# 8강: 신경망 파트3 - 학습 및 평가

# 그래디언트 확인(gradient check)

- 분석적 그래디언트는 미적분학 지식을 통해 그래디언트 계산
- 수치적 그래디언트를 아래 식을 활용하여 근사적으로 그래디언트를 계산하는 방식

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x)}{h} \quad \text{(bad, do not use)}$$

- 그래디언트 확인이란 이 두 그래디언트를 비교하는 과정임
- 그래디언트 확인 과정은 생각보다 복잡하고 오류가 발생하기 쉬움

# 팁1: 중심 차분 공식(centered difference formula) 사용

- 수치적 그래디언트 계산 시 중심 차분 공식을 사용하는 것이 훨씬 더 잘 계산됨

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x-h)}{2h} \quad \text{(use instead)}$$

- 중심차분 공식은 그래디언트의 각 차원에 적용을 해야함
- 중심차분 형태는 각 차원 당 손실 함수를 2번 계산하게 함
- 계산비용은 2배인 대신 그래디언트 근사값이 훨씬 더 정확히 계산

# 팁2: 비교를 위해 상대 오차(relative error) 사용

- 상대 오차를 사용하는 것이 좋음
- 만약 $|f'_a - f'_n|$ 혹은 $|f'_a - f'_n|^2$ 이 크면 그래디언트 확인이 실패라고 판단하는 경우에 문제가 있음
- 둘의 차이가 $10^{-4}$인 경우 두 그래디언트가 1.0 근처이면 이 차이는 매우 적절한 차이이므로 두 그래디언트가 일치한다 생각
- 두 그래디언트가 모두 $10^{-5}$ 이하라면 $10^{-4}$는 매우 큰 차이이므로 그래디언트 확인은 실패라고 판단될 것임
- 따라서 항상 상대 오차를 고려하는 것이 더 적절함

# 팁2: 비교를 위해 상대 오차(relative error) 사용

$$\frac{|f'_a - f'_n|}{\max(|f'_a|, |f'_n|)}$$

- 대칭성을 위해 분모에서는 $f'_a, f'_n$ 값 두 개를 모두 활용함
- 두 값중 0인 것이 있는 경우 0으로 나눠지지 않아야하므로 두 값의 최대값으로 나누는 것이 선호됨
- 두 값 모두 0인 경우도 잘 체크해야함

# 팁2: 비교를 위해 상대 오차(relative error) 사용

- 상대오차 $> 10^{-2}$
  - 그래디언트가 잘못됐다는 것을 의미
- $10^{-2} >$ 상대오차$>10^{-4}$
  - 불편함을 느낌
- $10^{-4} >$ 상대오차
  - 일반적으로 꺾임점이 있는 목표에 대해서는 괜찮음
  - 꺾임점이 없는 경우(ex. tanh, softmax 사용) $10^{-4}$는 너무 높음
- $10^{-7} >$ 상대오차
  - 만족

# 팁2: 비교를 위해 상대 오차(relative error) 사용

- 신경망이 깊을 수록 상대 오차는 더 커짐
- 10층 신경망 입력 데이터에 대해 그래디언트를 확인하는 경우 오류가 누적되어 $10^{-2}$의 상대 오차가 괜찮을 수도 있음
- 단일 미분 가능함수에 대한 $10^{-2}$ 오차는 잘못된 그래디언트를 나타낼 가능성이 높음

# 팁3: 배정밀도(double precision) 사용

- 일반적인 실수는 그래디언트 확인을 계산할 때 단정밀도(single precision) 부동소수점을 사용하는 것
- 그래디언트 구현이 올바르더라도 상대오차가 높게 나타나는 경우 있음 (ex. 0.01)
- 배정밀도로 전환함으로써 상대오차가 급격이 떨어지는 경우가 종종 있음

# 팁4: 부동 소수점의 활성범위 주변에 머무르기

- 손실 함수를 배치 수로 나눌 때 매우 작은 숫자를 생성하는 경우 많은 수치적인 문제를 초래할 수 있음
- 수치적/분석적 그래디언트 비교 시 두 값이 극도로 작지는 않은지 확인하는 것이 좋음
- 매우 작아진다면, 일시적으로 손실함수를 스케일링하여 더 좋은 부동소수점 범위로 가져오는 것이 좋음
  - ex) 1.0 주변

# 팁5: 목적 함수의 꺾임 주의

- ReLU 함수, SVM 손실, maxout 뉴런 등의 함수는 꺾인 지점(즉, 미분 불가능한 지점)이 있음
  - ReLU = max(0,x)
  - maxout: $\max(w_1^T x + b_1, w_2^T x + b_2)$

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)$$

- 이런 부분에서의 그래디언트 확인은 에러가 생길 수 있음
- ex) ReLU 함수의 경우 $x = 10^{-6}$에서의 그래디언트 계산 시 해석적 그래디언트는 0이지만 수치적 그래디언트는 0이 아닐 수 있음
- 많은 꺾임 함수들이 있기 때문에 이런 에러가 꽤 흔할 수 있음

# 팁5: 목적 함수의 꺾임 주의

- 손실을 계산할 때 꺾임이 교차되었는지 여부를 체크할 수 있음
- 손실 계산 시 max(x,y) 형태의 함수에서 어떤 것이 큰지도 추적함
- 즉, 순방향 패스 동안 x,y 중 어떤 것이 큰지에 대한 정보도 저장
- 그러면 수치적 그래디언트 계산 시 f(x+h), f(x-h) 계산에서 꺾임이 교차되었는지 체크할 수 있고, 꺾임이 교차된 경우라면 수치적 그래디언트가 정확하지 않을 것임

# 팁6: 몇 가지의 데이터 포인트만 사용

- 꺾임 문제를 해결하기 위한 하나의 방법은 이미지 수를 줄이는 것
- 꺾임을 포함하는 손실 함수는 이미지가 많을 수록 꺾임이 적게 발생
- 가령 2-3개의 이미지에 대해서만 그래디언트 확인을 수행하여도 그래디언트 계산 식에 이상 여부를 확인하는데 충분할 수 있음
- 아주 적은 수의 이미지를 사용하면 그래디언트 확인이 더 빠르고 효율적임

# 팁7: 스텝 크기 h에 주의

- 일반적으로 수치적 그래디언트 계산 시 h가 작을 수록 더 좋은 근사값을 얻음

$$[f(x+h) - f(x-h)]/2h$$

- 그렇지만 h가 매우 작아질 경우 수치 정밀도 문제로 더 좋은 근사값을 얻지 못할 수도 있음
- 때때로, 그래디언트 확인에 실패했을 때, h를 $10^{-4}$ 혹은 $10^{-6}$으로 키우면 그래디언트 확인에 성공할 수 있음

# 팁8: "특성화"된 동작 모드에서 기울기 확인

- 단일 지점에서 그래디언트 확인이 성공하더라도 그래디언트가 전체적으로 올바로 구현된 건지는 확실치는 않음
- 또한, 랜덤 초기화는 파라미터 공간에서 "특성화"된 지점이 아닐 수 있고, 실제로 그래디언트가 올바로 구현되지 않았음에도 된 것처럼 보일 수 있음
- 예를 들어, 가중치 초기화가 매우 작은 SVM은 모든 이미지에 대해 거의 0의 점수를 할당하면, 일부 점수가 다른 점수보다 큰 특성화된 모드로 일반화되지 못할 수 있음
- 안전한 그래디언트 확인을 위해서는, 신경망이 먼저 잠시 학습하면서 손실이 감소하기 시작한 후에 그래디언트 확인을 해야함

# 팁9: 정규화가 데이터를 압도하지 않도록 함

- 데이터 손실이 정규화 손실을 압도하지 않도록 해야함
- 그렇지 않으면 데이터 손실의 잘못된 구현을 파악하지 못하게 할 수 있음
- 먼저, 정규화를 끄고 데이터 손실만 확인한 후, 독립적으로 정규화 항을 확인하는 것이 권장됨
- 정규화 항만 독립적으로 확인하는 방법
  - 데이터 손실을 포함 안 하도록 코드 수정
  - 정규화 강도를 늘려 그래디언트 확인에서의 큰 비중을 차지하도록 함

# 이성 검사 (sanity check)

- 시간이 많이 걸리는 훈련 과정 전에 초기화나 구현에 문제가 없는지 확인하는 것은 이성 검사(sanity check)라고 불림
- 방법1: 작은 파라미터로 초기화할 때 예상하는 손실을 얻는지 확인

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta) \qquad L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

  - CIFAR-10에서 소프트맥스 분류기의 경우 초기손실이 2.302, SVM ($\Delta = 1$ 인 경우) 의 경우 9일 것으로 예상됨

# 이성 검사 (sanity check)

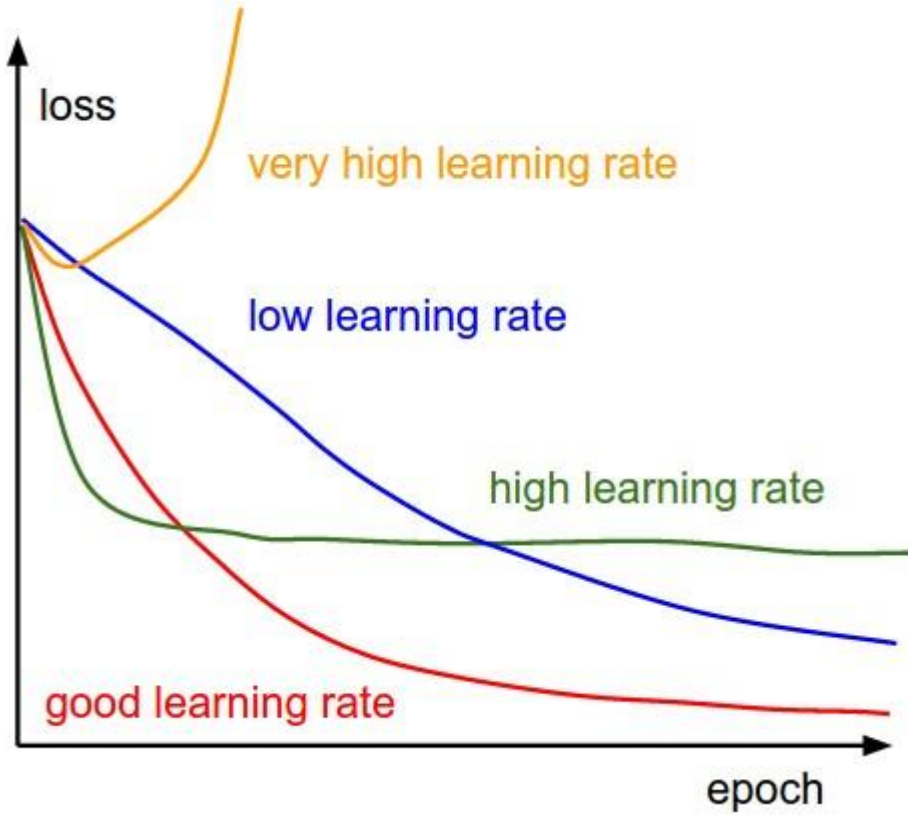
- 방법2: 정규화 강도를 늘릴 때 손실이 증가하는지 확인
- 방법3: 매우 작은 데이터셋에 과적합 시켜봄
  - 데이터의 아주 작은 부분(ex, 20개 이미지)에 대해 학습시키고 손실이 0에 도달하는지 확인해봄
  - 손실이 0이 되는 것을 방해할 수 있기 때문에 정규화는 0으로 설정
  - 작은 데이터셋에 대해 이 이성검사를 통과하지 못하면 전체 데이터셋으로 넘어가는 것은 가치가 없음

# 훈련 과정 중 모니터링

- 신경망 훈련 중 중요한 수치들을 모니터링 하여 그래프로 그리면 , 다양한 하이퍼파라미터 설정에 대한 직관을 얻고, 어떻게 하이퍼파라미터를 변경해야할지 파악하는데 사용할 수 있음
- 다음에 나올 그래프들은 x축이 에폭(epoch) 단위
- 에폭이란 학습 중 각 이미지를 대략 몇 번 볼 수 있는지를 측정
- 배치 크기의 설정에 따라 반복횟수가 달라지므로, 반복 횟수보다는 에폭을 모니터링하는 것이 더 좋음

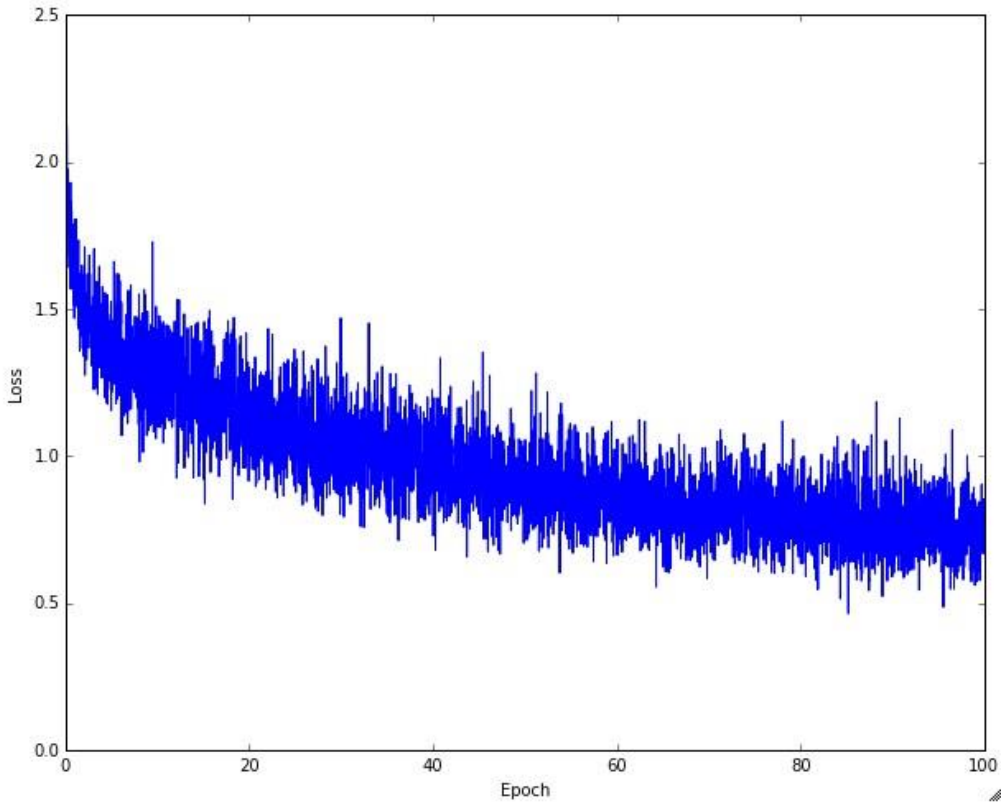
# 모니터링1: 손실

- 손실은 학습 중 추적하는 것이 중요한 수치 중 하나
- 낮은 학습률 사용 시 손실 그래프는 선형적임
- 높은 학습률 사용 시 더 지수적이지만 안 좋은 손실 값에 수렴할 수 있음
- 좋은 학습률은 어느 정도 빨리 손실이 감소하면서도 낮은 손실 값에 수렴함



# 모니터링1: 손실

- CIFAR-10 데이터셋에서 작은 신경망을 학습하는 동안의 일반적인 손실 그래프
- 각 배치의 손실 값들을 그래프로 그림
- 손실의 진동 정도는 배치 크기와 관련됨
- 배치 크기가 1이면 진동은 상대적으로 높음
- 배치 크기가 전체 데이터셋이면 진동은 최소화됨

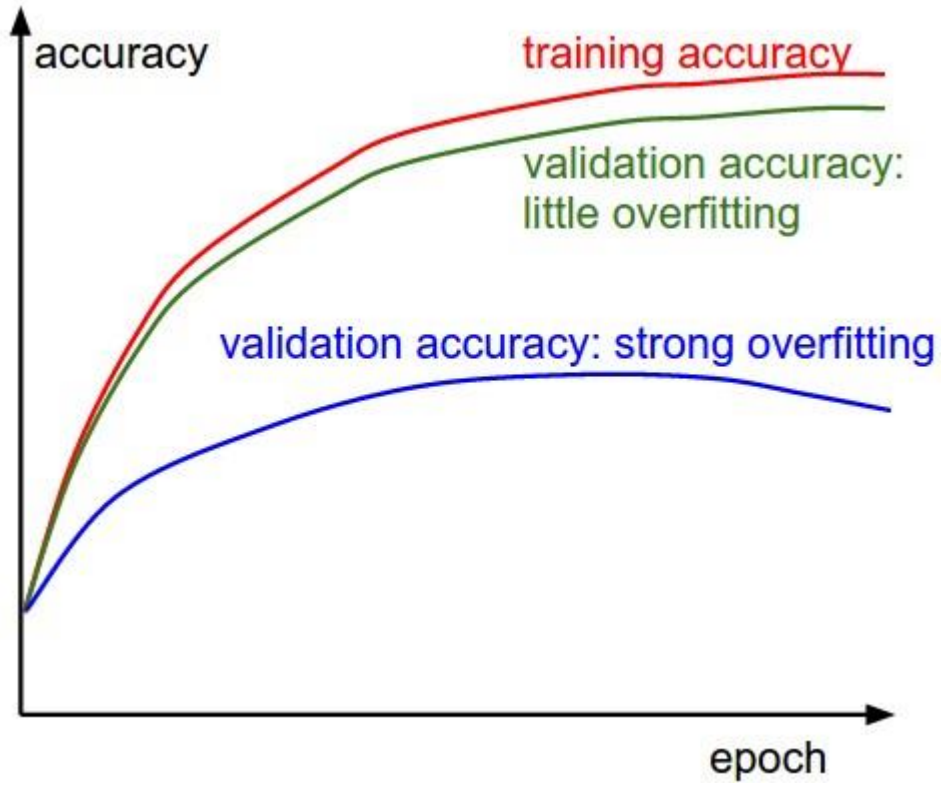


# 모니터링1: 손실

- 일부 사람들은 손실 함수를 로그 도메인에서 그래프를 그리는 것을 선호함
- 그래프가 약간 더 해석하기 쉬운 직선으로 나타나게 됨

# 모니터링2: 학습/검증 정확도

- 학습 정확도와 검증 정확도의 차이는 과적합의 정도를 나타냄
- 파란색 검증 정확도 곡선은 강한 과적합을 나타냄
- 과적합 해결을 위해서는 정규화 강도를 늘리거나 데이터셋을 더 수집할 수 있음
- 초록색 검증 정확도 곡선은 검증 정확도가 훈련 정확도를 꽤 잘 따라감
- 이 경우는 신경망 크기가 충분히 크지 않은 늘려 신경망을 크게 만드는 것 고려



# 모니터링3: 업데이트 / 가중치값 비율

- 업데이트 크기와 값 크기의 비율을 모니터링할 수 있음
- 업데이트란 그래디언트에 학습률이 곱해진 것
- 이 비율은 모든 파라미터 세트에 대해 독립적으로 평가하고 모니터링하고자 할 수 있음
- 대략 이 비율은 $10^{-3}$ 정도 될 것이 권장됨
- 이보다 낮으면 학습률이 너무 낮을 수 있음
- 이보다 높으면 학습률이 너무 높을 가능성이 있음

# 모니터링3: 업데이트 / 가중치값 비율

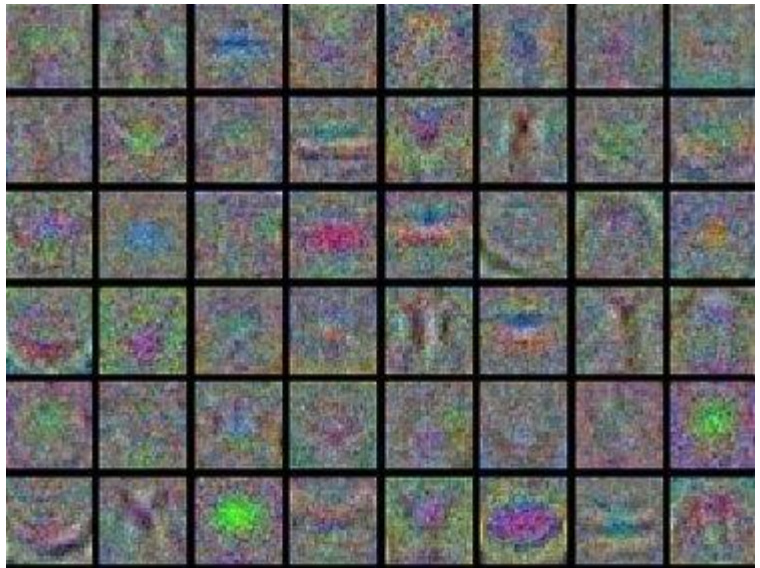
```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

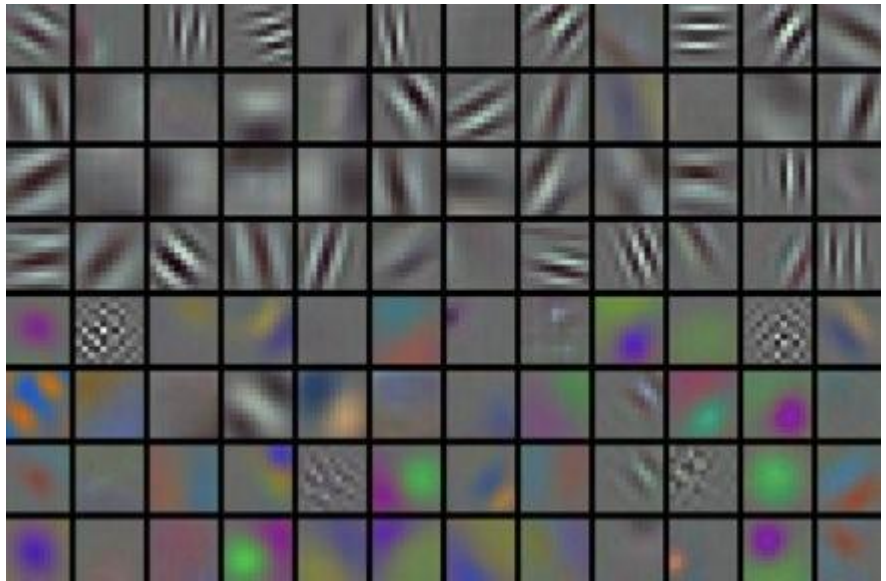
# 모니터링4: 계층별 활성화 / 그래디언트 분포

- 잘못된 초기화는 학습 과정을 느리게 하거나 완전히 정지시킬 수 있음
- 이 문제는 상대적으로 쉽게 진단 가능
  - 모든 계층의 활성화/그래디언트 히스토그램 그림
- 이상한 분포를 보는 것은 좋은 징후가 아님
- 예를 들어, tanh 뉴런을 사용할 때는, 모든 뉴런이 0을 출력하거나 -1 혹은 1에서 포화되는 대신, [-1,1]의 전체 범위에서 분포하기를 원함

# 모니터링5: 첫 번째 계층 시각화

- 첫 번째 계층에 대한 가중치 시각화 예시
- 왼쪽 그림
  - 잡음이 많음
  - 미수렴, 잘못된 학습률 설정, 매우 낮은 가중치 정규화 페널티 가능성
- 오른쪽 그림
  - 깨끗하고 매끄럽고 다양한 특징들이 있음
  - 학습이 잘 이행되고 있다는 것을 나타내는 좋은 징후





# 파라미터 업데이트

- 바닐라(vanila) 경사하강법 코드

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

- 이 외에도 다양한 파라미터 업데이트 방법이 존재함

# 접근방식1: SGD와 추가 기능

- 바닐라 업데이트

```
# Vanilla update
x += - learning_rate * dx
```

  - 가장 간단한 형태의 파라미터 업데이트
  - learning_rate: 학습률 (하이퍼파라미터)
  - 그래디언트가 전체 데이터셋에 대해 계산되었고, 학습률이 충분히 낮다면 손실함수가 항상 감소하는 것을 보장함

# 접근방식1: SGD와 추가 기능

- 모멘텀(momentum) 업데이트
  - 심층 신경망에서 거의 항상 더 좋은 수렴률을 경험하게 함
  - 최적화 문제의 물리적 관점에서 동기부여됨
  - 손실: 산등성이 지형의 높이 (h)
  - 퍼텐셜에너지 U=mgh 식에 따르면 $U \propto h$이므로 손실을 퍼텐셜에너지로 생각 가능

# 접근방식1: SGD와 추가 기능

- 모멘텀(momentum) 업데이트
  - 랜덤 초기화는 랜덤 위치에서 초기속도 0인 입자를 설정하는 것과 동일
  - 최적화 과정은 산에서 파라미터벡터(입자)를 시뮬레이션하는 것과 동일
  - $F = -\Delta U$ 식과 비슷하게, 입자가 느끼는 힘은 손실 함수의 음의 그래디언트에 비례함
  - $F = ma$ 식에 따르면 음의 그래디언트 $\Delta U$는 입자의 가속도(속도 변화)에 비례함

# 접근방식1: SGD와 추가 기능

- 모멘텀(momentum) 업데이트

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

  - 손실 함수의 음의 그래디언트(dx)가 속도 변화에 영향을 줌
  - mu: 모멘텀이라 불리며 보통 0.9 정도의 값을 가짐
  - 물리적 의미로는 모멘텀보다는 마찰계수와 더 관련이 높음
  - mu는 속도를 감소시키고 시스템의 운동에너지를 줄이는 역할을 하여 입자가 산의 바닥에서 멈추도록 함

# 접근방식1: SGD와 추가 기능

- 모멘텀(momentum) 업데이트

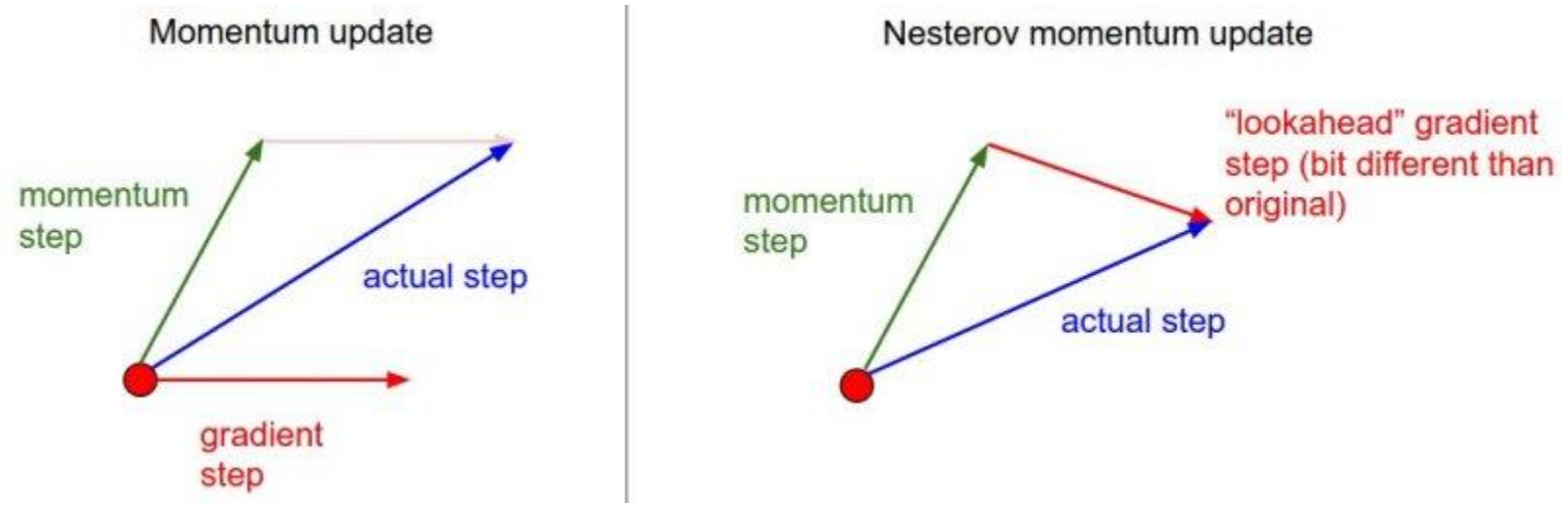
```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

  - 교차검증을 통해 모멘텀 파라미터는 일반적으로 0.5, 0.9, 0.95, 0.99와 같은 값으로 설정됨
  - 모멘텀 스케쥴링(값을 고정하지 않고 시간에 따라 바꿈) 시 더 도움됨
  - 모멘텀은 훈련의 후반부에 증가함
  - 일반적으로는 모멘텀을 0.5정도에서 시작하고, 여러 에폭을 거치고 나면 0.99 정도로 설정됨
  - 모멘텀 업데이트의 경우, 그래디언트가 일관된 방향이면 그 방향으로 속도를 빠르게 늘림

# 접근방식1: SGD와 추가 기능

- 네스테로프 모멘텀(Nesterov momentum)

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




```
x_ahead = x + mu * v
# evaluate dx_ahead (the gradient at x_ahead instead of at x)
v = mu * v - learning_rate * dx_ahead
x += v
```

# 접근방식2: 학습률 서서히 감소

- 심층신경망 학습 시, 일반적으로 시간이 지남에 따라 학습률을 줄이는 것이 도움됨
- 학습률이 높을 때는, 시스템에 많은 운동에너지가 있어서 파라미터벡터가 튀어다니면서, 손실함수의 깊고 좁은 부분에 정착할 수 없음
- 너무 천천히 줄이면 시스템이 오랫동안 튀어다니며 계산을 낭비
- 너무 빨리 줄이면 파라미터 벡터가 너무 빨리 느려져서 최선의 위치에 도달하지 못함

# 접근방식2: 학습률 서서히 감소

- 단계적 감소
  - 몇 에폭마다 학습률을 일정한 비율로 줄임
  - 일반적으로는 5 에폭마다 절반으로 줄이거나, 20 에폭마다 0.1로 줄임
  - 이 숫자들은 유형과 모델에 크게 의존함
  - 실제적으로 볼 수 있는건 고정된 학습률로 훈련하면서 검증정확도를 관찰하고 개선되지 않을 때마다 학습률을 상수(ex. 0.5)배만큼 줄이는 것

# 접근방식2: 학습률 서서히 감소

- 지수적 감소
  - 학습률을 지수적으로 감소시킴
  - 수학적 형태는 $\alpha = \alpha_0 e^{-kt}$
  - $\alpha_0, k$는 하이퍼파라미터이며 $t$는 반복횟수임 (에폭 단위 사용도 가능)

# 접근방식2: 학습률 서서히 감소

- 1/t 감소
  - 학습률이 $1/t$ 형태로 감소
  - 수학적인 형태는 $\alpha = \alpha_0/(1 + kt)$
  - $\alpha_0, k$: 하이퍼파라미터, $t$: 반복 횟수

# 접근방식2: 학습률 서서히 감소

- 단계적 감소, 지수적 감소, 1/t 감소 중 실제적으로는 단계적 감소가 약간 더 선호됨
- 이는 단계적 감소가 포함하는 하이퍼파라미터들 (감소비율 혹은 에폭 단위 단계 타이밍)이 $k$보다 해석하기 더 쉽기 때문
- 계산적으로 자원만 허용된다면, 더욱 느리게 감소시키는 쪽으로 하고 더 긴 시간 동안 훈련하는 것이 좋음

# 접근방식3: 이차 방법(second order methods)

$$x \leftarrow x - [Hf(x)]^{-1} \nabla f(x)$$

$$H(f) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

- 헤시안은 직관적으로 손실 함수의 지역적인 곡률을 설명하며, 이를 통해 더 효율적인 업데이트 수행 가능
- 역행렬 헤시안 곱셈은 곡률이 완만한 방향으로는 더 공격적인 스텝을, 곡률이 가파른 방향으로는 더 짧은 스텝을 취함
- 학습률 하이퍼파라미터가 필요 없다는 장점이 있음

# 접근방식3: 이차 방법(second order methods)

- 대부분의 응용에서는 실용적이지 않음
- 헤시안 행렬을 계산하고 역행렬을 구하는 것은 공간과 시간 면에서 매우 비용이 많이 들기 때문
  - 예) 백만개 파라미터 신경망은 약 3725GB의 RAM을 차지하는 1000000x1000000 크기 헤시안 가짐
- 역행렬 헤시안을 근사하는 다양한 quasi-Newton 방법들이 개발됨
- 그 중 L-BFGS는 가장 인기 있는 방법

# 접근방식3: 이차 방법(second order methods)

- L-BFGS는 메모리 문제 외에도 전체 훈련 세트에 대해 계산되어야한다는 문제가 있음
- 미니배치 SGD와 달리, L-BFGS를 미니 배치에서 작동하게 하는 것은 까다로움
- 실제로 L-BFGS나 유사한 이차방법들보다는 모멘텀에 기반한 SGD 변형이 더 표준적임

# 접근방식4: 파라미터별 적응적 학습률 방법

- 지금까지 논의된 접근법들은 모든 파라미터에 대해 전역적이고 동등하게 학습률을 조정
- 학습률을 적응적으로 조정하는 방법이 많이 연구됨
- 심지어 파라미터 별로도 수행 가능
- 여전히 다른 하이퍼파라미터 설정을 필요로 할 수 있지만, 원래의 학습률보다 더 넓은 범위의 하이퍼파라미터 값에 대해 잘 작동한다고 주장됨

# 접근방식4: 파라미터별 적응적 학습률 방법

- Adagrad
  - 처음으로 제안된 적응적 학습률 방법
  - 파라미터 별로 다른 학습률을 적용한다는 특징

```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

  - 큰 그래디언트를 가진 파라미터는 이미 빠르게 변화했으므로 이후 업데이트는 더 조심스럽게 이루어짐
  - 작은 그래디언트를 가진 파라미터는 느리게 변화했으므로 이후 업데이트는 더 빠르게 이루어짐

# 접근방식4: 파라미터별 적응적 학습률 방법

- Adagrad
  - 제곱근 연산이 중요함
  - eps: 스무딩 항으로 일반적으로 $10^{-4}$~$10^{-8}$ 범위에서 설정
  - 딥러닝의 경우 Adagrad의 단조감소하는 학습률은 너무 공격적이어서 너무 일찍 학습을 멈추게 함

```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

# 접근방식4: 파라미터별 적응적 학습률 방법

- RMSprop
  - 매우 효과적이지만 현재는 발표되지 않은 적응적 학습률 방법
  - 이 방법을 사용하는 모든 사람들이 Geoffrey Hinton의 Coursera 강의 6번째 강의의 29번 슬라이드를 인용함
  - Adagrad 방법을 매우 간단히 조정하여 그것의 공격적으로 단조감소하는 학습률을 완화함
  - 특히, 제곱 그래디언트의 이동 평균을 사용함

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

# 접근방식4: 파라미터별 적응적 학습률 방법

- RMSprop
  - decay_rate: 하이퍼파라미터이며 일반적으로 0.9, 0.99, 0.999 값 사용
  - cache 변수는 "leaky"함. 즉, 최근 그래디언트에 더 큰 가중치를 두고 과거의 그래디언트는 점차 잊어버림

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

# 접근방식4: 파라미터별 적응적 학습률 방법

- Adam
  - 최근 제안된 파라미터 업데이트 방식으로 RMSProp과 유사
  - 원시적이고 아마 노이즈가 있는 그래디언트 dx 대신 좀 더 부드러운 버전의 그래디언트 m을 곱함

```
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += - learning_rate * m / (np.sqrt(v) + eps)
```

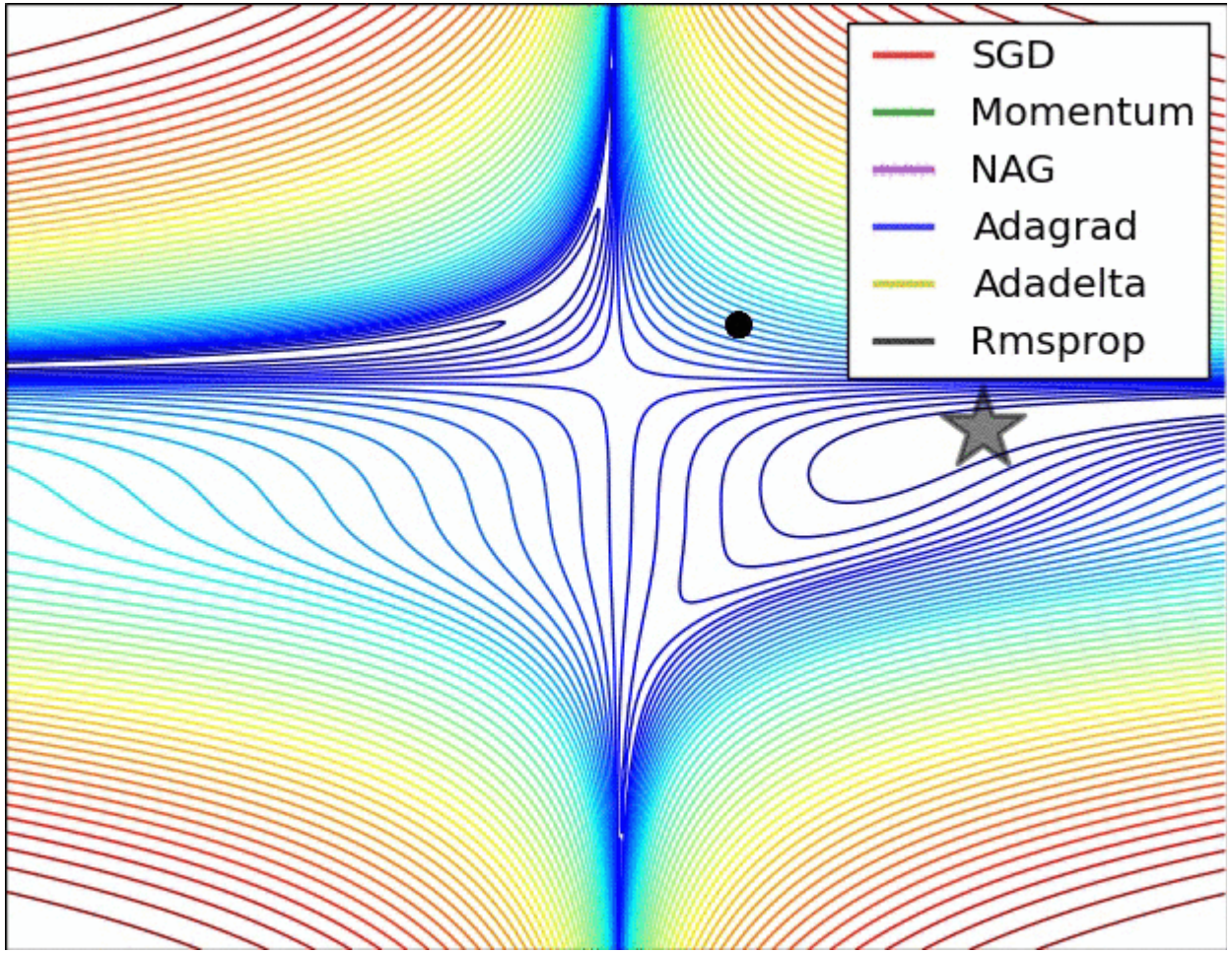
  - 논문에서 추천하는 값은 eps=$10^{-8}$, beta1=0.9, beta2=0.999
  - Adam은 현재 기본적으로 사용되는 알고리즘으로 권장되며 종종 RMSProp보다 더 잘 작동함
  - SGD+Nesterov 모멘텀을 대안으로 시도해보는 것도 종종 가치가 있음

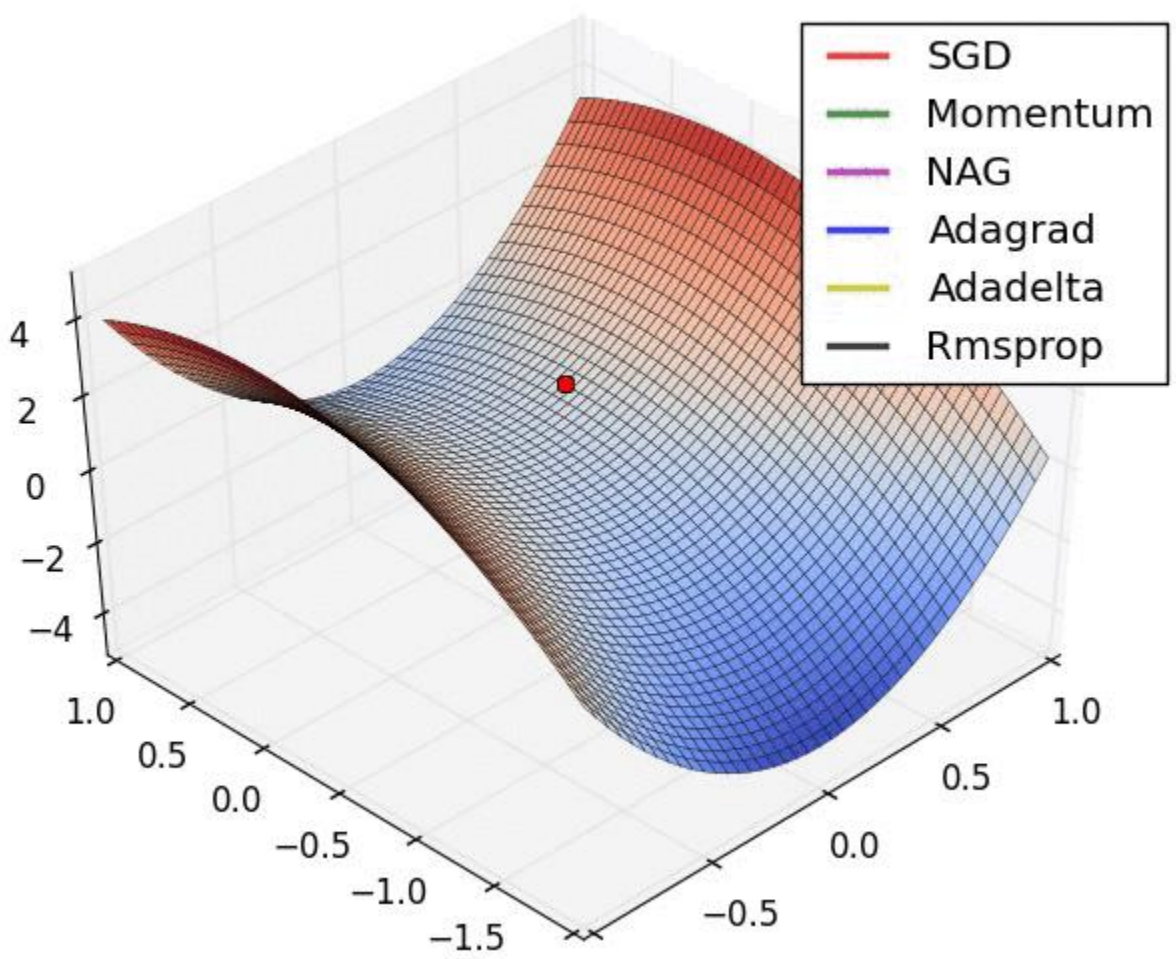
# 접근방식4: 파라미터별 적응적 학습률 방법

- Adam
  - 전체 Adam 파라미터 업데이트에는 편향 수정 메커니즘이 포함됨
  - 이는 처음 벡터 m, v가 모두 초기화되어있어 0으로 편향되어있기 때문에 이를 보상
  - 편향 수정 메커니즘을 사용하면 파라미터 업데이트는 다음과 같음

```
# t is your iteration counter going from 1 to infinity
m = beta1*m + (1-beta1)*dx
mt = m / (1-beta1**t)
v = beta2*v + (1-beta2)*(dx**2)
vt = v / (1-beta2**t)
x += - learning_rate * mt / (np.sqrt(vt) + eps)
```

# 파라미터 업데이트 방식 애니메이션



# 하이퍼파라미터 최적화

- 신경망에서 가장 일반적인 하이퍼파라미터
  - 초기 학습률
  - 학습률 감소 스케쥴 (ex. 감소 상수)
  - 정규화 강도 (ex. L2페널티, 드롭아웃 강도)
- 훈련시키는데 있어 상대적으로 덜 민감한 하이퍼파라미터들도 많이 있음

# 하이퍼파라미터 최적화 팁 - 구현

- 큰 신경망은 일반적으로 훈련하는데 오랜 시간이 걸림
- 지속적으로 랜덤 하이퍼파라미터를 샘플링하고 최적화를 수행하는 워커(worker)를 구현하는 것이 좋음
- 워커란 동시에 여러 작업을 수행하도록 설계된 시스템에서 사용됨
- 다양한 하이퍼파라미터 설정으로 모델을 학습하는 여러 개의 병렬 작업을 수행
- 각 워커는 독립적인 학습 과정을 거치고, 다양한 하이퍼파라미터 조합에 대한 성능을 탐색함

# 하이퍼파라미터 최적화 팁 - 구현

- 훈련 동안 워커는 모든 에폭 이후 검증 정확도를 추적하고, 시간에 따른 손실 등 훈련 통계 정보들을 기록함
- 검증 정확도를 직접 파일 이름에 포함시키면 진행상황을 간단하게 검토하고 정렬하는데 유용함
- 컴퓨팅 클러스터에서 워커를 시작하거나 종료하는 마스터라고 불리는 두 번째 프로그램이 있음
- 워커가 기록한 파일들을 검사하고 훈련 통계를 그래프로 그리는 등의 추가 작업 수행 가능

# 하이퍼파라미터 최적화 팁 - 검증

- 대부분의 경우 적절한 크기의 단일 검증세트를 사용하는 것이 좋음
- 사람들이 "교차검증"했다고 말하는 경우 대부분은 사실은 여전히 단일 검증 세트만을 사용한 것임

# 하이퍼파라미터 최적화 팁 – 하이퍼파라미터 범위

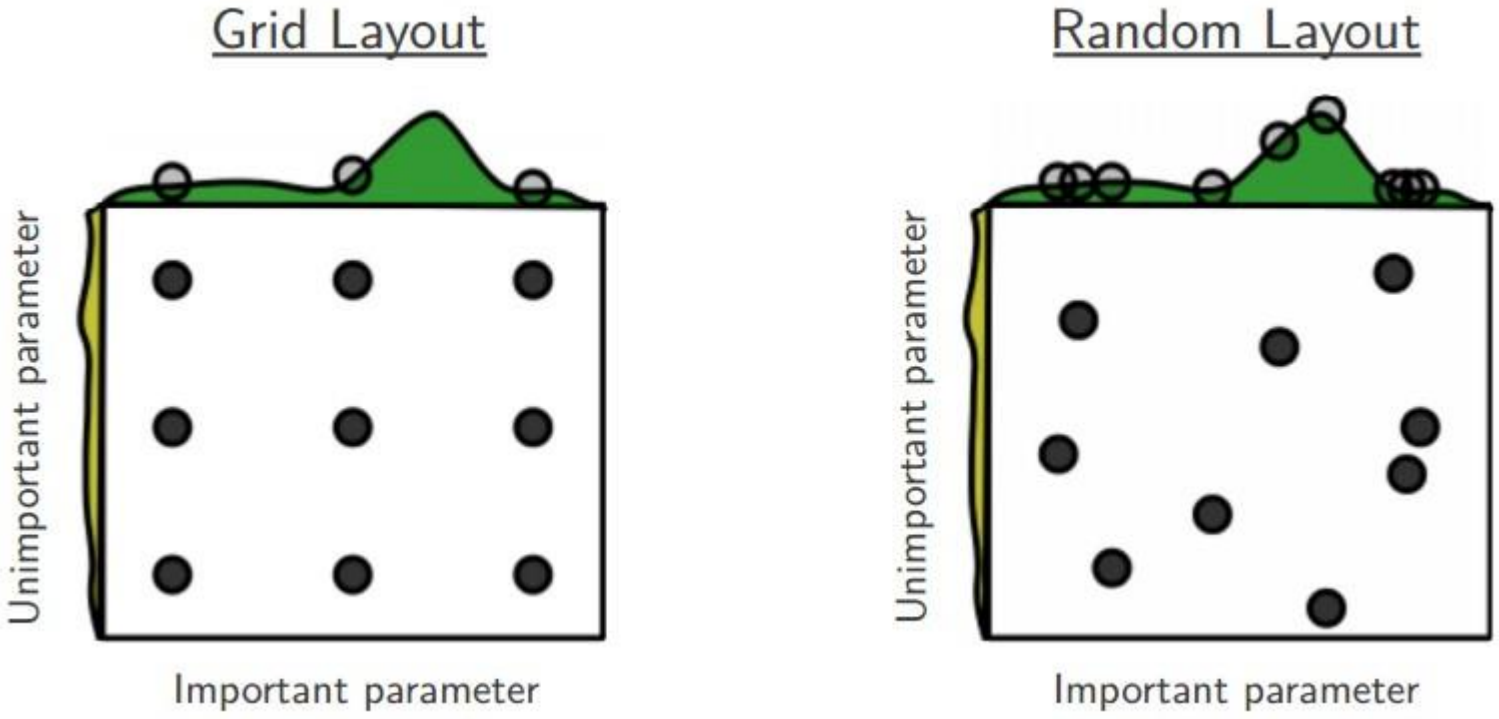
- 하이퍼파라미터는 로그 스케일로 탐색하는 것이 좋음
- 학습률의 전형적인 샘플링 형태

```
learning_rate = 10 ** uniform(-6, 1)
```

- 이와 같은 전략을 정규화 강도에 대해서도 사용해야함
- 이는, 학습률과 정규화 강도가 훈련 동작에 곱셈 효과를 가지기 때문
- 일부 파라미터(ex. 드롭아웃)는 로그 스케일이 아닌 원래의 스케일에서 주로 탐색됨
  - dropout = uniform(0,1)

# 하이퍼파라미터 최적화 팁 – 그리드 검색 보다는 랜덤 검색

- 랜덤 탐색이 그리드에서의 탐색보다 하이퍼파라미터 최적화에 더 효율적
- 많은 경우, 일부 하이퍼파라미터는 다른 하이퍼파라미터보다 훨씬 중요
- 랜덤 탐색 시 중요한 하이퍼파라미터에 대해 더 다양한 값을 탐색하게 되며, 따라서 더 좋은 결과를 얻을 가능성이 높음



# 하이퍼파라미터 최적화 팁 – 최적값이 경계에 있는 경우 주의

- 가끔 하이퍼파라미터를 좋지 않은 범위에서 탐색하는 경우가 있을 수 있음
  - ex) learning_rate = 10 ** uniform(-6, 1)
- 결과를 받게 되면, 최종 학습률이 이 구간의 가장자리에 있는지 확인하는 것이 중요

# 하이퍼파라미터 최적화 팁 – 탐색을 대략적으로부터 세밀하게

- 먼저 대략적인 범위에서 탐색한 후 가장 좋은 결과가 나타나는 곳에 따라 범위 좁히는 것이 도움될 수 있음
  - ex) 10 ** [-6, 1]
- 1에폭 혹은 그보다 적게만 훈련하면서 초기의 대략적인 검색 수행이 유용할 수 있음
- 그 다음 단계는 5 에폭 동안 더 좁은 범위로 검색 수행
- 마지막 단계는 최종 범위에서 더 많은 에폭 동안 상세한 검색 수행

# 모델 앙상블(model ensembles)

- 모델 앙상블은 여러 독립적인 모델을 훈련시키고 테스트 시에 그들의 예측을 평균 내는 것
- 실제로 신경망의 성능 몇 퍼센트 향상 가능
- 앙상블에 포함된 모델의 수가 증가함에 따라 성능은 일반적으로 단조롭게 향상됨
- 앙상블 내의 모델 다양성이 높을 수록 향상은 더욱 두드러짐

# 모델 앙상블(model ensembles)

- 모델 앙상블 유형
  - 동일한 모델, 다른 초기화
  - 교차 검증 중 발견된 상위 몇 개의 모델 선택
  - 단일 모델의 다른 체크포인트를 갖고 앙상블 구성
- 모델 앙상블 단점
  - 테스트 예제를 평가하는데 더 오랜 시간이 걸림