# 요약

|  |
| --- |
| * 신경망의 목적 예측의 “정확도”를 높이자. 정확도의 지표는? 손실함수(Loss function)의 값이 최소화 되는것. * 신경망 학습의 목적   손실함수의 값을 가능한 낮추는 매개변수를 찾는 것 (=최적 매개변수를 찾는 문제, 최적화) ※ 매개변수 : weight   * weight 갱신 - Gradient Descent (경사하강법) - SGD (Stochastic Gradient Descent, 확률적경사하강법) - Momentum (모멘텀) - AdaGrad (Adaptive Gradient) - RMSProp - Adam * weight 초기값 - Xavier 초기값 - He 초기값 * 배치정규화 (Batch normalization) * 활성화 함수  : 입력신호의 총합을 출력신호로 변환하는 함수 - Step - Sigmoid   \*\* Vanishing gradient 문제를 해결하는 방법 - ReLU(Rectified Linear Unit)  - MaxOut   * 올바른 학습 (overfitting 문제 해결)   + weight 감소 - L1 regularization - L2 regularization - Elastic-Net regularization ※ 실제 신경망 학습에 적용할 때, 만약 특정한 feature selection 후 학습하는 것이 아니라면 많은 경우에 L2 regularization을 사용하면 훨씬 좋은 성능을 기대할 수 있다.   + Dropout * 앙상블 |

## <1> 매개변수 갱신

우리는 이제까지,

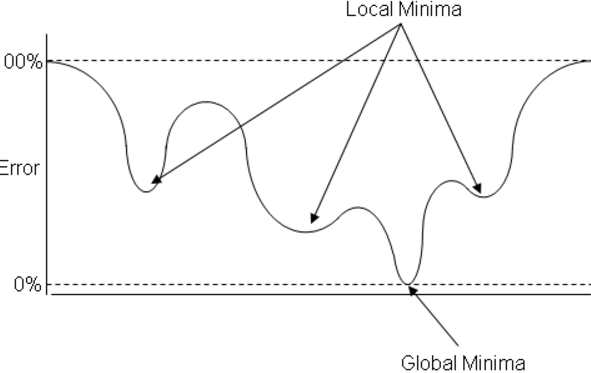
최적의 매개변수값을 찾는 방법으로 매개변수의 기울기(미분)을 이용했음.

매개변수의 기울기를 구해 기울어진 방향으로 배개변수값을 갱신하는 일을 몇번 반복해서 점점 최적의 매개변수를 찾음

\*\* 요즘은 Adam을 많이 사용하기는 하나, 모든 문제에서 항상 뛰어난 기법은 없음.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Gradient Descent**  **(경사하강법)** | **SGD  (Stochastic Gradient Descent, 확률적경사하강법)** | **Momentum  (모멘텀)** | **AdaGrad (Adaptive Gradient)** | **RMSProp** | **Adam** |
| 설명 |  | loss function을 계산할 때 전체 데이터(batch) 대신 일부 데이터의 모음(mini-batch)에 대해서만 loss function을 계산 | Gradient Descent를 통해 이동하는 과정에 일종의 ‘관성’을 주는 것.  과거에 이동했던 방식을 기억하면서 그 방향으로 일정 정도를 추가적으로 이동하는 방식 | 개별 매개변수에 적응적(adaptive)으로 학습률을 조정하면서 학습을 진행  (‘각각의’ 매개변수에 ‘맞춤형’값을 만들어줌) | gradient의 제곱값을 더해나가면서 구한 Gt 부분을 합이 아니라 지수평균으로 바꾸어서 대체. 이는 과거기울기의 반영 규모를 기하급수적으로 감소시킴 | RMSProp과 Momentum 방식을 합친 것 같은 알고리즘.  Momentum 방식과 유사하게 지금까지 계산해온 기울기의 지수평균을 저장하며, RMSProp과 유사하게 기울기의 제곱값의 지수평균을 저장 |
| 수식 |  |  |  |  |  |  |
| 장점 |  | 계산 속도가 빠르기 때문에 같은 시간에 더 많은 step을 갈 수 있으며 여러 번 반복할 경우 보통 batch의 결과와 유사한 결과로 수렴 | 1) SGD에 비해 상대적으로 빠르게 이동  2) local minima를 빠져나오는 효과가 있을 것이라고도 기대 (why? 기존에 이동했던 방향에 관성이 있어 이 local minima를 빠져나오고 더 좋은 minima로 이동할 것을 기대할 수 있게 된다.) |  |  |  |
| 단점 | 1) 속도 문제 : Loss Function을 계산할 때 전체 train set을 사용하는 것. 너무 많은 계산량이 필요  2)  local minima\*\* | 1) 이동속도가 현저하게 느림 2) local minima 여전히 발생  3)  oscilation(진동) 현상 | 과거에 이동했던 양을 변수별로 저장해야하므로 변수에 대한 메모리가 기존의 두 배로 필요 | 과거의 기울기를 제곱하여 계속 더해가는 방법. 그래서 학습을 진행할 수록 갱신강도가 약해짐. 실제로 무한히 계속 학습을 하게 되면 순간 갱신량이 0이 되어 전혀 갱신되지 않음. 이를 보완하여 고친 알고리즘이 RMSProp과 AdaDelta |  |  |

\*\* local minima란?



### Python code

|  |  |
| --- | --- |
| **Gradient Descent** | optimizer = tf.train.GradientDescentOptimizer() |
| **Momentum** | tf.train.MomentumOptimizer() |
| **AdaGrad** | tf.train.AdagradOptimizer() |
| **RMSProp** | tf.train.RMSPropOptimizer() |
| **Adam** | tf.train.AdamOptimizer() |

## <2> 가중치(weight) 초기값

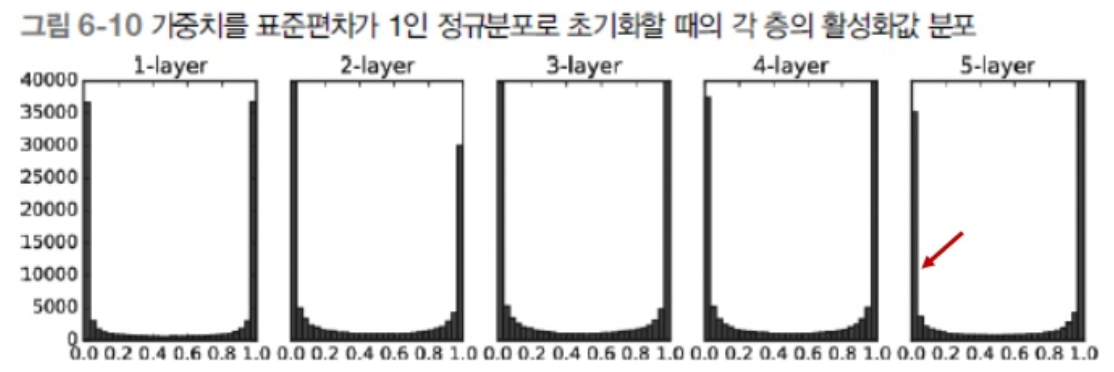
※ 여전히 연구중인 분야

* 가중치 초기값을 정하는 방법은 학습에 있어서 매우 중요함
* 목적은 output layer의 활성화값의 분포를 넓게 만들 수 있도록 초기가중치를 조정하는 것

그림 6-10과 6-11은 가중치의 초기값에 따라 은닉층 활성화값들이 어떻게 변화하는지 간단한 실험을 해본 결과

\*\* 층은 5개, 각 층의 뉴런은 100개, 활성화 함수로는 sigmoid함수

|  |
| --- |
|  |



* 각 층(layer)의 활성화값들이 0과 1에 치우쳐 분포되어 있음.
* 데이터가 0과 1에 치우쳐 분포하게 되면, 역전파시에 기울기 값이 점점 작아지다가 사라짐 (= Gradient vanishing 문제)



* 0.5부근에 집중되어 있음
* 앞의 예처럼 0과 1로 치우치진 않았으니 기울기 소실문제는 일어나지 않았지만 활성화값들이 치우쳤다는 것은 문제가 있다. 즉, 뉴런을 여러개 둔 의미가 없어진다!

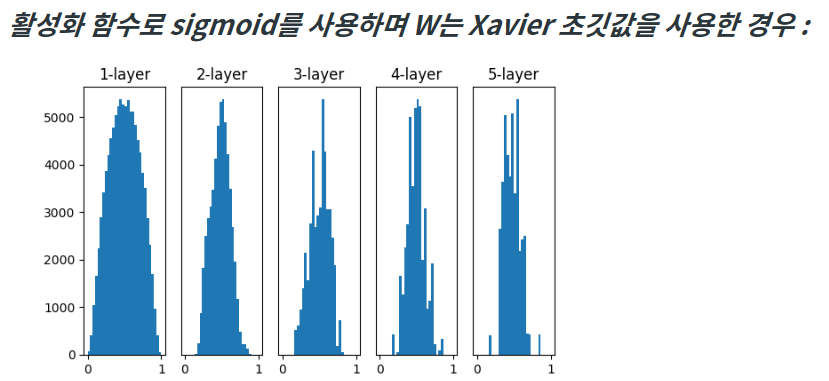
그러므로, 각 층(layer)의 활성화 값은 적당히 고루 분포되어야 함. 층과 층사이에 적당하게 다양한 데이터가 흐르게 해야 신경망 학습이 효율적으로 이뤄지기 때문. 반대로 치우친 데이터가 흐르면 기울기 소실이나 표현력 제한 문제에 빠져서 학습이 잘 이뤄지지 않는 경우가 생김.

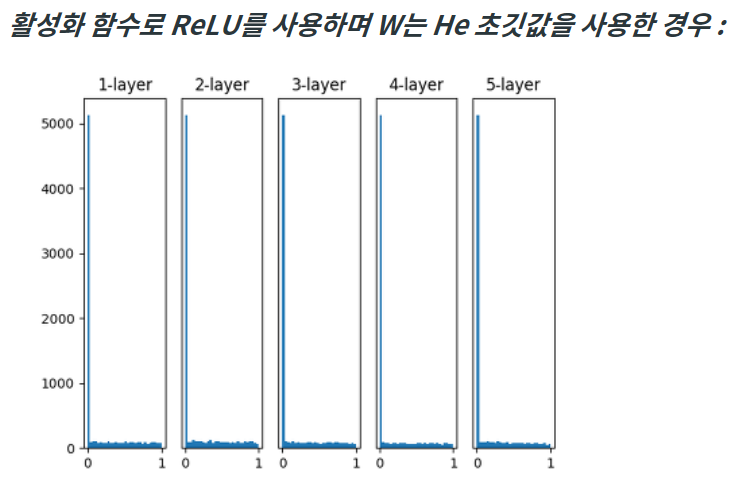
### Xavier 초기값과 He초기값

초기 weight값을 학습이 잘 되도록 딱! 맞게 주고 싶었다. 어떻게 하면 좋을까?

입력층의 노드수(fan\_in)와 출력층의 노드수(fan\_out)의 개수를 사용하여 초기값을 설정하는 방법을 사용했다.

|  |  |
| --- | --- |
| **Xavier(사비에르) 초기값** | **He초기값** |
| Xavier Glorot(사비에르 글로로트)와  Yoshua Bengio(요슈아 벤지오) | KaimingHe (카이밍히) |
|  |  |
| (weight의 초기값을 random하게 주고 싶은데,)  앞 계층의 노드가 n개라면 표준편차가 인 정규분포를 사용하면 된다. | 앞 계층의 노드가 n개라면 표준편차가 인 정규분포를 사용하면 된다. |
| 활성화함수가 선형인것을 전제로 이끈 결과이다. 하여, sigmoid함수와 tanh함수는 좌우대칭이라 중앙부근이 선형인 함수로 볼 수 있다. 그래서 sigmoid함수와 tanh함수의 경우 Xavier초기값이 적당함. | ReLU  ※ ReLU 함수는 *x*<0 일 경우 활성화가 되지 않기 때문에 입력 뉴런의 개수의 절반만큼만 해당하는 뉴런들로 조정할 경우 괜찮은 분포가 나타나는 것을 확인 할 수 있음. |





#### Python code

|  |
| --- |
| ## xavier 초기화  xavier\_init = tf.contrib.layers.xavier\_initializer()  ## He 초기화  he\_init = tf.contrib.layers.variance\_scaling\_initializer()  hidden1 = fully\_connected(X, n\_hidden1, weights\_initializer=he\_init, scope="h1") |

### (참고) Bias 초기화

바이어스(b)는 0으로 초기화 하는 것이 가장 보편적이다.

ReLU를 사용한다면 0.01와 같은 작은 값으로 초기화 하면 성능이 향상된다는 보고도 있지만 모든 경우에 그런것은 아니고 오히려 그냥 0으로 초기화 하는 것이 더 일반적인 방법이다.

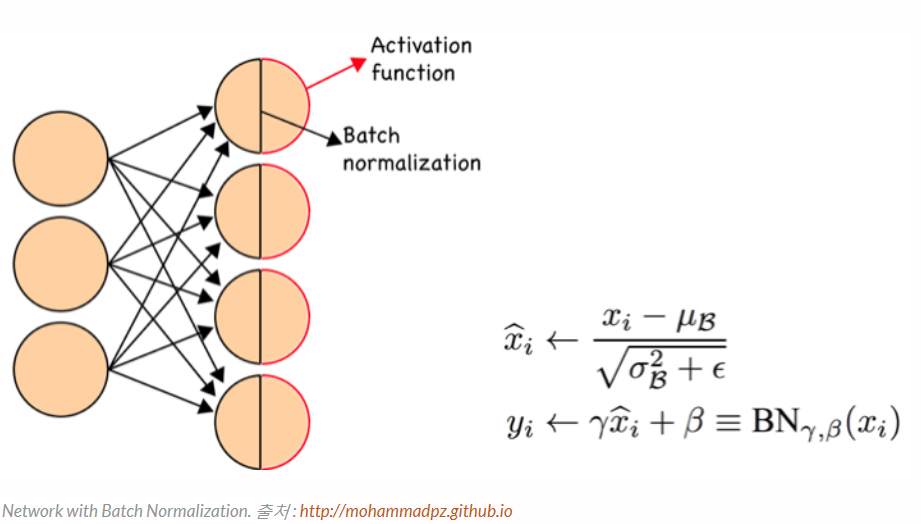
## <3> Batch Normalization(배치 정규화)

(참고) <https://arxiv.org/abs/1502.03167>

* Ioffe and Szegedy에 의해서 제안
* 신경망 학습단계에서 activation 값이 표준정규분포를 갖도록 강제하는 기법

### 학습방법

가중치의 초기값을 적절히 설정하면 각 층의 활성화 값의 분포가 적당히 퍼지면서 학습이 원할하게 수행된다. 하여, 각층이 활성화를 적당히 퍼트리도록 “강제”화 하는 방법을 “배치정규화”라고 한다.



* 실제로 이 Batch Normalization을 네트워크에 적용시킬 때는, 특정 Hidden Layer에 들어가기 전에 Batch Normalization Layer를 더해주어 input을 modify해준 뒤 새로운 값을 activation function으로 넣어주는 방식으로 사용한다.

### 장점

배치 정규화의 장점은 다음과 같다.

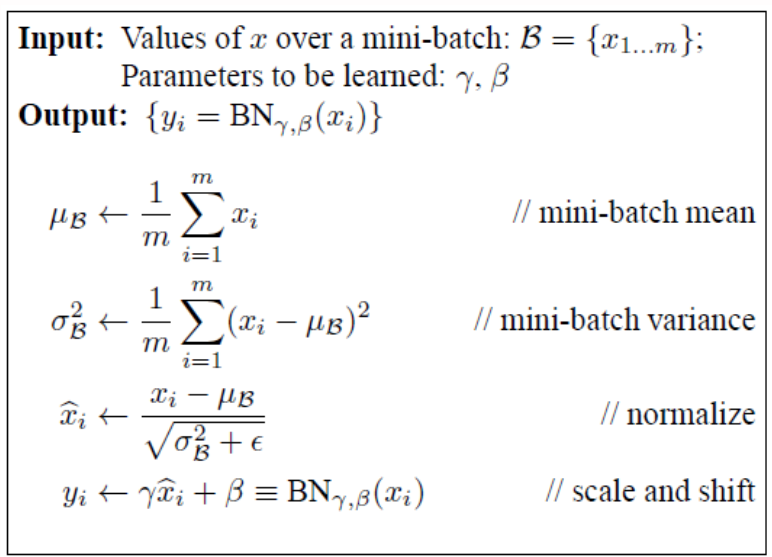
•초깃값에 크게 의존하지 않는다.

•gradient vanishing / gradient exploding을 방지한다.

•오버피팅을 억제한다. ( 그래도 드롭아웃을 병행해서 사용하기는 한다. )

### 알고리즘

뉴럴넷을 학습시킬 때 보통 mini-batch 단위로 데이터를 가져와서 학습을 시키는데, 각 feature 별로 평균과 표준편차를 구해준 다음 normalize 해주고, scale factor와 shift factor를 이용하여 새로운 값을 만들어준다. 알고리즘의 개요는 다음과 같다.



미니배치 http://cfile8.uf.tistory.com/image/25638D3658D6D05035DF2F에 대해 이를 평균이 0, 분산이 1인 표준정규분포를 따르는 http://cfile10.uf.tistory.com/image/24208B3958D6D0BC0908BF으로 정규화한다.

이 때 B가 미니배치이므로, http://cfile22.uf.tistory.com/image/22648A3558D6D1323B21E6는 단일 원소 값이 아니라 단일 입력 데이터(행렬)임에 유의한다.

즉, 배치 정규화는 단일 입력 데이터 단위가 아니라 **미니배치 단위로 정규화**한다.

정규화 하고 나서 배치 정규화 계층마다 이 정규화된 데이터에 고유한 확대(scale,http://cfile1.uf.tistory.com/image/2262504658D6D507346F73), 이동(shift, http://cfile22.uf.tistory.com/image/2201E13B58D6D5A902E34F) 변환을 수행한다. 초깃값은 http://cfile1.uf.tistory.com/image/2262504658D6D507346F73 = 1, http://cfile22.uf.tistory.com/image/2201E13B58D6D5A902E34F= 0이며 학습하면서 적절한 값으로 조정해나간다.

~@.@~

단, 지금까지의 설명은 ‘일반적인 Network 일 때’ 에 한해서 통용된다. 만약 Batch Normalization을 CNN에 적용시키고 싶을 경우 지금까지 설명한 방법과는 다소 다른 방법을 이용해야만 한다. 먼저, convolution layer에서 보통 activation function에 값을 넣기 전 Wx+b 형태로 weight를 적용시키는데, Batch Normalization을 사용하고 싶을 경우 normalize 할 때 beta 값이 b의 역할을 대체할 수 있기 때문에 b를 없애준다. 또한, CNN의 경우 convolution의 성질을 유지시키고 싶기 때문에, 각 channel을 기준으로 각각의 Batch Normalization 변수들을 만든다. 예를 들어 m의 mini-batch-size, n의 channel size 를 가진 Convolution Layer에서 Batch Normalization을 적용시킨다고 해보자. convolution을 적용한 후의 feature map의 사이즈가 p x q 일 경우, 각 채널에 대해 m x p x q 개의 각각의 스칼라 값에 대해 mean과 variance를 구하는 것이다. 최종적으로 gamma와 beta는 각 채널에 대해 한개씩 해서 총 n개의 독립적인 Batch Normalization 변수들이 생기게 된다.

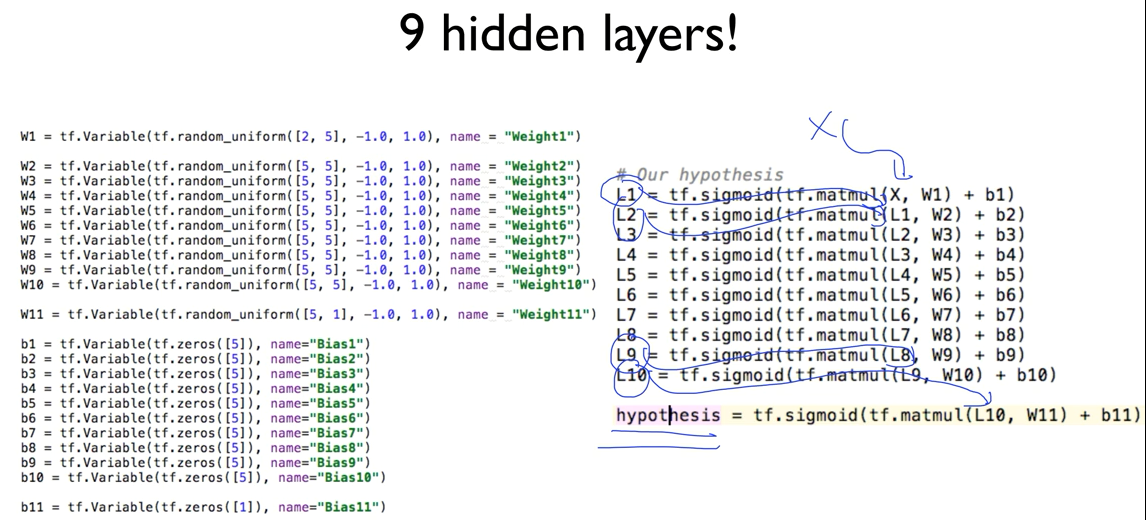
### Python code

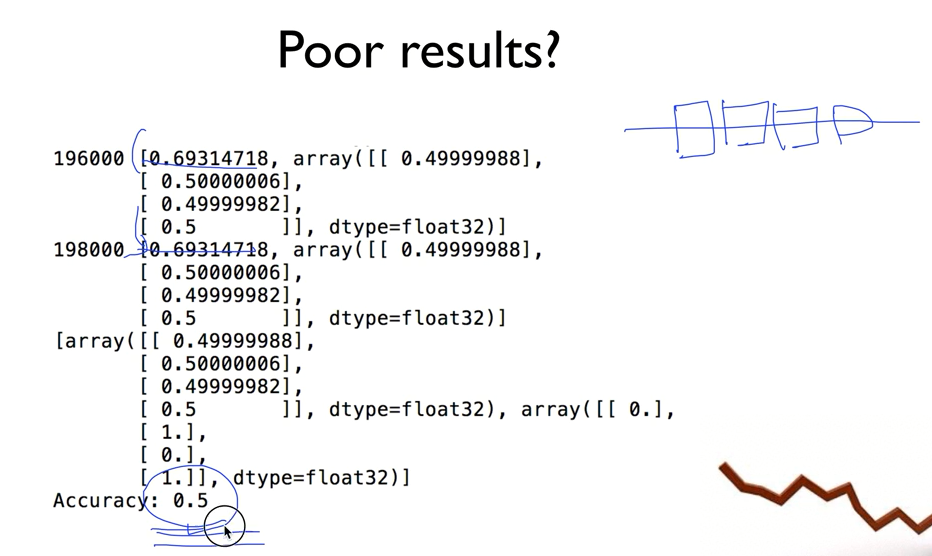
|  |
| --- |
| ## batch\_norm()  import tensorflow as tf from tensorflow.contrib.layers import batch\_norm  hidden1 = fully\_connected(X, n\_hidden[0], scope="hidden1", normalizer\_fn=batch\_norm, normalizer\_params=bn\_params)  hidden2 = fully\_connected(hidden1, n\_hidden[1], scope="hidden2", normalizer\_fn=batch\_norm, normalizer\_params=bn\_params)  logits = fully\_connected(hidden2, n\_outputs, activation\_fn=None,scope="outputs", normalizer\_fn=batch\_norm, normalizer\_params=bn\_params) |

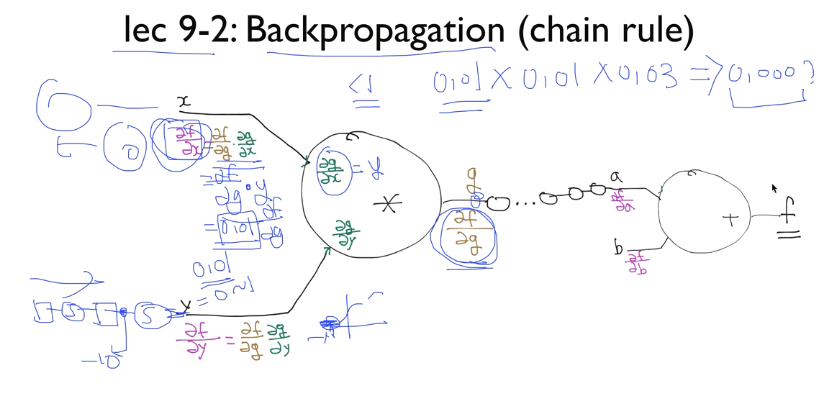
## <4> Activation function (활성화 함수)

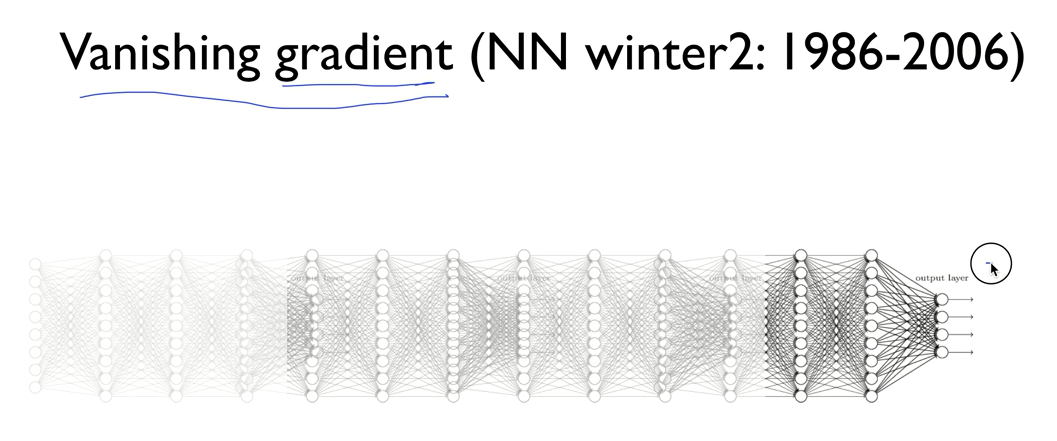
### ReLU의 탄생배경

(참고) Sigmoid 보다 ReLU가 더 좋아 (lec 10-1)









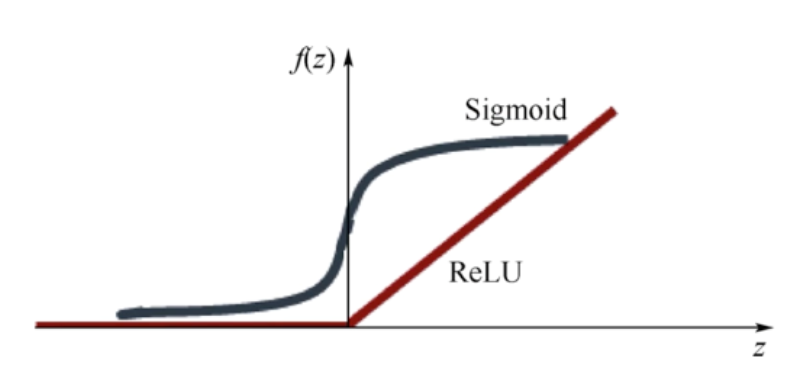
hidden layer를 무려 9개나 두었음에도 불구하고 정확도는 겨우 50%! (결과는 좋지 않다.)

왜 그럴까?

backpropagation시에 layer가 많을 때는 미분 결과를 최초 layer까지 전달하는 것이 거의 불가능하다. 보통 2~3개의 layer에 대해서는 잘 동작하지만, 그 이상이 되면 문제가 발생한다.

sigmoid는 전달된 값을 0과 1 사이의 값으로 변형이 된다. 일정 비율로 줄어들기 때문에 왜곡은 아니지만, 값이 현저하게 작아지는 현상이 벌어진다. 3개의 layer를 거치면서 계속해서 1/10로 줄어들었다면, 현재 사용 중인 값은 처음 값의 1/1000이 된다. 이렇게 작아진 값을 갖고 원래의 영향력을 그대로 복원한다는 것은 불가능하다. 이 문제를 vanishing gradient라고 부르고 1986년부터 2006년까지 아무도 해결하지 못했다.

hinton 교수가 2006년에 방법을 찾아냈다. 그 이유는 non-linearity에 대해 잘못된 방법을 사용했다는 것이었다. non-linearity는 sigmoid 함수를 말한다. vanishing gradient 문제의 발생 원인은 sigmoid 함수에 있었다. sigmoid 함수가 값을 변형하면서 이런 문제가 생긴 것이었다. 이걸 알아내는데, 10년이 넘게 걸렸다.



hinton 교수님은 sigmoid 함수 대신 ReLU 함수를 제안했다. ReLU 함수는 그림에 있는 것처럼 0보다 작을 때는 0을 사용하고, 0보다 큰 값에 대해서는 해당 값을 그대로 사용하는 방법이다.

### 다양한 활성화 함수

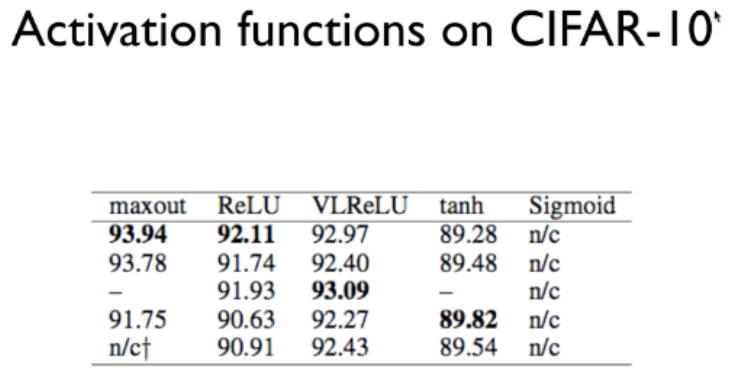
**tanh** - sigmoid 함수를 재활용하기 위한 함수. sigmoid의 범위를 -1에서 1로 넓혔다.  
**ReLU** - max(0, x)처럼 음수에 대해서만 0으로 처리하는 함수

\*\* ReLU의 확장판 (음수에 대해서 어떻게 처리할까?)  
**> Leaky ReLU** - ReLU 함수의 변형으로 음수에 대해 1/10로 값을 줄여서 사용하는 함수  
**> ELU** - ReLU를 함수식을 적용 (a(exp(x)-1) )하여 사용하는 함수

**maxout** - 두 개의 W와 b 중에서 큰 값이 나온 것을 사용하는 함수



#### 성능비교



### Python code

|  |  |
| --- | --- |
| sigmoid | tf.sigmoid() |
| ReLU | tf.nn.relu() |
| Leaky ReLU | ## TensorFlow does not have a predefined function for Leaky ReLUs  def leaky\_relu(z, name=None):  return tf.maximum(0.01 \* z, z, name=name)  ex)  hidden1 = fully\_connected(X, n\_hidden1, activation\_fn=leaky\_relu) |
| ELU | tf.nn.elu() |

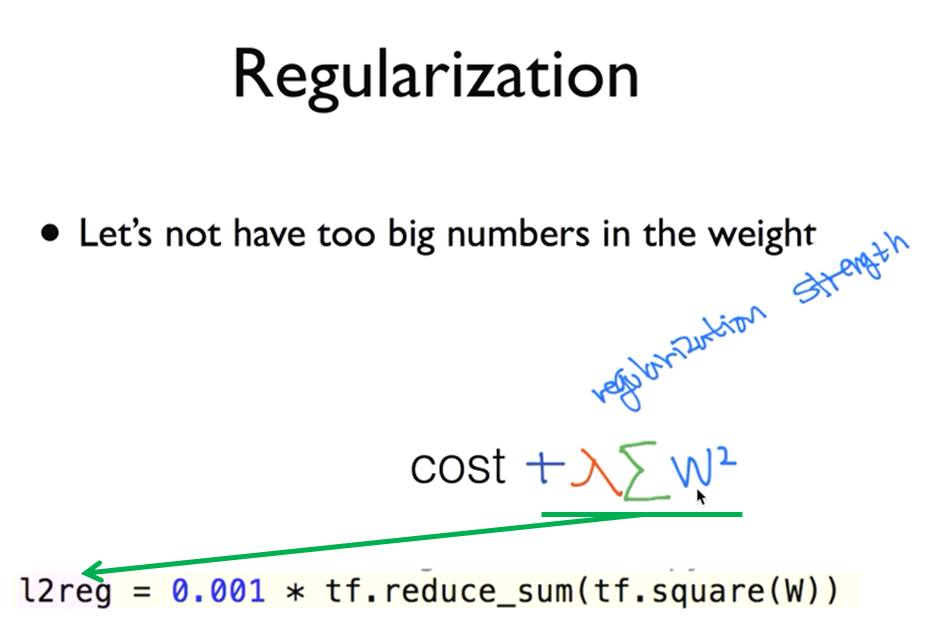
## <5> 과적합(overfitting)문제 해결

오버피팅은 주로 다음의 두 경우에 일어난다.

- 매개변수가 많고 표현력이 높은 모델

- 훈련데이터가 적음

### 가중치(weight) 감소



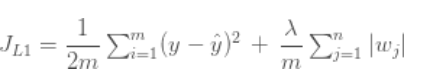
학습하는 과정에서 큰 가중치에 대해서는 그에 상응하는 큰 패널티를 부과하여 오버피팅을 억제하는 방법이다. 원래 오버피팅은 가중치의 값이 커서 발생하는 경우가 많기 때문

신경망 학습의 목적은 손실함수의 값을 줄이는 것.

예를 들어 “손실함수 + 가중치의 00법칙”을 하면, 가중치가 커지는 것을 억제할 수 있음.

#### L1 regularization

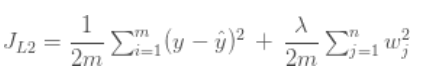
각 가중치 파라메타의 절대 값을 더함. 추가되는 항을 L1 페널티(penalty)라고 부르기도 하고 이런 방식의 선형회귀 분석을 라쏘(LASSO) 방식이라고 부르기도 합니다.



가중치 벡터*w* 가 있을때, 목적 함수에 *λ*∣*w*∣ 를 더한다.

#### L2 regularization

가중치의 제곱 법칙을 손실함수에 더함. 릿지(Ridge) 회귀 분석이라고도 합니다.



모든 파라미터 제곱 만큼의 크기를 목적 함수에 제약을 거는 방식으로 구현된다. 다시말해, 가중치 벡터 *w* 가 있을때, 목적 함수에 를 더한다 (여가서 *lambda* 는 regulrization의 강도를 의미). 부분이 항상 존재하는데 이는 앞서 본 regularization 값을 *w* 로 미분했을 때 2*λw* 가 아닌 *λw* 의 값을 갖도록 하기 위함이다.

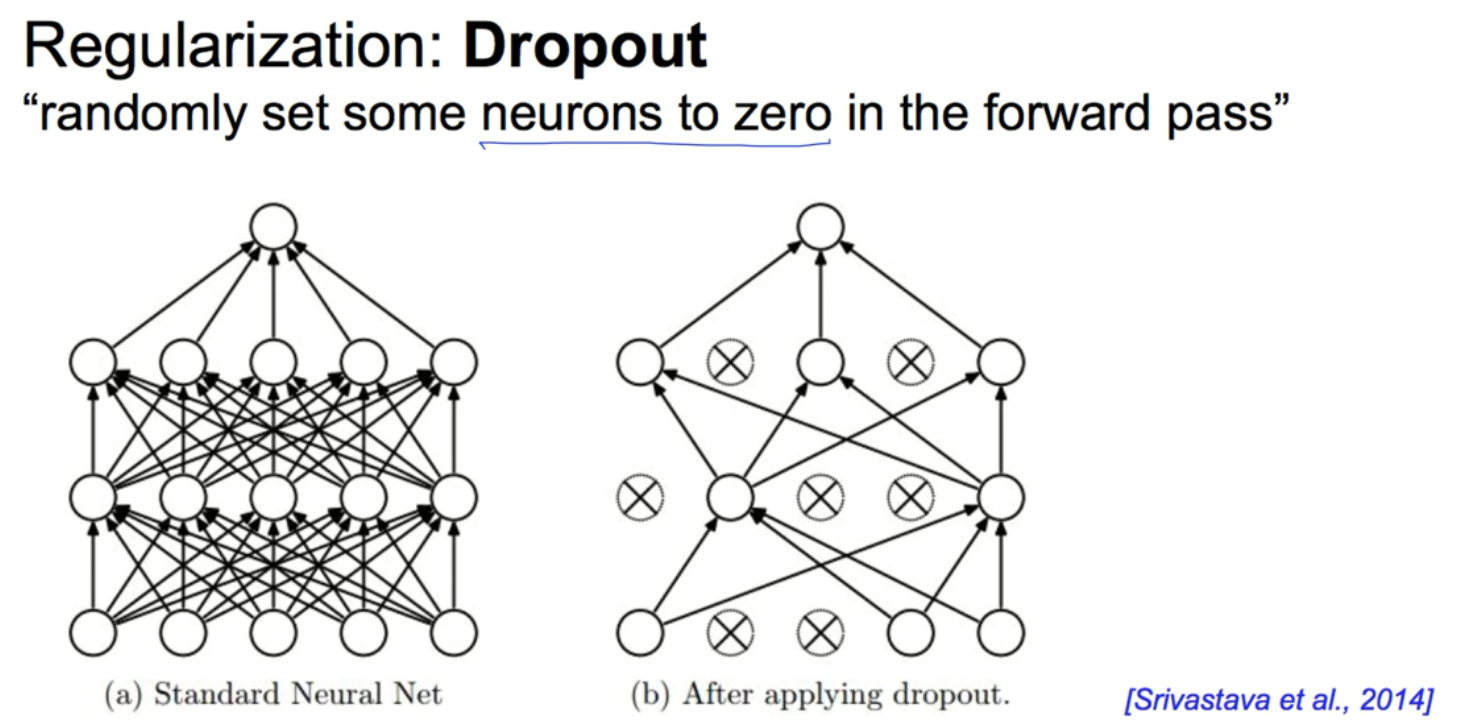
gradient descent 업데이트 과정에서 L2 regularization을 적용하는 것은 모든 가중치 값이 선형적으로 감소하게 된다

#### Elastic-Net regularization

L1 regularization과 L2 regularization을 동시에 사용

### Dropout

가중치 감소(L1, L2)방법을 사용해서 overfitting을 억제할수는 있지만, 신경망 모델이 복잡해지면 가중치감소 방법만으로는 대응하기 어려워진다. 이럴때 흔히 Dropout이라는 기법을 이용한다.



Dropout이란? 뉴런을 임의(random)로 삭제하면서 학습하는 방법이다.

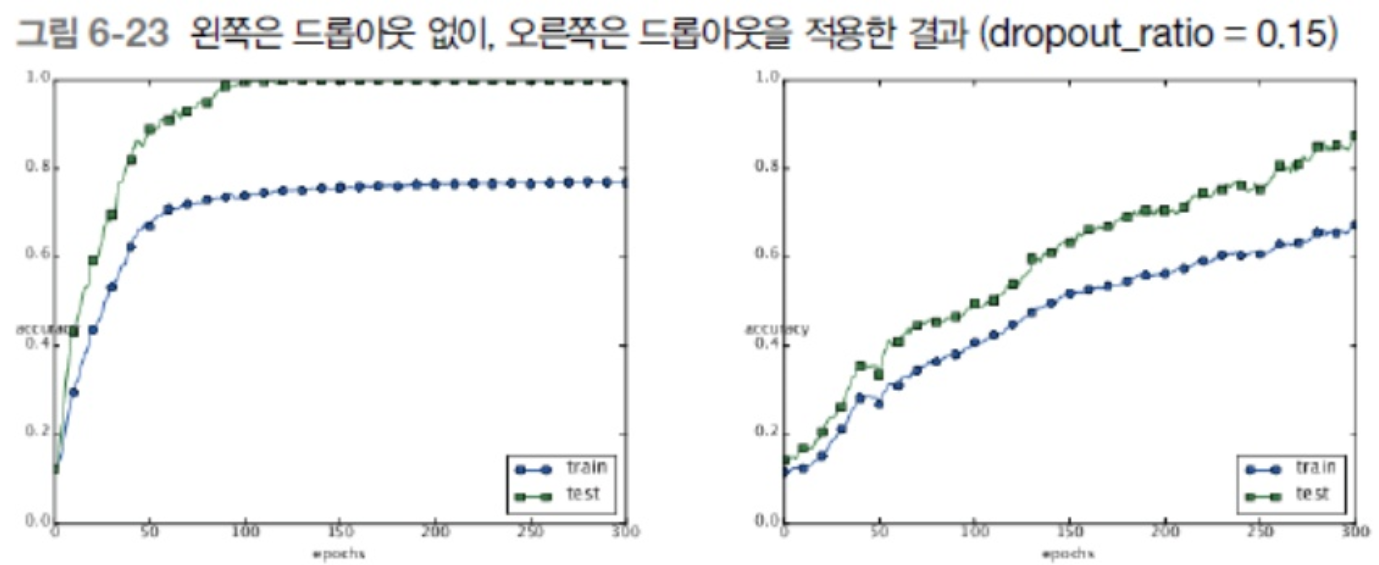
학습중에 중간층의 유닛값을 일정비율로 0으로 바꿔서 결합을 누락시킴으로써 신경망을 제약하고 overfitting을 억제하게 된다.

- 출력을 0으로 할 유닛을 무작위로 선택

- 원래대로라면 모든 유닛의 출력으로 오차를 구하지만, 그 중 일부가 누락되기 때문에 일종의 규제화와 같은 작용을 한다. 그리고 출력을 0으로 한 유닛에 대한 오차역전파 계산에는 원래의 값을 사용한다.

- 일반적으로 드롭아웃비율은 50%로 한다.

- 학습시킬때만 dropout을 적용. (실전일때는 모든 노드가 투입되어야 함)

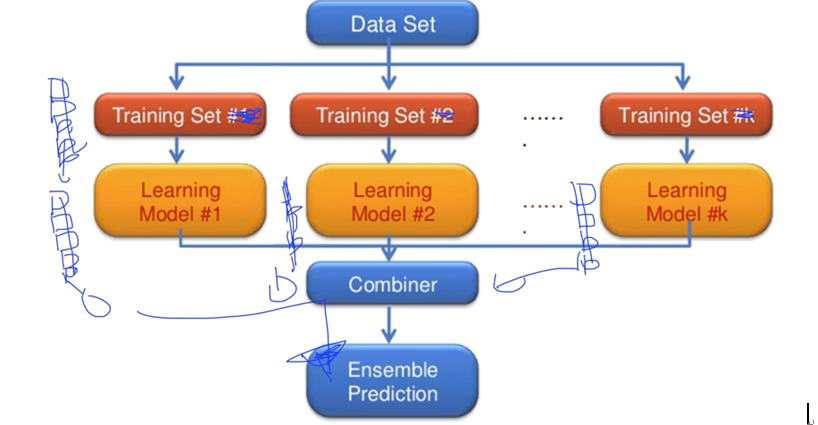


#### Python code

|  |
| --- |
| ## dropout()  from tensorflow.contrib.layers import dropout  [...]  **## train**  is\_training = tf.placeholder(tf.bool, shape=(), name='is\_training')  keep\_prob = 0.5  X\_drop = dropout(X, keep\_prob, is\_training=is\_training)  hidden1 = fully\_connected(X\_drop, n\_hidden1, scope="hidden1")  hidden1\_drop = dropout(hidden1, keep\_prob, is\_training=is\_training)  hidden2 = fully\_connected(hidden1\_drop, n\_hidden2, scope="hidden2")  hidden2\_drop = dropout(hidden2, keep\_prob, is\_training=is\_training)  logits = fully\_connected(hidden2\_drop, n\_outputs, activation\_fn=None, scope="outputs") |

## <5> Ensemble(앙상블)





기계학습에서 앙상블 학습을 애용한다. 앙상블 학습은 개별적으로 학습시킨 여러 모델의 출력을 평균내어 추론하는 방식이다(voting을 사용하기도 한다) 신경망의 맥락에서 얘기하면 가령 같은 구조의 네트워크를 5개 준비하여 따로따로 학습시키고, test시에 그 5개의 출력을 평균내어 답하는 것이다. 앙상블 학습을 수행하면 신경망의 정확도가 몇 %정도 개선된다는 것이 실험적으로 알려져 있다.

앙상블 학습은 드롭아웃과 밀접하다. 드롭아웃이 학습때 뉴런을 무작위로 삭제하는 행위를 매번 다른 모델을 학습시키는 것으로 해석할 수 있기 때문이다. 그리고 추론때는 뉴런의 출력에 삭제한 비율을 곱함으로써 앙상블 학습에서 여러 모델의 평균을 내는것과 같은 효과를 얻는 것이다. 즉, 드랍아웃은 앙상블 학습과 같은 효과를 (대략) 하나의 네트워크로 구현했다고 생각할 수 있다.