결과적으로, update함수의 경우처럼, integer 자료형 w_offset와 h_offset변수를 새로 선언하여, 곱셈 연산을 신경망의 각 층마다 lim_fil_w * lim_fil_h 배 줄일 수 있었다. 프로파일링을 해본 결과는 다음과 같았다.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
75.60	15.41	15.41	5000	3.08	3.08	update
23.88	20.27	4.87	26452800	0.00	0.00	convolution
0.25	20.32	0.05	15030	0.00	0.33	conv_forward
0.15	20.35	0.03	5010	0.01	0.01	load_img
0.10	20.37	0.02	26452800	0.00	0.00	leaky_relu
0.05	20.38	0.01	5010	0.00	0.00	output
0.02	20.39	0.01	1	5.00	5.00	softmax_init
0.00	20.39	0.00	5000	0.00	0.00	loss
0.00	20.39	0.00	5000	0.00	4.07	train
0.00	20.39	0.00	10	0.00	0.99	inference
0.00	20.39	0.00	10	0.00	0.00	print_img
0.00	20.39	0.00	3	0.00	0.00	conv_init

이전 단계와 비교해보자면, update함수의 실행 시간이 17.54초에서 15.41초로 12.1%감소하였고, convolution 함수의 경우 5.66초에서 4.87초로 13.9% 개선되었다. 전체 코드 실행 시간은 23.29초에서 20.39초로 12.4% 정도 개선되었다.

2.4 Version3

3번째 최적화 과정에서는 지금까지의 최적화 과정에서 계속 확인할 수 있었던, convolution함수와 leaky_relu함수의 function call로 인한 overhead를 줄이고자, 상위 함수인 conv_forward로 병합해 보고자 하였다. 코드의 실행 과정을 보면, 모델이 임의의 이미지 데이터를 가지고 한단계 훈련을 할 때 마다, 각각의 신경망 층마다 conv_forward 함수를 호출하게 되며 conv_forward 안에서는 이미지를 입력 받아 입력 이미지에서 일정한 간격을 두고 convoltion 연산을 적용하는 계층적인 실행 절차를 가지고 있다. 즉, 현재까지 최적화 한 코드 상에서는 conv_forward → convolution → leaky relu 형태의 호출을 하고 있는데, 이를 모두 conv forward 함수로 합쳐 버리고자 한 것이다.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
77.64	15.09	15.09	5000	3.02	3.02	update
22.19	19.40	4.31	15030	0.29	0.29	conv_forward
0.15	19.43	0.03	5010	0.01	0.01	output
0.05	19.44	0.01	5010	0.00	0.00	load_img
0.00	19.44	0.00	5000	0.00	0.00	loss
0.00	19.44	0.00	5000	0.00	3.88	train
0.00	19.44	0.00	10	0.00	0.87	inference
0.00	19.44	0.00	10	0.00	0.00	print_img
0.00	19.44	0.00	3	0.00	0.00	conv_init
0.00	19.44	0.00	1	0.00	0.00	softmax_init

프로파일링 결과를 확인해보니, convolution 함수의 실행시간이 4.87초에서 4.31초로 11.5% 감소하였다. update함수의 경우 전혀 수정하지 않았으므로, update함수의 실행시간이 15.41초에서 15.09초로 감소한 것은 그냥 측정에 외부적인 노이즈가 있는 것으로 판단하였다. 전체 코드 실행시간은 20.39초에서 19.44초로 4.65% 개선되었다.

2.5 Version4

3번째 최적화를 진행한 후, loop unrolling을 적용해보고자 하였다. Loop unrolling을 적용하기 위해, 이미지의 행과 열방향으로 반복적으로 수행되었던 convolution 연산을, 반복문 구문을 제거하고 모든 케이스에 대해 직접 convolution을 계산하는 방식으로 loop unrolling을 시도해보았으나, 오히려 convolution 함수와 update 함수의 실행시간이 각각 5초 가량 증가하여, loop unrolling은 적용하지 않았다. 결국, 더 이상 적용할만한 최적화가 없다고 판단하여, 마지막으로 gcc 컴파일러를 이용해 다양한 최적화 수준으로 컴파일을 진행해보았다.

-O1 옵션으로 최적화한 후 프로파일링 결과는 다음과 같다.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
77.44	13.06	13.06	5000	2.61	2.61	update
22.41	16.85	3.78	15030	0.25	0.25	conv_forward
0.12	16.87	0.02	5010	0.00	0.00	load_img
0.06	16.88	0.01	5010	0.00	0.00	output
0.00	16.88	0.00	5000	0.00	0.00	loss
0.00	16.88	0.00	5000	0.00	3.37	train
0.00	16.88	0.00	10	0.00	0.76	inference
0.00	16.88	0.00	10	0.00	0.00	print_img
0.00	16.88	0.00	3	0.00	0.00	conv_init
0.00	16.88	0.00	1	0.00	0.00	softmax_init

-O2 옵션으로 최적화한 후 프로파일링 결과는 다음과 같다.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
68.70	8.22	8.22	5000	1.64	1.64	update
31.01	11.93	3.71	15030	0.25	0.25	conv_forward
0.17	11.95	0.02				load_img
0.08	11.96	0.01	5010	0.00	0.00	output
0.08	11.97	0.01	5000	0.00	0.00	loss

-O3 옵션으로 최적화한 후 프로파일링 결과는 다음과 같다.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
71.82	8.68	8.68				update
28.05	12.07	3.39				conv_forward
0.17	12.09	0.02				frame_dummy

3가지 최적화 수준에서 프로파일링 결과를 종합적으로 비교해보면, 특이하게도 가장 높은 최적화 수준인 O3 레벨의 최적화보다 O2 레벨의 최적화가 미묘한 차이로 코드 실행시간이 더 빨랐다. 원인은 커널의 작동원리와 관련된 것으로 추측할 수 있었으나, 본 강의의 영역을 벗어나므로 생략하고자 한다. 높은 최적화를 진행하면, 단순히 코드의 명령어 배치, 연산 과정뿐만 아니라 함수를 새로 생성하거나 제거하기도 한다는 것을 확인할 수 있었다. 그러나, 이런 행위들의 영향이 무엇인지는 정확하게 알 수 없었다.

3. 결론

Version 1

- 불필요한 for문을 제거, 그 외 반복들에서 Code Motion을 수행하였다. 또한 중첩 for문에서 일부 연산을 상위단계 loop로 옮겨 loop inefficiency를 감소시켰다.
- 전체적으로 실행시간이 변화가 없거나 오히려 증가하였다. Code Motion중 사용하는 변수의 수가 레지스터의 수를 넘어 메모리를 사용한 것이 원인으로 추측된다.

Version 2

- Code Motion에서 발생한 불필요한 변수들을 제거, loop inefficiency를 추가적으로 낮추기 위해 일부 연산들을 다시 상위의 loop로 이동하였다.
- update함수의 실행시간이 12.1%, convolution함수의 실행시간이 13.9% 감소하여 전체 실행시간이 12.4% 감소하였다.

Version3

- convolution함수와 leaky_relu함수의 과도한 function call로 인한 overhead를 줄이기 위하여 convolution함수와 leaky_relu함수를 상위함수인 conv_forward에 병합하였다.
- convolution 함수의 실행시간이 11.5% 감소하였다.

Version4

- -O1, -O2, -O3 최적화 수준에서 컴파일하여, 실행시간을 비교하였더니 -O2 수준의 최적화가 가장 코드 실행시간을 줄여주었다. -O2 최적화 수준의 컴파일을 한다면 전체 코드 실행시간을 이전 단계에 비해 38.4% 감소시켰다.

결과적으로, 이번 프로젝트를 통해 본 팀은 수업시간에 배운 최적화 기법들 중 일부를 C 프로그램에 직접 적용해보고, 성능을 개선하는데 성공하였으며, 더 나아가 gcc 컴파일러의 최적화 효과도 간단히 테스트해볼 수 있었다. 불행히도, 수업시간에 등장한 예제 만큼의 비약적인 최적화는 수행할 수 없었다. 하지만 개발 과정에서 이미 수업에서 배운 C 프로그램 최적화 지식이 다소 반영되었다는 것을 고려한다면, 최적화 과정을 통해 프로그램 성능을 준수하게 향상시킬 수 있었다고 판단할 수 있었다.