4강: 최적화 - 경사하강법

개요

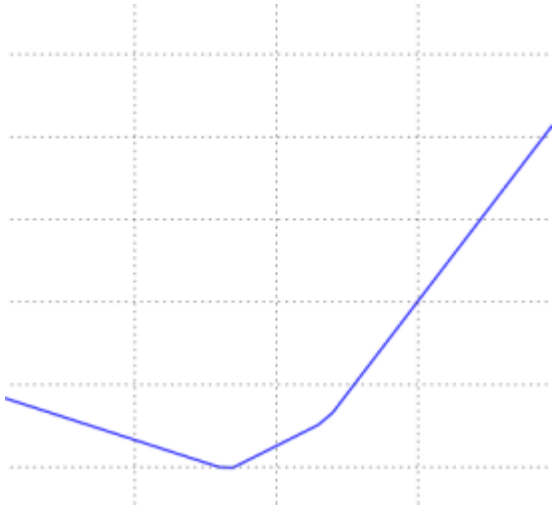
- 지금까지 이미지 분류 작업에서의 두 핵심 요소를 소개함
  - 원본 이미지 픽셀을 클래스 점수로 맵핑하는 (파라미터화된) 점수 함수
    - ex) 선형 함수
  - 훈련 데이터 내의 기본 정답 레이블과 유도된 점수가 얼마나 잘 일치하는지에 기초하여 특정 파라미터 세트의 품질을 측정하는 손실 함수
    - ex) SVM/softmax
- 훈련세트 이미지 x_i에 대한 예측이 정답 레이블 y_i와 일관되게 만들면 손실도 매우 낮을 것임

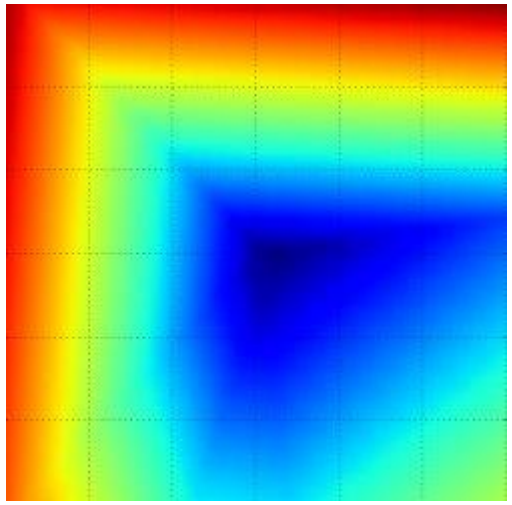
개요

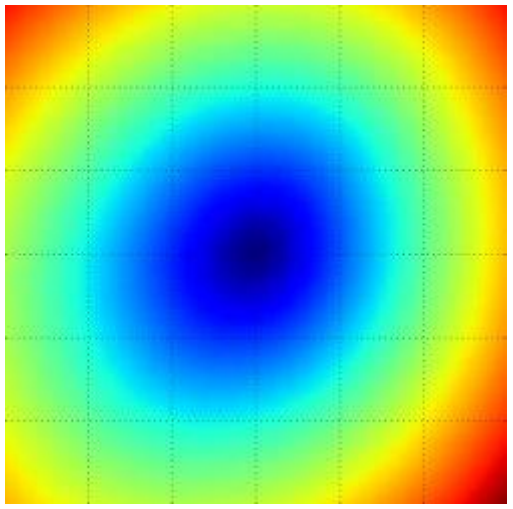
- 세 번째 핵심 구성 요소인 최적화(optimization)에 대해 알아봄
- 최적화는 손실 함수를 최소화하는 파라미터 W 세트를 찾는 과정

손실함수의 시각화

- 손실함수는 매우 높은 차원의 공간에서 정의되므로 시각화가 어려움
- 1차원, 2차원을 따라 자르면서 직관을 얻을 수 있음
  - ex) 다양한 a에 대해 $L(W+aW_1)$ 계산 가능
  - ex) 다양한 a,b에 대해 $L(W+aW_1+bW_2)$ 계산 가능
- 손실 그래프는 piece-wise linear 형태







손실함수의 시각화

$$L_i = \sum_{j \neq y_i} \left[\max(0, w_j^T x_i - w_{y_i}^T x_i + 1) \right]$$

- 간단하게 훈련세트 이미지가 3개이고 1차원이며, 클래스가 3개인 경우를 고려

$$L_0 = \max(0, w_1^T x_0 - w_0^T x_0 + 1) + \max(0, w_2^T x_0 - w_0^T x_0 + 1)$$

$$L_1 = \max(0, w_0^T x_1 - w_1^T x_1 + 1) + \max(0, w_2^T x_1 - w_1^T x_1 + 1)$$

$$L_2 = \max(0, w_0^T x_2 - w_2^T x_2 + 1) + \max(0, w_1^T x_2 - w_2^T x_2 + 1)$$

$$L = (L_0 + L_1 + L_2)/3$$

손실함수의 시각화

- 손실 식을 다음과 같이 ω_0에 대한 함수로 생각

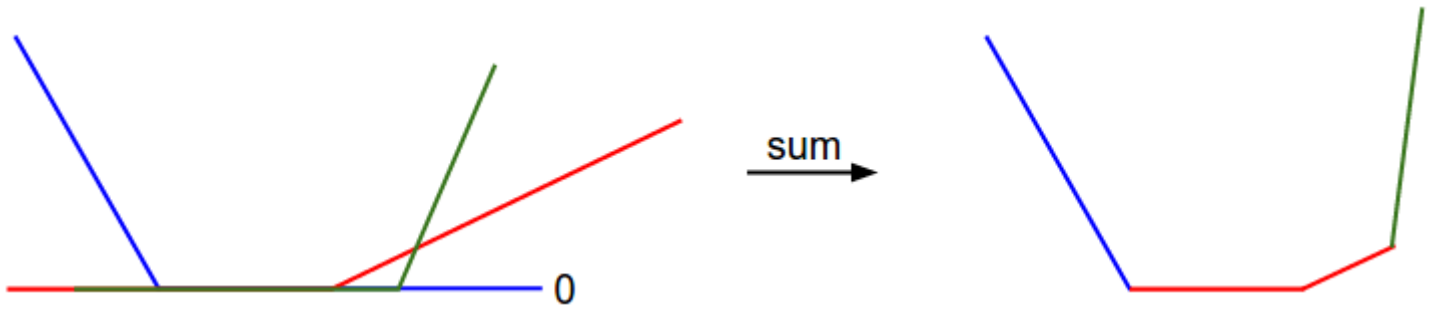
$$L_0 = \max(0, w_1 x_0 - w_0 x_0 + 1) + \max(0, w_2 x_0 - w_0 x_0 + 1)$$

$$L_1 = \max(0, w_0 x_0 - w_1 x_1 + 1) + C_1$$

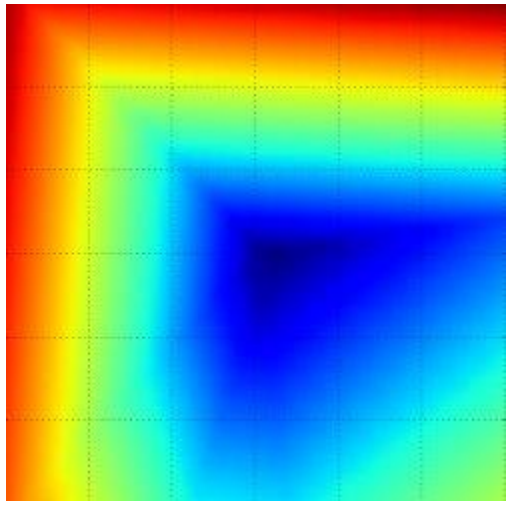
$$L_0 = \max(0, w_0 x_2 - w_2 x_2 + 1) + C_2$$

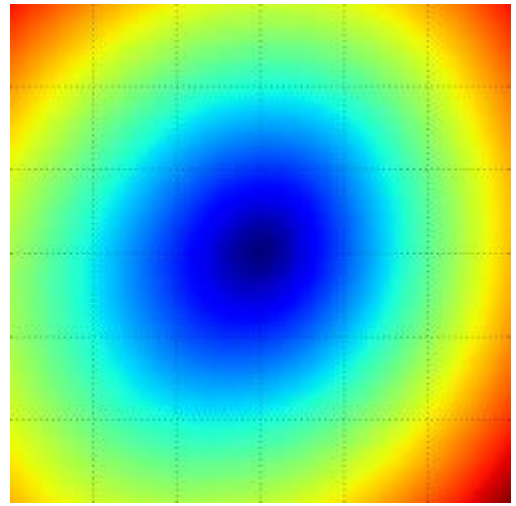
$$L = (L_0 + L_1 + L_2)/3$$

- Piece-wise linear 형태가 됨



손실함수의 시각화





- 손실 함수를 최소화하는 가중치 W를 찾는 과정은 최적화 문제에 해당
- 점수 함수가 선형 함수인 경우 볼록 함수에 대한 최적화 문제
- 점수 함수를 신경망으로 확장하면 비볼록(non-convex) 함수가 됨

손실함수의 시각화

- 손실 함수의 꺾인 부분들은 기술적으로 손실 함수를 미분 불가능하게 만듦
- 이 경우 기울기 대신 부분 기울기(subgradient)를 사용
- 본 수업에서는 부분 기울기와 기울기를 교환하여 사용

최적화

- 최적화의 목표는 손실 함수를 최소화하는 W를 찾는 것
- 신경망의 경우 잘 알려져있는 볼록 문제에 대한 최적화 기법을 사용할 수 없음
- 따라서, 일반적으로 비볼록 함수에 대해 실행 가능한 최적화 방법을 소개

전략1: 랜덤 검색

- W값으로 다양한 랜덤 가중치 값을 시도한 후 가장 잘 작동하는 W 값을 선택

```
# assume X_train is the data where each column is an example (e.g. 3073 x
# assume Y_train are the labels (e.g. 1D array of 50,000)
# assume the function L evaluates the loss function

bestloss = float("inf") # Python assigns the highest possible float value
for num in range(1000):
  W = np.random.randn(10, 3073) * 0.0001 # generate random parameters
  loss = L(X_train, Y_train, W) # get the loss over the entire training se
  if loss < bestloss: # keep track of the best solution
    bestloss = loss
    bestW = W
  print 'in attempt %d the loss was %f, best %f' % (num, loss, bestloss)

# prints:
# in attempt 0 the loss was 9.401632, best 9.401632
# in attempt 1 the loss was 8.959668, best 8.959668
# in attempt 2 the loss was 9.044034, best 8.959668
# in attempt 3 the loss was 9.278948, best 8.959668
# in attempt 4 the loss was 8.857370, best 8.857370
# in attempt 5 the loss was 8.943151, best 8.857370
# in attempt 6 the loss was 8.605604, best 8.605604
# ... (truncated: continues for 1000 lines)
```

전략1: 랜덤 검색

- 찾은 최적의 가중치 W를 사용하여 테스트 세트에 대해 평가

```
# Assume X_test is [3073 x 10000], Y_test [10000 x 1]
scores = Wbest.dot(Xte_cols) # 10 x 10000, the class scores for all test e
# find the index with max score in each column (the predicted class)
Yte_predict = np.argmax(scores, axis = 0)
# and calculate accuracy (fraction of predictions that are correct)
np.mean(Yte_predict == Yte)
# returns 0.1555
```

아이디어

- 랜덤 검색보다 더 잘하기 위한 핵심 아이디어는 특정 가중치 W를 약간 개선하는 문제는 상당히 덜 어렵다는 것
- 랜덤한 W로 시작하고, 그것을 반복적으로 세밀화하여 매번 약간씩 개선하는 접근법을 취함
- 눈이 가려진 등산객 비유
  - 눈가리개를 쓴 채로 산악 지형을 등산하며 바닥에 도달하는 경우 생각

전략2: 랜덤 로컬 검색

- 랜덤한 방향으로 한 발 뻗어보고, 내리막길로 이어지는 경우에만 걸음을 내딛음
- 구체적으로, 랜덤한 W로 시작하여, 랜덤한 변동 dW를 더한 후 W+dW에서의 손실이 더 낮으면 업데이트를 진행
- 테스트 세트 분류 정확도 21.4%를 달성
- 더 나은 결과이지만 여전히 낭비적임

전략3: 그래디언트를 따라감

- 좋은 방향을 무작위로 찾을 필요 없음
- 수학적으로 가장 가파른 하강 방향을 최적의 방향으로 계산
- 이는 손실 함수의 그래디언트와 관련이 있음

그래디언트(gradient)

- 일반 함수에서의 기울기

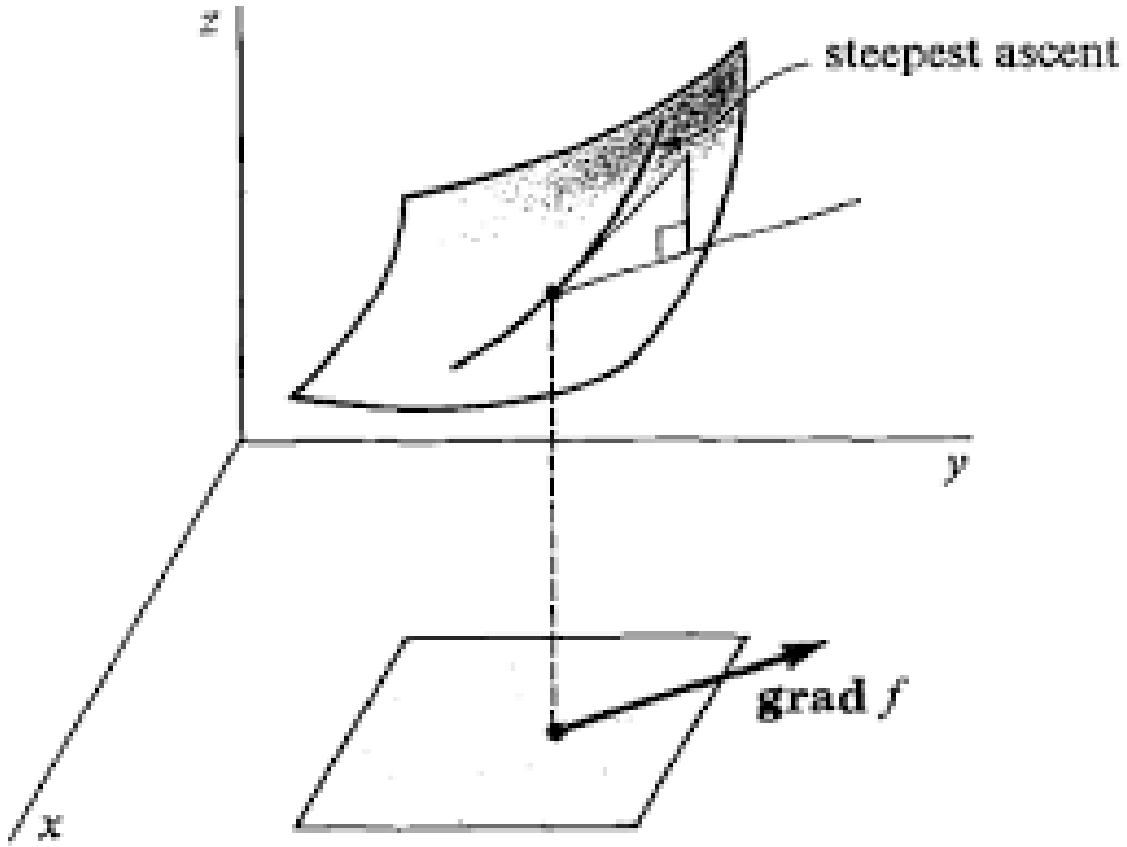
$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- 다변수 함수 $f = f(x_1, x_2, \cdots, x_n)$에서의 그래디언트(gradient)

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

그래디언트(gradient)

- $z = f(x, y)$ 에 대한 그래디언트 gradf는 가장 가파르게 올라가는 방향



그래디언트(gradient)

- 그래디언트 계산 방법
  - 수치적 방법: 느리고 대략적이지만 쉬움
  - 해석적 방법: 빠르고 정확하지만 미적분학이 필요하고 오류 발생 쉬움

그래디언트 수치적 계산 방법

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- 위 식을 활용하면 그래디언트를 수치적으로 계산할 수 있음
- 위 식을 각 차원에 대해 적용해 편미분(ex, $\frac{\partial f}{\partial x_1}$, $\frac{\partial f}{\partial x_2}$ 등)을 계산

```python
def eval_numerical_gradient(f, x):
  """
  a naive implementation of numerical gradient of f at x
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """

  fx = f(x) # evaluate function value at original point
  grad = np.zeros(x.shape)
  h = 0.00001

  # iterate over all indexes in x
  it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
  while not it.finished:

    # evaluate function at x+h
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h # increment by h
    fxh = f(x) # evalute f(x + h)
    x[ix] = old_value # restore to previous value (very important!)

    # compute the partial derivative
    grad[ix] = (fxh - fx) / h # the slope
    it.iternext() # step to next dimension

  return grad
```

nditer 객체

- np.nditer는 넘파이에서 제공하는 객체이며, 다차원 배열을 효율적으로 순회(iterate)하기 위한 반복자(iterator)

```
import numpy as np

arr = np.array([[1, 2], [3, 4]])
for x in np.nditer(arr):
    print(x)
```

```
1
2
3
4
```

```
it = np.nditer(arr, flags=['multi_index'])
for x in it:
    print(it.multi_index, x)
```

```
(0, 0) 1
(0, 1) 2
(1, 0) 3
(1, 1) 4
```

nditer 객체

```
it = np.nditer(arr, op_flags=['readwrite'])
for x in it:
    x[...] = x*2  # 각 요소를 2배로 만듭니다.
```

그래디언트 수치적 계산 방법

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

```python
def eval_numerical_gradient(f, x):
  """
  a naive implementation of numerical gradient of f at x
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """

  fx = f(x) # evaluate function value at original point
  grad = np.zeros(x.shape)
  h = 0.00001

  # iterate over all indexes in x
  it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
  while not it.finished:

    # evaluate function at x+h
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h # increment by h
    fxh = f(x) # evalute f(x + h)
    x[ix] = old_value # restore to previous value (very important!)

    # compute the partial derivative
    grad[ix] = (fxh - fx) / h # the slope
    it.iternext() # step to next dimension

  return grad
```

그래디언트 계산 시 실제적인 고려사항

- 극한을 사용하는 대신 매우 작은 h값을 사용하는 것으로 종종 충분 (ex. 10^{-5})
- 중앙 차분 공식을 사용하여 수치 그래디언트를 계산하는 것이 더 잘 작동하는 경우가 많음

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

그래디언트 계산 코드

- 가중치 공간에서의 임의의 지점에서 CIFAR-10 손실 함수의 그래디언트 계산

```
# to use the generic code above we want a function that takes a single argument
# (the weights in our case) so we close over X_train and Y_train
def CIFAR10_loss_fun(W):
  return L(X_train, Y_train, W)

W = np.random.rand(10, 3073) * 0.001 # random weight vector
df = eval_numerical_gradient(CIFAR10_loss_fun, W) # get the gradient
```

가중치 업데이트 코드

```
loss_original = CIFAR10_loss_fun(W) # the original loss
print 'original loss: %f' % (loss_original, )

# lets see the effect of multiple step sizes
for step_size_log in [-10, -9, -8, -7, -6, -5,-4,-3,-2,-1]:
  step_size = 10 ** step_size_log
  W_new = W - step_size * df # new position in the weight space
  loss_new = CIFAR10_loss_fun(W_new)
  print 'for step size %f new loss: %f' % (step_size, loss_new)

# prints:
# original loss: 2.200718
# for step size 1.000000e-10 new loss: 2.200652
# for step size 1.000000e-09 new loss: 2.200057
# for step size 1.000000e-08 new loss: 2.194116
# for step size 1.000000e-07 new loss: 2.135493
# for step size 1.000000e-06 new loss: 1.647802
# for step size 1.000000e-05 new loss: 2.844355
# for step size 1.000000e-04 new loss: 25.558142
# for step size 1.000000e-03 new loss: 254.086573
# for step size 1.000000e-02 new loss: 2539.370888
# for step size 1.000000e-01 new loss: 25392.214036
```

스텝 사이즈의 효과

- 그래디언트는 얼마나 멀리 나아가야하는지는 알려주지 않음
- 이를 결정하는 파라미터는 스텝 사이즈(혹은 학습률)
- 등산객 비유에서 조심스럽게 움직이면 지속적이지만 매우 작은 진전을 얻음
- 큰 스텝으로 움직이는 것이 항상 효과가 있지는 않으며, 어느 정도를 넘어서면 과도한 걸음으로 판단되어 더 높은 손실을 가져옴

효율성 문제

- 수치적 그래디언트 계산 시간은 파라미터 수에 비례
- 기존 예시에서 총 30730개의 파라미터를 가지며, 단 한번의 파라미터 업데이트를 위해 손실 함수를 30731번 계산해야함
- 수천만개 이상의 파라미터를 가진 현대의 신경망에서는 훨씬 더 문제가 악화됨

그래디언트 분석적 계산 방법

- 수치적 그래디언트 계산은 근사치이며 계산 비용이 크다는 단점을 가짐
- 미적분학을 사용하여 분석적으로도 그래디언트 계산 가능
- 이 분석적 방법은 근사치가 없이 직접적인 수식을 얻을 수 있으며, 매우 빠르게 계산 가능
- 실제로는 분석적 그래디언트를 계산하고 이를 수치적 그래디언트와 비교하여 구현의 정확성을 확인하는 그래디언트 체크(gradient check) 과정을 흔히 거침

SVM 손실 함수에 대한 그래디언트

- 단일 훈련세트 이미지에 대한 SVM 손실 함수 식

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

- ω_{y_i}에 대한 그래디언트

$$\frac{\partial L_i}{\partial w_{y_i}} = -(\sum_{\ell \neq y_i} 1(w_\ell^T x_i - w_{y_i}^T x_i + \Delta > 0))x_i$$

SVM 손실 함수에 대한 그래디언트

- 1. $f = x^T y$ 형태에서 $\nabla_x f$는?
  - $\nabla_x f = y$
- 2. $f = w_j^T x_i - w_{y_i}^T x_i + \Delta$ 형태에서 $\nabla_{w_{y_i}} f$는?
  - $\nabla_{w_{y_i}} f = -x_i$ (w_{y_i}외에는 상수 취급하고 미분)
- 3. $f = \max(0, w_\ell^T x_i - w_{y_i}^T x_i + \Delta)$형태에서 $\nabla_{w_{y_i}} f$는?
  - $\nabla_{w_{y_i}} f = -x_i \cdot 1(w_\ell^T x_i - w_{y_i}^T x_i + \Delta > 0)$
- 4. 위 3번에서의 f를 y_i가 아닌 모든 ℓ에 대해 더한 값인 L_i에서 $\nabla_{w_{y_i}} L_i$는?

- 1. $f = x^T y$ 형태에서 $\nabla_x f$는?
  - $\nabla_x f = y$
- 2. $f = w_j^T x_i - w_{y_i}^T x_i + \Delta$ 형태에서 $\nabla_{w_{y_i}} f$는?
  - $\nabla_{w_{y_i}} f = -x_i$ (w_{y_i}외에는 상수 취급하고 미분)
- 3. $f = \max(0, w_\ell^T x_i - w_{y_i}^T x_i + \Delta)$형태에서 $\nabla_{w_{y_i}} f$는?
  - $\nabla_{w_{y_i}} f = -x_i \cdot 1(w_\ell^T x_i - w_{y_i}^T x_i + \Delta > 0)$
- 4. 위 3번에서의 f를 y_i가 아닌 모든 ℓ에 대해 더한 값인 L_i에서 $\nabla_{w_{y_i}} L_i$는?

SVM 손실 함수에 대한 그래디언트

- $\omega_j\ (j \neq y_i)$에 대한 그래디언트

$$\frac{\partial L_i}{\partial w_j} = 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0)x_i$$

- 가중치 행렬 W에 대한 그래디언트

$$W = \begin{matrix} 3073 \\ \left[\begin{matrix} w_0^T \\ w_1^T \\ \vdots \end{matrix}\right] \end{matrix}^{10}$$

$$\frac{\partial L_i}{\partial W} = \nabla_W L_i = \begin{matrix} 3073 \\ \left[\begin{matrix} (\frac{\partial L_i}{\partial w_0})^T \\ (\frac{\partial L_i}{\partial w_1})^T \\ \vdots \end{matrix}\right] \end{matrix}^{10}$$

SVM 손실 함수에 대한 그래디언트

- 가중치 행렬 W에 대한 전체 손실 L의 그래디언트

$$L = \frac{1}{N} \sum L_i$$

$$\frac{\partial L}{\partial W} = \frac{1}{N} \sum_i \frac{\partial L_i}{\partial W}$$

소프트맥스 분류기 손실 함수에 대한 그래디언트

- 단일 훈련세트 이미지에 대한 소프트맥스 손실 함수 식

$$L_i = -f_{y_i} + \log \sum_{\ell} e^{f_\ell}$$

$$f_\ell = w_\ell^T \cdot x_i$$

- 점수 값에 대한 그래디언트

$$\frac{\partial L_i}{\partial f_j} = \frac{e^{f_j}}{\sum_{\ell} e^{f_\ell}}, \qquad j \neq y_i$$

$$\frac{\partial L_i}{\partial f_j} = \frac{e^{f_j}}{\sum_{\ell} e^{f_\ell}} - 1, \quad j = y_i$$

연쇄 법칙(chain rule)

- 기본 연쇄 법칙

$$z = g(y), y = f(x)$$

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

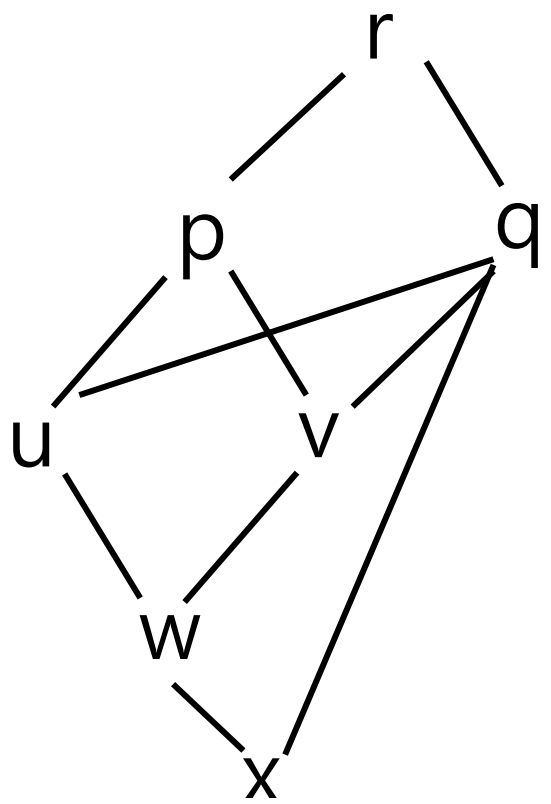
- 길이가 긴 경우

$$v = p(u), u = h(z), z = g(y), y = f(x)$$

$$\frac{dv}{dx} = \frac{dv}{du} \cdot \frac{du}{dz} \cdot \frac{dz}{dy} \cdot \frac{dy}{dx}$$

연쇄 법칙(chain rule)

- 두 변수 사이의 경로가 다양한 경우
  - 두 변수 사이의 모든 가능한 경로들에 대한 미분 값들 곱을 더해주어야함



$$\frac{\partial r}{\partial x} = \frac{\partial r}{\partial p} \cdot \frac{\partial p}{\partial u} \cdot \frac{\partial u}{\partial w} \cdot \frac{\partial w}{\partial x} + \frac{\partial r}{\partial p} \cdot \frac{\partial p}{\partial v} \cdot \frac{\partial v}{\partial w} \cdot \frac{\partial w}{\partial x} + \frac{\partial r}{\partial q} \cdot \frac{\partial q}{\partial x}$$

연쇄 법칙(chain rule)

- 스칼라가 아닌 벡터(혹은 행렬)인 변수가 있는 경우
  - 일반적으로는 자코비안 행렬을 사용하여 연쇄법칙을 서술함
  - $z = g(y), y = f(x)$ (z: 스칼라, x: $\mathbb{R}^n$ 벡터, y: $\mathbb{R}^m$ 벡터)
  - $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot J$ ($\frac{\partial z}{\partial y}$: 크기 m인 행벡터, $\frac{\partial z}{\partial x}$: 크기 n인 행벡터)
  - J는 y를 x에 대해 미분한 자코비안 행렬

$$J(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

소프트맥스 분류기 손실 함수에 대한 그래디언트

- 점수 값에 대한 그래디언트

$$\frac{\partial L_i}{\partial f_j} = \frac{e^{f_j}}{\sum_\ell e^{f_\ell}}, \qquad j \neq y_i$$

$$\frac{\partial L_i}{\partial f_j} = \frac{e^{f_j}}{\sum_\ell e^{f_\ell}} - 1, \quad j = y_i$$

$$f_\ell = w_\ell^T \cdot x_i$$

- 점수 값을 w_ℓ에 대해 미분

$$\frac{\partial f_\ell}{\partial w_\ell} = x_i$$

소프트맥스 분류기 손실 함수에 대한 그래디언트

- 연쇄법칙 적용

$$\frac{\partial L_i}{\partial w_j} = \frac{\partial L_i}{\partial f_j} \cdot \frac{\partial f_j}{\partial w_j} = \left(\frac{e^{f_j}}{\sum_\ell e^{f_\ell}}\right) \cdot x_i, \qquad j \neq y_i$$

$$\frac{\partial L_i}{\partial w_j} = \frac{\partial L_i}{\partial f_j} \cdot \frac{\partial f_j}{\partial w_j} = \left(\frac{e^{f_j}}{\sum_\ell e^{f_\ell}} - 1\right) \cdot x_i, \quad j = y_i$$

- 하나의 식으로 표현

$$\frac{\partial L_i}{\partial w_j} = \frac{\partial L_i}{\partial f_j} \cdot \frac{\partial f_j}{\partial w_j} = \left(\frac{e^{f_j}}{\sum_\ell e^{f_\ell}} - 1(j = y_i)\right) \cdot x_i$$

경사하강법(gradient descent)

- 경사 하강법은 그래디언트를 반복적으로 계산하고 가중치 W 업데이트를 수행하는 절차
- 경사하강법의 바닐라(vanila) 버전 코드

```
# Vanilla Gradient Descent

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

미니-배치 경사하강법

- 대규모 응용 프로그램에서는 수백만개의 이미지를 훈련세트로 사용하는 경우가 존재
- 단 한번의 가중치 업데이트를 위해 전체 훈련세트에 대한 손실 함수를 계산하는 것은 낭비일 수 있음
- 이를 해결하기 위해 훈련세트 이미지들의 배치(batch)에 대해 그래디언트를 계산

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

미니-배치 경사하강법

- 예) 훈련세트 이미지가 128*4=512개인 경우 크기 128인 배치 4개로 나누어질 수 있음
- 순서대로 각 배치에 대해 그래디언트를 계산한 후 파라미터 업데이트

$$\frac{\partial L}{\partial W} = \frac{1}{N} \sum_i \frac{\partial L_i}{\partial W}$$

$$\frac{\partial L}{\partial W} = \frac{1}{n_b} \sum_i \frac{\partial L_i}{\partial W}$$

미니-배치 경사하강법

- 미니배치 경사하강법이 잘 작동하는 이유는 훈련세트 이미지들이 서로 연관되어있기 때문
- ILSVRC
  - 매년 개최되는 이미지넷 인식 대회
  - 훈련세트는 120만개의 이미지를 가짐
- 만약, 120만개의 이미지가 사실 1000개의 고유한 이미지의 복사본들로 구성되었다고 가정하면, 전체 이미지에 대한 손실 평균은 1000개의 이미지(배치)에 대한 손실 평균과 동일
- 미니배치 그래디언트는 전체에 대한 그래디언트의 좋은 근사값

미니-배치 경사하강법

- 미니 배치 크기는 하이퍼파라미터이며, 교차 검증하는 것은 흔치 않음
- 일반적으로 16, 32, 64, 128, 256 등 2의 거듭제곱으로 설정됨
- 이는 벡터화된 연산 구현이 2의 거듭제곱인 경우 더 빠르게 작동하기 때문

확률적 경사 하강법(stochastic gradient descent)

- 확률적 경사하강법(stochastic gradient descent, SGD)는 한번에 단 한 개의 훈련 이미지에 대해서만 그래디언트를 계산하는 방식
- 미니배치 경사하강법을 언급할 때도 종종 확률적 경사하강법(SGD) 용어를 사용함

미니배치의 선택

- 순차적 선택
  - 훈련세트를 여러 부분으로 나누어 순차적으로 각 배치를 선택하는 방법
  - 모든 이미지를 사용한 학습이 가능하지만, 데이터의 순서가 학습에 영향을 미침
- 랜덤 선택
  - 매 반복마다 훈련세트에서 데이터를 샘플링하여 배치를 생성
  - 데이터의 다양성을 확보하지만 일부 이미지가 누락될 수 있음
- 실제로는 종종 각 반복에서 훈련 데이터셋을 랜덤하게 셔플링한 후 작은 배치들로 나누고, 순차적으로 각 배치를 사용하여 파라미터를 업데이트함