CS231n Convolutional Neural Networks for Visual Recognition

3.最適化:確率勾配降下法

序論

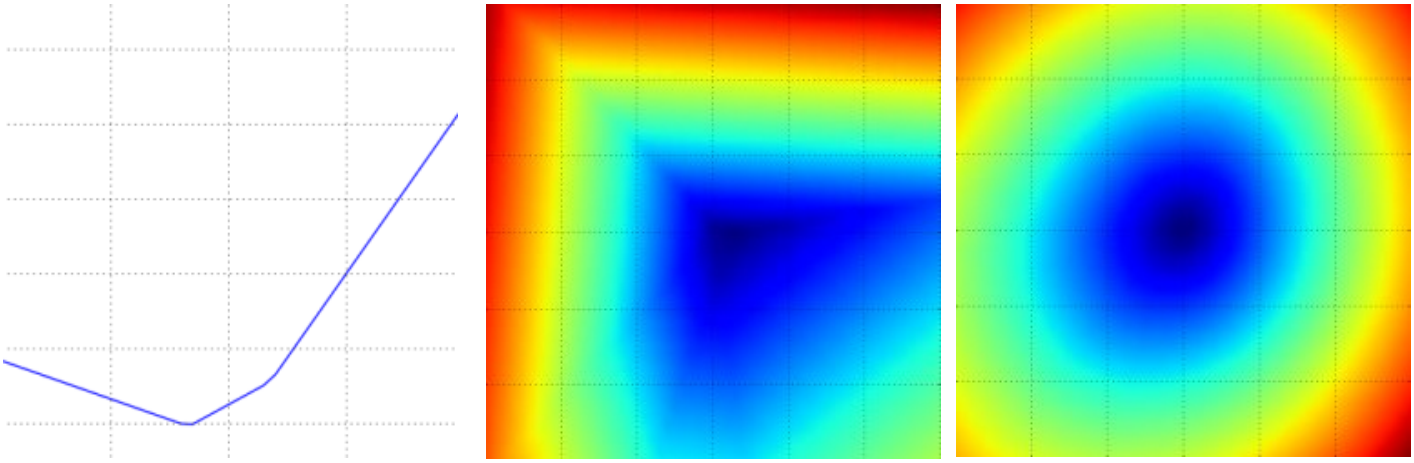
前のセクションでは，画像分類タスクの文脈で2つの重要な要素を紹介しました.
・生の画像のピクセルを分類スコアにマッピングする（パラメータ化された）スコア関数（例：線形関数).
・誘導されたスコアが訓練データの正解ラベル(教師データ) とどれだけ一致しているかに基づいて，特定のパラメータセットの品質を測定する損失関数.
これには多くの方法やバージョンがあることがわかりました（例：ソフトマックス/SVM）．具体的には，線形関数が$f(x_i, W) = Wx_i$として定式化され，開発したSVMは

$$L = \frac{1}{N}\sum_i \sum_{j \neq y_i} [\max(0, f(x_i; W)_j - f(x_i; W)y_i) + 1)] + \alpha R(W)$$

例x_iの予測値が基底真理ラベルy_iと一致するようなパラメータWの設定では，損失Lが非常に小さくなることがわかりました．ここで，3番目で最後の重要な要素を紹介します．**最適化**です．最適化とは，損失関数を最小化するパラメータWのセットを見つけるプロセスです.
ちょっと先取り：これら3つのコアコンポーネントがどのように相互作用するかを理解したら最初の構成要素（パラメータ化された関数の写像）を再検討し，線形写像よりもはるかに複雑な関数に拡張していきます．最初にニューラルネットワーク全体，次に畳み込みニューラルネットワークにです．損失関数と最適化プロセスは相対的にみて変更されません.

損失関数の可視化

この授業で見ていく損失関数は，通常，非常に高次元の空間（例えば，CIFAR-10では，線形分類器の重み行列のサイズは[10 x 3073]で，合計30,730個のパラメータ）で定義されているため，可視化するのが難しいです．しかし，線（1次元）や平面（2次元）に沿って高次元空間をスライスすることで，1つの空間についていくつかの感覚的理解を得ることができます． 例えば，ランダムな重み行列W（空間内の1点に対応）を生成し，線に沿って進み，途中で損失関数の値を記録することができます．すなわち，ランダムな方向W_1を生成し，異なるaの値について$L(W + aW_1)$を評価することにより，この方向に沿った損失を計算することができ,この処理により，aの値をx軸，損失関数の値をy軸とする単純なプロットが生成されます．また，a, bを変化させたときの損失$L(W + aW_1 + bW_2)$を評価することで，2次元でも同じ手順を行うことができます．プロットでは，a,bはx軸とy軸に対応し，損失関数の値を色の変化で表現しています.



CIFAR-10におけるMulticlass SVM(正則化なし)の1つの例(左,中)と100個の例(右)の損失関数の表面図．左: a を変化させただけの一次元の損失． 中，右: 二次元の損失スライス，青 = 低損失，赤 = 高損失．損失関数の断片的な線形構造に注目してください．複数の例の損失は平均値と結合しているので，右のボウル形状は多くの区分線形関数のボウル（中央のようなもの）の平均値です．

損失関数の断片的な線形構造は，数学的に調査することで説明できます．一つの例を挙げると

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

式から明らかなように，各例のデータ損失は，Wの(max(0,−)関数によるゼロ閾値の)線形関数の和である．さらに，Wの各行(すなわち，W_j)の前に正の符号を持つこともあれば (例題の間違った分類に対応する場合)，負の符号を持つこともある (例題の正しい分類に対応する場合)．これをより明確にするために，3つの1次元の点と3つの分類を含む単純なデータセットを考えてみましょう．完全なSVMの損失(正則化なし)は次のようになります．

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

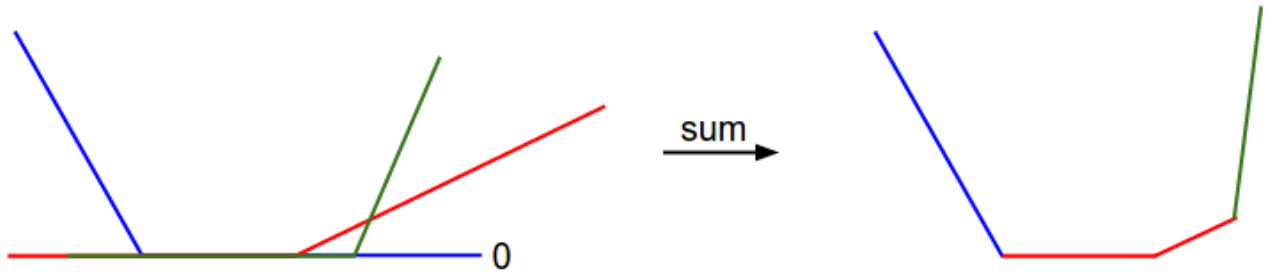
$$L_1 = \max(0, w_0^T x_1 - w_1^T x_1 + 1) + \max(0, w_2^T x_1 - w_1^T x_1 + 1)$$

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

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

これらの例は 1 次元なので，データx_iと重み w_jは数値である．例えば，w_0を見てみると，上のいくつかの項はw_0の線形関数であり，それぞれがゼロでクランプされています．これを次のように可視化することができます

(次ページに続きます)



データの損失を示す1次元図．x軸は重みで，y軸は損失です．データ損失は複数の項の和であり，それぞれは特定の重みから独立しているか，またはゼロで閾値が設定されている線形関数です．完全なSVMデータ損失は，この形状の30,730次元版である．

余談ですが，お椀型の外観から，SVM のコスト関数が凸関数の一例であることがお分かりになったかもしれません．このようなタイプの関数を効率的に最小化するための文献は数多く存在します．スコア関数fをニューラルネットワークに拡張すると，目的関数は非凸型になり，上の可視化ではボウルではなく，複雑で凹凸な地形が特徴的になります．

微分不可能な損失関数．技術的な注意点として，(max演算による)損失関数の変曲点が，技術的に損失関数を微分不可能にしています．しかし，劣勾配が一般的に使われています．この授業では，サブグラジエント(劣勾配)とグラジエント(勾配)という用語を同義に使います．

最適化

最適化の目標は，損失関数を最小化するWを見つけることです．ここでは，損失関数を最適化するためのアプローチを，モチベーションを高めながらゆっくりと開発していきます．この授業に参加したことがある人にとっては，このセクションは凸問題であるため，奇妙に見えるかもしれませんが，私たちの目標は，最終的には，凸最適化の文献で開発されたツールを簡単には使用できないニューラルネットワークを最適化することであることを覚えておいてください．

作戦その1．非常に悪い解決策：ランダム探索

与えられたパラメータWのセットがどれだけ良いかをチェックするのは非常に簡単なので，最初に思いつくかもしれない（非常に悪い）アイデアは，単に多くの異なるランダムな重みを試してみて，どれが最も良く機能するかを追跡することです．この手順は次のようになります．

(次ページに続きます)

```
# assume X_train is the data where each column is an example (e.g. 3073 x 50,000)
# 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 set
  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
# ... (trunctated: continues for 1000 lines)
```

上のコードでは，いくつかのランダムな重みベクトルWを試してみましたが，いくつかはうまくいきました．この検索で見つかった一番良い重みWをテストセットで試してみることができます．

```
# 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 examples
# 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
```

最高のWでは約15.5%の精度が得られます．分類を完全にランダムに推測しても10%しか得られないことを考えると，このような脳死状態ランダム検索の解決策にしては，それほど悪い結果ではありません．

基本的な考え方: 反復的な洗練．もちろん，もっと良いことができることが判明しています．中心となる考え方は，最適な重みの集合Wを見つけることは非常に難しい，あるいは不可能な問題です（特にWが複雑なニューラルネットワーク全体の重みを含んでいる場合）が，特定の重みの集合Wをわずかに良くするように洗練させる問題は，それほど困難ではないということです．言い換えるならば，このアプローチは，ランダムなWから始めてそれを反復的に洗練させ，その都度わずかずつ良いものにしていくというものです．

☞我々の戦略は，ランダムな重みから始めて，時間をかけてそれを反復的に改良し，より低い損失を得ることです．

目隠しされたハイカーとの類似．目隠しをしたまま丘陵地をハイキングしていて，底にたどり着こうとしていると考えてみてはいかがでしょうか．CIFAR-10の例では，Wの寸法が10×3073なので，丘は30,730次元です．丘の上のすべての点で，我々は特定の損失（地形の高さ）に到達します．

作戦その2．ランダムローカル探索(ランダム局所探索)．最初に思いつく戦略は，ランダムな方向に片足を伸ばしてみて，下り坂になっている場合のみ一歩を踏み出してみるというものです．具体的には，まずランダムなWから始めて，これにランダムな摂動δWを発生させ，摂動した$W+\delta W$での損失が低ければ更新を実行します．この手順のコードは以下の通りです．

```
W = np.random.randn(10, 3073) * 0.001 # generate random starting W
bestloss = float("inf")
for i in range(1000):
  step_size = 0.0001
  Wtry = W + np.random.randn(10, 3073) * step_size
  loss = L(Xtr_cols, Ytr, Wtry)
  if loss < bestloss:
    W = Wtry
    bestloss = loss
  print 'iter %d loss is %f' % (i, bestloss)
```

以前と同じ数の損失関数評価（1000）を使用して，このアプローチはテストセットの分類精度21.4%を達成しました．以前より改善されていますが無駄が多く，計算量も多くなっています．

戦略その3．勾配追跡．前のセクションでは，重みの空間の中で，重みベクトルを改善する（そして，より低い損失を与える）方向を見つけようとしました．良い方向をランダムに探す必要はないことがわかりました：重みベクトルを変更すべき方向に沿って最適な方向を計算することができますが，それは数学的に最も急峻な降下の方向であることが保証されています（少なくとも，ステップサイズがゼロに向かう限界では）．この方向は損失関数の勾配に関係します．我々のハイキングのアナロジーでは，このアプローチは，足元の丘の勾配を感じて，最も急だと感じる方向に降りることに大体対応しています．
一次元関数では，勾配は観測点での関数の瞬間的な変化率です．勾配は，1つの数値を取るのではなく，数値のベクトルを取る関数の勾配の一般化です．さらに，勾配は入力空間の各次元の傾き（より一般的には微分と呼ばれる）のベクトルにすぎません．入力に対する1次元関数の微分の数式は次のようになります．

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

対象の関数が1つの数(スカラー)の代わりに数のベクトルをとるとき，その微分を部分微分と呼び，勾配は各次元の部分微分のベクトルです．

勾配の計算

勾配を計算するには2つの方法があります．遅くて近似的だが簡単に計算できる方法(数値勾配)と，速くて正確だが微積分を必要とし、間違いが起こりやすい方法(解析的勾配)です．ここでは両方の方法を紹介します．

有限差分を用いた勾配の数値計算

上で与えられた式により，数値的に勾配を計算することができます．ここでは，関数 **f**，勾配を評価するためのベクトル **x**，そして **x** における **f** の勾配を返す汎用関数を示します．

```
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
```

上で述べた勾配の公式に従って，上のコードはすべての次元を1つずつ反復処理し，その次元に沿って小さな変化hを加え，その次元に沿って関数がどのくらい変化したかを見て損失関数の偏微分を計算します．変数**grad**は最後に完全な勾配を保持します．

実用性を考慮する．数学的な定式化では，勾配はhがゼロに向かう限界で定義されますが，実際には非常に小さな値を使えば十分であることに注意してください (例で見たように 1e-5 など)．理想的には，数値的な問題にならない最小のステップサイズを使いたいでしょう．さらに，実際には，数値的な勾配を計算する際には，中心差分の式を使った方がよく機能します．中心差分：$[f(x+h) - f(x-h)]/2h$. 詳細はwikiを参照してください．

上で与えられた関数を使って，任意の点で任意の関数の勾配を計算することができます．CIFAR-10 の損失関数の勾配を，重み空間のランダムな点で計算してみましょう．

(次ページに続きます)

```
# 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
```

勾配から，各次元に沿った損失関数の傾きがわかります.

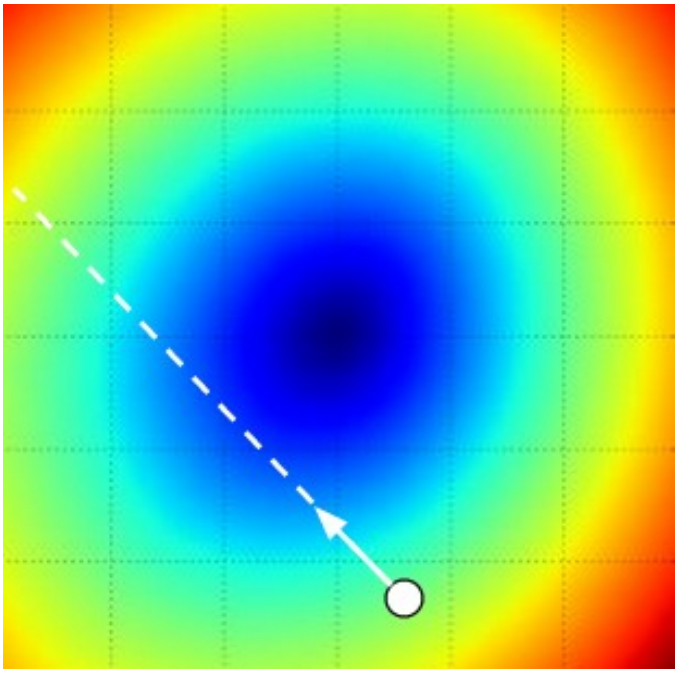
```
loss_original = CIFAR10_loss_fun(W) # the original loss
print 'original loss: %f' % (loss_original, )

# 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
```

負の勾配方向の更新．上のコードでは，`W_new` を計算するために勾配 `df` の負の方向に更新を行っていることに注目してください.

ステップの大きさの影響．勾配は関数が最も急峻な増加率を持つ方向を教えてくれますが，その方向に沿ってどのくらいの距離を踏むべきかは教えてくれません.

この授業で後述するように，ステップサイズ（学習率とも呼ばれる）を選択することは，ニューラルネットワークを訓練する上で最も重要な（そして最も頭痛の種となる）ハイパーパラメータ設定の一つになります．目隠しをして坂道を下るという例えでは，足下の坂道が何かの方向に傾斜しているのを感じますが，どのくらいの長さのステップを踏むべきかはわかりません．慎重に足を引き摺る様に歩けば，一貫した前進が期待できますが，非常に小さな前進になります（これは小さなステップサイズを持つことに対応します）．逆に，より速く降りるために，自信を持って大きなステップを踏むこともできますが，これではうまくいかないかもしれません．上のコード例にもあるように，ある時点で大きなステップを踏むと，「踏み越えた」ことになり，より大きな損失が発生します.

(次ページに続きます)



ステップサイズの効果の可視化．ある特定のスポットWからスタートして，損失関数の最も急峻な減少の方向を示す勾配（=負の方向，白い矢印）を評価します．
ステップが小さいと，一貫性はありますが，ゆっくりとした進行になる可能性があります．大きなステップは，より良い進歩につながりますが，リスクが高くなります．ステップサイズが大きいと，最終的にはオーバーシュートして損失を悪化させてしまうことに注意してください．ステップサイズ（後に学習率と呼ぶことにします）は，慎重に調整しなければならない最も重要なハイパーパラメータの1つになります．

効率の問題．数値勾配の評価は，パラメータの数に比例して複雑さが増すことにお気づきでしょう．私たちの例では，合計で30730個のパラメータを持っていたので，勾配を評価するために30,731個の損失関数を評価しなければならず，1回のパラメータ更新しかできませんでした．最近のニューラルネットワークは数千万個のパラメータを簡単に持つことができるので，この問題はさらに悪化します．明らかに，この戦略はスケーラブルではないので，もっと良いものが必要です．

微積分を使って勾配を解析的に計算する
数値勾配は有限差分近似を使って計算するのは非常に簡単ですが，欠点は近似的であること (小さな値のhを選ばなければならないので，真の勾配はhがゼロになる限界として定義されます) と，計算に非常にコストがかかることです．勾配を計算する2つ目の方法は微積分を使った解析的な方法で，これにより勾配の直接式を導出することができます (近似なし)．しかし，数値勾配とは異なり，実装する際にエラーが発生しやすいため，実際には解析的勾配を計算して数値勾配と比較し，実装の正確性を確認することが一般的です．これをグラジエントチェック(gradient check)と呼びます．
1つのデータポイントに対するSVM損失関数の例を使ってみましょう．

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

重みを基準にして関数を微分することができます．例えば，w_{y_i}に対する勾配を取ると，次のようになります．

$$\nabla_{w_{y_i}} L_i = -\left(\sum_{j \neq y_i} \mathbb{1}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) \right)$$

ここで$\mathbb{1}$はインジケータ関数で，内部の条件が真であれば 1，そうでなければ 0 となります．式を書くと一見不気味ですが，コードでこれを実装するときには，希望するマージンを満たさなかった分類数を単純に数えます (そのため損失関数に寄与した)．これは，正しい分類に対応するWの行に関してのみの勾配であることに注意してください．$j \neq y_i$を満たす他の行については，勾配は次のようになります:

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

勾配の式を導出したら，それを実装して勾配の更新を実行するのは容易です．

勾配降下法

損失関数の勾配を計算できるようになったので，勾配の評価とパラメータの更新を繰り返すことを勾配降下と呼びます．基本版は以下のようになります．

```
# Vanilla Gradient Descent

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

この単純なループは，すべてのニューラルネットワークライブラリの中核をなしています．最適化を実行する方法は他にもありますが（例：LBFGS），現在のところ勾配降下法がニューラルネットワークの損失関数を最適化する最も一般的で確立された方法です．この分類では，このループの詳細（例えば，更新式の正確な詳細など）にいくつかの工夫を凝らしていきますが，結果に満足するまで勾配を追うという核心的な考え方は変わりません．

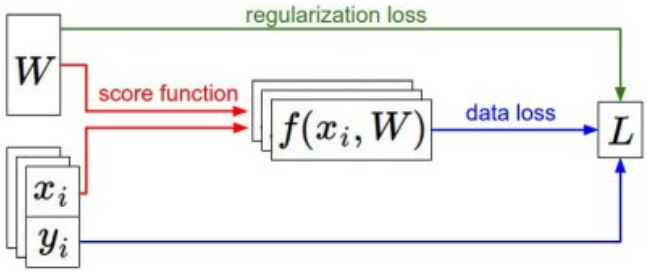
ミニバッチ勾配降下法．大規模なアプリケーション(ILSVRCチャレンジのような)では，学習データは数百万個の例題を持つことがあります．そのため，単一のパラメータ更新だけを行うために，訓練セット全体の損失関数を計算するのは無駄なように思えます．この課題に対処するための非常に一般的なアプローチは，訓練データのバッチに対する勾配を計算することです．例えば，現在の最新のConvNetsでは，典型的なバッチには120万個の訓練セット全体から256個の例が含まれています．その後，このバッチは以下パラメータ更新プログラムを実行するために使用されます．

```
# 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
```

これがうまく機能する理由は，学習データの例が相関関係にあるからです．これを見るために，ILSVRC の 120 万枚の画像が，実際には 1000 個のユニークな画像（各分類に 1 つずつ，言い換えれば各画像の 1200 個の同一のコピー）の完全な複製で構成されているという極端なケースを考えてみましょう．そして，120 万枚の画像全体のデータ損失を平均すると，1000 枚のうちの小さなサブセットのみで評価した場合と全く同じ損失が得られることがわかります．もちろん，実際には，

データセットには重複した画像は含まれていないので，ミニバッチからの勾配は，完全な目的物の勾配の良い近似値です．したがって，より頻繁にパラメータ更新を行うためにミニバッチの勾配を評価することによって，実際にははるかに速い収束を達成することができます．
最たる例としては，ミニバッチに1つの例しか含まれていない設定があります．この処理は確率的勾配降下 (SGD) と呼ばれます (オンライン勾配降下と呼ばれることもあります)．これは比較的一般的ではありませんが，実際にはベクトル化されたコードの最適化により，1つの例の勾配を100回評価するよりも，100個の例の勾配を100回評価する方がはるかに効率的に計算できるからです．SGD は技術的には一度に一つの例を使って勾配を評価することを指しますが，ミニバッチ勾配降下に言及している場合でも SGD という用語を使う人を耳にすることがあります (つまり，「ミニバッチ勾配降下」の MGD や「バッチ勾配降下」の BGD という言及はめったに見られません)．ミニバッチのサイズはハイパーパラメータですが，クロスバリデーションを行うことはあまりありません．通常はメモリ制約（もしあれば）に基づいているか，32，64，128などの値に設定されています．実際には2の累乗を使用していますが，これは多くのベクトル化された演算の実装が入力のサイズを2の累乗にした方が(論理シフト演算で済むため)高速に動作するからです．

まとめ



(x, y)のペアのデータセットが与えられ，固定される．重みは，最初は乱数であり，変更可能である．フォワードパスの間，スコア関数は，ベクトルfに格納された分類スコアを計算します．データ損失はスコアfとラベルyの間の互換性を計算します．正則化損失は重みの関数でしかありません．勾配降下の間，我々は重みの勾配を計算し（必要に応じてデータの勾配も計算します），勾配降下の間にパラメータの更新を実行するためにそれらを使用します．

このセクションでは
・底辺に到達しようとしている高次元の最適化環境としての損失関数の感覚的アプローチを開発しました．我々が開発したワーキングアナロジーは，目隠しされたハイカーが底にたどり着きたいと願っているようなものであった．特に，SVMのコスト関数は区分線形関数であり，お椀型であることがわかりました．
・私たちは，損失関数を反復的な絞り込みで最適化するというアイデアを思いつきました．
・関数の勾配が最も急な上昇方向を与えることがわかり，有限差分近似を用いて数値的に計算する単純だが非効率的な方法について議論しました（有限差分とは，数値勾配の計算に用いられるhの値のことです）．

・パラメータの更新には，ステップサイズ(または学習率)のトリッキーな設定が必要で，これを適切に設定しなければならないことがわかりました．低すぎると，進捗は安定していますが遅いです．高すぎると，進捗は速くなりますが，よりリスクが高くなります．このトレードオフについては，今後のセクションで詳しく説明します．
・数値勾配と解析的勾配の計算の間のトレードオフについて議論しました．数値勾配は単純ですが，近似的で計算にコストがかかります．解析的勾配は正確で計算が速いですが，数学で勾配を導出する必要があるため，よりエラーが発生しやすいです．そのため，実際には常に解析的勾配を使用し，その実装を数値勾配と比較する勾配チェックを行います．
・反復的に勾配を計算し，ループ内でパラメータの更新を行う勾配降下アルゴリズムを導入しました．