**Implicit Gradient Descent**

**김지환, 김하연**

1. **Introduction**

미분방정식의 해를 수치적으로 계산하는 방법인 Euler method은 크게 forward(explicit) Euler method와 backward(implicit) Euler method 두 개로 나뉜다. Forward Euler method는 가장 보편적으로 알려진 형태로 가 주어졌을 때, 로 다음 함수 값이 계산된다. Backward Euler method는 y값을 update할 때, 현재 상태의 값이 아닌 다음 상태의 값으로 미분을 계산해서 update하는 방식으로 n자리에 n+1이 들어가고 와 같이 계산할 수 있다. Backward Euler method는 뒤에서 추가적으로 더 설명하겠지만, forward Euler method와 h가 수렴하지 않는 대부분의 영역에서 수렴하게 된다. 따라서 h를 forward에 비해 굉장히 크게 할 수 있고 목표하는 값에 더 빠르게 도달할 수 있게 된다. Gradient descent의 기본적인 형태는 Loss function을 로 놓았을 때 와 같이 explicit 하게 주어진다. Euler method는 함수를 근사하는 것이고 Gradient descent는 주어진 loss function을 최소화하는 parameter를 찾는 것이라서 문제의 본질은 많이 다르다고 할 수 있다. 하지만, 식의 형태가 비슷하고, Nesterov momentum이 조금 더 앞선 time step의 gradient를 사용하는 것의 효용성에 대해 검증한 바 있었기 때문에, Gradient descent를 backward Euler method와 같이 계산을 하면 explicit 한 형태와 비교했을 때 learning rate가 큰 경우에는 더 빠르게 수렴하는 효과를 기대할 수 있을 것 같아서 본 연구를 진행하게 되었다.

1. **Preliminaries**
   1. **Backward (Implicit) Euler Method**

Forward Euler method 와 backward Euler method의 수렴성에 대해 조금 알아보자. 조금 문제를 단순화하기 위해 라고 하자(A는 행렬). 이를 이용해서 Forward Euler method에 대한 difference equation을 쓰면 , 가 되고, Backward Euler method의 경우 , 가 된다. 여기서 u의 수렴은 forward의 경우 , backward의 경우 의 고유값이 결정하게 되고, 단순하게 생각했을 때 고유값의 절대값이 1보다 작은 경우 수렴한다고 볼 수 있다. 와 의 고유값이 서로 역수 관계이기 때문에 위의 두 행렬의 고유값을 1로 맞춰 주기 위해 가능한 범위가 서로 반대가 되게 된다. Condition number가 큰 행렬로 표현되는 방정식을 stiff equation이라고 하는데, stiff한 system에서는 forward Euler method의 가 매우 작아야 한다. 본 프로젝트에서는 MNIST 데이터를 2-layer fully connected network에 학습을 시키는 예제를 활용했는데, 이런 image classification을 위한 network의 weight matrix는 stiff할 가능성이 크다. 각 neuron들은 특정한 방향이나 형태의 input들에 더 높은 forward pass 값들을 전달해줄 것이고, 이는 condition number가 높다는 것을 의미한다. 따라서 learning rate가 높게 설정해 준다면, implicit 한 방법이 더 효과적일 수 있다.

* 1. **Implicit Gradient Descent**

앞서 설명한 backward Euler method의 식 형태에 맞게 gradient descent식을 변형해보자.

,

에서 Taylor expansion을 해주면

위 식에서 좌변에 에 관한 항을 모두 모아서 다시 정리하면 다음을 얻는다.

최종적으로 얻은 식에 conjugate gradient method를 적용해 계산하였다. Conjugate gradient method는 sparse matrix의 inverse를 계산하는데 굉장히 효과적인 방법으로 matrix의 condition number를 k, non-zero entry의 개수를 m이라고 하면 총 시간 복잡도가 인 iterative method이다. k의 경우 preconditioning matrix를 행렬의 대각성분만 추출해서 사용을 해도 꽤 많이 줄어들기 때문에 구하려고 하는 이 충분히 sparse 하다면 matrix multiplication과 유사한 수준으로 시간 복잡도를 줄일 수 있다. 의 각 entry는 를 구성하는 parameter 중 2개로 미분한 2계 미분값이다. Neural Network의 parameter 수가 많아질수록 loss에 크게 영향을 안주는 parameter값이 많아질 것이고, 그에 따라 그 parameter로 loss를 편미분한 값이 0이 되는 경우가 많아져 non-zero entry인 m이 줄어들 것이라고 예상하였기에 conjugate gradient method를 이용하였다.

* 1. **Calculating Hessian**

Implicit gradient descent의 식을 계산하기 위해선 결국 hessian을 계산해야 되고, hessian도 gradient와 마찬가지로 backpropagation의 형태로 계산이 가능하다. 번째 layer의 pre-activation value z에 대한 hessian을 , weight matrix에 대한 hessian을 로 놓자. Loss function은 L, activation function은 f, 번째 layer의 gradient는 이다. 벡터는 column vector를 기준으로 계산하였다.

H는 symmetric matrix인 것을 고려해서 위 식을 행렬 형태로 다시 정리하면 아래와 같은 식을 얻을 수 있다.

이제 K를 계산해보자. 우선 K는 W에 gradient를 계산하는 것에서부터 출발한다. 아래 식에서 a는 z의 activation 값이다.

계산을 하면 위와 같은 4차원 tensor가 나오게 된다. 4차원 tensor에는 conjugate gradient를 적용할 수 없기 때문에 matrix 를 flatten을 해서 weight matrix의 총 entry 수를 m이라고 했을 때 mxm matrix를 만들어줄 수 있다. 그렇게 했을 때 얻어지는 행렬의 ij번째 row는 아래 식과 같이 계산할 수 있다.

1. **Code** 
   1. **Eve Master Code**

먼저 기존에 존재하는 github source 와 논문을 참고하여, 1 layer binary classification을 implicit gradient descent 방법을 진행하였다. 1 layer 이기 때문에 딱 하나의 layer 에 대해서만 gradient descent 를 짰다. 주어진 dataset에 대하여 weight vector 에 대한 hessian function을 짜고, 계산한 gradient 와 I – hessian function 을 이용하여 conjugate gradient method 를 이용하여 dx( = ) 의 값을 계산하였다. Conjugate gradient descent 는 scipy.sparse.linalg library 를 이용하였다. testTime 이라는 변수를 선언하여, epoch 이 testTime 배가 될때마다 (즉 I % testTime == 0 일때마다) test loss 와 train loss 를 구하여 lossesTrain, lossesTest 의 list 에 추가하였다. 이후 이렇게 구한 losses 들을 matplotlib library 를 이용해 test 와 train 의 각 경우에서 그래프를 구하였다. Learning rate 는 5000, 10000 iteration에 절반씩 줄어들도록 하였다. Hessian function 의 구현은, 기존에 주어진 gradient descent 방법인 weight와 logistic gradient 를 직접 2번 미분하여 계산하였다. 먼저

hess = np.zeros((len(weight), len(weight)))

를 선언하여 weight vector x weight vector 만큼의 matrix 를 만들어준다. 이후에

*for* k *in* range(len(features)):

val = np.exp(-label[k]\*features[k].dot(weight))

const += label[k] \* (val/(1 + val)\*\*2)

const = const/len(features)

를 이용하여 feature (처음 입력된 input value [281, 33]) 를 이용하여 한번 미분을 완료한 후에,

*for* i *in* range(len(weight)):

*for* j *in* range(len(weight)):

hess[i][j] = const \* weight[j] \* (weight[i] \*\*2)

를 이용하여 각 hessian matrix 에 계산한 matrix 를 구하여 계산한다. 그렇게 만들어진 matrix [33, 33] 를 output으로 return 한다. 이렇게 구해진 hessian function을

I = np.identity(len(weight))

hess = hessian(features, label, weight)

A = I - hess

grad = np.zeros(weight.shape)

*for* j *in* range(len(features)):

grad += logisticGrad(features[j], label[j], weight)/len(features)

를 이용해 각 우변과 좌변을 만들어주고 마지막으로 conjugate gradient 로 dx를 계산하여준다. 구한 dx는 weigh 에 직접 바로 빼주어 값을 구한다.

* 1. **MNIST**

MNIST dataset에 대한 세개의 gradient descent 방법을 사용하였다. 첫번째는 그냥 단순한 stochastic gradient descent방법, 두번째는 adam gradient descent 방법, 마지막으로는 implicit gradient descent 방법이다. 전체적인 구조는 fully connected layer W1, b1, W2, b2 2개로 되어있으며 각 사이의 output에 대해서 forward(X, params) 함수를 통해 Z2, A2, Z3, A3 를 계산한다. 우선 function.py 에 sigmoid, dsig, compute\_loss, backprop\_IGD backprop\_origin function을 선언한다. Dsig() 함수는 sigmoid 결과에 대해 미분한 결과를 의미한다. 우선 forward 함수는 각 Input X는 초기에 넣어주는 input이고, params 는 W1, b1, W2, b2 가 들어있는 dictionary이다. Original backpropagation 방법인 def backprop\_origin(X, Y, params, cache, N\_batch, lr)은 일반적인 backward 계산을 시행하여 diff dictionary에 저장한다. Implicit gradient 방법을 시행한 bacoprop\_IGD(X, Y, params, cache, N\_batch, lr) 은 처음 dZ3는 일반적인 방법으로 계산하고 H3를 다음과 같이 구한다(softmax function의 hessian).

H3 = np.zeros((dZ3.shape[0], dZ3.shape[0]))

*for* i *in* range(N\_batch):

H3 += np.matmul(dZ3[:, i], dZ3[:, i].T)

H3 \*= (1. / N\_batch)

그 다음 second layer 의 hessian function 을 다음과 같이 구한다.

*# hessian of second layer*

dfz = np.zeros((dZ2.shape[0], dZ2.shape[0]))

*for* i *in* range(N\_batch):

dfz\_i = dsig(cache["Z2"][:, i])

dfz += np.matmul(dfz\_i, dfz\_i.T)

dfz \*= (1. / N\_batch)

H2 = (params["W2"].T @ H3 @ params["W2"]) \* dfz

이렇게 구한 H3 와 H2 를 이용하여 각 db를 계산한다. 이때 위와 마찬가지로 scipy.sparse.linalg library 를 이용하여 conjugate gradient를 진행하였다.

db2 = sc.cg(np.identity(db2.shape[0]) - lr \* H3, lr \* db2)[0]

db2 = np.array([[i] *for* i *in* db2])

db1 = sc.cg(np.identity(db1.shape[0]) - lr \* H2, lr \* db1)[0]

db1 = np.array([[i] *for* i *in* db1])

Bias 값들은 vector 로 되어있어서 상대적으로 implicit gradient 를 계산하기 용이하였지만, weight 값들은 matrix로 되어있어 hessian 결과값이 4차원 텐서가 나온다는 단점이 있다. 이를 위에서 언급한 것과 같이 flatten을 이용해 2차원 matrix로 바꿔준 뒤에 계산이 가능은 하지만, fully connected network는 weight가 node 수의 제곱의 scale을 갖기 때문에 K2에 대해서만 아래 방식을 이용해 근사적으로 계산하였다. 모든 Wij에 대한 2계 미분을 구하는 것이 아니라 column 별로 각각 따로 hessian을 계산하여 conjugate gradient를 계산하였다. 아래 식은 weight matrix의 j번째 column에 대한 식이고, 아래 코드에서 K2에 해당한다.

a = np.zeros(dZ2.shape[0])

*for* n *in* range(N\_batch):

a\_n = cache["A2"][:, n]

a += a\_n \* a\_n

a \*= (1. / N\_batch)

K2 = [H3\*a\_i *for* a\_i *in* a]

dW2 = [sc.cg(np.identity(dW2.shape[0])-lr\*K2[i],lr\*dW2[:,i])[0] *for* i *in* range(dW2.shape[1])]

dW2 = np.array(dW2).T

1. **Experiment & Result**
   1. **Eve-master**

위에 서술한 Eve-master 식을 이용하여 iteration 1000, regularization = 0.001, test time = 10과 learning rate 를 조절하여 결과를 내었다. 각각 learning rate 가 0.01, 0.1일때의 결과는 부록의 그림 1, 2 와 같다.

그림을 보게 되면 learning rate 이 0.01일때는 implicit gradient descent 방법의 test와 train result 결과가 나머지 다른 gradient descent 방법보다 크다는 것을 확인할 수 있다. 그렇지만 learning rate 가 0.1로 증가하고 나서는 특히 test 단계에서의 결과가 나머지 gradient descent 방법에 비해 converge 를 더 빨리 한다는 것을 확인할 수 있다. 그리고 learning rate 가 커지면, adam, eve 의 batch gradient descent 방법은 굉장히 osciliation 이 심한것을 확인할 수 있다. 그 이유는 learning rate 가 커지면 커질수록 weight update 시에 optimum 방향으로 이동하지 않을 가능성이 크고, batch 단위 내에서 계산하는 방식이 그 정확도를 떨어트렸을 가능성이 크기 때문이라고 사료된다. 특히 eve 와 adam 은 기존에 계산한 mean 과 varience 의 합을 이용해서 계산을 진행하기 때문에 learning rate 가 커질수록 더 잘못된 결과가 나올 가능성이 커진다고 예상할 수 있다. Learning rate 가 커질 때 implicit gradient descent 의 방법이 다른 gradient descent 보다 더 converge를 잘하는 이유는 우리가 만든 implicit gradient descent 방법이 hessian funtion을 이용해서 t+1 time 의 결과를 역으로 구하여 계산하기 때문에 다른 방법에 비해 learning rate 의 값이 중요하게 작용하지 않는다는 것으로 생각할 수 있다. 그래서 타 방법들이 learning rate 이 증가하면 할수록 그래프 전체적으로 변동이 심해지는데 비하여 Implicit gradient descnet 방법은 learning rate의 값에 상관없이 변동이 거의 없었다.

* 1. **MNIST**

MNIST의 dataset 을 이용하여 3가지의 방법으로 계산한 결과는 부록의 3 부터 6까지와 같다. 총 epoch 100에 대하여 10번마다 test 와 train 을 진행하여 총 10개의 점에 대하여 그래프를 찍어보았다. 특히 db1 과 db2의 값만 implicit gradient descent 방법을 진행하여 계산한 결과와, dW2, db1, db2의 세 값을 implicit gradient descent 방법을 진행한 결과를 비교하여 보았다. 우선 그냥 db1, db2, dW2를 implicit gradient descent 방법을 시행하여 learning rate 이 각각 0.01, 0.1, 0.2, 0.7를 진행한 결과는 아래와 같다. Eve master 결과와 마찬가지로 learning rate 이 증가할수록 implicit gradient descent 방법이 다른 방법에 비해 converge 를 더 빨리 한다는 것을 확인할 수 있다. 초반 learning rate 이 0.01일때는 IGD 방법이 가장 loss 가 크다가, 이후 0.1로 증가하고 나서는 다른 방법보다 낫거나 거의 작아지는 모습을 확인할 수 있다. SGD 방법과 비교해서는 거의 대부분의 경우에 더 작은 것을 확인할 수 있다. 즉 그냥 일반적인 backpropagation 방법에 비하여 implicit gradient descent 방법이 더 효과적으로 작용한다는 것을 내포한다. Eve master 에서 시행한 결과와 마찬가지로 adam gradient descent 방법은 learning rate 이 증가할수록 optimum 으로의 converge 도 느리고 oscillation 도 심한 것을 확인할 수 있다. Db1 과 db2 만 IGD를 실시한 결과와, db1, db2, dW2 까지 IGD를 실행한 결과를 비교하면 다음과 같다. 우선 learning rate 이 각각 0.1, 0.2 일 때는 근소한 차이이지만 db1, db2만 IGD로 계산하는 것 보다 dW2까지 포함하여 IGD를 계산하는 것이 더 성능이 좋은 것을 확인할 수 있다. 그렇지만 learning rate 이 0.01 로 작을 때는 오히려 dW2를 IGD로 계산하지 않는 방식이 IGD로 계산하는 것보다 더 성능이 좋음을 확인할 수 있었고 다른 gradient descent 방법과 비교해서도 더 나았다. 이는 적은 learning rate에서는 IGD의 방법이 오히려 다른 방법보다 더 converge 가 느리게 일어날 수 있음을 의미하며 lmplicit gradient 방법이 learning rate 가 작을 때는 수렴하지 않는다는 기존의 수학적 결과를 설명할 수 있다.

1. **Conclusion**

Implicit Descent를 사용하는 경우, 다른 optimization 방법에 비해 learning rate가 큰 경우에 수렴을 잘 하는 것을 확인할 수 있었다. Learning rate가 작을 때 성능히 현저히 떨어지는 것과 결합해서 보면 이는 Implicit Euler method의 특징과도 굉장히 유사하다는 것을 확인할 수 있다. Euler method에서 time step을 크게 가져갈 수 있다는 것은 시뮬레이션 분야에서 프레임은 떨어지더라도 전체적인 시뮬레이션의 속도를 높여주는데 크게 기여한다. 딥러닝 알고리즘에서는 learning rate와 성능이 직접적인 연관성이 없기 때문에 결정적인 역할을 하기는 어렵지만, system이 dimension이 굉장히 커져서 learning rate가 큰 상태로 학습을 시켜야 되는 시간이 길어지는 경우 활용 가치가 충분히 있을 것으로 예상된다.

기존에 SGD algorithm에서 가장 오래걸리는 computation이 matrix multiplication이지만, IGD에서는 가장 오래걸리는 computation이 matrix K를 만드는 과정이다(weight matrix의 hessian). 위에서 언급했던것과 같이 Fully connected network의 weight parameter 수가 이기 때문에 실제로 이를 hessian으로 만들면 크기의 행렬이 만들어진다. 따라서 node의 수와 parameter 수가 비슷한 CNN등의 구조에서 활용한다면 크게 속도 저하를 시키지 않고 활용할 수 있을 것이다.

결국 Hessian의 inverse를 계산하는 과정을 거치기 때문에 Newton’s method와 비교가 될 수 있다. 하지만 본 프로젝트는 Newton’s method와 출발 지점이 조금 다르다 inverse를 계산하는 것 보다 식의 형태를 implicit한 form으로 바꾸는 것에 더 의의가 있었고, Newton’s method가 learning rate 자체를 계산하는 거라면 본 프로젝트는 learning rate가 클 때 효과적인 방법을 찾는 것이기 때문에 방향성이 다르다. 또 learning rate라는 parameter를 여전히 사용하기 때문에 본 프로젝트에서 처럼 부분적으로 적용이 가능하다는 이점도 있다.

1. **Contribution**

김지환 : implicit gradient 의 수학적 공식, 보고서, MNIST 코드의 IGD 부분 구현

김하연 : Eve-master 의 구현, 보고서, MNIST 코드의 SGD, ADAM plot 구현

1. **Reference**

[1] EVE-master 코드 : <https://github.com/Jeanselme/Eve>

[2] An Introduction to the Conjugate Gradient Method Without the Agonizing Pain - Jonathan Richard Shewchuk, august 4 1994, Carnegie Mellon University

[3] linear algebra and its applications 4th edition, Gilbert Strang

[4] <https://zhenye-na.github.io/2018/09/09/build-neural-network-with-mnist-from-scratch.html>

[5] IMPROVING STOCHASTIC GRADIENT DESCENT WITH FEEDBACK, Jayanth Koushik & Hiroaki Hayashi (2016)

**8. 부록**

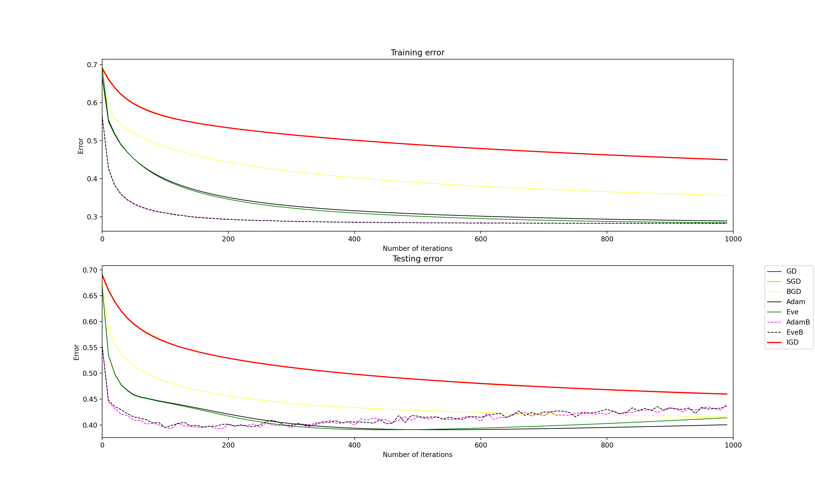


그림 1 learning rate = 0.01 epoch = 1000

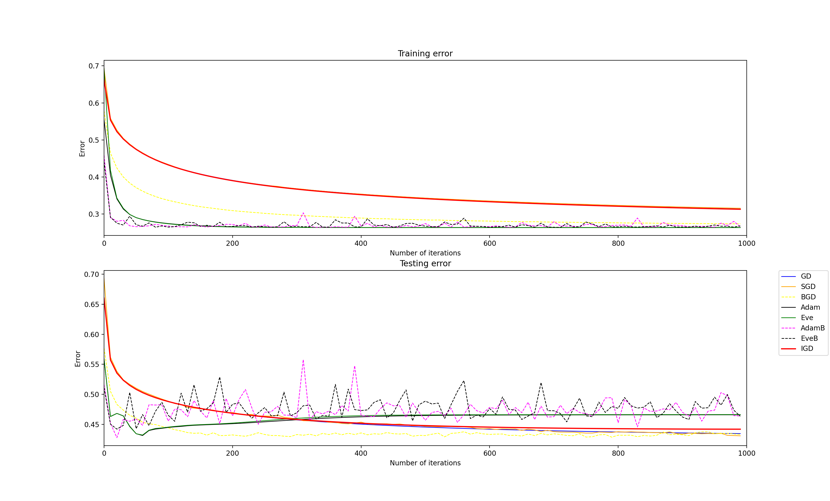


그림 2 learning rate = 0.1, epoch = 1000

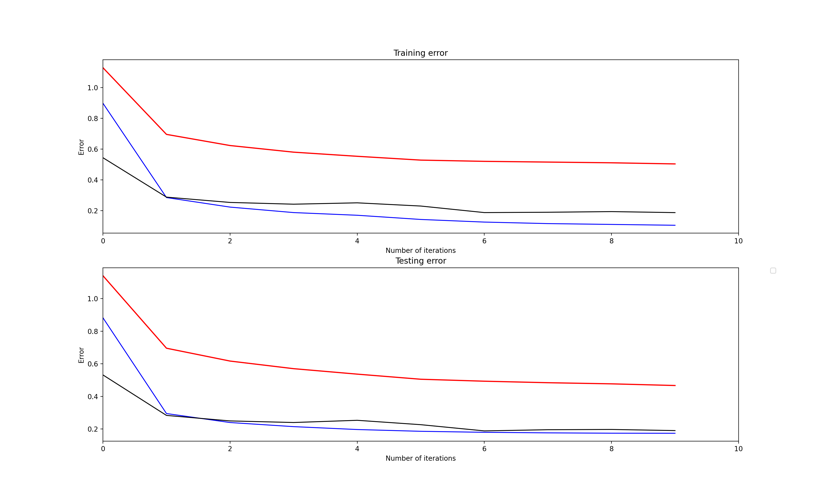


그림 3 db1, db2 dW2 learning rate = 0.01

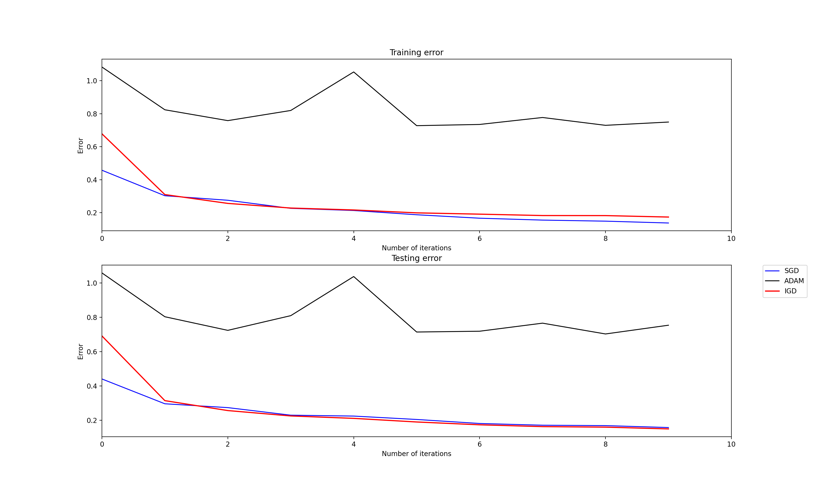


그림 4 db1, db2 dW2 learning rate = 0.1

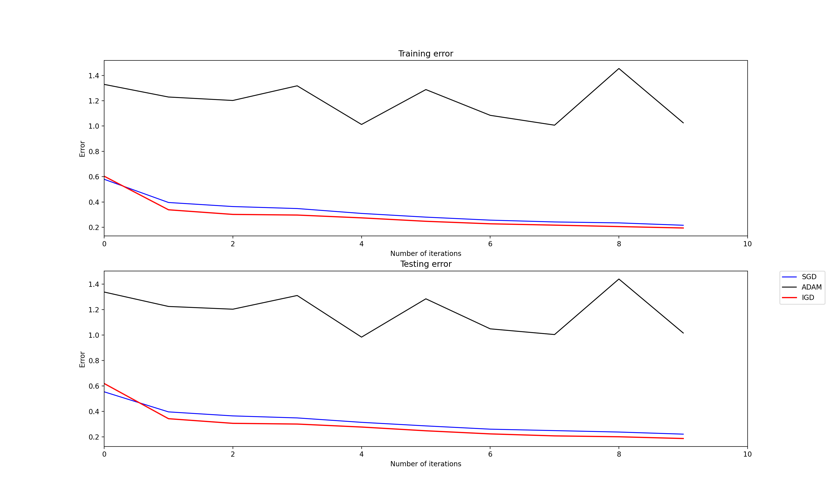


그림 5 db1, db2 dW2 learning rate = 0.2

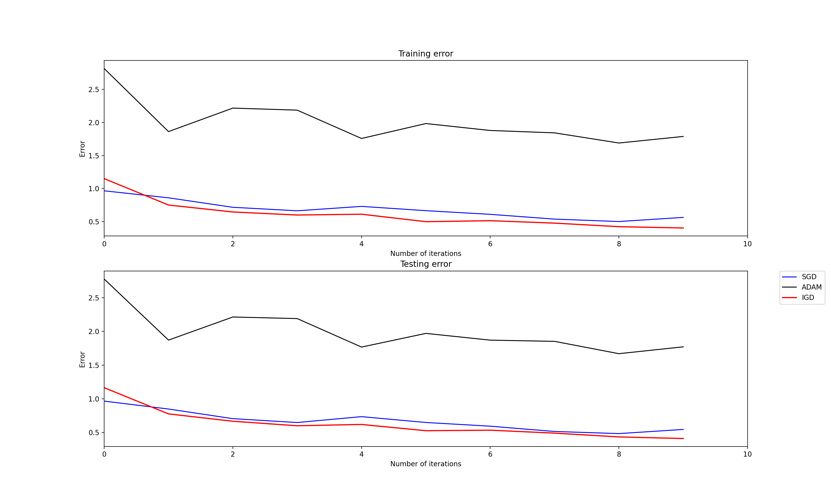


그림 6 db1, db2 dW2 learning rate = 0.7

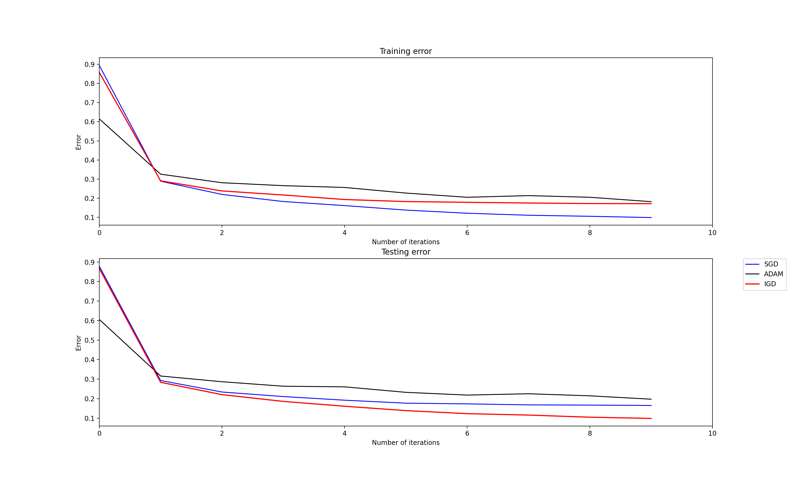


그림 7 db1, db2 learning rate = 0.01

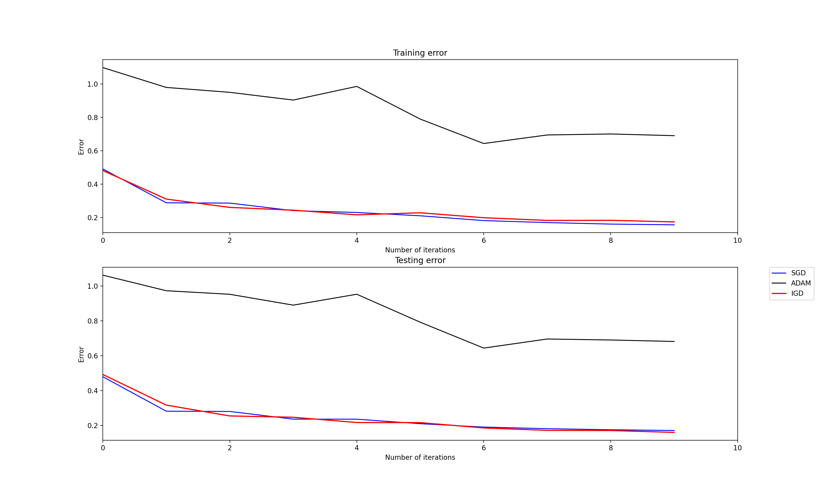


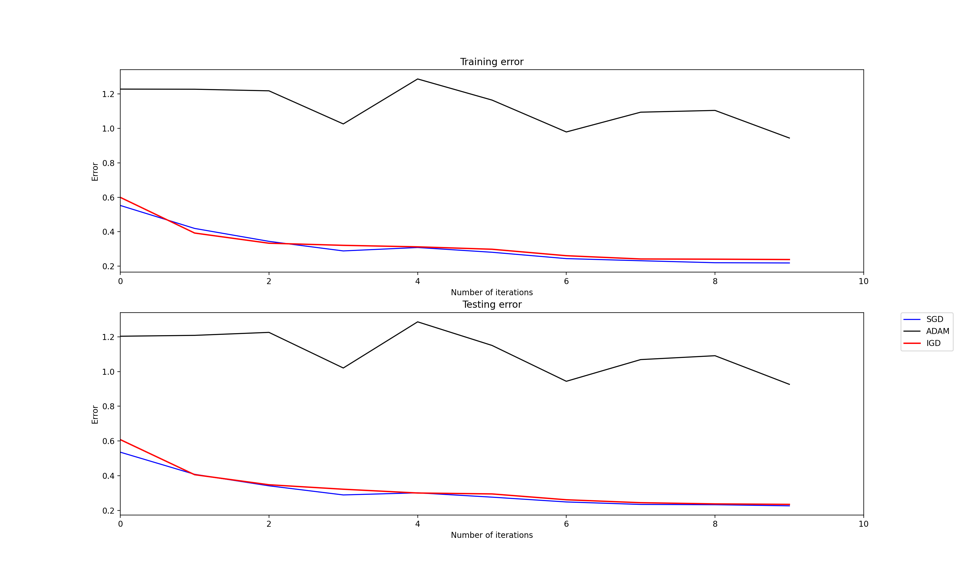
그림 8 db1, db2, learning rate = 0.1

그림 9 db1, db2, learning rate = 0.2