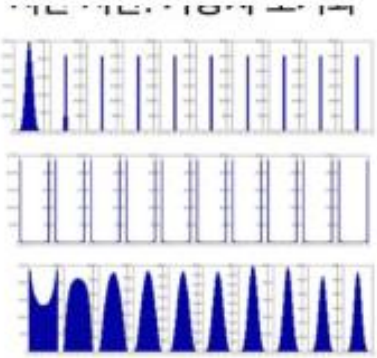
Lecture 7

6강 remind

1. 보통 활성화 함수로 ReLU를 사용.(그냥 일반적으로)

2. W값 초기화(Initualization)



초기화가 너무 작음:
활성이 영으로 감, 경사도 0,
학습 안됨

초기화가 너무 큼:
활성이 포화 (tanh 경우),
경사가 영, 학습 안됨

초기화가 적당함:
모든 계층에서 좋은 활성 분포
학습이 좋게 진행됨

3. 데이터 전처리: 데이터 정규화(normalization)하는 것.

4. batch normalization.

5. babySitting Learning: 학습률에 따른 손실. 정확도에 대한 얘기.

6. 그리드 서치, 랜덤 서치 이야기

-하이퍼파라미터.

1. 최적화(fancier optimization)

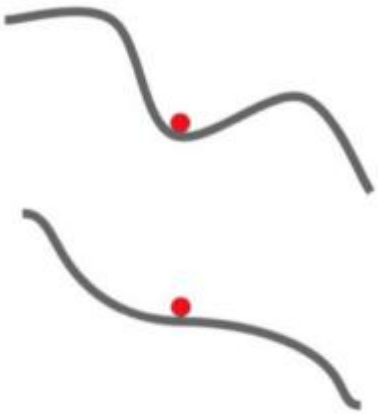
2. 정규화(Regularization)

3. 전이 학습(Transfer Learning)

-최적화(optimization)

SGD(확률적 경사 하강법)

목표 함수까지 가는데, 매우 지그재그 형태. (좋은 성능 안나옴)



문제점:

1.만약 손실함수가 위의 형태를 띄고 있다면(경사가 0인 형태), 멈추게 됨.(기울기가 0이어서)

2.시간도 오래 걸림

SGD + 모멘텀

SGD + 모멘텀 (momentum)

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

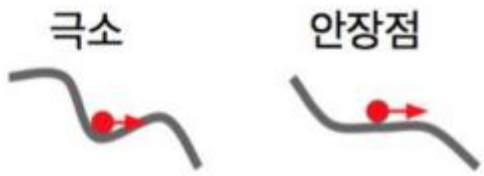
```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

경사의 방향이 아닌, 속도의 방향으로.



이런식으로, 속도의 방향으로 가기 때문에, 극소, 안장점을 만났을 때, 멈추지 않음.

Nesterov

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

짜증남, 우리는 보통 $x_t, \nabla f(x_t)$ 에 대해 업데이트 하고자 함

$\tilde{x}_t = x_t + \rho v_t$ 변수의 변경과 재구성

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$
$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1}$$
$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$

```
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```

네스테로프는 현재 속도와 이전 속도 사이 오류 수정하는 항을 포함

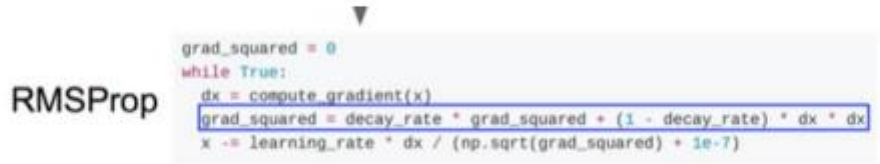
SGD, SGD Momentum보다 속도는 느리지만, 오버슈팅 동작이 최소화 됨.

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

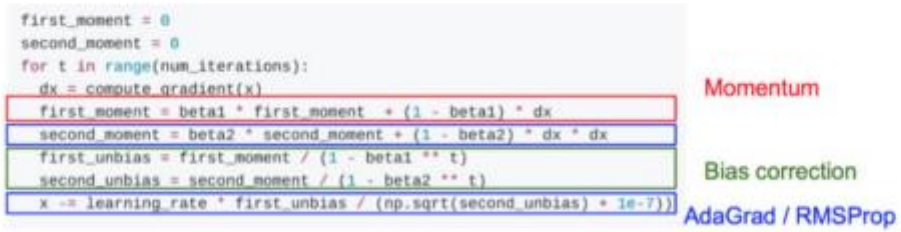
뒤에 나누기 때문에, 만약 grad_squared가 큰 값이라면, 전체 진행이 느려지게 됨.(업데이트 할수록 느려짐)

RMSProp



AdaGrad의 속도 느려지는 부분을 보완.

Adam

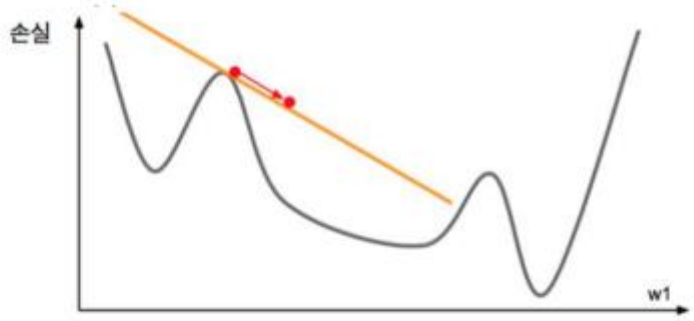


SGD Momentum, Adagrad, RMSProp 합친 형태.

변수 두개 0으로 고정하는 이유가 시작때부터 크게 움직이는 걸 막기 위함.

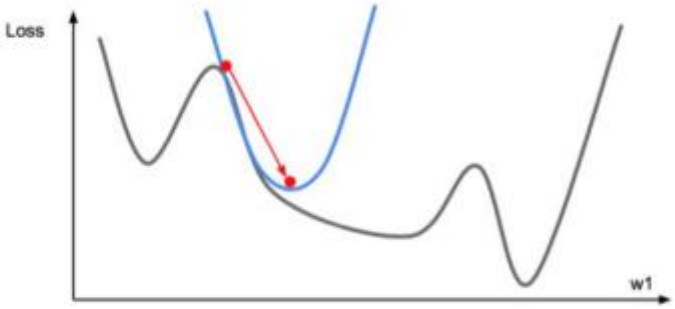
괜찮은 최적화 알고리즘.

1차 최적화



가장 기본적 경사. 미분값.

2차 최적화



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

뉴턴 스텝(다차원 2차 최적화에 사용.)

학습률 존재 하지 않음.

매 걸음마다 최소의 걸음.

문제점:

O(N^2)원소 가짐. 역행렬에 O(N^3) 걸림. ->딥러닝에 안좋음.

BFGS(제한 조건 없는 함수에서 x를 최소화 시키는 것) : 뉴턴 방법 사용. 역행렬 만드는 대신, 1로 업데이트 해서.

L-BFGS(제한된 BFGS): 역행렬 전체 생성 혹은 저장 안함.

full batch에서는 적합. mini-batch에서는 안 좋음.

보통 Adam이 가장 일반적. full-batch 할 수 있음 L-BFGS쓸 것.

모델 앙상블(Model Ensembles):

하나의 모델을, 여러 개로 조화롭게 학습(여러 모델의 예측값의 평균).

훈련 과정에서 여러 snapshot 갖고 있다가 앙상블로 활용.

```
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
```

훈련하는 동안, 계속 앙상블 갖는 것.

-정규화.

드롭아웃(Dropout):

forward pass 할 때, 임의의 뉴런들을 0으로 설정. (계층 몇 개 뺀다)

```
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)
```

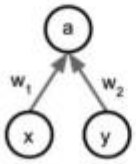
드롭아웃 사용 이유:

1.과적합 해결에 탁월.

2. 모델 내부에서 모델 앙상블을 하고 있음.

드랍아웃: 테스트시

적분을 근사하고 싶음 $$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$



단일 뉴런을 생각해 보기.

테스트시 갖고 있는 것: $E\big[a\big] = w_1 x + w_2 y$

훈련동안 갖고 있는 것:

$$\begin{aligned} E\big[a\big] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

테스트 시 출력 = 훈련시 기대 출력. 이 두개가 같도록 해야 함.

드롭아웃:

```
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
```

테스트시 배율 조정 (scale)

역 드랍아웃

더 흔히 사용됨: "역 드랍아웃(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
```

테스트 시간은 변하지 않음!

드롭아웃에서 predict할 때, '*p'가 되어있는 것을 확인 가능.

시간 절약을 위해, 역 드랍아웃 train때, '/ p' 해줌.

배치 정규화(Batch Normalization):

임의의 미니배치로 정규화

노이즈를 갖는다는 것만 보면 드롭아웃과 비슷.

파라미터값 제어는 불가능.

데이터 증강(data augmentation):

CNN에서 훈련하는 동안 이미지 변형.

ex) 1.고양이 사진 좌우 반전. 구역별 자르고 그걸 모아다가 평균.

2. 밝기 변경. 각 색들을 샘플링. 이걸 다 더하기. -> 좀 어려워서 잘 안 씀

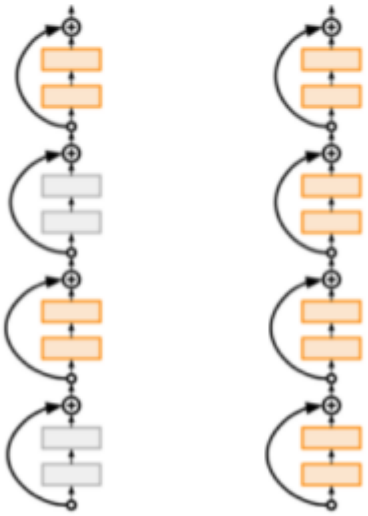
드랍커넥트(DropConnect):

드랍아웃처럼 활성을 0값 말고, 행렬 값 일부를 0으로.

Fractional Max Pooling(작은 맥스 풀링):

필터들을 더 작은 필터들로. (ex. 2x2 풀링을 임의로 1x1, 1x2, 2x1, 2x2로 설정해서 이걸 조합해 풀링)

확률적 깊이(stochastic depth):



훈련중에 계층을 임의로 드랍 하고, 테스트 할 때는 전체 계층 사용.

알려주는 사람이 극찬.

-전이학습(Transfor Learning).

학습된 모델을 다른 작업에 이용하는 것. CNN에 적용.

	매우 비슷한 데이타셋	**매우 다른 데이타셋**
매우 적은 데이타	제일 위 계층에서 선형분류기 사용	곤란한 상황에 처함... 여러 단계로부터 선형 분류기 시도 할 것
꽤 많은 데이타	몇개의 계층을 미세조정	더 많은 계층을 미세조정할 것

이미지 처리에 많이 사용.

Lecture 8

1. CPU와 GPU

2. Deep Learning Frameworks

   2.1 Caffe/ Caffe2

   2.2 Theano / TensorFlow

   2.3 Torch / PyTorch

CPU(central processing unit): 적은 수의 코어로 처리할 수 있는 작업이 많다.

GPU(graphics processing unit): 코어가 많고, 단순한 계산을 매우 빠르게 처리 가능함.

-deep learning Framework

1. Caffe 에서 Caffe2로 발전.

2. Torch에서 PyTorch로 발전.

3. Theano가 TensorFlow 로 발전.

딥러닝 프레임워크의 특징:

1. 대규모 계산 그레프를 쉽게 구축 가능.

2. 계산 그래프에서 기울기 계산 쉽게 가능.

3. GPU라이브러리 활용하여 효율적 GPU 실행 가능.

넘파이로는 GPU를 들릴 수 없어서, TensofFlow, Pytorch 함께 사용.

Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

TensorFlow

```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
y = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
z = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

각각의 형태.

Torch:

- 단점: Lua 언어를 사용한다.
- 단점: 자동 미분 기능이 없다.
- 장점: 안정적인 소프트웨어이다.
- 장점: 많은 기존 코드가 있다.
- 공통점: 빠르다.

PyTorch:

- 장점: Python 언어를 사용한다.
- 장점: 자동 미분 기능이 있다.
- 단점: 상대적으로 새로운 프레임워크이며 아직 변화하는 중이다.
- 단점: 기존 코드 양이 적다.
- 공통점: 빠르다.

Static

Once graph is built, can **serialize** it and run it without the code that built the graph!

Dynamic

Graph building and execution are intertwined, so always need to keep code around

->

Static: Pytorch나 TensorFlow같은 프레임워크 사용하면, 그래프 재사용이 가능.

Dynamic: 계산 그래프의 구성과 실행이 강하게 연결, 그래프 구성 코드를 항상 유지해야 함.