

Training Neural Net

(How to set the Neural networks?)

정재영

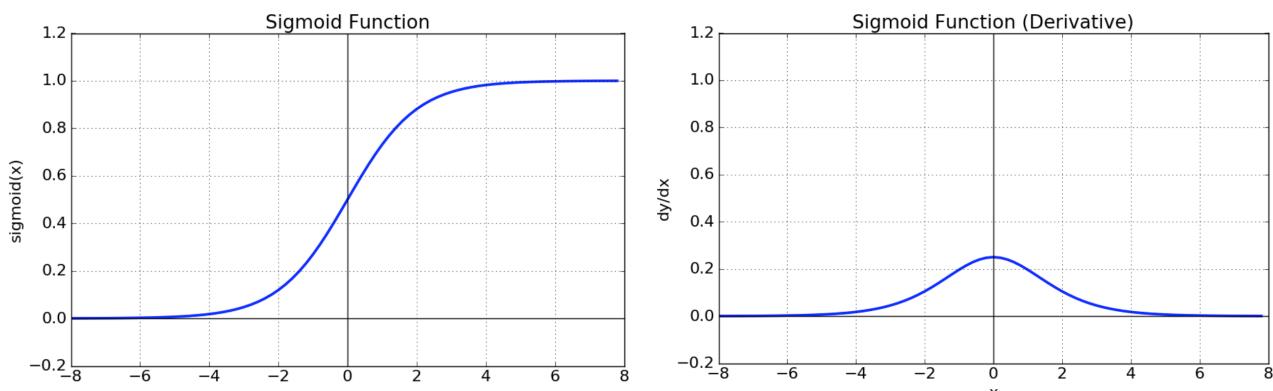
Content

1. 활성화 함수 Activations
2. 전처리 과정 Preprocessing
3. 가중치 초기화 Initialization of weights
4. MiniBatch Normalization
5. 학습 과정 돌보기 Babysitting the learning process
6. 모수 최적화 Parameter Optimization
7. Fancier Optimizations
8. 평가 Evaluation
9. 정규화로서의 드롭아웃 Dropout as the regularization

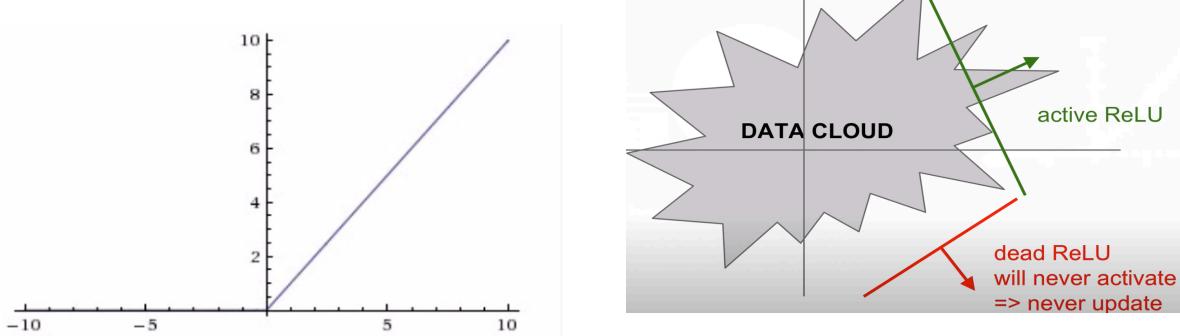
1. Activations

본격적인 논의에 앞서, 그 배경이 되는 활성화 함수 쟁점에 대해 간략히 언급하는 것이 적절할 것 같다. ReLU 가 활성화 함수로서 인기를 끌기 이전에 주로 사용되었던 sigmoid 함수는 모델을 학습시킬 때 다음의 3 가지 주된 단점을 갖고 있다고 알려져 있다:

- 1) Not 0-centered, 2) Vanishing Gradient (stop to update grads), 3) Expensive cost in computation (by exp f)



활성화 함수를 사용하는 데 있어서 이 단점들을 해결하기 위해, 2012년부터 ReLU 함수를 본격적으로 사용하기 시작하였다. ReLU 함수는 정의역이 0일 때 미분 불가능하다는 단점이 있지만, 다른 모든 영역에서는 위의 2) 같은 곤경이 나타나지 않았고, 1)에 대해서는 여전히 극복하지는 못하지만 연산적으로 매우 효율적이게 해주고, 더욱이 3)을 해결해주어 (exp를 사용하지 않기 때문에) 모델을 학습시킬 때 sigmoid 함수 보다 훨씬 더 좋은 성능을 얻게 해준다.

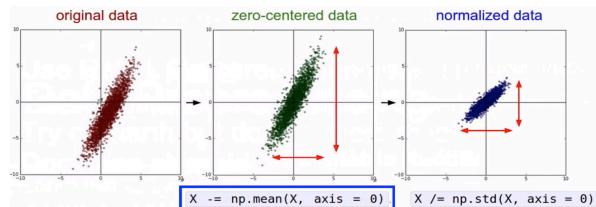


Q. ReLU 역시도 음수인 경우에 grads 가 모두 0 아닌가?

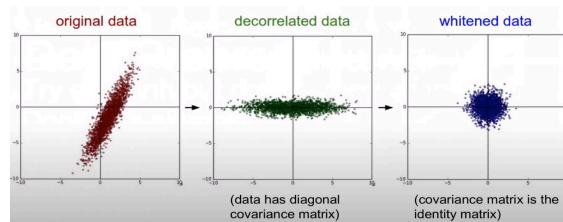
맞다. 그래서 음수인 경우에는 active 하지 않고, 그래서 grads를 업데이트 시키지 못한다. 하지만 양수인 경우에는 항상 grads를 업데이트 시킬 수 있다. 이처럼 ReLU가 deactivate 한 경우를 소위 dead ReLU 라고 부르는데, 대체로 이러한 현상이 나타나는 원인은 잘못된 초기화, 너무 큰 학습률 lr 설정¹ 때문이다. 물론 이것은 분명 문제적이고, 따라서 ReLU 역시 아직 이상적인 함수가 아니다. 우리는 더 좋은 함수를 찾던가, 아니면 다른 방법으로 이 문제를 극복할 필요가 있다. (이후에 이 학습률 문제를 극복하기 위해 Leaky-ReLU, PReLU 등이 등장했다.)

2. 전처리 과정 (참고)

- 일반적인 전처리 (1): 0 centered 해주기 위해, 각각의 입력값 X에서 평균을 뺀 뒤 정규화를 해준다. (img분석 시 자주 사용)



- 일반적인 전처리 (2): 위와 동일한 목표로 PCA(주성분분석; 차원 줄이기) 수행 뒤에, 백색화 whitening 해준다. (이미지 분석 시 일반적으로 사용 x)



¹ lr이 너무 큰 경우에는 뉴런의 값이 한번 dead ReLU zone에 빠지게 되면 다시 돌아오지 못하게 되어 vanishing grads 현상이 나타난다.

3. 가중치 초기화

우리는 방금 이미지 처리 시 전처리 과정이 0-centered를 수행한다는 것을 알아보았다. 이번에는 가중치 초기화에 대해 알아보자. 위에서 말했듯이 이것이 적절히 되지 않는다면, dead ReLU 현상이 일어날 수 있다. 따라서 초기화를 잘 하는 것은 매우 중요하다. cs231n을 참고할 때, 가중치 초기화의 종류는 4 가지가 있다.

3.1 여러가지 가중치 초기화 방법

1) $\forall \mathbf{W}\mathbf{s} = 0$,

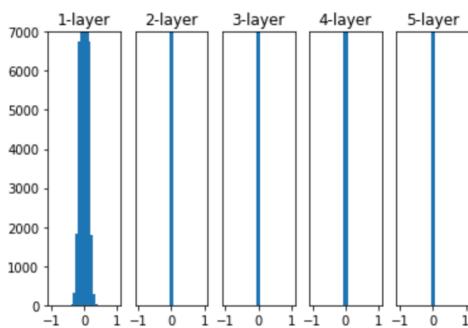
문제: 이 설정 때문에 모두 동일한 일만을 수행할 것. 즉, 이 경우에는 모두 동일한 출력값을 산출해낼 것이다. 그러면 역전파 시 모든 뉴런이 동일한 그래디언트 값을 갖게 된다. 학습이 잘 되려면, 각 뉴런의이 가중치에 따라 비대칭적이어야 (어떤 뉴런은 가중치가 크고, 다른 뉴런은 작은 경우) 한다. (\Rightarrow 가중치를 부여할 의미가 없어진다)

2) Randomly setting $\forall \mathbf{W}\mathbf{s}$ (better)

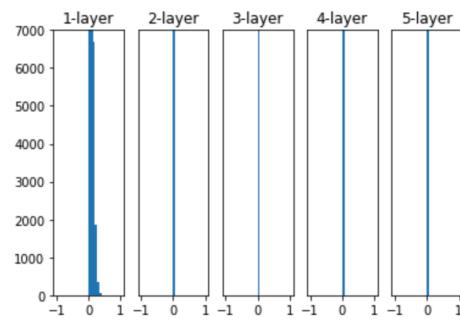
한계: 소규모 small 연결망에서 이것은 좋은 선택일 수 있지만, 더 깊은 deeper 신경망에서는 모든 활성화가 0이 되기 때문에 학습이 문제가 나타날 수 있다:

① 작은 난수: 학습률이 0.01일 때, 평균은 0으로 수렴하지만 std는 급속하게 0으로 수렴해서 모든 activation이 0이 된다.

그리고 이러한 설정을 고려할 때, 역전파 시의 그래디언트는 어떤 형태인가? 위 설정에 따를 때, X 는 0에 가깝기 때문에, $dW_1 = X * dW_2$ 이니까 dW_1 또한 0에 가깝게 되고, 그러면 gradient는 0에 수렴하게 된다. 즉 Vanishing gradient 현상이 나타나게 된다.



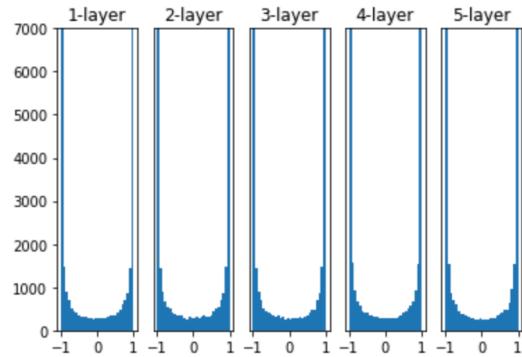
[small random numbers — tanh]²



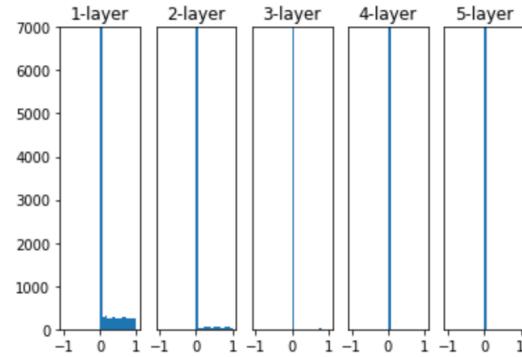
[small random numbers — ReLU]

²https://github.com/musicjae/cs231n/blob/master/assignment2/weight_initializations.ipynb 에서, 초기화와 활성화 함수를 바꿔 가며 그 래프를 비교할 수 있다.

② 큰 난수: 마찬가지로, 학습률이 1인 경우에도, 뉴런이 -1 또는 1로 saturated 되어 gradients는 0으로 수렴하기 때문에, 손실 loss가 업데이트 되지 않는다.



[large random numbers —tanh]

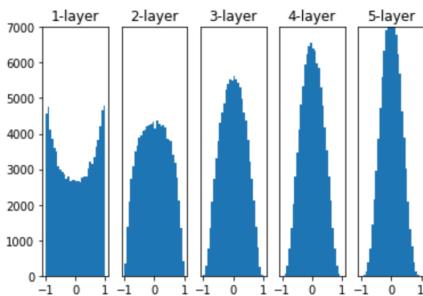


[large random numbers —ReLU]

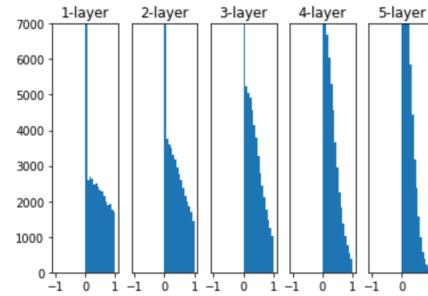
3) Xavier 초기화 (2010)

$$w = \text{np.random.rndn(fan_in, fan_out^3)} / \text{sqrt(fan_in)}$$

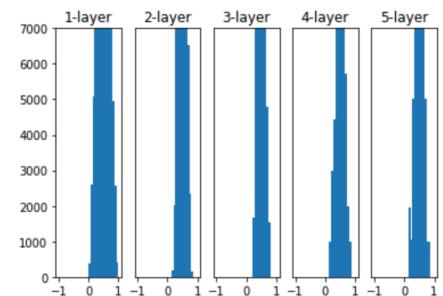
Tanh 함수에 잘 적용된다. 하지만 ReLU 함수에 사용 시 std가 0으로 수렴하는 곤경을 지닌다.



[Xavier—tanh]



[Xavier — relu]



[Xavier — sigmoid]

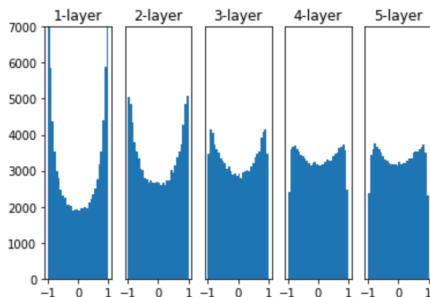
위 실습에서 나타나듯이, tanh 경우에는 학습이 잘 일어나는 데 반하여, relu, sigmoid 경우에는 점점 std가 0에 수렴하는 양상을 보인다.

³ fan_in: Input Layer의 뉴런 개수, fan_out: Output Layer의 뉴런 개수

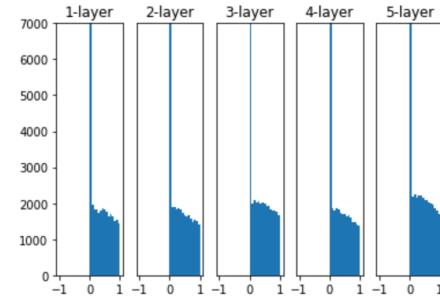
4) Kaming He 초기화 (2015)

$$W = \text{np.random.rndn}(n_input, n_output) / \text{sqrt}(n_input / 2)$$

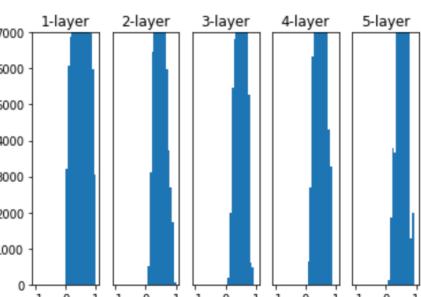
위에서 보았듯이, Xavier 초기화에서는 ReLU 활성화 적용 시 레이어가 깊어질수록 output이 0에 가까워지는 현상이 나타난다. He 초기화는 아래 그림에서처럼 이런 문제를 개선해준다. 하지만 He 활성화 때 조차도 sigmoid 활성화 적용 시 나타나는 0으로 수렴하는 현상은 Xavier에 비해 유의미하게 개선되지 않는다.



[he-tanh]

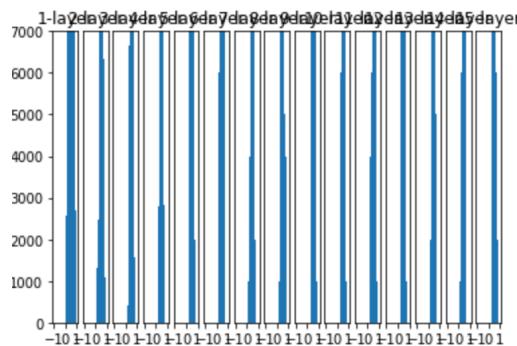


[he-relu]

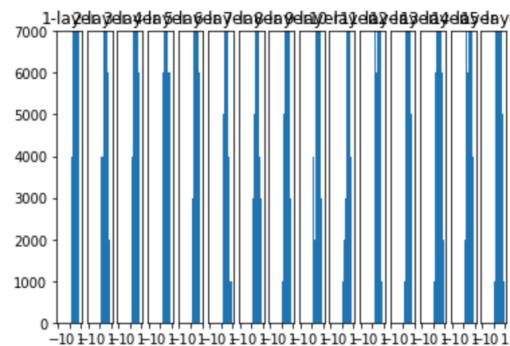


[he-sigmoid]

아래와 같이 은닉층이 15 개인 경우에도 he 초기화는 Xavier 초기화에 비해 개선되지 않는 것 같다. 따라서 이번 실습에 따르면 sigmoid 경우에 Xavier 초기화와 He 초기화 간의 뚜렷한 차이는 없어 보인다.



[Xavier--sigmoid]



[he -- sigmoid]

3.2 이번 실습을 통해 도출된 잠정적 결론

어떤 초기화가 가장 좋은가? 이 단순한 질문에 대한 항상적인 답은 없다. 하지만 활성화 함수는 일반적으로 ReLU를 사용하고, 가중치 초기화는 매우 다양하게 쓰이지만, 가장 좋은 선택이 될 수 있는 것은 tanh 함수 사용 시에는 Xavier 초기화, ReLU 함수 사용 시에는 he 초기화일 것이다.

4. Batch Normalization ⁴

:Vanishing grad 현상이 나타나지 않게 해주면서 학습 속도를 가속화 시켜보자는 생각에서 고안된 방법.

“One way to make deep networks easier to train is to use more sophisticated optimization procedures such as SGD+momentum, RMSProp, or Adam. Another strategy is to change the architecture of the network to make it easier to train.”⁵

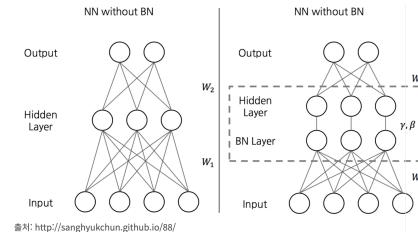
4.1 구체적인 고안 동기:

모델의 학습 과정에서 불안정성이 나타나는 원인은 내부에서 일어나는 covariate shift⁶ 때문인 것 같다. 즉, 각 layer를 거치면서 input의 분포가 달라지는 현상이 나타나기 때문에 학습 과정에서 불안정성이 나타난다. 이러한 탓에 gradient가 0으로 수렴하는 현상, 즉 vanishing gradient가 나타난다.⁷ 이것을 극복하기 위해, 각 layer를 거칠 때마다 Normalization을 해주는 것이 바로 이 Batch Normalization이다.

“실제에서, *saturation* 문제와 *vainishing gradient* 문제는 *ReLU*와 주의깊은 초기화를 사용함으로써 처리된다. (우리는 이것을 앞에서 보았다) 하지만 만약 신경망이 학습할 때 *nonlinearity inputs*의 분포가 더 안정적인 채로 있게 해준다면, 최적화는 덜 *saturated* 될 것이고, 학습은 더 빨라질 것이다”⁸

4.2 Batch Normalization의 이점:

- 학습 효율 ↑
- 가중치 초기값에 의존 ↓
- 과적합 발생 가능성 ↓
- Vanishing Gradients 발생 가능성 ↓
- 높은 학습률을 허용해도 더 빠른 학습이 가능



⁴ “Batch normalization is now considered a standard part of the neural network toolkit” from NIPS conference 2015

⁵ cs231n assignment2 batch nomalizataion

⁶ “Covariate Shift: 학습 기간 동안, <신경망 parameter>가 변화함에 따라 <신경망 활성화의 분포>가 변화하는 것”, Ibid p.2

⁷ “심층 신경망 학습은 이전 layer의 모수가 변할 때, 학습 시간 동안 각 layer의 분포가 변한다는 사실에 의해 곤경과 마주하게 된다complicated. 이 것은 lower 학습률, 신중한 W 최적화를 요구함으로써 학습을 느리게 하고, saturating nonlinearities로 모델 학습을 어렵게 한다. 우리는 이 현상을 internal covariate shift 라고 지칭한다.” , Sergey Ioffe 외, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (2015)

⁸ ibid p.2

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;
 Parameters to be learned: γ, β

Output: $\{y_i = BN_{\gamma, \beta}(x_i)\}$

$$\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

```
mu_b = np.mean(x, axis = 0) # 평균
var = 1/float(N)*np.sum((x-mu_b)**2, axis = 0) # variance 구하기
x_hat = (x-mu_b)/np.sqrt(var+eps) # 정규화, eps: 수치적 안정성의 상수
y = gamma*x_hat + beta # scale & shift
out = y
```

4.3 실용적 접근

입력값 x 에서 임의의 작은-batch를 골라 만들어진 $N \times D$ 인 mini-batch 입력값 x' 이 있다고 해보자. 이 batch에 대해 (1) mean, std 를 계산해주고, (2) 정규화해준다:

FC-Net =====> BN =====> Activation (보통 배치 정규화는 이와 같은 위치에 나열된다)

5. 학습 과정 돌보기 Babysitting the Learning Process (학습률 찾아가기)

전처리: zero centered



신경망의 아키텍처 결정: 은닉층은 몇 개인가? 몇 개의 node를 둘 것인가?



loss 체크: reg를 바꿔가면서 loss 변화 확인



Training data 일부분으로 과적합 여부 확인: (1) 일부의 data를 고른다.
 (2) reg = 0.0에서 sgd를 사용
 (3) 이 경우 반드시 과적합이 나와야 한다. ($loss \approx 0$, $accuracy \approx 1$) // meaning: 가중치 업데이트, 역전파가 제대로 작동하고 있고, 앞에서 설정된 학습률도 적절하다. 만약 과적합이 나오지 않으면, 문제가 있는 것.



학습률 learning rate 찾아가기: * lr too high \rightarrow loss 값 = Nan
 * 정확한 lr 결정은 교차 검증으로 해야 해

6. 모수parameter 최적화 (교차검증을 통해 hyperparameter 결정하기)

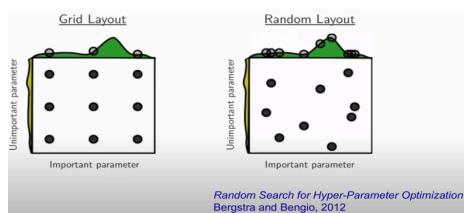
6.1 교차 검증의 전략:

Coarse (A few epoch) ==> Fine (Longer running time)

6.2 Hyperparameter에 대한 두 가지 탐색 과정

(1) 랜덤 탐색: logspace에 랜덤하게 값을 골라오기 ==> val_acc ↑

(2) 그리드 탐색: 등간격으로 모든 범위를 커버할 수 있도록 exhaustive 탐색하기 // worse.



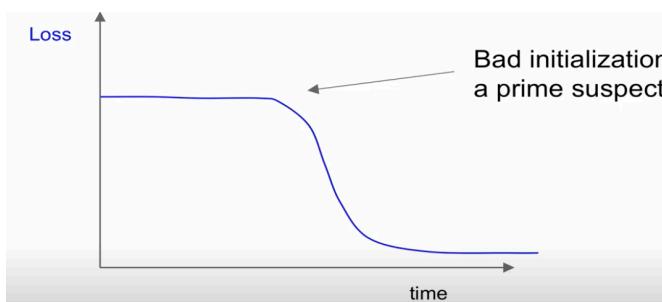
(1) 위 그림을 참고해보면, Grid 탐색은 중요한 모수의 3 지점만을 탐색. 반면에, 랜덤 탐색은 9 개의 지점을 탐색. 더 많은 중요한 모수의 지점을 탐색하는 랜덤 탐색이 better.

(2) 랜덤 탐색에서 중요치 않은 모수는 하나의 중요한 모수에 대해 한번만 탐색하는 데 반하여, 그리드 탐색에서 중요치 않은 모수는 하나의 중요한 모수에 대해 3 번을 탐색한다 => 비효율적.

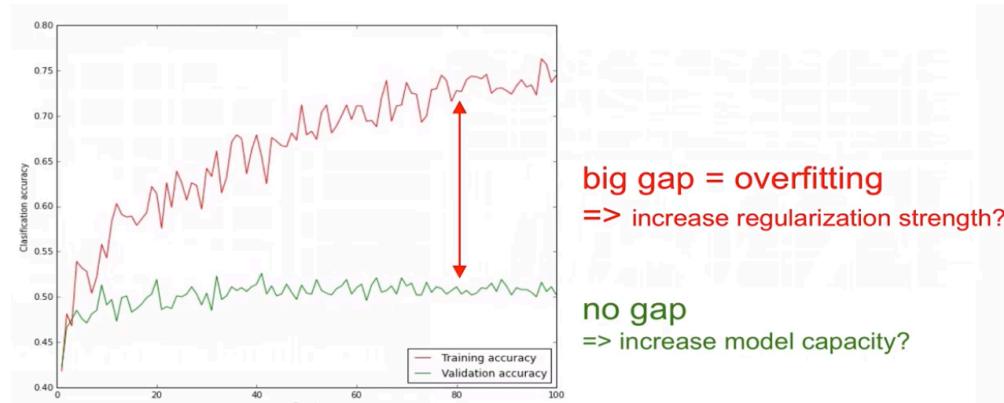
6.3 모니터링 대상

(1) loss

- 손실 함수가 bump 하는 경우: 초기화를 잘못했음을 의심



(2) 정확도 (학습 정확도, 검증 정확도)



(3) 가중치 업데이트 / 가중치 크기 ($1/10000$ ideal)

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

ratio between the values and updates: $\sim 0.0002 / 0.02 = 0.01$ (about okay)
want this to be somewhere around 0.001 or so

7. 파라미터 업데이트

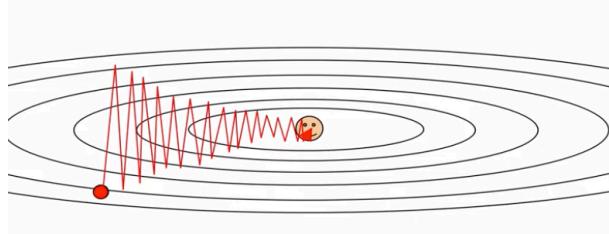
neural network, main loop:

```
while True:  
    data_batch = dataset.sample_data_batch()  
    loss = network.forward(data_batch)  
    dx = network.backward()  
    x += - learning_rate * dx
```

simple gradient descent update
now: complicate.

이 모수 업데이트하는 부분을 알아보자. 위 코드에서 이것은 매우 단순하다. 하지만 이것은 너무 느리기 때문에 실제로는 잘 사용하지 않는다. SGD는 수직으로는 간격이 좁은데, 수평으로는 간격이 다소 넓다. 이 경우에 SGD에 따르면, 우리는 최소 지점으로 이동하는 경로가 (수직은 경사가 급해서 빠른데, 수평은 경사가 낮아서 느리니까) 벡터합을 이용하면 지그재그로 이동할 것이다.

ss function is steep vertically but shallow hori



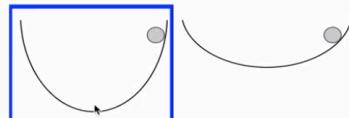
at is the trajectory along which we converge to the minimum with SGD? very slow pro

이것을 개선해보자!

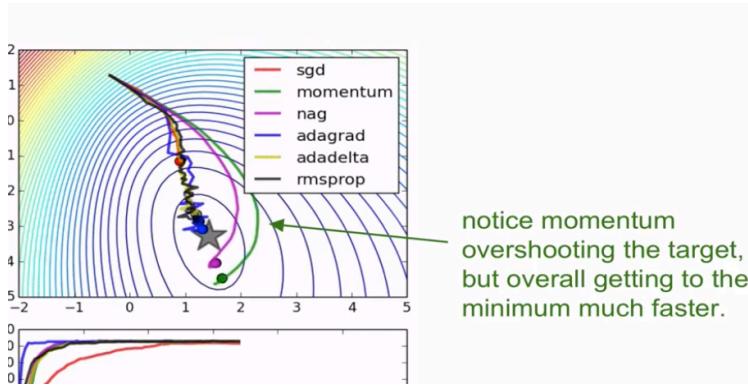
Momentum update

v라는 변수를 도입하자. 이것은 속도를 의미한다. 우리는 x의 위치를 이 속도를 통해 업데이트한다. 이것은 마치 언덕에서 공을 굴리는 것과 같다. 여기서 mu는 마찰계수와 같다.

```
# Gradient descent update  
x += - learning_rate * dx  
  
# Momentum update  
v = mu * v - learning_rate * dx # integrate velocity  
x += v # integrate position
```



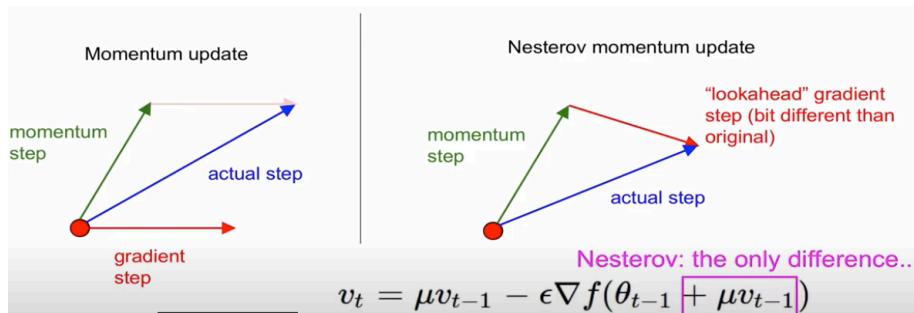
그래서 이것은 경사 하강과는 다르게, 경사가 낮은 지점에서는 속도가 천천히 빌드업되고, 경사가 깊은 곳에서는 속도가 빨라져서 최저 지점을 그냥 지나가지만 마치 추 운동처럼 결국 최저 지점으로 수렴한다.



이것은 초반에 위 초록 선 같이 오버슈팅이 일어나는데, 이것은 속도가 빌드업되는 현상 때문이다. 하지만 결국 최저 손실점으로 온다.

Nesterov Momentum update

이것은 위 모멘텀의 변형이다. 이것은 모멘텀 업데이트 보다 항상 컨버전스 레이트가 더 좋다는 것이 증명되었다. 그 이유는 보통 아까의 모멘텀은 두 개의 파트로만 나눠져 있다: 모멘텀 스텝, 그래디언트 스텝. 반면에, 네스테로브 모멘텀 업데이트에서는, 이미 모멘텀 스텝으로 이동한 것으로 예상한 지점에서 그래디언트 스텝을 시작하게 한다. 즉, 그래디언트 스텝을 계산하기 전에, 모멘텀 스텝을 미리 고려를 해서, 시작점을 모멘텀 스텝의 종료점으로 변경을 한 뒤, 그래디언트 스텝을 평가한다.



불편한 점: 네스테로프를 사용 시, 역전파 및 순전파 과정에서, 세타(파라미터 벡터)와 그 위치에서의 그래디언트를 구하는 경우에, 우리는 이때 세타와 다른 위치에서의 그래디언트를 구해야 하기 때문에, 일반적인 최적화 가령 API 코드들과 호환성이 떨어진다.

Variable transform and rearranging saves the day: $\phi_{t-1} = \theta_{t-1} + \mu v_{t-1}$

Replace all thetas with phis, rearrange and obtain:

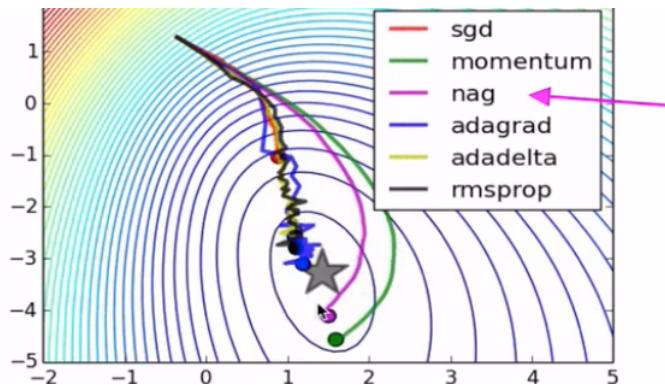
$$v_t = \mu v_{t-1} - \epsilon \nabla f(\phi_{t-1})$$

$$\phi_t = \phi_{t-1} - \mu v_{t-1} + (1 + \mu) v_t$$

vanilla update

```
# Nesterov momentum update rewrite
v_prev = v
v = mu * v - learning_rate * dx
x += -mu * v_prev + (1 + mu) * v
```

이것을 계산하기 위해 위 같이 pi를 도입한다.



이것은 방향을 미리 예측하고 가기 때문에 단순 모멘텀 업데이트 보다 더 빠르게 수렴하는 것을 볼 수 있다.

Adagrad update

캐시 개념이 도입된다. 일반적인 SGD로 보이지만, 여기서는 캐시의 루트를 씌워준 다음에 나눠준다는 것이 SGD와의 차이점이다.

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

이것은 모든 파라미터가 동일한 학습률을 제공받는 것이 아니라, 캐시가 계속 빌딩업 되니까, 하나의 패러미터마다 다른 학습률을 부여받게 된다. 여기서 e^{-7} 은 0으로 나누는 일을 방지하기 위한 역할을 하는 것이다. 이렇게 아다그래드를 사용하면, 수직 축은 그래디언트가 큰 데, (그래서 캐시 값이 커진다 \rightarrow 분모의 캐시 값 커진다 \rightarrow x의 업데이트 속도 낮아진다) 수평 축은 그래디언트가 작다. (캐시 작아 \rightarrow 업데이트 속도 빨라져) 그래서, 수직 축에서는 업데이트 속도 낮춰주고, 수평 축에서는 그 속도를 빠르게 해줘서, 경사에 경도되지 않는 효과를 가져온다.

문제점: 스텝사이즈가 시간이 흐름에 따라서 캐시 값이 계속 증가하는데, 그러면 결국 학습률은 0에 매우 가깝게 되어 학습이 종료된다. 따라서 학습이 종료되지 않게 해줘야 한다.

RMSProp 업데이트

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)

↓

# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

위 adagrad를 개선한 것이 이것이다. 여기서는 decayrate가 도입된다. 이것은 하나의 하이퍼-파라미터로 볼 수 있다. 이것은 그래디언트 제곱의 합을 구하는 것은 동일한데, 이 디케이蓐을 구함으로써 캐시의 값이 서서히 커지게 된다. 그래서 adagrad의 장점인 경사에 경도되지 않는 평형 효과는 유지하면서, 이것의 단점이었던 stepsize가 0이 되는 문제점을 해결해준다.

Adam 업데이트

RMSprop과 모멘텀을 더한 것이 이것이다. 베타1,2 는 초매개변수이다.

Adam update [Kingma and Ba, 2014]

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

RMSProp-like

Looks a bit like RMSProp with momentum

Adam update

[Kingma and Ba, 2014]

```
# Adam
m,v = ... initialize caches to zeros
for t in xrange(1, big_number):
    dx = ... evaluate gradient
    m = beta1*m + (1-beta1)*dx # update first moment
    v = beta2*v + (1-beta2)*(dx**2) # update second moment
    mb = m/(1-beta1**t) # correct bias
    vb = v/(1-beta2**t) # correct bias
    x += - learning rate * mb / (np.sqrt(vb) + 1e-7)
```

momentum
bias correction
(only relevant in first few iterations when t is small)
RMSProp-like

이것이 좀 더 완벽한 형태의 코드인데, 여기서 bias correction은 최초의 m과 v가 0으로 초기화 되었을 때, 즉 t가 작은 값인 초기 iter 과정에서 이들을 scale up 해준다.

정리

이 업데이트 방법 중 어떤 것도 최선이 아니다. 학습률은 초기에 다소 큰 것을 적용하고, 그 다음에는 서서히 그 것을 decay 해주면서, 값을 작게 만들면서 적용을 해주는 것이 최선이다. 그 디케이 방법은 아래 같은 것들이 있다. 현실적으로는 2번째 것이 가장 자주 사용된다. 그리고 위 업데이트 중 adam이 현재 가장 인기 있다. 이렇게 그래디언트 정보만 사용하는 것을 1st order 방법이라고 한다.

- step decay
- exponential decay
- $1/t$ decay

Second order optimization 방법

해시안을 이용해서 곡면이 어떻게 구성되었는지를 알 수 있다. 이 곡면의 구성을 알면 학습이 요구되지 않고서 바로 최저점으로 갈 수 있다. 그러면 학습률 업데이트도 필요가 없다.

Second order optimization methods

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

이것을 사용할 때의 장점은 수렴이 매우 빨라지고, 학습률 같이 초매개변수가 요구되지 않는다. 하지만 이것은 우리의 DNN에서는 현실적으로 사용이 불가능하다. 왜냐하면 우리가 만약 매개변수가 1억개 있다면, 이 해시안 행렬은 1억 x 1억의 행렬이 필요하고, 이것의 역행렬을 구하는 연산이 요구된다.

8. 평가: 모델 양상블

단일 모델을 학습시키는 대신에, 복수 개의 독립적인 모델들을 학습시킨다. 그리고 테스트 시간 때 이들 결과의 평균을 내준다. 그러면 거의 항상 성능이 2% 정도 향상된다.

단점: 트레이닝 시, 모델이 여러 개여서 이것을 관리해야 한다. 또한, 테스트 시에도, 이것의 평균을 내려면, 선형적으로 테스트 속도가 느려진다.

Tricky 방법:

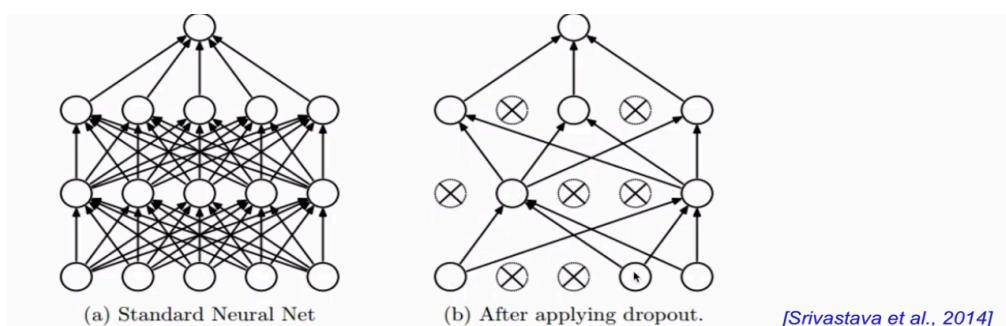
여러 개 모델이 아닌 단일 모델 내에서 한번 epoch를 돌 때마다 체크포인트를 생성하는데, 이 체크 포인트 간의 양상블을 하더라도 성능 향상을 볼 수 있다. 더욱이, 파라미터 벡터들 간의 양상블도 성능 향상을 가져온다:

```
while True:
    data_batch = dataset.sample_data_batch()
    loss = network.forward(data_batch)
    dx = network.backward()
    x += - learning_rate * dx
    x_test = 0.995*x_test + 0.005*x # use for test set
```

이것이 가능한 이유는 bowl 형태의 함수를 가지고 있을 때, 우리가 이것을 최적화 하려고 한다고 해보자. 이때 우리는 최솟값에 가야 하는데 자꾸 이것을 지나칠 수 있다. 즉 stepsize가 클 수 있다. 그런데 이것 하나하나를 다 평균화하면 이것은 위같이 효과적일 수 있다.

9장. Dropout

정규화 하기 위해 사용된다. 이것은 매우 간단한데 성능을 높인다. 최근에는 BN을 사용해서 이것을 잘 안 사용하는 경우도 있기는 하다.



(b)를 보면 몇몇 노드를 0으로 처리한 것을 볼 수 있다.

```

p = 0.5 # probability of keeping a unit active. higher = less dropout

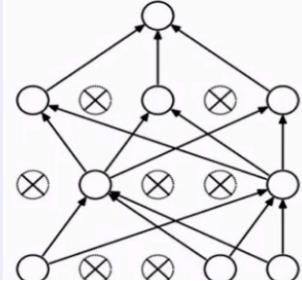
def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

```

Example forward pass with a 3-layer network using dropout



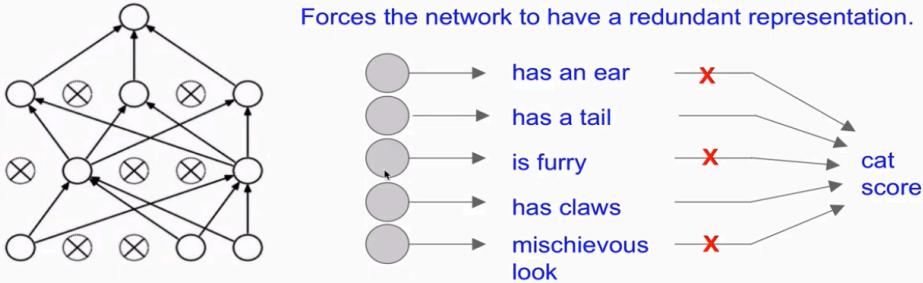
첫번째 줄을 보면, 선형 변환 해준 다음에, U1을 통해 이진 마스크를 해준다. 그래서 p=0.5 이니까 이것보다 클 경우에는 1, 그렇지 않으면 0. (드롭아웃) 시킨다.

두번째 경우에도 이진 마스크를 해줌으로써 (U2) 를 사용. (*U2 ...)에서 아스테리카는 튜플을 unpack 해주기 위해 사용한다.

* 드롭아웃 장점:

Waaaait a second...
How could this possibly be a good idea?

Forces the network to have a redundant representation.



우리 신경망이 redundancy를 갖는다. 원래 모든 노드가 살아 있으면, 위 그림 같이 귀, 털 등을 인지하여 고양이 점수를 매긴다.

(해석 1) 그런데 이들 중 일부 노드가 없어진다고 해서 그 label에 해당하는 것을 인지하지 못하는 것은 아니다. 가령, 귀 노드를 없앤다고 해서 귀 노드를 인지하지 못하는 것은 아니다. 꼬리 노드가 귀 노드를 인지하려고 노력할 것이고 다른 것들도 마찬가지일 것이다.

(해석 2) 이것 역시 매개변수를 공유하는 모델의 큰 양상을 훈련시키는 것이다. 즉, 여기에서는 가중치를 공유하는 각각의 모델의 파라미터 값을 평균값으로써 양상을 효과를 낼 수 있다.

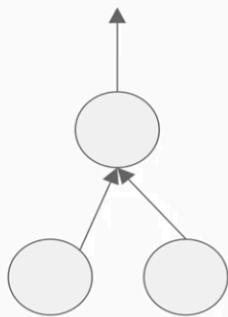
* 테스트 타임 때 드롭아웃:

몬테카를로 근사법: 테스트 시에도 드롭아웃을 활용하자. (=> 매우 비효율적)

At test time....

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



Q: Suppose that with all inputs present at test time the output of this neuron is x .

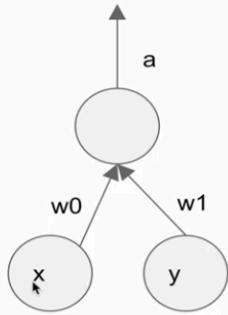
What would its output be during training time, in expectation? (e.g. if $p = 0.5$)

$p=0.5$ 일 때, 우리가 얻을 수 있는 기대치는?

At test time....

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



during test: $a = w0*x + w1*y$

during train:

$$\begin{aligned} E[a] &= \frac{1}{4} * (w0*0 + w1*0 \\ &\quad w0*0 + w1*y \\ &\quad w0*x + w1*0 \\ &\quad w0*x + w1*y) \\ &= \frac{1}{4} * (2 w0*x + 2 w1*y) \\ &= \frac{1}{2} * (w0*x + w1*y) \end{aligned}$$

마지막 줄의 $1/2$ 는 결국 0.5 와 같다. 이것이 함의하는 바는 다음과 같다: $p=0.5$ 를 사용하는 경우에, 우리가 트레이닝 때 사용했던 활성화 보다 2 배가 더 부풀려지는 그런 경우를 테스트 때 얻게 되는 것이다.

We can do something approximate analytically

```
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p, # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

At test time all neurons are active always

=> We must scale the activations so that for each neuron:
output at test time = expected output at training time

힌튼 교수가 처음에 드롭아웃을 시도했을 때 잘 안되었는데, 그 이유는 테스트 타임 때 트레이닝 타임 만큼 scaling을 해줘야 했기 때문이다.

정리하자면, 위 그림과 같이, 테스트 타임 시 모든 뉴런은 살아있고, 활성화 값들을 트레이닝 타임 때의 기대치 만큼 스케일링(*p)을 해줘야 한다.

More common: “Inverted dropout”

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

test time is unchanged!

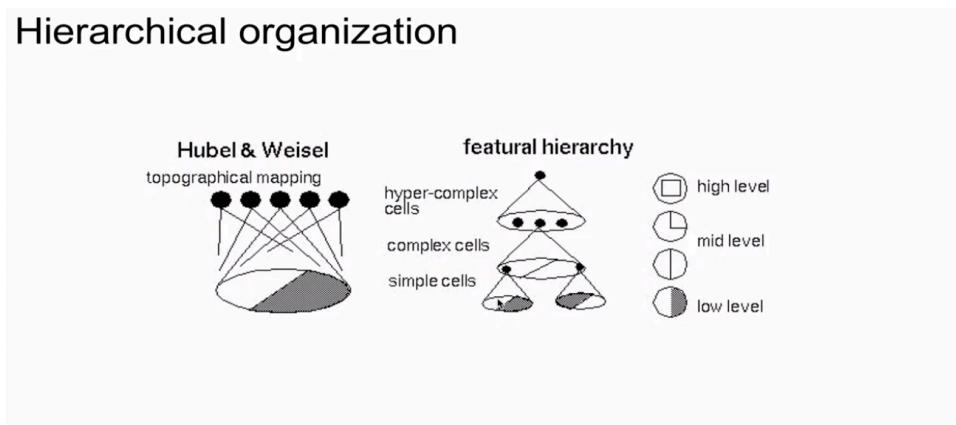


하지만 보다 현실적으로는 테스트 타임은 그대로 두고, 대신에 트레이닝 타임 때 p를 나눠준다. 즉, 트레이닝 타임 때 스케일링을 미리 처리해준다.

10. CNN

1959에 수행되었던 생물학 실험이 ANN의 동기가 되었다. 그것은 고양이의 시각 반응에서 특정 이미지에서 어떤 신경이 활성화된다는 것을 밝힌 것이다. 즉 이것은 cortex에서 지역성이 보장이 된다는 것을 보였다. (?)

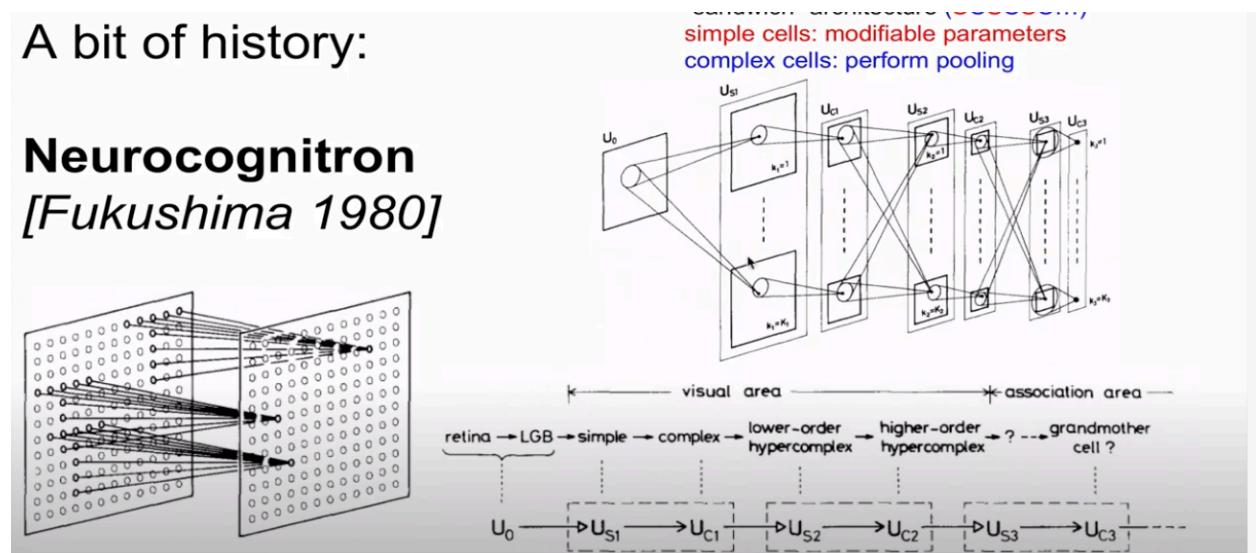
Hierarchical organization



휴블과 위즐은 여기서 맨왼쪽에 그림처럼 지역성이 유지된다는 것을 보였고, 가운데 그림을 보면, 아주 간단하고 작은 영역을 보는 cell이 있고, 이들을 관장하는 좀 더 복합적 cell이 있고, 이 전체를 관장하는 초복합적 cell이 있음을 밝혔다.

Neurocognitron

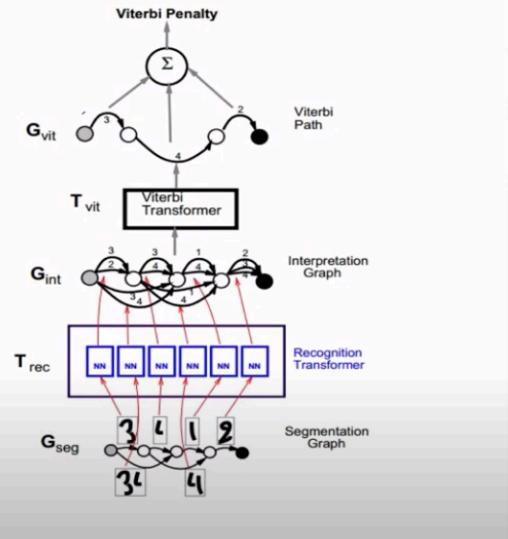
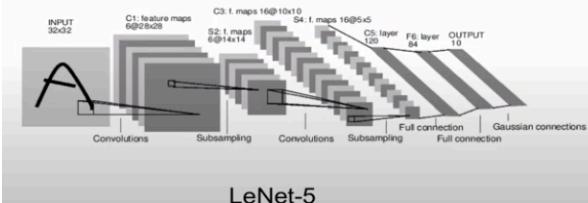
A bit of history: **Neurocognitron** [Fukushima 1980]



이때는 역전파가 불가능했다.

A bit of history: Gradient-based learning applied to document recognition

[LeCun, Bottou, Bengio, Haffner
1998]

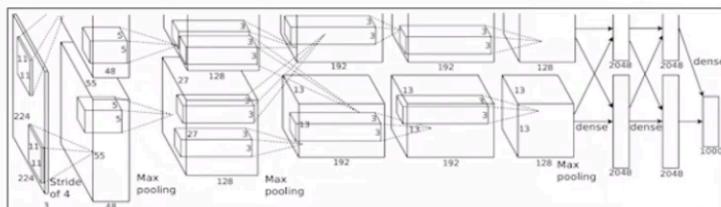


기계가 zip 코드를 분류해준 것. 이때는 역전파가 가능했었고, 그래서 CNN의 가능성성이 확인됐었다.

아래의 알렉스넷 덕분에 CNN이 인기를 끌게 됐다. 하지만 이전 것과 큰 차이는 없었다. 여기서는 이전과 다르게 BN을 사용했고, GPU가 발전했고, 더 깊은 신경망을 사용했다.

A bit of history: ImageNet Classification with Deep Convolutional Neural Networks

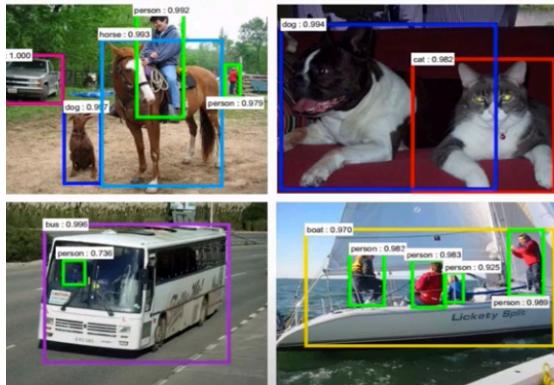
[Krizhevsky, Sutskever, Hinton, 2012]



“AlexNet”

Fast-forward to today: ConvNets are everywhere

Detection



[Faster R-CNN: Ren, He, Girshick, Sun 2015]

Segmentation



[Farabet et al., 2012]

Segmentation: 오브젝트의 형태를 따내는 것

Detection: 분류화에서는 localization하는 것이 아닌, 하나의 이미지 내에서 여러 개의 오브젝트를 찾아내는 것.

Reference

[1] cs231n lectures

[2] Sergey Ioffe, Christian Szegedy, ‘Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift’, 2015