EE488 Machine Learning basics and practice

Mini-Assignment 1 보고서

20180155 김준범

Part2 Logistic Regression

Step1. Implement Logistic Regression using Numpy

Step 1-1. Implement Sigmoid Function

Numpy의 exp함수를 사용하면 입력 받은 행렬의 모든 원소에 e^x 연산을 수행한다. 따라서 아래의 8번 셀과 같이 코드를 작성하면 z의 모든 원소에 대한 sigmoid function 계산 결과를 얻을 수 있다. 9번 셀은 8번 셀에서 정의 한 sigmoid 함수를 실험한 것으로 0, 4.5, -6의 값을 넣었을 때 sigmoid 함수의 정의에 따라 잘 계산된 값이 출력되는 것을 확인할 수 있다.

```
[8]  def sigmoid(z):
         ######################### Blank ########################
         sig = 1/(1 + np.exp(-z))
         ########################################################
         return sig
```

```
[9]  # Print out the result of sigmoid function

     print ("sigmoid(0) = " + str(sigmoid(0)))
     print ("sigmoid(4.5) = " + str(sigmoid(4.5)))
     print ("sigmoid(-6) = " + str(sigmoid(-6)))
```

```
sigmoid(0) = 0.5
sigmoid(4.5) = 0.9890130573694068
sigmoid(-6) = 0.0024726231566347743
```

Step 1-2. Implement forward propagation

predict: prediction for each data point x_i; $h(w^T x_i) = \dfrac{1}{1+e^{-w^T x_i}}$

loss: $J(w) = \frac{1}{n}\sum_{i=1}^{n}[-y_i \log\{h(w^T x_i)\} - (1-y_i)\log\{1-h(w^T x_i)\}]$

위 식은 forward propagation에서 예측 값과 오차를 계산하는 식이다. Predict 식을 계산하는 것은, 우선 Numpy의 matmul 함수를 통해 model의 weight와 input의 행렬 곱을 구한 뒤, Step 1-1에서 만든 sigmoid 함수에 넣어 주었다. Loss를 계산하는 것은, input x의 행의 개수를 통해 n을 구한 것을 시작한다. 그리고 Σ 와 Σ 안의 첫 번째 항은 모든 input에 대해 대응되는 label과 계산한 예측 값의 곱이므로 y의 transpose와 예측 값을 matmul 함수를 통해 행렬 곱한 것과 같다. Σ 안의 두 번째 항의 계산 역시 이와 동일하게 transpose와의 matmul을 사용했고 각각의 계산 결과를 loss1, loss2에 저장했다가 둘의 합을 앞서 구한 n으로 나눠 주면 loss를 계산할 수 있다.

```
[10] def forward(x, y, w, eps=1e-8):
        ######################### Blank ########################
        predict = sigmoid(np.matmul(w, x.T))
        n = x.shape[0]
        loss1 = - np.matmul(y.T, np.log(predict + eps))
        loss2 = - np.matmul((np.ones(y.shape) - y).T, np.log(np.ones(predict.shape) - predict + eps))
        loss = (1/n) * (loss1 + loss2)
        ########################################################
        return predict, loss
```

Step 1-3. Implement backward propagation

$$\begin{aligned}\frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w}} &= -\frac{1}{n}\sum_{i=1}^{n}\left[y_i\left(1-h\left(\boldsymbol{w}^T\boldsymbol{x}_i\right)\right)-\left(1-y_i\right)h\left(\boldsymbol{w}^T\boldsymbol{x}_i\right)\right]\boldsymbol{x}_i \\ &= \frac{1}{n}\sum_{i=1}^{n}\left[h\left(\boldsymbol{w}^T\boldsymbol{x}_i\right)-y_i\right]\boldsymbol{x}_i \\ &= \frac{1}{n}(\hat{Y}-Y)X\end{aligned}$$

위 식은 back propagation 과정에서 구해지는 model weight의 gradient이다. 이 식의 핵심은 예측한 label ŷ 와 label y의 차이를 X로 곱한 뒤, n으로 나누면 그것이 model weight의 gradient가 된다는 것이다. 이를 바탕으로 implement한 것은 아래에 있다.

```
[11] def backward(x, y, predict):
        ######################### Blank ########################
        n = x.shape[0]
        grad_w = (1/n) * (np.matmul(predict - y, x))
        ########################################################
        return grad_w
```

Step 1-4. Add bias unit

$$\bar{x} := \begin{bmatrix} x_1^T & 1 \\ x_2^T & 1 \\ \vdots & \\ x_n^T & 1 \end{bmatrix} \in \mathbf{R}^{n\times(d+1)}, \bar{w} := \begin{bmatrix} w \\ b \end{bmatrix} \in \mathbf{R}^{d+1}$$

위 식은 bias unit을 추가하기 위해 input에 1 column을 추가하고, weight w에 b row를 추가하는 것을 보여준다. Numpy에는 c_, r_함수가 있는데 각각 column과 row에 원하는 행렬을 끼워 넣어주는 함수이다. 따라서 size가 nx1인 1 행렬을 만들고 c_함수로 x에 붙여주고, b를 w에 r_로 붙여주면 bias unit을 추가할 수 있게 된다. 이를 implement한 것은 아래에 있다.

```
[12] def bias_unit(x, w, b):
        ######################### Blank ########################
        n = x.shape[0]
        ones = np.ones((n,1))
        x_bar = np.c_[x, ones]
        w_bar = np.r_[w, b]
        ########################################################
        return x_bar, w_bar
```

Step 1-5. Model initialization

초기 모델의 weight는 임의로 설정하고 b의 값을 0으로 설정한다. W의 값은 np.random.normal을 이용해서 아래와 같이 설정해주었다.

```
[13] def initialize_params(X_train, verbose=False):

        ######################### Blank #######################
        w = np.random.normal(0, 1, X_train.shape[1])
        b = 0
        ########################################################

        X_train_bar, w_bar = bias_unit(X_train, w, b) # add bias unit
        if verbose:
            print('Before adding the bias unit')
            print('shape of X_train:, ', X_train.shape)
            print('w: ', w.__repr__())
            print('b: ', b.__repr__(), end='₩n₩n')
            print('After adding the bias unit')
            print('shape of X_train_bar:, ', X_train_bar.shape)
            print('w_bar: ', w_bar.__repr__())

        return X_train_bar, w_bar
```

Step 1-6. Implement accuracy function for computing the accuracy

$$\text{acc: } \frac{1}{n}\sum_{i=1}^{n}\mathbf{1}_{[\hat{y}_i=y_i]} \times 100(\%) \text{ where } \hat{y}_i = \mathbf{1}_{[h(w^T x_i)\geq 0.5]}.$$

위 식은 accuracy를 계산하는 식이다. 예측한 label은 h($w^T x_i \geq 0.5$ 인 경우 1로 간주하여, label과 일치하는 %를 계산하는 것이 원리이다. Label과 예측 값이 일치하는 지 for loop을 돌면서 확인할 수 있지만 이렇게 하는 경우 accuracy 계산 속도가 너무 느리게 된다. 따라서 예측 값이 0.5보다 크거나 같으면 1, 아니면 0인 행렬에 2를 곱하고 모든 요소에 1를 빼, 1과 -1로 구성된 행렬을 만들었다. 또, label 행렬을 2로 곱하고 모든 요소에 1를 빼서, 이전에 만든 행렬과의 inner product를 구했다. 그러면 예측과 label이 일치하는 경우 +1, 아닌 경우 -1이 더 해지게 되어, 이 값에 n을 더하고 2로 나누면 일치하는 것의 개수를 구할 수 있게 된다. 이를 implement한 것은 아래에 있다.

```
[14] def accuracy(predict, y):
        ######################### Blank #######################
        n = predict.size
        pred_label = 2 * (predict >= 0.5) - 1
        inner_product = np.matmul(pred_label.T, 2 * y - 1)
        acc = 100 * ( inner_product + n)/(2 * n)
        ########################################################
        return acc
```

Step 2 Apply to the MNIST Dataset

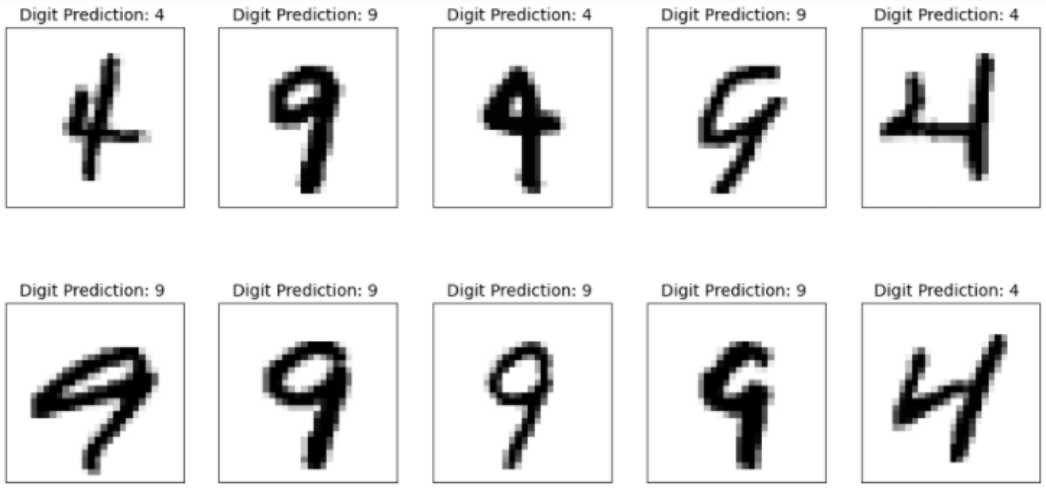
Step 2-1 Implement training code combining all the methods above

모델을 training 하는 과정은 다음과 같이 4가지로 분류될 수 있다. 1) forward 과정을 통해 예측 값과 loss를 계산한다. 2) 예측 값과 label을 통해 accuracy를 계산한다. 3) back propagation을 통해 model weight의 gradient를 계산한다. 4) 이 gradient를 gradient descent method에 적용해 변화된 weight를 계산한다. 각각 과정에 대한 함수는 이미 모두 만들었고 조심해야 할 것은 input으로 X_train_std를 바로 넣어주는 것이 아닌, bias unit이 추가된 X_train_bar과 w_bar을 forward에 넣어주어야 한다는 것이다. 이를 implement한 것은 아래에 있다.

```
######################### Blank ########################
predict, loss = forward(X_train_bar, Y_train, w_bar, eps=1e-8)
train_acc = accuracy(predict, Y_train)
gradient = backward(X_train_bar, Y_train, predict)
w_bar = w_bar - learning_rate * gradient
########################################################
```

이 식이 포함된 셀을 실행한 결과와 예측 결과는 아래에 있다.

```
100%|██████████| 5000/5000 [01:41<00:00, 49.26it/s, accuracy=97.30, loss=0.0774]
train accuracy: 97.32
test accuracy: 96.38
```



Step 3 Apply to the CIFAR10 dataset

Step 3-1 Implement training code combining all the methods above

이 단락에서 implement하는 과정은 Step 2-1과 도일하다. Implement한 결과는 아래에 있다.

```
######################### Blank ########################
predict, loss = forward(X_train_bar, Y_train, w_bar, eps=1e-8)
train_acc = accuracy(predict, Y_train)
gradient = backward(X_train_bar, Y_train, predict)
w_bar = w_bar - learning_rate * gradient
########################################################
```

이 식이 포함된 셀을 실행한 결과는 아래에 있다.

```
100%|██████████| 5000/5000 [03:48<00:00, 21.88it/s, accuracy=74.10, loss=1.6516]
train accuracy: 74.21
test accuracy: 73.80
```

Step 4 Implement Logistic Regression with L2 Regularization

Step 4-1. Implement forward propagation with regularization term

predict: Prediction for each data point x_i : $h(w^T x_i) = \dfrac{1}{1+e^{-w^T x_i}}$

loss: $J(w) = \frac{1}{n}\sum_{i=1}^{n}[-y_i \log\{h(w^T x_i)\} - (1-y_i)\log\{1-h(w^T x_i)\}] + \frac{\lambda}{2n}\|w\|_2^2$

위 식은 L2 Regularization을 포함한 predict와 loss를 계산하는 식이다. Step 1-2와 매우 유사한 구조이며 loss의 식에 regularization 항이 추가된 것만이 다르다. 따라서 1-2의 코드를 그대로 쓰고 regularization 항 만 loss에 더해주는 식으로 implementation을 구상했다. Regularization항은 norm의 제곱을 계산해야 하기 때문에 np.linalg.norm을 활용해 norm을 계산하고 제곱하여 loss에 더 해주었다.

```
[20] from numpy.linalg import norm

     def forward_with_regularization(x, y, w, lambda_, eps=1e-8):
         ######################### Blank ########################
         predict = sigmoid(np.matmul(w, x.T))
         n = x.shape[0]
         loss1 = - np.matmul(y.T, np.log(predict + eps))
         loss2 = - np.matmul((np.ones(y.shape) - y).T, np.log(np.ones(predict.shape) - predict + eps))
         loss = (1/x.shape[0]) * (loss1 + loss2 + (lambda_/2) * (np.linalg.norm(w) ** 2))
         ########################################################
         return predict, loss
```

Step 4-2. Implement backward propagation

$$\begin{aligned}\frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w}} &= -\frac{1}{n}\sum_{i=1}^{n}\left[y_i\left(1-h\left(\boldsymbol{w}^T\boldsymbol{x}_i\right)\right)-\left(1-y_i\right)h\left(\boldsymbol{w}^T\boldsymbol{x}_i\right)\right]\boldsymbol{x}_i+\frac{\lambda}{n}w \\ &= \frac{1}{n}\sum_{i=1}^{n}\left[h\left(\boldsymbol{w}^T\boldsymbol{x}_i\right)-y_i\right]\boldsymbol{x}_i+\frac{\lambda}{n}w \\ &= \frac{1}{n}(\hat{Y}-Y)X+\frac{\lambda}{n}w\end{aligned}$$

위 식은 L2 Regularization을 포함한 model weight의 gradient를 계산하는 식이다. Step 1-3의 식과 비교하면 마찬가지로 뒤에 regularization항이 추가된 것 외에는 모두 일치한다. 따라서 1-3의 implementation 뒤에 regularization항을 단순히 더해 주었다.

```
[21] def backward_with_regularization(x, y, w, predict, lambda_):
         ######################### Blank ########################
         n = x.shape[0]
         grad_w = (1/n) * (np.matmul(predict - y, x) + lambda_ * w)
         ########################################################
         return grad_w
```

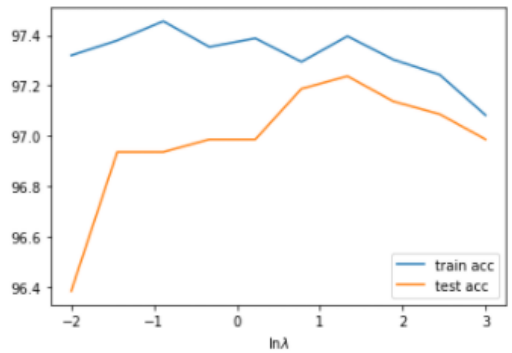
Step 4-3. Training Regularized logistic regression with different λ (Apply to the MNIST dataset)

해당 단락의 implementation은 2-1과 3-1과 거의 동일하다. 한 가지 다른 점은 함수 step 4-1과 step 4-2에서 만든 regularization을 포함한 forward와 backward 함수를 사용해야 하는 것이다. 해당 implementation은 아래와 같다.

```
######################## Blank ########################
predict, loss = forward_with_regularization(X_train_bar, Y_train, w_bar, lambda_, eps=1e-8)
train_acc = accuracy(predict, Y_train)
gradient = backward_with_regularization(X_train_bar, Y_train, w_bar, predict, lambda_)
w_bar = w_bar - learning_rate * gradient
#######################################################
```

이 식을 포함한 코드 셀을 실행한 결과는 아래에 있다.

```
100%|██████████| 5000/5000 [00:58<00:00, 86.10it/s, accuracy=97.29, loss=0.0812]
  0%|          | 9/5000 [00:00<00:59, 83.52it/s, accuracy=51.05, loss=5.9100]train accuracy: 97.32
test accuracy: 96.38
100%|██████████| 5000/5000 [01:00<00:00, 83.09it/s, accuracy=97.37, loss=0.0825]
  0%|          | 8/5000 [00:00<01:03, 78.33it/s, accuracy=43.19, loss=2.5830]train accuracy: 97.38
test accuracy: 96.94
100%|██████████| 5000/5000 [00:58<00:00, 85.68it/s, accuracy=97.43, loss=0.0852]
  0%|          | 9/5000 [00:00<01:00, 83.16it/s, accuracy=59.98, loss=2.2368]train accuracy: 97.46
test accuracy: 96.94
100%|██████████| 5000/5000 [00:57<00:00, 86.77it/s, accuracy=97.35, loss=0.0925]
  0%|          | 9/5000 [00:00<01:02, 80.36it/s, accuracy=49.71, loss=2.8139]train accuracy: 97.35
test accuracy: 96.99
100%|██████████| 5000/5000 [00:59<00:00, 84.64it/s, accuracy=97.38, loss=0.0994]
  0%|          | 8/5000 [00:00<01:04, 76.84it/s, accuracy=51.79, loss=4.1822]train accuracy: 97.39
test accuracy: 96.99
100%|██████████| 5000/5000 [00:59<00:00, 83.66it/s, accuracy=97.30, loss=0.1045]
  0%|          | 9/5000 [00:00<00:59, 84.57it/s, accuracy=40.22, loss=2.8347]train accuracy: 97.29
test accuracy: 97.19
100%|██████████| 5000/5000 [00:59<00:00, 84.09it/s, accuracy=97.40, loss=0.1058]
  0%|          | 9/5000 [00:00<01:00, 82.06it/s, accuracy=52.21, loss=3.1562]train accuracy: 97.40
test accuracy: 97.24
100%|██████████| 5000/5000 [00:59<00:00, 83.48it/s, accuracy=97.29, loss=0.1040]
  0%|          | 8/5000 [00:00<01:06, 74.80it/s, accuracy=42.92, loss=4.2759]train accuracy: 97.30
test accuracy: 97.14
100%|██████████| 5000/5000 [01:03<00:00, 78.32it/s, accuracy=97.24, loss=0.1068]
  0%|          | 7/5000 [00:00<01:11, 69.89it/s, accuracy=51.78, loss=3.7943]train accuracy: 97.24
test accuracy: 97.09
100%|██████████| 5000/5000 [01:02<00:00, 80.13it/s, accuracy=97.08, loss=0.1168]
train accuracy: 97.08
test accuracy: 96.99
```



위 그래프를 보면 test accuracy는 regularization parameter가 증가함에 따라서 점점 증가하다가, 최고점을 찍은 뒤 감소하는 것을 확인 할 수 있다. 이는 너무 regularization의 기여도가 커지며 model이 underfitting 되었기 때문이다.