천체의 펄서 예측 신경망

정보컴퓨터공학부 강민진

천체가 펄서인지 묻는 것에 참/거짓으로 나뉘므로 이는 이진 판단 문제이다.

참/거짓을 구분하는 실숫값을 만들어주고 이를 확률값의 성질에 맞게 변환해주는 함수를 이용해야 하는데 이 함수가 sigmoid 함수이다.

또한 손실함수를 이용해서 gradient descent를 이용한 학습을 해야 하는데, 이를 해결하기 위해서 cross entropy라는 개념을 사용한다.(두 확률 분포가 얼마나 비슷한 지를 숫자로 나타내는 개념.)

sigmoid function 의 cross entropy값을 loss function으로 정의하는 방법을 통해 이진 판단 문제를 다루는 NN를 학습할 수 있게 되었다.

정밀도(precision) : NN이 참으로 추정한 것 중에 참인 것

재현율(recall) : 참인 것 중에서 NN이 참으로 추정한 것.

모두 분자는 실제 참인 것이고, NN이 참으로 추정한 것이고, 분모는 위의 것으로 각각 들어간다.

하나만 높아도 정확도가 상승하기는 하지만 정확도를 더 높이려면 정밀도, 재현율을 함께 높아야 한다.

```
%run ../chap02/pulsar.ipynb
```

```
def pulsar_exec(epoch_count=10, mb_size=10, report=1, adjust_ratio=False):
    load_pulsar_dataset(adjust_ratio)
    init_model()
    train_and_test(epoch_count, mb_size, report)
```

```
def load_pulsar_dataset(adjust_ratio):
    pulsars, stars = [], []
    with open('../../data/chap02/pulsar_stars.csv') as csvfile:
        csvreader = csv.reader(csvfile)
        next(csvreader, None)
        rows = []
        for row in csvreader:
            if row[8] == '1': pulsars.append(row)
            else: stars.append(row)

    global data, input_cnt, output_cnt
    input_cnt, output_cnt = 8, 1

    star_cnt, pulsar_cnt = len(stars), len(pulsars)

    if adjust_ratio:
        data = np.zeros([2*star_cnt, 9])
        data[0:star_cnt, :] = np.asarray(stars, dtype='float32')
        for n in range(star_cnt):
            data[star_cnt+n] = np.asarray(pulsars[n % pulsar_cnt], dtype='float32')
    else:
        data = np.zeros([star_cnt+pulsar_cnt, 9])
        data[0:star_cnt, :] = np.asarray(stars, dtype='float32')
        data[star_cnt:, :] = np.asarray(pulsars, dtype='float32')
```

```
def eval_accuracy(output, y):
    est_yes = np.greater(output, 0)
    ans_yes = np.greater(y, 0.5)
    est_no = np.logical_not(est_yes)
    ans_no = np.logical_not(ans_yes)

    tp = np.sum(np.logical_and(est_yes, ans_yes))
    fp = np.sum(np.logical_and(est_yes, ans_no))
    fn = np.sum(np.logical_and(est_no, ans_yes))
    tn = np.sum(np.logical_and(est_no, ans_no))

    accuracy = safe_div(tp+tn, tp+tn+fp+fn)
    precision = safe_div(tp, tp+fp)
    recall = safe_div(tp, tp+fn)
    f1 = 2 * safe_div(recall*precision, recall+precision)

    return [accuracy, precision, recall, f1]

def safe_div(p, q):
    p, q = float(p), float(q)
    if np.abs(q) < 1.0e-20: return np.sign(p)
    return p / q
```

```
def train_and_test(epoch_count, mb_size, report):
    step_count = arrange_data(mb_size)
    test_x, test_y = get_test_data()

    for epoch in range(epoch_count):
        losses = []

        for n in range(step_count):
            train_x, train_y = get_train_data(mb_size, n)
            loss, _ = run_train(train_x, train_y)
            losses.append(loss)

        if report > 0 and (epoch+1) % report == 0:
            acc = run_test(test_x, test_y)
            acc_str = ','.join(['%5.3f']*4) % tuple(acc)
            print('Epoch {}: loss={:5.3f}, result={}'. ₩
                  format(epoch+1, np.mean(losses), acc_str))

    acc = run_test(test_x, test_y)
    acc_str = ','.join(['%5.3f']*4) % tuple(acc)
    print('₩nFinal Test: final result = {}'.format(acc_str))
```

```
%run ../chap02/pulsar_ext.ipynb
```

```
pulsar_exec()
```

```
Epoch 1: loss=0.139, result=0.965,0.975,0.624,0.761
Epoch 2: loss=0.128, result=0.971,0.969,0.696,0.810
Epoch 3: loss=0.130, result=0.973,0.951,0.734,0.828
Epoch 4: loss=0.131, result=0.973,0.917,0.762,0.832
Epoch 5: loss=0.130, result=0.975,0.953,0.755,0.843
Epoch 6: loss=0.127, result=0.974,0.893,0.809,0.849
Epoch 7: loss=0.130, result=0.972,0.950,0.718,0.818
Epoch 8: loss=0.119, result=0.974,0.960,0.743,0.837
Epoch 9: loss=0.116, result=0.974,0.971,0.727,0.832
Epoch 10: loss=0.120, result=0.967,0.976,0.649,0.780

Final Test: final result = 0.967,0.976,0.649,0.780
```

result = 정확도(accuracy), 정밀도(precision), 재현율(recall), F1 값(F1 score) 순.

[그림 1]

기존의 데이터로 NN의 성능을 추정한 것. 데이터 중 약 91%가 일반 별이고, 그 외 나머지인 약 9%만이 펄서이기 때문에, 학습이 거의 되지 않은 신경망도 무조건 펄서가 아니라고 답하는 것만

으로 약 91%의 정확도를 얻을 수 있다.(물론 제대로 학습되는 것은 아니다.) 데이터가 균형을 이루지 않고 치우쳐져 있기 때문에 재현율이 낮아도 96.7%의 높은 정확도를 달성하는 것을 볼 수 있다. 그러나, F1 score로 살펴보면 0.78 밖에 되지 않는데, 이는 정밀도와 재현율의 조화평균이 F1 score 이기 때문이다.

```
pulsar_exec(adjust_ratio=True)
```

```
Epoch 1: loss=0.405, result=0.923,0.939,0.902,0.920
Epoch 2: loss=0.376, result=0.918,0.983,0.848,0.910
Epoch 3: loss=0.377, result=0.904,0.879,0.934,0.906
Epoch 4: loss=0.364, result=0.915,0.987,0.839,0.907
Epoch 5: loss=0.378, result=0.925,0.941,0.905,0.923
Epoch 6: loss=0.367, result=0.923,0.982,0.859,0.916
Epoch 7: loss=0.369, result=0.917,0.986,0.844,0.909
Epoch 8: loss=0.363, result=0.927,0.940,0.910,0.925
Epoch 9: loss=0.349, result=0.926,0.981,0.867,0.920
Epoch 10: loss=0.378, result=0.915,0.909,0.919,0.914

Final Test: final result = 0.915,0.909,0.919,0.914
```

result = 정확도(accuracy), 정밀도(precision), 재현율(recall), F1 값(F1 score) 순.

[그림 2]

[그림 1]의 방식에서 일반 별 데이터를 상당 수 버려서 균형을 맞추거나, noise를 이용해서 data augmentation을 통해 펄서의 개수를 늘이게 되면 출현 빈도가 낮은 펄서를 학습할 수 있다.

위의 학습은 데이터의 균형을 맞춰서 학습한 결과이다.

얼핏 보기에는 91.5%의 정확도를 이루기 때문에 위의 96.7%에 비해 정확도가 떨어지는 것으로 보이지만 위의 데이터는 91% 정도가 일반 별이기 때문에 96.7%의 정확도를 달성한 것이므로 착시 효과가 걷어진 것으로 볼 수 있다.

데이터의 균형을 맞추면 일반 별에 치우쳤던 학습의 균형을 잡으면서 신경망의 추정 성능을 향상시킬 수 있지만, 착시 효과때문에 성능 향상을 오인할 수 있다. 이를 해결하기 위해서 F1 score를 이용해서 성능 향상을 보여줄 수 있다.