Part 2 Project:
Data Analysis
Using Neural Network

경영학과 20122682

류성호

1. 프로젝트 목적

특정 데이터를 선택하고, 이를 Neural Network를 사용한 예측 모델로 구축한다. 'How to implement'에 초점을 맞추어, 주어진 데이터에 최적화된 Neural Network 구조를 찾는다.

A. Neural Network(NN)

Neural Network는 뉴런 시냅스로 이루어진 뇌의 구조가 바탕이 되는, Deep learning 기법이다. 입력 신호(Input data)가 결합 세기(Weight, Bias)를 통해 다음 인공 뉴런(node)으로 전달되며, 전 node의 output이 새 node에 input이 되는 node 그룹 구조이다. 이를 통해 데이터의 '비지도 학습'을 하고, 전처리 데이터를 여러 층 쌓아 최적화된 결과를 예측해, 이전의 SVM 등 알고리즘보다 높은 효율을 가능케 한다.

2. 프로젝트 수행

A. MNIST 데이터

MNIST 데이터는 훈련 데이터(train data) 55,000개, 테스트 데이터(test data) 10,000개로 이루어진 컴퓨터 비전 데이터 셋으로, 손으로 쓰여진 숫자 이미지의 집합이다. 각 이미지는 0~9까지의 수를 나타내며, 28x28의 픽셀 사이즈를 가진다. 해당하는 숫자가 무엇인지 나타내는 label이 붙어 있는 데이터이다.

이러한 이미지 데이터를 TensorFlow를 통해 들여다보고, 그 이미지가 어떤 숫자인지 예측하는 CNN 학습 모델을 구축한다. CNN은 이미지 특성을 추출하는데 주로 사용되는 NN 기법으로, 입력 이미지의 필터를 달리하고, 이미지 사이즈를 변형하면서 데이터 특성을 찾는다. 현재 세계 최고 수준의 MNIST 예측 정확도는 약 99.7%이며, 본 프로젝트는 이를 목표로 'Yoshua' 의 'Hyperparameter Tuning guide'를 따른다.

B. 초기 CNN 디자인

수업 시간에 다룬 cnn_mnist.py을 바탕으로 train 과 test 데이터의 Cost를 볼 수 있도록 코드를 추가해 초기 모델을 디자인했다.

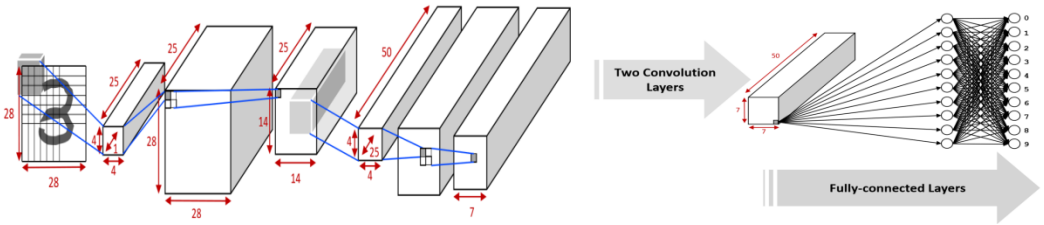
```
temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
train_loss.append(np.sqrt(temp_loss))

test_temp_loss = sess.run(loss2, feed_dict={eval_data: eval_x, eval_target: eval_y})
test_loss.append(np.sqrt(test_temp_loss))
```

사용된 CNN 모델은 Convolutional, Detector, Pooling stage 라는 세 가지 과정을 2번 반복 수행한다. (자세한 과정은 코드 주석 참고)

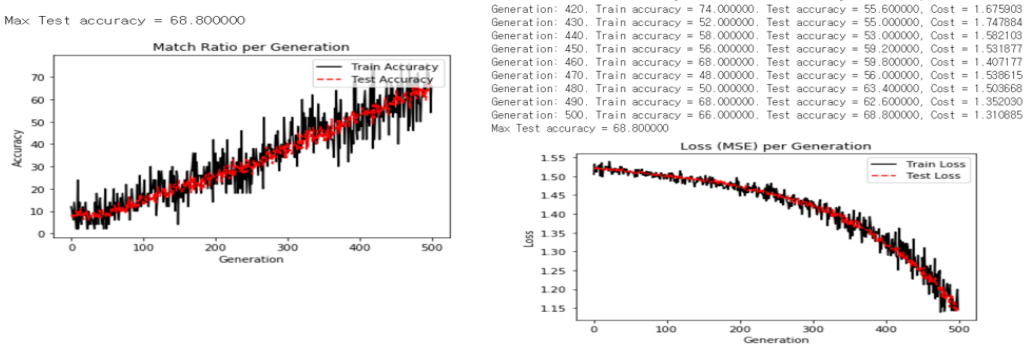
- *Convolutional stage*: 일정한 크기(weight)의 filter를 사용해 train data를 부분화하고, 이를 Convolution layer에 적용한다. 즉, Input data의 크기 보다 작은 filter의 크기를 Input data의 일부분과 Matrix Multiplication 하는 과정이다. 여기서는 28x28의 초기 train data가 4x4의 filter를 지나 초기 데이터와 동일한 사이즈와 차원을 가진 feature map을 생성한다.

- *Detector stage*: 형성된 feature map에 Activation function을 사용해 각 Convolution layer에 해당하는 bias 값을 더해준다. 초기 모델은 ReLU함수를 사용해 ReLU(Wx+b) (=결과값) 의 Convolution layer을 형성해 feature map을 완성한다.

- *Pooling stage*: Convolution layer의 feature map들을 pooling stage의 input으로 사용해 크기를 줄여 나간다. Sub sampling 이라고도 하며, pooling 하는 filter의 크기만큼 데이터가 요약된다. 위 경우, 2x2 max pooling을 거쳐 크기가 1/2이 줄어든 14x14x25, 7x7x50의 데이터를 차례로 만들었다. 최종적으로 변한 7x7x50의 각 데이터(item)는 10개의 노드로 연결이 되는 2번의 fully connected layer를 거쳐 예측 값을 도출한다.



사용된 Hyperparameter들과 초기 모델의 정확도와 Cost는 다음과 같다.



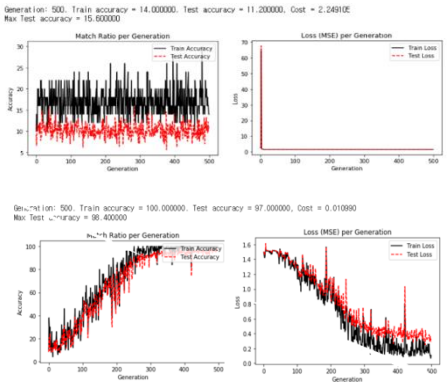
C. Tuning HyperParameter

일반적으로 데이터 모델의 학습 예측 정확도를 높이기 위해서는 모델이 표현할 수 있는 표현력, 즉 Capacity를 높이는 것이다. 얼마나 다양한 경우를 모델이 반영할 수 있는지, 아래의 조건들이 그 방법들이다. (Tuning은 Hyperparameter별로 각각 수행)

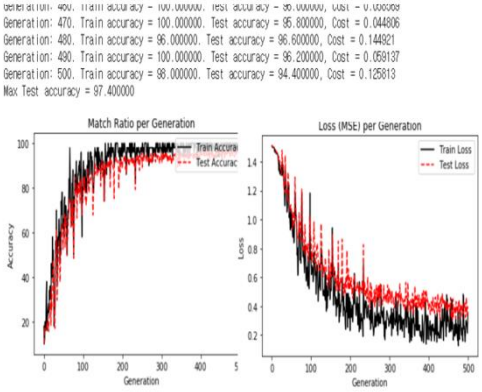
i. Learning rate 증가

'Yoshua' 의 'Hyperparameterr Tuning guide' 를 따르면 learning rate는 0.01로부터 시작하는 것이 가장 좋은 tuning 방법이라고 한다. 보통 learning rate 는 데이터 예측에 상당히 많은 영향을 끼치고, 값이 올라가면 올라갈수록 train error가 줄어든다고 한다. 즉, 한번에 많이 검색 할수록 속도도 빨라지지만, 그 만큼 error 도 낮아진다는 것이다. 그러나 어느 시점에서 learning rate가 큼에도 error가 발산하게 되는데 이 지점을 찾아야 한다.

우선, 초기 0.005의 learning rate 너무 낮기에 한번에 1000배수를 곱하고, Accuracy가 이상하게 나올 경우 0과 1000배수 사이에 발산 직전의 지점을 찾는다. 200배수 간격으로 줄여서 나가기로 한다.



<learning rate*1000, learning rate*100>

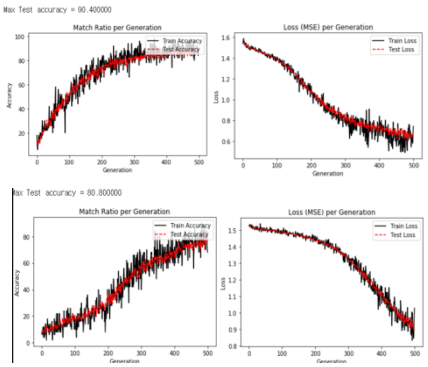


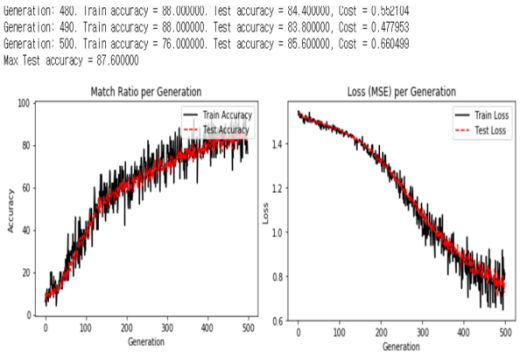
< learning rate*20 = 0.1>

Learning rate는 1000배 에서 완전 발산하며, 약 100배로 줄일 때까지 높은 예측을 보이지 못했다. 그러나 100배이하부터 Max Test Accuracy가 약 95~98.4 에 달하는 높은 예측 결과를 보여준다. 100~80 사이의 어느 것을 선택해도 될 것 같으나, 'Yoshua' 의 Guide에 따른 0.01의 값에 10배를 한 learing rate*20, 즉 default를 0.1로 설정해보니, 높은 초기 학습이 이뤄졌다는 것을 도표로 확인 할 수 있었다. Max Test Accuracy도 97.4에 해당하는 높은 정확도를 보이며, 무엇보다 전반적인 Generation에서 높은 수치를 유지한다. 도표를 보면 Test Accuracy가 특정 부분에서의 발산이 가끔 일어나는데, 그 정도 역시 20배수가 더 적게 발견된다.

ii. Final fully connected layer에 연결된 hidden node의 수 증가

Hidden node의 수를 늘릴수록 그만큼 거쳐야 하는 node들이 많아지므로 좀 더 정교한 예측이 가능할 것이다. 'Yoshua' 의 Guide에 따르면, Hidden node의 수의 최적 값이 있다고 하면, 그 최적 값을 넘어서는 node를 추가한다고 해서 error가 올라가는 것이 아니라, 필요한 node들만 작동해 error에 영향을 미치지 않는다고 한다



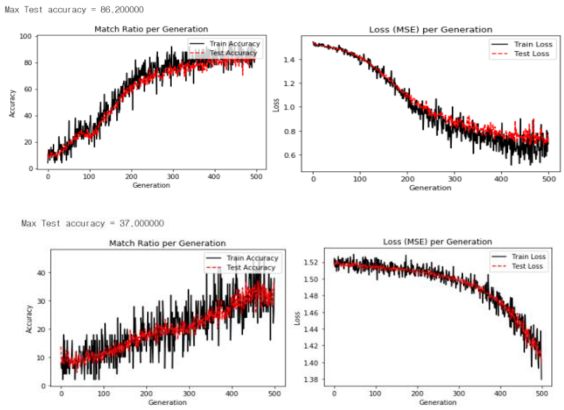


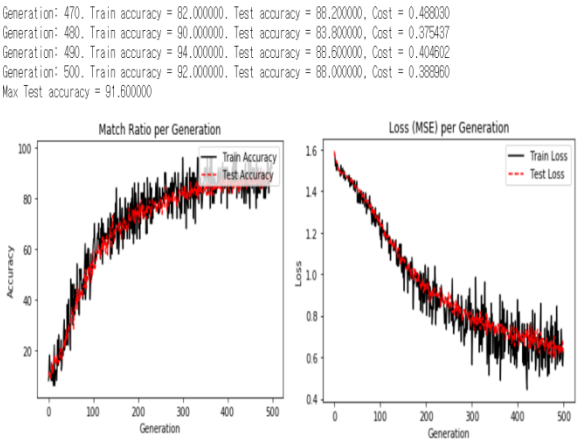
<hidden_layer_nodes = 100, hidden_layer_nodes = 30 > < hidden_layer_nodes = 40 >

Node의 수는 100부터 줄여나갔고, 40~100개의 Node 수에는 약 87~90.4 정도의 비슷한 Max Test Accuracy가 발견된다. Node의 수가 30~35개에 이르면, Accuracy 80으로 현저하게 줄었기 때문에, 비록 Running Time은 늘어났지만, 그 중 최소 범위인 40개가 적당하다. 즉, 40이상의 Node 수는 train에 관여하지 않고 불필요하다. .

iii. Convolution kernel width, depth 조정

일반적으로 filter(kernel)의 크기를 증가시키고, depth를 높이면 Capacity가 올라간다. 전자는 learning 하는 범위가 늘어나고, 후자는 learning을 하는 feature의 개수가 증가하기 때문이다. filter의 크기는 8x8로, depth는 Conv layer를 추가해 25, 50, 75로 설정하고 조정해 나간다.





<conv1,2_size = 8, conv1,2,3_num_features = 30, 45, 60> < conv1,2_size = 8, conv1,2_num_features = 50, 75>
<conv1,2_size = 4, conv1,2,3_num_features = 50, 75, 90>

filter의 크기와 depth(차원)의 수를 모두 증가시키면, Max Test Accuracy가 약 89.2에 달했고, 더 늘릴수록 높은 정확도를 가진다는 것을 예상할 수 있었다. 그러나 그에 따라 learning을 하는 범위와 learning의 대상이 되는 feature의 수가 증가해 Running Time이 상당히 늘어났다.

그래서 어느 Hyperparameter가 정확도에 더 영향을 주고 있는지 알고, 이를 이용해

선택적으로 Tuning할 필요가 있다. 두 가지 경우를 나눠 살펴보았다. 위 표에서 알 수 있듯이 filter의 크기는 4x4로 유지한 채, feature의 수만 늘리면 별다른 효과를 볼 수 없었고, 대체적인 cost의 감소가 더뎠다. 반면, Running Time은 다소 빨라졌다.

이와 반대로, filter의 수를 8x8로 바꾸고 차원을 바꾸지 않았을 때는 처음 결과와 비슷한 높은 정확도를 보여주는 듯했다. 그렇기 때문에 feature의 수는 filter의 수보다 정확도에 적은 영향을 끼치는 것을 알 수 있었다. 속도 향상을 위해 영향을 덜 주는 feature 수는 상대적으로 낮게 올리고, filter의 수는 높게 올려야 할 것이다. 위의 오른쪽 그림으로 수정한 뒤, 실행한 결과, 그러나 아직도 속도 문제는 가시질 않았다.

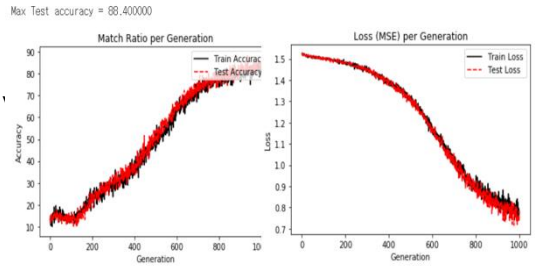
iv. Padding = "SAME"

일반적으로 Input 데이터의 내부에서만 Convolutional stage를 수행해 결국 feature map의 크기가 줄어드는 "VALID" 방식보다 Input 데이터와 같은 크기의 feature map을 만들기 위해 zero padding 을 사용하는 "SAME" 방식이 예측 정확도가 높다. 이 경우에는 초기 모델의 Convolution Padding 방식을 "SAME"으로 했기에 수정하지 않고 진행한다.

```
conv1 = tf.nn.conv2d(input_data, conv1_W, strides=[1, 1, 1, 1], padding='SAME')
relu1 = tf.nn.relu(conv1 + conv1_b)
max_pool1 = tf.nn.max_pool(relu1, ksize=[1, max_pool_size1, max_pool_size1, 1],
                           strides=[1, max_pool_size1, max_pool_size1, 1], padding='SAME')
```

v. 적절한 Batch size, Generation

Batch size와 Generation을 늘리면 데이터를 train할 수 있는 sample 수와 기회가 많아지므로 예측 정확도가 높아지는 것은 당연하다. 'Yoshua'의 가이드에 따르면, Batch size는 1~수백, Generation의 수는 원칙에 따라 overfitting이 생기는 시점을 발견한 후, 그 때 test를 멈추는 게 최적 값이라 한다.




```
Generation: 780. Train accuracy = 80.600000. Test accuracy = 77.400000, Cost = 0.774027
Generation: 790. Train accuracy = 78.000000. Test accuracy = 74.000000, Cost = 0.785501
Generation: 800. Train accuracy = 80.800000. Test accuracy = 79.800000, Cost = 0.814318
Generation: 810. Train accuracy = 77.000000. Test accuracy = 75.000000, Cost = 0.830330
Generation: 820. Train accuracy = 79.600000. Test accuracy = 79.200000, Cost = 0.764236
Generation: 830. Train accuracy = 78.800000. Test accuracy = 80.600000, Cost = 0.752940
Generation: 840. Train accuracy = 82.200000. Test accuracy = 78.600000, Cost = 0.691187
Generation: 850. Train accuracy = 78.000000. Test accuracy = 78.600000, Cost = 0.750490
Generation: 860. Train accuracy = 78.200000. Test accuracy = 78.600000, Cost = 0.770195
Generation: 870. Train accuracy = 80.400000. Test accuracy = 75.600000, Cost = 0.686635
Generation: 880. Train accuracy = 78.400000. Test accuracy = 79.600000, Cost = 0.723467
```

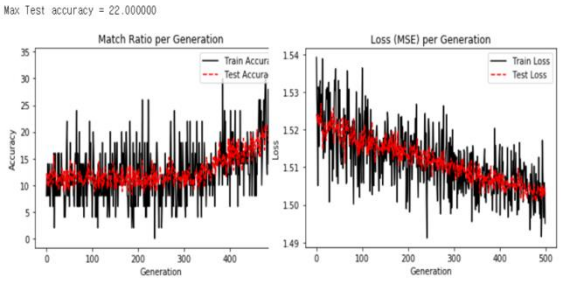
<batch_size = 500, iter_count = 1000 >

Batch size와 Generation를 각각 500, 1000으로 늘려보고, 100씩 줄여서 반복 수행

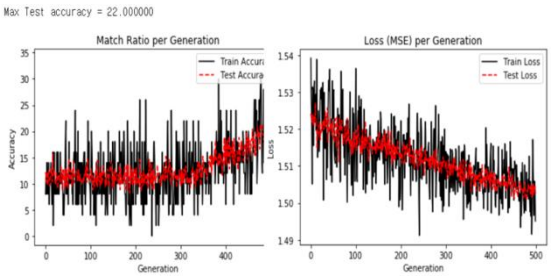
한 결과, batch size는 늘릴수록 좋다. 또한 Generation은 약 800번을 중심으로 test accuracy의 증가가 주춤하고 있는 모습을 볼 수 있었다. 즉, 800번의 train 시도를 최적의 시점이라 판단했다. Batch size의 경우, batch size만을 늘린다고 해서 뚜렷한 예측의 변화가 이뤄지지 않았다. 반면, Batch size는 Running Time에 큰 영향을 끼치며, 다른 hyperparameter 들을 조정 후, 너무 큰 값이 되지 않도록 조정해야 할 것이다..

vii. 적절한 Activation function

초기 모델에서 사용한 Activation function은 ReLU 함수로, 0 이하의 값이 Input으로 들어오면 0을 출력하고 0 이상의 값이 오면 비례함수로 내보낸다. 반면, Sigmoid 함수는 계단형식의 함수를 곡선화 해주는 기능을 가지며, 어떤 현상을 단순화하여 1과 0으로 분류하는 데 사용이 된다.



<Sigmoid 함수: all layers 적용>



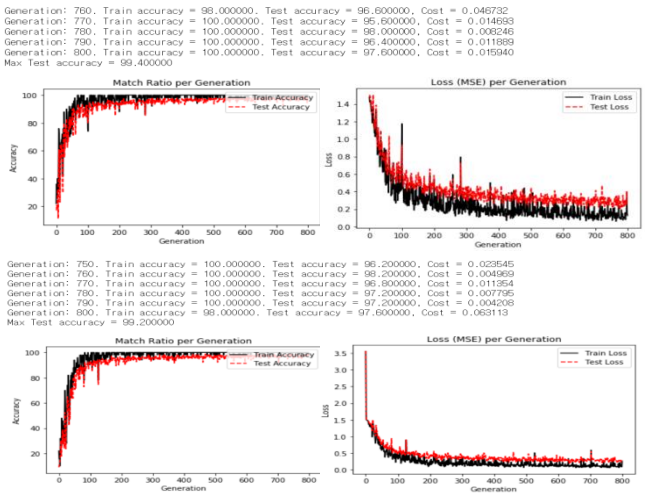
< Sigmoid 함수: last fullyl layers 적용 >

모든 Convolution layer에 적용된, 혹은 일부분에 적용된 Sigmoid 함수는 모두 ReLU 함수보다 낮은 정확도를 가졌다. 특히 Max Test Accuracy가 17.2까지 감소하는 비효율적인 현상을 보였으며, 특히 마지막 Fully connected layer에 적용할 경우, ReLU함수보다 정확한 예측을 보일 것이라는 가정은 틀렸다는 사실을 확인했다. 즉, 초기 모델에 쓰인 ReLU 함수를 그대로 사용한다.

D. 학습 결과(optimal한 Neural Network 구조)

위 프로젝트 수행 단계를 거친 최적의 Hyperparameter들을 종합해 Optimal한 CNN 구조를 만들었다.

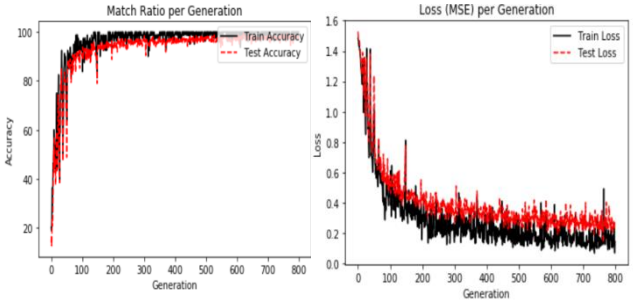
```
batch_size = 50              # 실제 학습 시킬 데이터 sampling
learning_rate = 0.1        # 한번에 가능한 이동,학습률
evaluation_size = 500       # test accuracy를 위한 sampling
iter_count = 800            # Generation : 학습 반복 수
target_size = max(train_y) + 1
num_channels = 1            # 색 -> only gray
conv1_num_features = 50
conv2_num_features = 75     # conv filter의 수
max_pool_size1 = 2
max_pool_size2 = 2          # pooling filter의 크기
hidden_layer_nodes = 40   # fully connected layer에 쓰인 노드의 수
conv1_size = ?
conv2_size = ?              # conv filter의 크기
# activation function : ReLU
```



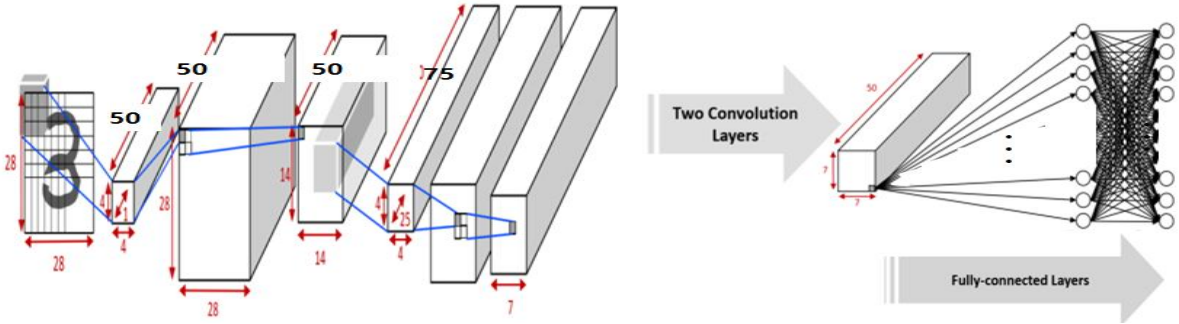
위의 두 도표는 설정된 Hyperparameter 중 Convolution filter size에 4를 넣고 8을 넣은 예측 정확도와 Cost를 나타낸다. 사실 8x8의 크기로 설정한 filter의 Max Test Accuracy와 Cost 변화 정도가 뚜렷했지만, Running Time이 약 5~8분 가량이 소비된다. 수행 시간은 메모리와도 관련이 깊기 때문에 4x4의 filter 크기를 선택하는 것이 옳다고 판단했고, Tuning 과정에서 수정하지 않았던 batch size를 80으로 수정해 보완했다.

```
## Hyperparameters
batch_size = 80              # 실제 학습 시킬 데이터 sampling
learning_rate = 0.005*20        # 한번에 가능한 이동,학습률
evaluation_size = 500        # test accuracy를 위한 sampling
iter_count = 800             # Generation : 학습 반복 수
target_size = max(train_y) + 1
num_channels = 1             # 색 -> only gray
conv1_num_features = 50
conv2_num_features = 75      # conv filter의 수
max_pool_size1 = 2
max_pool_size2 = 2           # pooling filter의 크기
hidden_layer_nodes = 40   # fully connected layer에 쓰인 노드의 수
conv1_size = 4
conv2_size = 4               # conv filter의 크기
# activation function : ReLU
```

```
Generation: 750. Train accuracy = 100.000000. Test accuracy = 97.400000, Cost = 0.017363
Generation: 760. Train accuracy = 97.500000. Test accuracy = 98.000000, Cost = 0.049725
Generation: 770. Train accuracy = 100.000000. Test accuracy = 97.400000, Cost = 0.016658
Generation: 780. Train accuracy = 98.750000. Test accuracy = 97.000000, Cost = 0.048054
Generation: 790. Train accuracy = 100.000000. Test accuracy = 98.200000, Cost = 0.011371
Generation: 800. Train accuracy = 100.000000. Test accuracy = 96.200000, Cost = 0.020529
Max Test accuracy = 99.600000
```



이로써, Optimal한 CNN 구조를 만들었고 Max Test Accuracy 또한 99.6에 해당하는 높은 수치를 보여준다. 참고로 batch size를 100으로 늘리면 4~5분 가량 시간이 소요되지만 위 결과 이상의 정확도를 보여주기도 한다. 구조를 살펴보면 다음과 같다



Convolution, pooling layer의 수는 예측 성능에 비해 수행 속도가 크게 증가해 수정하지 않았지만 그럼에도 높은 예측 정확도를 기록했고, Cost를 구하는 softmax 함수는 학습 속도를 높이기 위해 고정시켜 사용했다.

3. 결론

그 동안 배웠던 deep learning 기법과 데이터 분석 사례를 돌아봤다. 많은 사람들이 연구 과정에서 수 많은 시행착오를 겪었고, 실패에 대한 반성과 보완을 통해 지금 우리에게 보다 나은 학습 환경을 주고 있었다. 작은 parameter 하나하나에 담긴 의미와 그 의미로 새로운 의미를 만들어 내는 그들의 수고에 항상 감사한 마음을 가져야겠다. 좋은 가르침 감사합니다.