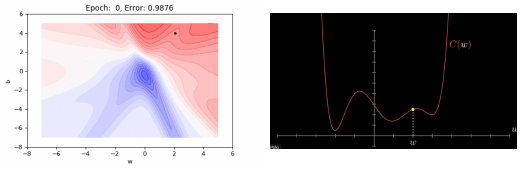
(언어는 오해의 원천이다.)

오늘 수업에서는 여러 용어들이 나올 텐데 용어들에 대해 명확한 컨셉을 잡고 넘어가는 것이 중요하다.

Gradient Descent

First-order iterative optimization algorithm for finding a local minimum of a differentiable function.

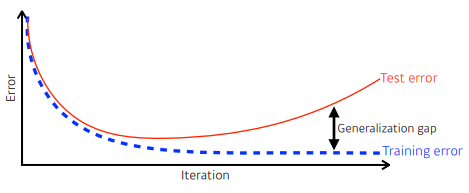


저번 수업에서 slope와 bias로 이루어진 linear network가 있을 때 이것에 대한 gradient즉, loss function에 대한 partial derivative를 구해서 이것으로 파라미터를 빼주는 gradient descent에 대해 알아보았다. 이것이 위에서 말하는 gradient descent이다. 결국은 함수값이 줄어들었을 때 어떤 optimer를 이룰 것이라고 기대하는 loss function이 존재하고 내가 찾고자 하는 파라미터를 가지고 loss function과 편미분한 값을 이용해 학습을 하겠다는 것이다.

**1. Important Concepts in Optimization**

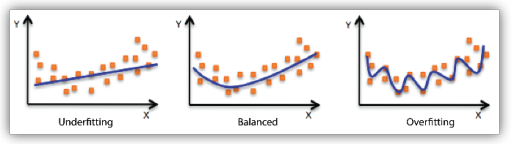
\* Generalization

많은 경우에 일반화 성능(Generalization)을 높이는 것이 우리의 목적이다.

  
일반화라는 것이 무엇인지 알아보자. 일반적으로 학습을 시키게 되면 iteration이 지나감에 따라 학습 데이터에 대한 Training error는 줄어든다. 하지만 Training error가 0이 된다고 해서, 원하는 최적값에 도달한다는 보장은 없다. 왜냐하면 일반적으로는 Training error는 감소하지만 어느 정도 시점 이후 부터 Test error는 증가하게 된다. 즉, Generalization performance는 일반적으로 Training error와 Test error의 차이를 말한다.

정리하자면 Generalization이 좋은 모델은 '이 네트워크의 성능이 학습 데이터와 비슷하게 나올거다.'라는 보장을 해준다. 근데 만약 우리 네트워크가 굉장히 안 좋아서 학습 데이터의 성능 자체가 안 좋을 수도 있다. 그러면 Generalization이 좋다고 해서 테스트 데이터의 성능이 좋다고 할 수는 없다.

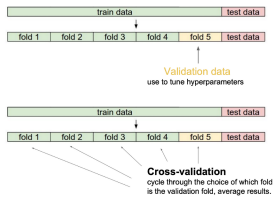
\* Underfitting vs. Overfitting



일반적으로 학습 데이터에서는 잘 동작하지만 테스트 데이터에 대해 잘 동작하지 않는 현상을 Overfitting이라고 말한다. 반대로 네트워크가 너무 단순하거나 학습을 너무 조금 시켜서 학습 데이터조차 잘 못 맞추는 경우를 Underfitting이라고 한다.

\* Cross-validation

Cross-validation is a model validation technique for assessing how the model will generalize to an independent (test) data set



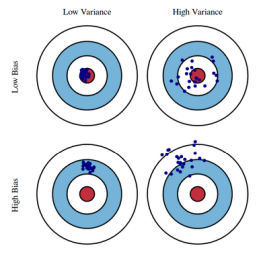
일반적으로 train data와 test data를 나눠줄 때가 많다. 일반적으로 train data와 validation data로 나눠서 학습데이터를 준다. train data로 학습시킨 모델이 학습에 사용되지 않은 validation data를 기준으로 얼마나 잘되는 지를 보는 것이다. 그러면 train data와 validation data를 얼마의 비율로 나누는 것이 좋을까? 이를 해결하기 위한 것이 cross-validation이다. 혹은 k-fold validation이라고도 불린다.

cross-validation은 data를 train data와 test data로 나누고 train data를 k개로 나누어 학습시키는 것이다. 위 그림을 기준으로 하면 처음에 fold1을 제외한 다른 fold들로 학습을 시키고 fold1으로 성능을 확인한다. 그리고 fold2를 제외한 다른 fold들로 학습을 시키고 fold2로 성능을 확인한다. 이 과정을 반복하는 것이 cross-validation이다.

Neural Network를 학습하는데 있어서 network를 얼마나 크게 가져갈지, loss function은 어떤 것을 쓸지 등 많은 Hyper parameter가 존재한다. 그래서 cross-validation을 통해 최적의 Hyper parameter셋을 찾고, 이 Hyperparameter셋을 고정한 상태에서 학습시킬 때는 모든 데이터를 다 사용한다.

test data는 학습에 어떤 방법으로도 사용되서는 안 된다. test data를 활용해서 cross-validation을 한다거나, Hyper parameter search를 하는 것은 그 자체로 cheating이다. 그래서 사실상 학습에서는 train data와 validation data만을 사용해야 하고, test data는 어떤 식으로도 사용되어서는 안된다.

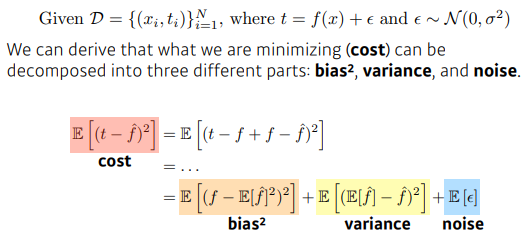
\* Bias and Variance



variance라는 것은 내가 어떤 입력을 넣었을 때 출력이 얼마나 일관적으로 나오는 지를 말한다. 그래서 variance가 낮은 모델들은 간단한 모델들이 많다. 그에 반해 variance가 큰 모델들은 비슷한 입력이 들어와도 출력이 많이 달라진다. 이렇게 되면 overfitting될 가능성이 크다.

bias란 출력이 많이 분산되더라도 평균적으로 봤을 때 어떤 출력 타겟에 접근하게 되면 bias가 낮다고 한다. bias가 높은 것은 내가 원하는 어떤 값이 평균과 많이 빗나가는 것이다.

\* Bias and Variance Tradeoff

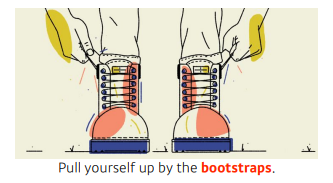


내 학습 데이터에 노이즈가 껴있다고 가정했을 때 내가 이 노이즈가 껴있는 target data를 minimize하는 것은 3가지 part로 나뉠 수 있다. 즉, 위 내용은 내가 minimize하는 것은 한 가지 값인데 사실 이 값이 3가지 conference로 이루어져 있어서 이 3개 각각을 minimize하는 것이 아니라 하나가 줄어들면 하나가 커질 수밖에 없다라는 trade off 이론을 말한다. 그럼 결과론 적으로 위와 같은 식이 나온다.

t가 target이고 f\_hat이 neural network의 출력값이다. 그래서 cost를 minimize하는 것은 bias, variance, noise를 minimize하는 것과 같다. 그래서 우리 모델 특성이 bias를 많이 줄이게 되면 variance가 많이 높아질 가능성이 크고 variance가 많이 줄은 모델은 bias가 높아질 가능성이 크다. 근본적으로 train\_data에 노이즈가 껴있는 경우 이 bias와 variance 둘 다 줄이기는 힘들다는 것이 Bias and Variance Tradeoff이다.

\* Bootstrapping

Bootstrapping is any test or metric that uses random sampling with replacement.



Bootstrapping은 통계학에서 굉장히 많이 활용된다. 용어 자체는 그림처럼 부츠의 끈을 말한다. 신발 끈을 위로 들어서 하늘을 날겠다는 꿈을 말한다고 한다. 학습 데이터가 고정되어 있을 때 그 안에서 subsampling을 통해 학습 데이터를 여러 개를 만들고 그것을 가지고 여러 모델을 만들어서 무언가를 하겠다는 것이 Bootstrapping이다. 예를 들어, 학습 데이터 100개를 랜덤으로 80개를 선택해 여러 모델을 만드는 것이다.

\* Bagging vs. Boosting

Bagging(Bootstrapping aggregating)

- Multiple models are being trained with bootstrapping.

- ex) Base classifiers are fitted on random subset where individual predictions are aggregated(voting or averaging).

(참고) aggregate 합계, 총액; 종합하다

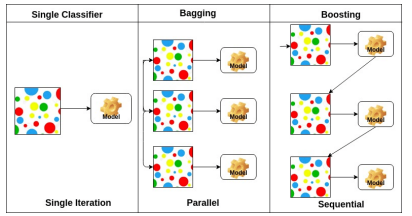
이전에 말한 것과 비슷하다. 학습 데이터가 고정되어 있을 때, 이 학습 데이터를 한 번에 다 써서 모델 하나를 만드는 것이 아니라 학습 데이터를 여러 개를 만드는 것이다. 이러한 것은 일반적으로 앙상블이라고도 부른다.

Boosting

- It focuses on those specific training samples that are hard to classify

- A strong model is built by combining weak learners in sequence where each learner learns from the mistakes of the previous weak learner.

학습 데이터가 100개가 있으면 이를 Sequential하게 바라보고 모델 하나를 간단하게 만들고 이 모델을 학습 데이터에 대해 돌려본다. 이 모델은 간단하기 때문에 80개의 데이터는 잘 예측하고, 20개는 잘 예측하지 못할 수도 있다. 그럼 2번째 모델은 이 20개의 잘 안되는 데이터에 대해서만 잘 동작하도록 모델을 만든다. 이렇게 여러 개의 모델을 만들어서 이 모델들을 합치는 것이 Boosting이다. 합칠 때 여러 개의 모델을 어떤 독립적인 모델로 보고 n개의 결과를 뽑는 것이 아니라 하나하나의 모델들(weak learner들)을 Sequential하게 합쳐서 하나의 strong learner를 만드는 것이다.



**2. Practical Gradient Descent Methods**

Gradient를 분류하려면 세 가지로 분류할 수 있다. 대부분의 딥러닝에서는 Mini-batch gradient descent를 사용한다.

(1) Stochastic gradient descent

- Update with the gradient computed from a single sample

(2) Mini-batch gradient descent

- Update with the gradient computed from a subset of data

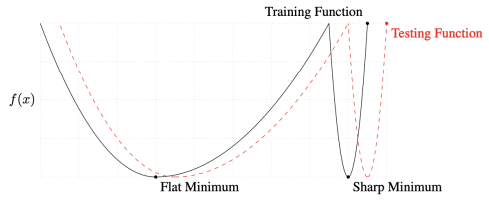
(3) Batch gradient descent

- Update with ghe gradient computed from the whole data

\* Batch-size Matters

- "It has been observed in pactice that when using a larger batch there is a degradation in the quality of the model, as measured by its ability to generalize" (degradation: 저하)

- "We ... present numerical evidence that supports the view that large batch methods tend to converge to sharp minimizers of the training and testing functions. In contrast, small-batch methods consistently converge to flat minimizers... this is due to the inherent noise in the gradient estimation"(inherent: 내재하는)



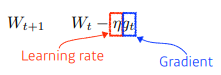
sharp minimizer보다는 flat minimizer가 더 낫다는 것을 기억하자. 위 그림에서 우리의 목적은 Testing Function의 minimum을 찾는 것이다. 왜냐하면 우리는 train data에 잘 동작하는 것이 목적이 아니라 test data에서 잘 동작하는 모델을 찾고 싶어하기 때문이다.

flat minimum의 특징은 training function에서 조금 멀어져도 testing function에서도 적당히 낮은 값이 나온다. train data에서 잘 되면 test data도 어느정도 잘 된다는 의미이다.

sharp minimum은 training function에서 local minimum이 되는 점에 도달해도 testing function에서는 약간만 멀어져 있어도 testing function이 굉장히 높은 값이 나온다. training 단계에서 얻어지는 어떤 값들이 test data에서 잘 동작하지 못할 수 있다는 의미이다. 그래서 generalization performance가 떨어진다.

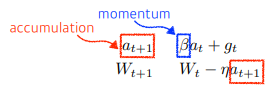
그래서 위 논문에서 말하고 싶은 것은 batch size를 줄이면 일반적으로 generalization performance가 좋아진다는 것을 실험적으로 보이고 그러면 우리가 large batch size를 활용하려면 어떻게 하면 좋을지에 대한 테크닉에 관한 것이다. 배울 점이 많은 논문이므로 읽어보는 것이 좋다고 한다.

\* Gradient Descent



일반적으로 W라는 것은 neural network의 weight를 의미한다. gradient에 Learning rate를 곱한 값을 W에 빼주면 업데이트가 된다. 이것이 기본적인 gradient descent이다. 이때 문제는 Learning rate(혹은 step size)를 잡는 것이 너무 어렵다는 것이다. 너무 크면 이전에 이야기했던 것 처럼 학습이 잘 안되고 너무 작으면 아무리 학습을 시켜도 학습이 안된다. 그래서 learning rate를 적절히 잡아주는 것이 굉장히 중요하다.

\* Momentum

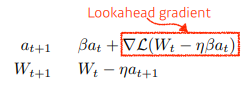


그래서 이를 좀 더 발전시키고자 혹은 '어떻게 하면 똑같이 gradient information만 활용해서 더 좋은 성능, 더 빠른 성능을 낼 수 있을까?'라는 취지로 여러 가지 Optimize 테크닉이 나오게 된다. 그중 가장 기본이 되는 것 중 하나가 Momentum이다. 직관적으로 한 번 gradient가 한 쪽 방향으로 흐르면 다음 gradient가 조금 다르게 흘러도 해당 방향으로 흐르던 정보를 조금 더 이어가자는 것이다.

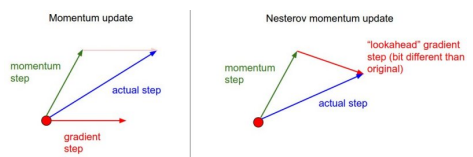
아이디어는 간단하다. B라고 불리는 Hyper parameter를 추가한다. 이것이 momentum을 잡게 된다. 즉, g라고 불리는 gradient가 현재 들어왔다면 t+1번째에는 이 gradient를 그냥 버리고 그때 나오는 g\_t+1만 사용하는 것이 아니라 a라고 불리는 항이 계속 그 값을 가지고 있는다. 그래서 momentum과 현재 gradient를 합친 것을 accumulation이라고 부른다.

이것의 장점은 한 번 흘러가기 시작한 gradient 옵션을 어느 정도 유지시켜주므로 gradient를 굉장히 왔다 갔다 해도 어느 정도 잘 학습을 하게 된다.

\* Nesterov Accelerated Gradient

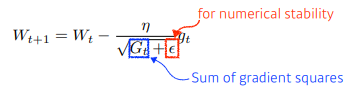


Momentum이랑 조금 비슷하지만 약간 컨셉이 다른 것이 NAG(Nesterov Accelerated Gradient)이다. NAG는 gradient를 계산할 때, Lookahead gradient를 계산하게 된다. a라는 현재 정보가 있으면 그 방향으로 한 번 가보고 이동한 곳에서 gradient를 계산한 것을 가지고 accumulation을 하는 것이다.



\* Adagrad

Adagrad adapts the learning rate, performing larger updates for infrequent and smaller updates for frequent parameters.

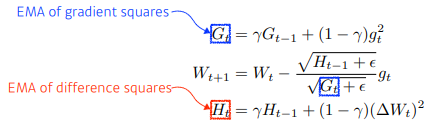


Neural Network의 parameter가 얼만큼 지금까지 변해왔는지, 그리고 안 변했는지를 보게 된다. Neural Network의 많이 변한 parameter들에 대해서는 더 적게 변화시키고 변화되지 않은 parameter는 많이 변화시키고 싶은 것이다. 그럼 지금까지 각 parameter가 얼마나 많이 변했는지를 저장해야 한다. 그 값이 바로 G이다. 많이 변화된 것은 G값이 크므로 G값이 분모에 있게 되어 좀더 적게 변화시킬 수 있다.

가장 큰 문제는 G라고 불리는 것이 계속 커지기 때문에 결국에는 G가 무한대로 가게 되면 W의 업데이트가 되지 않는다는 것이다. 즉, 뒤로 갈수록 학습이 점점 멈추게 되는 현상이 생기는 것이다. 그래서 이러한 문제를 해결하기 위해 고안된 것이 뒤에서 나오는 Adam같은 것이다.

\* Adadelta

Adadelta extends Adagrad to reduce its monotomically decreasing the learning rate by restricting accumulation window.

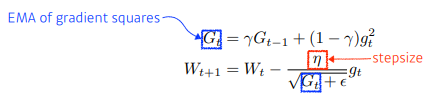


There is no learning rate in Adadelta

Adadelta는 앞에서 말한 Adagrad에서 말한 문제점을 최대한 막기 위해 만들어 졌다. 이를 이론적으로 막을 수 있는 가장 쉬운 방법은 현재 timestep t가 주어졌을 때 이것을 어느 정도 window size만큼의 parameter, 시간에 대한 gradient의 제곱의 변화를 보겠다는 의미이다. 이것도 문제가 있다. window size를 100으로 잡게 되면 이전 100개 동안 g라는 정보를 들고 있어야 한다. 이런 것을 막을 수 있는 방법이 Exponential Moving Average 방법이다. 어떤 값이 있을 때 그것을 감마와 이전값에 있는 감마를 곱하고, 1 - 감마만큼을 더해주면 어느정도 time window만큼의 값을 저장하고 있다고 볼 수 있다. 이러한 EMA를 이용해 G를 update하는 것이 Adadelta가 된다.

\* RMSprop

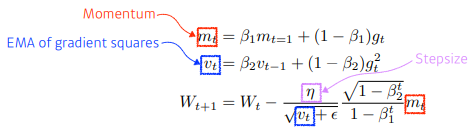
RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in his lecture.



Adadelta는 learning rate를 사용하지 않아 바꿀 수 있는 값이 없었다. 그래서 많이 사용하지 않았고 대신 RMSprop을 많이 사용했었다. 지금은 많이 사용하진 않는다. RMSprop은 논문으로 제안된 것은 아니고 Geoff Hinton이 강의 중 이렇게 하니까 잘된다고 말하자, 다른 사람들이 그 방식대로 시도해 보았고 실제로 좋은 성과를 보였다.

아이디어는 간단하다. gradient squares를 그냥 더하는 것이 아니라 Exponentail Moving Average를 더해준다. 그리고 그것을 분모에 넣고, 대신에 stepsize를 분자에 넣어준다.

\* Adam



Adam effectively combines momentum with adaptive learning rate approach.

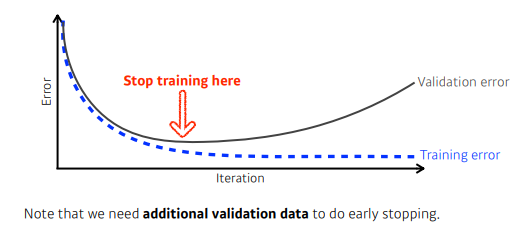
일반적으로 가장 무난하게 사용하는 것이 Adam이다. gradient squares를 Exponential Moving Average로 가져감과 동시에 Momentum을 같이 활용하는 방법이다.

Hyper parameter에는 momentum을 얼마난 유지시킬지 정하는 B1, gradient squares에 대한 EMA 정보 그리고 Stepsize, 입실론이라고 불리는 총 4개의 파라미터가 있다. 이 4개의 파라미터를 조정하는 것도 굉장히 중요하다.

**3. Regularization**

Regularization은 앞에서 이야기했던 Genelarization을 잘되게 하고 싶은 것이다. 즉, 뭔가 규제를 건다는 의미이다. 이때 규제는 학습에 반대되도록 규제를 걸게 된다. 엄밀히 말하면 학습을 방해하는 것 목표이며 학습을 방해하면서 얻는 이점은 학습 데이터에만 잘 동작하는 것이 아니라 테스트 데이터에서도 잘 동작하게 만들어 주는 것이 Regularization이다.

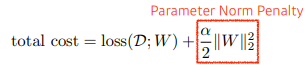
\* Early stopping



iteration이 증가함에 따라 training error는 감소하고 일정 시점 이후부터 validation error는 증가한다. early stopping은 말 그대로 validation error가 증가하기 전에 학습을 멈추는 것이다.

\* Parameter Norm Penalty

It adds smoothness to the function space

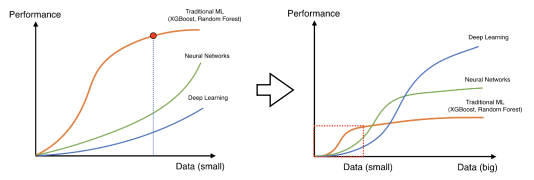


neural network의 parameter가 너무 커지지 않게 하는 것을 말한다. 쉽게 말하면 network parameter들을 모두 제곱한 다음 더하면 어떠한 숫자가 나올 것이다. 이를 같이 줄이는 것이다.

이왕이면 network를 학습할 때 network weight 숫자들이 (크기 관점에서)작으면 작을수록 좋다. 물리적인 의미는 이 neural network가 만든 function space에서 이 함수를 최대한 부드러운 함수로 보자는 것이다. 부드러운 함수일수록 generalization performance가 높을 것이다'라는 가정이 깔려있다.

\* Data Augmentation

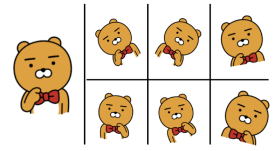
More data are always welcomed



데이터는 많을수록 좋지만 데이터는 한정적이다.

However, in most cases, training data are given in advance

In such cases, we need data augmentation

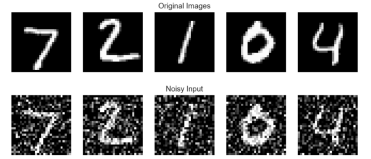


그래서 Data Augmentation이 필요하다. 어떤 식으로든 데이터를 변형해 데이터를 늘리는 것이다.

예를 들어, 강아지 고양이 분류문제를 생각해보자. 강아지 사진을 45, 90도 기울이거나 이미지를 조금 찌그리거나 해도 강아지일 가능성이 크다. 이렇게 변환을 했을 때도 이미지의 라벨이 바뀌지 않는 한도 내에서 변환을 시키는 것이 Data Augmentation중에 하나이다.

\* Noise Robustness

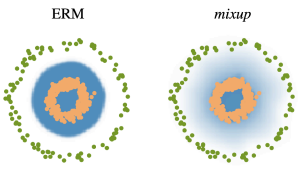
Add random noises inputs or weights



입력 데이터와 weight에 noise를 집어넣는다. weight를 학습시킬때 neural network의 weight를 매번 흔들어주면 성능이 잘 나온다는 실험적인 결과가 있다.

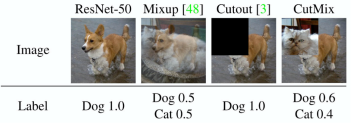
\* Label Smoothing

Mix-up constructs augmented training examples by mixing both input and output of two randomly selected training data.



Label Smoothing은 Data Augmentation과 비슷하다. 굳이 차이점을 말하자면 Label Smoothing은 데이터 두 개를 뽑아서 학습 데이터를 하나만 보는게 아니라 train 단계에서 가지고 있는 학습 데이터 두 개를 뽑아 섞어주는 것이다. 일반적으로 분류 문제를 푼다고 하면 내 이미지들이 살고있는 공간 속에서 decision boundary를 찾고싶은 것이다. 이 decision boundary를 부드럽게 만들어 줄 수 있다.

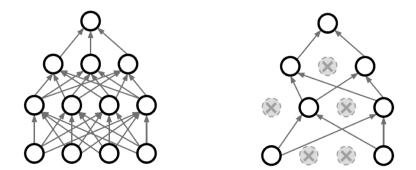
CutMix constructs augmented training examples by mixing inputs with cut and paste and outputs with soft labels of two randomly selected training data.



위처럼 Mixup은 두 개의 이미지를 고르고 두 개의 이미지와 label을 섞는다. Cutout은 이미지가 주어지면 일정 영역을 빼는 것이고 CutMix는 이미지를 섞을 때 특정 영역은 고양이, 특정 영역은 강아지로 섞어준다.

\* Dropout

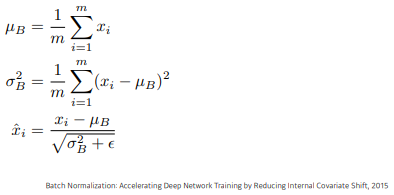
In each forward pass, randomly set some neurons to zero.



neural network의 weight를 일반적으로 0으로 바꾸는 것이다. 일반적으로 사용하면 성능이 많이 올라간다.

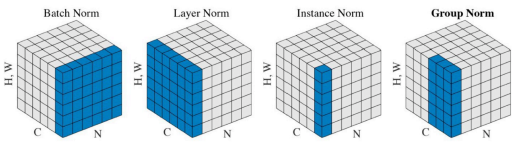
\* Batch normalization

Batch normalization compute the empirical mean and variance independently for each dimension (layers) and normalize



Batch normalization은 내가 적용하고자 하는 layer의 statistics를 정교화시키는 것이다.

There are different variances of normalizations.



간단한 분류문제를 풀 때 batch normalization을 사용하는 것이 성능이 더 좋다.