CS231n Convolutional Neural Networks for Visual Recognition

8.実装してみましょう

このセクションでは、2次元のトイ(お遊び用の)・ニューラル・ネットワークの完全な実装を説明します。まず、単純な線形分類器を実装し、次にそのコードを2層のニューラルネットワークに拡張します。見ての通り、この拡張は驚くほど簡単で、ほとんど変更の必要はありません。

データの生成

簡単には線形分離できない分類データセットを作ってみましょう。好例はスパイラルデータセットで、次のようにして生成できます。

```
N = 100 # number of points per class
D = 2 # dimensionality
K = 3 # number of classes
X = np.zeros((N*K,D)) # data matrix (each row = single example)
y = np.zeros(N*K, dtype='uint8') # class labels
for j in range(K):
  ix = range(N*j,N*(j+1))
  r = np.linspace(0.0,1,N) # radius
  t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
  X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
  y[ix] = j
# lets visualize the data:
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
plt.show()
```

トイ・スパイラルのデータは、3つの分類（青、赤、黄）で構成されており、直線的に分離できません。

通常であれば、各特徴量の平均値がゼロ、標準偏差が1単位になるようにデータセットを前処理したいところですが、今回は特徴量がすでに-1から1の範囲に収まっているので、このステップは省略します。

ソフトマックス線形分類器の学習

パラメータの初期化

まず、この分類データセットを使って、ソフトマックス分類器を学習しましょう。前のセクションで見たように、Softmax分類器は線形スコア関数を持ち、クロスエントロピー損失を利用します。線形分類器のパラメータは、各分類の重み行列`W`とバイアスベクトル`b`からなります。まず，これらのパラメータを乱数で初期化しましょう.

```
# initialize parameters randomly
W = 0.01 * np.random.randn(D,K)
b = np.zeros((1,K))
```

`D = 2`が次元の数、`K = 3`が分類の数であることを覚えておいてください。

分類スコアの計算

これは線形分類器なので，1回の行列乗算ですべての分類スコアを非常に簡単に並列に計算することができます.

```
# compute class scores for a linear classifier
scores = np.dot(X, W) + b
```

この例では、300個の2次元ポイントがあるので、この乗算の後、配列`score`はサイズ[300 x 3]となり、各行には3つの分類（青、赤、黄）に対応する分類スコアが表示されます。

損失の計算

損失関数とは，微分可能な目的関数であり，計算された分類スコアに対する不満足度を定量化するものです．直感的には，正しい分類が他の分類よりも高いスコアを持つようにしたいと思います．そうであれば，損失は小さく，そうでなければ，損失は大きくなります．この直感を定量化する方法はたくさんありますが、この例ではソフトマックス分類器に関連するクロスエントロピー損失を使用します。以前に取り上げましたが、fが1つの例に対する分類スコアの配列(例えばここでは3つの数字の配列)である場合、ソフトマックス分類器は損失関数を次のように計算します。

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

ソフトマックス分類器は、fの各要素を、3つの分類の（正規化されていない）対数確率を保持していると解釈できます。(正規化されていない)確率を得るためにこれらを指数化し、確率を得るためにそれらを正規化します。結果的にlogの中の式は，正しい分類の正規化された確率となります。この式はどのように機能するでしょうか。この量は常に0と1の間です。正しい分類の確率が非常に小さい(0に近い)場合、損失は(正の)無限大に向かっていきます。逆に、正しい分類の確率が1になると、$\log(1) = 0$なので、損失は0になります。したがって、損失関数L_iは、正解分類の確率が高いときは低く、低いときは非常に高くなります。

また、ソフトマックス分類器の完全な損失関数は、学習例と正則化の平均交差エントロピー損失として定義されることを思い出してください。

$$L_i = \underbrace{\frac{1}{N}\sum_i L_i}_{\text{data loss}} + \underbrace{\frac{1}{2}\lambda\sum_k\sum_l W_{k,l}^2}_{\text{regularization loss}}$$

上記で計算した`scores`の配列があれば、損失を計算することができます。まず、確率の求めます。やり方は簡単です。

```
num_examples = X.shape[0]
# get unnormalized probabilities
exp_scores = np.exp(scores)
# normalize them for each example
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
```

サイズ [300 x 3] の配列`probs`ができ、各行にクラスの確率が入ります。正規化したので、すべての行の合計が1になります。これで、各例で正しいクラスに割り当てられた対数確率を取り出すことができます。

```
correct_logprobs = -np.log(probs[range(num_examples),y])
```

配列`correct_logprobs`は，各例の正しいクラスに割り当てられた確率を表す 1 次元の配列です．完全な損失は、これらのログ確率と正則化損失の平均です。

```
# compute the loss: average cross-entropy loss and regularization
data_loss = np.sum(correct_logprobs)/num_examples
reg_loss = 0.5*reg*np.sum(W*W)
loss = data_loss + reg_loss
```

このコードでは、正則化強度λは`reg`の中に格納されています。正則化に`0.5`を乗じた便利な要素は、すぐに明らかになるでしょう。これを最初に（ランダムなパラメータで）評価すると、`loss = 1.1`となり、`-np.log(1.0/3)`となります。これは、初期のランダムな重みが小さい場合、すべてのクラスに割り当てられる確率が約3分の1になるためです。そこで、損失をできるだけ小さくしたいと考え、絶対的な下限として`loss = 0`を設定しました。しかし、損失が小さければ小さいほど、すべての例で正しいクラスに割り当てられる確率が高くなります。

バックプロパゲーションによる解析的勾配の計算

損失を評価する方法ができたので、今度はそれを最小化する必要があります。これには勾配降下法を使います。つまり、（上のように）ランダムなパラメータから始めて、パラメータに対する損失関数の勾配を評価し、損失を減らすためにどのようにパラメータを変更すべきかを知るのです。

中間変数pを導入しましょう。これは（正規化された）確率のベクトルです。ある例の損失は

$$p_k = \frac{e^{f_k}}{\sum_j e^{f_j}} \qquad L_i = -\log(p_{y_i})$$

ここで、この例が完全な目的に寄与する損失L_iを減少させるために、fの内部で計算されたスコアがどのように変化すべきかを理解したいと思います。言い換えれば、勾配$\partial L_i / \partial f_k$を導出したいわけです。損失$L_i$は$p$から計算され、これは$f$に依存します。偏微分の連鎖公式を使って式中の多くの項を消し勾配を導き出すのは面白くもあります。そして最終的には非常にシンプルで明快な式が導出されます

$$\frac{\partial L_i}{\partial f_k} = p_k - \mathbb{1}(y_i = k)$$

この表現がいかに優れておりシンプルであるかに注目してください。仮に、計算した確率が`p = [0.2, 0.3, 0.5]`で、正しいクラスが真ん中だったとします（確率は0.3）。この導出によると、スコアの勾配は`df = [0.2, -0.7, 0.5]`となります。勾配の解釈を思い出してみると、この結果は非常に直感的であることがわかります。スコアベクトル`f`の最初または最後の要素（間違ったクラスのスコア）を増加させると、損失が増加してしまうの（正の符号+0.2と+0.5のため） は予想通り悪いことです。しかし、正しいクラスのスコアを増やすことは、損失にマイナスの影響を与えます。0.7の勾配は、正しいクラスのスコアを増加させると、損失L_iが減少することを示しており、これは理にかなっています。
これらはすべて，以下のコードに集約されます．`probs`は，各例のすべてのクラスの確率を（行として）格納していることを思い出してください．`dscores`と呼ばれるスコアの勾配を得るためには，以下のようにします．

```
dscores = probs
dscores[range(num_examples),y] -= 1
dscores /= num_examples
```

最後に、`scores = np.dot(X, W) + b`という結果が出たので、（`dscores`に格納されている）`scores`の勾配を利用して、`W`と`b`にバックプロパゲートすることができます。

```
dW = np.dot(X.T, dscores)
db = np.sum(dscores, axis=0, keepdims=True)
dW += reg*W # don't forget the regularization gradient
```

ここでは、行列の乗算処理をバックプロップして、正則化による寄与を加えていることがわかります。正則化の勾配は非常にシンプルな形の`reg*W`であることに注意してください。なぜなら、その損失寄与に定数0.5を使用したからです。(すなわち、$\frac{d}{dw}(\frac{1}{2}\lambda w^2) = \lambda w$です。これは、勾配式を単純化するための一般的な便宜上のトリックです。)

パラメータの更新を行う

勾配を評価したことで、すべてのパラメータが損失関数にどのような影響を与えるかがわかりました。ここでは、損失を減らすために、負の勾配方向にパラメータの更新を行います。

```
# perform a parameter update
W += -step_size * dW
b += -step_size * db
```

おさらいしましょう:ソフトマックス分類器の学習

これらをまとめて、勾配降下法を用いてSoftmax分類器を学習するための完全なコードを以下に示します。(訳註：もともとはPython2で書かれたコードかもしれない。`for`文以外で`range`を使っている場合、`list(range(number))`の様に記述しないとエラーになると思われる)

```
#Train a Linear Classifier

# initialize parameters randomly
W = 0.01 * np.random.randn(D,K)
b = np.zeros((1,K))

# some hyperparameters
step_size = 1e-0
reg = 1e-3 # regularization strength

# gradient descent loop
num_examples = X.shape[0]
for i in range(200):

  # evaluate class scores, [N x K]
  scores = np.dot(X, W) + b

  # compute the class probabilities
  exp_scores = np.exp(scores)
  probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True) # [N x K]

  # compute the loss: average cross-entropy loss and regularization
  correct_logprobs = -np.log(probs[range(num_examples),y])
  data_loss = np.sum(correct_logprobs)/num_examples
  reg_loss = 0.5*reg*np.sum(W*W)
  loss = data_loss + reg_loss
  if i % 10 == 0:
    print "iteration %d: loss %f" % (i, loss)

  # compute the gradient on scores
  dscores = probs
  dscores[range(num_examples),y] -= 1
  dscores /= num_examples

  # backpropate the gradient to the parameters (W,b)
  dW = np.dot(X.T, dscores)
  db = np.sum(dscores, axis=0, keepdims=True)

  dW += reg*W # regularization gradient

  # perform a parameter update
  W += -step_size * dW
  b += -step_size * db
```

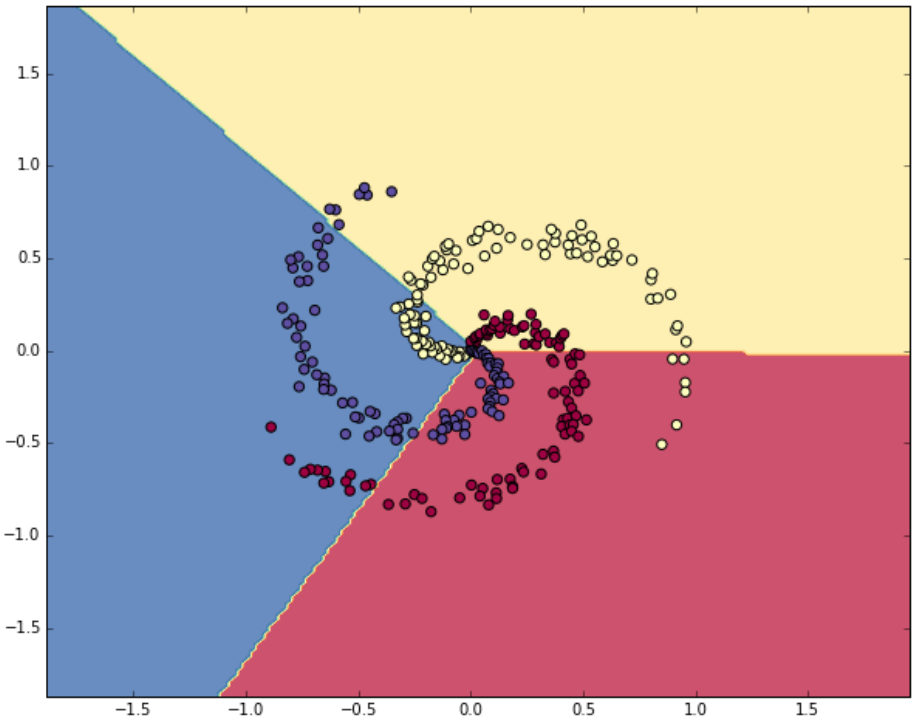
これを実行すると、出力が表示されます。

```
iteration 0: loss 1.096956
iteration 10: loss 0.917265
iteration 20: loss 0.851503
iteration 30: loss 0.822336
iteration 40: loss 0.807586

(中略)

iteration 160: loss 0.786431
iteration 170: loss 0.786373
iteration 180: loss 0.786331
iteration 190: loss 0.786302
```

約190回の繰り返しの後、値が収束したことがわかります。これで、トレーニングセットの精度を評価することができます。

```
# evaluate training set accuracy
scores = np.dot(X, W) + b
predicted_class = np.argmax(scores, axis=1)
print 'training accuracy: %.2f' % (np.mean(predicted_class == y))
```

これは49%と表示されます。あまり良い結果ではありませんが、データセットが線形分離できないようデータですので、驚くべきことではありません。学習した決定境界をプロットすることもできます。



線形分類器はトイ・スパイラル・データセットの学習に失敗

ニューラルネットワークの学習

このデータセットでは、明らかに線形分類器では不十分ですのでニューラルネットワークを使用したいと考えています。このおもちゃのデータでは、隠れ層を1つ追加すれば十分です。重みとバイアスを2セット（第1層と第2層）用意します。

```
# initialize parameters randomly
h = 100 # size of hidden layer
W = 0.01 * np.random.randn(D,h)
b = np.zeros((1,h))
W2 = 0.01 * np.random.randn(h,K)
b2 = np.zeros((1,K))
```

スコアを算出するためのフォワードパスの形が変わります。

```
# evaluate class scores with a 2-layer Neural Network
hidden_layer = np.maximum(0, np.dot(X, W) + b) # note, ReLU activation
scores = np.dot(hidden_layer, W2) + b2
```

ここでは、まず隠れ層の表現を計算し、次にこの隠れ層に基づいたスコアを計算しています。重要なのは、隠れ層でゼロでしきい値にする単純なReLUにより非線形性を導入したことです。
その他はすべて同じです。先ほどと同じようにスコアに基づいて損失を計算し、先ほどと同じようにスコア`dscores`の勾配を得ます。しかし、その勾配をモデルのパラメータに逆伝播させる方法は、当然ながら形を変えています。まず、ニューラルネットワークの第2層をバックパゲートします。これはソフトマックス分類器のコードと同じですが、`X`（生データ）を変数`hidden_layer`に置き換えています。

```
# backpropate the gradient to the parameters
# first backprop into parameters W2 and b2
dW2 = np.dot(hidden_layer.T, dscores)
db2 = np.sum(dscores, axis=0, keepdims=True)
```

なぜなら、`hidden_layer`自体が他のパラメータとデータの関数だからです。この変数を使ってバックプロパゲーションを続ける必要があります。その勾配は次のように計算できます。

```
dhidden = np.dot(dscores, W2.T)
```

これで、隠れ層の出力に勾配を掛けることができました。次に、ReLUの活性化関数をバックプロパゲートする必要があります。バックワードパスでのReLUは実質的にスイッチであるため、これは簡単であることがわかります。$r = \max(0,x)$であるため、$\frac{dr}{dx} = 1(x > 0)$となります。
.チェーンルールと組み合わせることで、ReLUユニットは、入力が0より大きければ勾配を変えずに通過させ、前進時に入力が0より小さければ殺してしまうことがわかります。したがって、ReLUをその場でバックプロパゲートするには、次のようにすればよいことがわかります

```
# backprop the ReLU non-linearity
dhidden[hidden_layer <= 0] = 0
```

そして、いよいよ第1層のウェイトとバイアスへと続きます。

```
# finally into W,b
dW = np.dot(X.T, dhidden)
db = np.sum(dhidden, axis=0, keepdims=True)
```

これで完了です。グラデーション`dW,db,dW2,db2`が得られ、パラメータの更新を行うことができるようになりました。線形分類の時と似た様な処理コードになります。

```
# initialize parameters randomly
h = 100 # size of hidden layer
W = 0.01 * np.random.randn(D,h)
b = np.zeros((1,h))
W2 = 0.01 * np.random.randn(h,K)
b2 = np.zeros((1,K))

# some hyperparameters
step_size = 1e-0
reg = 1e-3 # regularization strength

# gradient descent loop
num_examples = X.shape[0]
for i in range(10000):

  # evaluate class scores, [N x K]
  hidden_layer = np.maximum(0, np.dot(X, W) + b) # note, ReLU activation
  scores = np.dot(hidden_layer, W2) + b2

  # compute the class probabilities
  exp_scores = np.exp(scores)
  probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True) # [N x K]

  # compute the loss: average cross-entropy loss and regularization
  correct_logprobs = -np.log(probs[range(num_examples),y])
  data_loss = np.sum(correct_logprobs)/num_examples
  reg_loss = 0.5*reg*np.sum(W*W) + 0.5*reg*np.sum(W2*W2)
  loss = data_loss + reg_loss
  if i % 1000 == 0:
    print "iteration %d: loss %f" % (i, loss)

  # compute the gradient on scores
  dscores = probs
  dscores[range(num_examples),y] -= 1
  dscores /= num_examples

  # backpropate the gradient to the parameters
  # first backprop into parameters W2 and b2
  dW2 = np.dot(hidden_layer.T, dscores)
  db2 = np.sum(dscores, axis=0, keepdims=True)
  # next backprop into hidden layer
  dhidden = np.dot(dscores, W2.T)
  # backprop the ReLU non-linearity
  dhidden[hidden_layer <= 0] = 0
  # finally into W,b
  dW = np.dot(X.T, dhidden)
  db = np.sum(dhidden, axis=0, keepdims=True)

  # add regularization gradient contribution
  dW2 += reg * W2
  dW += reg * W
```

```
  # perform a parameter update
  W += -step_size * dW
  b += -step_size * db
  W2 += -step_size * dW2
  b2 += -step_size * db2
```

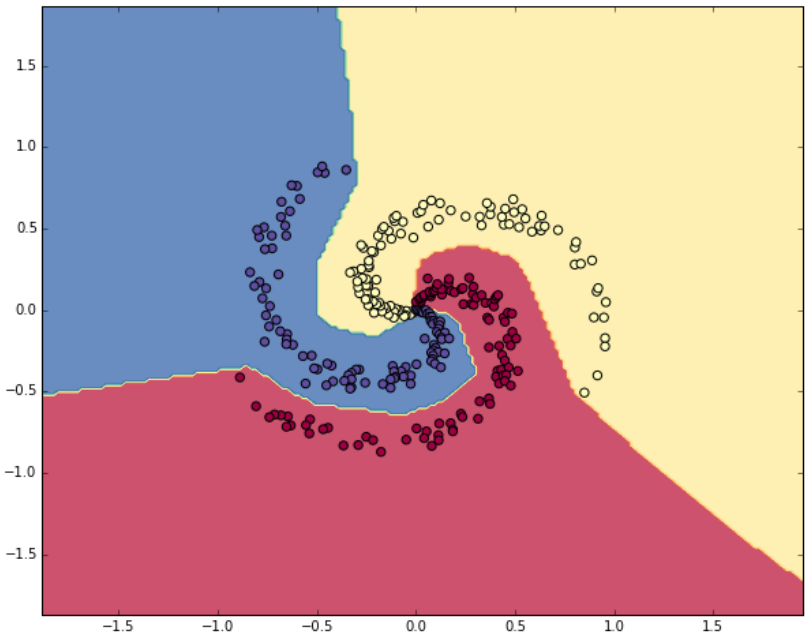
出力結果は

```
iteration 0: loss 1.098744
iteration 1000: loss 0.294946
iteration 2000: loss 0.259301
iteration 3000: loss 0.248310
iteration 4000: loss 0.246170
iteration 5000: loss 0.245649
iteration 6000: loss 0.245491
iteration 7000: loss 0.245400
iteration 8000: loss 0.245335
iteration 9000: loss 0.245292
```

訓練後の認識精度を測定します。

```
# evaluate training set accuracy
hidden_layer = np.maximum(0, np.dot(X, W) + b)
scores = np.dot(hidden_layer, W2) + b2
predicted_class = np.argmax(scores, axis=1)
print 'training accuracy: %.2f' % (np.mean(predicted_class == y))
```

出力結果は98%！そして、境界を可視化することもできます。



ニューラルネットワークによる分類でスパイラルデータセットを制覇！