

Kalici CD (Persistent Contrastive Divergence -PCD-)

Kisitli Boltzman Makinalari (RBM) yazisinda gosterilen egitim CD (contrastive divergence) uzerinden idi. Amac alttaki formilde, ozellikle eksiden sonraki terimi yaklasiksal olarak hesaplamaktir.

$$\sum_{n=1}^N \langle y_i y_j \rangle_{P(h|x^n; W)} - \langle y_i y_j \rangle_{P(x, h; W)}$$

Bu terime basinda eksi oldugu icin negatif parcaciklar (negative particles) ismi de veriliyor.

Simdi RBM’de gordugumuz CD’yi hatirlayalim, CD bir tur “tek adimlik Gibbs ornekleme” yapıyordu; bu tek adim ornekleme sonrasinda bir sonraki adim oncesi, veri, tekrar baslangic noktası olarak zincire veriliyordu. Yani her CD adiminin baslangici illa ki verinin kendisi olacaktır. Bu usul Gibbs’in veriden uzaklasma sansi cok azdir.

Bu yontemin avantajlari oldugu gibi tabii ki dezavantajlari da var; Cogugu ilginç yapay ogrenim verisi çokdorucludur (multimodal), yani optimizasyon baglamında dusunulurse birden fazla tepe (ya da cukur) noktası icerir. Eger eldeki veri egitim prosedurunu bu noktalara yeterince kanalizet edemiyorsa, o noktalar ogrenilmemis olur ve bu test verisi uzerindeki performansi kotu etkiler. Fakat bazen verinin bile soylediginden degisik yonleri gezebilen bir prosedur belki daha basarili olacaktır.

Bu dezavantajlari duzelten bir teknik Kalici CD teknigidir. PCD’ye gore modelden gelen “negatif parcaciklari” ornekleme arka planda, kendi baslarina ilerler, ve bu zincir hicbir zaman veriye, ya da baska bir seye set edilmez (hatta zincirin baslangic noktası bile veriden alakasiz olarak, rasgele secilir). Bu yonteme gore $h^0, x^0, h^1, x^1, \dots$ uretimi neredeyse tamamen “kapali devre” kendi kendine ilerleyen bir surec olacaktır. Diger yanda pozitif parcaciklar veriden geliyor (ve tabii ki her gradyan adimi sonrasi degisen W hem pozitif hem negatif parcaciklari etkiler), ve bu al/ver iliskisi, hatta bir bakima model ile verinin kapismasinin PCD’yi daha avantajli hale getirdigi iddia edilir, ki PCD, CD’den genellikle daha iyi ogrenim saglar [5].

```
from sklearn.utils import gen_even_slices
import numpy as np
import itertools

class RBM:
    def __init__(self, num_hidden, num_visible, learning_rate, max_epochs=10,
                 batch_size=10):
        self.num_hidden = num_hidden
        self.num_visible = num_visible
        self.learning_rate = learning_rate
        self.weights = 0.1 * np.random.randn(self.num_visible, self.num_hidden)
        self.weights = np.insert(self.weights, 0, 0, axis = 0)
        self.weights = np.insert(self.weights, 0, 0, axis = 1)
```

```

self.max_epochs = max_epochs
self.batch_size = batch_size

def run_visible(self, data):
    num_examples = data.shape[0]

    hidden_states = np.ones((num_examples, self.num_hidden + 1))

    data = np.insert(data, 0, 1, axis = 1)

    hidden_activations = np.dot(data, self.weights)
    hidden_probs = self._logistic(hidden_activations)
    hidden_states[:, :] = hidden_probs > \
        np.random.rand(num_examples, self.num_hidden + 1)
    hidden_states = hidden_states[:, 1:]
    return hidden_states

def run_hidden(self, data):
    num_examples = data.shape[0]

    visible_states = np.ones((num_examples, self.num_visible + 1))

    data = np.insert(data, 0, 1, axis = 1)

    visible_activations = np.dot(data, self.weights.T)
    visible_probs = self._logistic(visible_activations)
    visible_states[:, :] = visible_probs > \
        np.random.rand(num_examples, self.num_visible + 1)

    visible_states = visible_states[:, 1:]
    return visible_states

def _logistic(self, x):
    return 1.0 / (1 + np.exp(-x))

def _fit(self, v_pos):
    h_pos = self.run_visible(v_pos)
    v_neg = self.run_hidden(self.h_samples_)
    h_neg = self.run_visible(v_neg)
    lr = float(self.learning_rate) / v_pos.shape[0]
    v_pos = np.insert(v_pos, 0, 1, axis = 1)
    h_pos = np.insert(h_pos, 0, 1, axis = 1)
    v_neg = np.insert(v_neg, 0, 1, axis = 1)
    h_neg = np.insert(h_neg, 0, 1, axis = 1)
    update = np.dot(v_pos.T, h_pos).T
    update -= np.dot(h_neg.T, v_neg)
    self.weights += lr * update.T
    h_neg[np.random.rand(h_neg.shape[0], h_neg.shape[1]) < h_neg] = 1.0
    self.h_samples_ = np.floor(h_neg, h_neg)[:, 1:]

def fit(self, data):
    num_examples = data.shape[0]
    self.h_samples_ = np.zeros((self.batch_size, self.num_hidden))
    n_batches = int(np.ceil(float(num_examples) / self.batch_size))

```

```

batch_slices = list(gen_even_slices(n_batches * self.batch_size,
                                   n_batches, num_examples))

for iteration in xrange(1, self.max_epochs + 1):
    for batch_slice in batch_slices:
        self._fit(data[batch_slice])

if __name__ == "__main__":
    import numpy as np
    X = np.array([[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 1]])
    model = RBM(num_hidden=2, num_visible=3, learning_rate=0.1, batch_size=2)
    model.fit(X)
    print model.weights

```

Ustte gorulen kod daha once RBM icin kullanim kodla benzesiyor, sadece `fit` degisik, ve `_fit` eklendi. Bu kodda miniparça (minibatch) kavrami da var, her gradyan adimi ufak verinin mini parçaları üzerinden atılır. Bu parçalar hakikaten ufak, mesela 10 ila 100 satırlık veri arasındadırlar ve bu ilginç bir durumu ortaya cikartır, özellikle negatif parçacıklar için, ki bu parçacıklar W bağlantısı haricinde kendi baslarına ilerler, çok az veri noktası ile işlem yapabilmektedirler.

Metot `fit` içinde `self.h_samples_` degiskenine dikkat, bu degisken PCD'nin “kalici” olmasını sağlar, her `_fit` çağrı sonrası negatif parçacık ornekleme `self.h_samples_` 'in bıraktığı yerden baslar.

RBM için kullandığımız aynı veri seti üzerine k-katlama ile test edelim,

```

from sklearn.linear_model import LogisticRegression
from sklearn.cross_validation import KFold
import numpy as np, rbmp, sys

X = np.loadtxt('../stat/stat_mixbern/binarydigits.txt')
Y = np.ravel(np.loadtxt('../stat/stat_mixbern/binarydigitlabels.txt'))

np.random.seed(0)

scores = []
cv = KFold(n=len(X), n_folds=3)
for train, test in cv:
    X_train, Y_train = X[train], Y[train]
    X_test, Y_test = X[test], Y[test]
    r = rbmp.RBM(num_hidden=40, learning_rate=0.1, max_epochs=100,
                 num_visible=64, batch_size=10)
    r.fit(X_train)
    clf = LogisticRegression(C=1000)
    clf.fit(r.run_visible(X_train), Y_train)
    res3 = clf.predict(r.run_visible(X_test))
    scores.append(np.sum(res3==Y_test) / float(len(Y_test)))

print np.mean(scores)

```

! python test_rbm_kfold.py

0.989898989899

Daha cetrefil bir veri seti MNIST veri setine [2] bakalım. Veri 28x28 boyutunda ikisel veri olarak kodlanmış rakamların el yazısından alınmış resimlerini içerir. Veri seti unlu çünkü Derin Öğrenim'in ilk büyük başarıları bu veri seti üzerinde paylaşıldı. MNIST'i aldıktan sonra eğitim / test kısımlarının ilk 1000 tanesi üzerinde algoritmamızı kullanırsak, tek komşulu KNN (yani 1-NN) yüzde 85.4 başarı sonucunu verir. Altındaki parametreler üzerinden PCD ile RBM'in başarıları yüzde 86 olacaktır.

```
import numpy as np, gzip, sys
from sklearn import neighbors
from sklearn.cross_validation import train_test_split
from sklearn.linear_model import LogisticRegression

np.random.seed(0)
S = 1000

f = gzip.open('/tmp/mnist.pkl.gz', 'rb')
train_set, valid_set, test_set = cPickle.load(f)
f.close()

X_train, y_train = train_set
X_test, y_test = valid_set
X_train = X_train[:S]; y_train = y_train[:S]
X_test = X_test[:S]; y_test = y_test[:S]
print X_train.shape

clf = neighbors.KNeighborsClassifier(n_neighbors=1)
clf.fit(X_train, y_train)
print 'KNN', clf.score(X_test, y_test)

import rbmp
r = rbmp.RBM(num_hidden=500, learning_rate=0.1, max_epochs=200,
             num_visible=784, batch_size=20)
r.fit(X_train)
clf = LogisticRegression(C=1000)
clf.fit(r.run_visible(X_train), y_train)
res3 = clf.predict(r.run_visible(X_test))
print 'RBM', np.sum(res3==y_test) / float(len(y_test))
```

Kaynaklar

- [1] http://videolectures.net/icml09_tieleman_ufw/
- [2] <http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz>
- [3] Bengio, Y., *Learning Deep Architectures for AI*
- [4] Larochelle, H., Neural Networks class, <https://www.youtube.com/playlist?list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBHd>
- [5] Murphy, K. *Machine Learning A Probabilistic Perspective*