인공지능 과제 1 보고서

B635076 김승연

* singlelayer.py 코드

# B635076 김승연  
**import** numpy **as** np  
  
  
**class** singleLayer:  
 **def** \_\_init\_\_(self, W, Bias): # 제공. 호출 시 작동하는 생성자  
 self.W = W  
 self.B = Bias  
  
 **def** SetParams(self, W\_params, Bias\_params): # 제공. W와 Bias를 바꾸고 싶을 때 쓰는 함수  
 self.W = W\_params  
 self.B = Bias\_params  
  
 **def** ScoreFunction(self, X): # \Score값 계산 -> 직접작성  
 # 3.2  
 """  
 매개변수로 받은 X는 이미지 데이터임  
 X \* W(weight) + B(bias) 를 계산하여 결과값을 ScoreMatrix 에 저장하여 이를 리턴함  
 X는 [이미지 개수, 784]인 배열, W는 [784, 10(label 개수)]인 배열이므로 np.dot() 함수를 이용하여 곱셈을 하고  
 가중치인 B를 더해주면 결과로 나올 ScoreMatrix는 [이미지 개수, 10]인 배열임  
 예를 들어 ScoreMatrix의 3번째 행에서 가장 큰값을 가진 열(index)이 6이라면, 3번째 이미지가 숫자 6일 것이라 추측한 것임  
 """  
  
 ScoreMatrix = np.dot(X, self.W) + self.B  
  
 **return** ScoreMatrix  
  
 **def** Softmax(self, ScoreMatrix): # 제공.  
 **if** ScoreMatrix.ndim == 2:  
 temp = ScoreMatrix.T  
 temp = temp - np.max(temp, axis=0) # 가장 점수가 높은 인덱스의 값으로 뺌  
 y\_predict = np.exp(temp) / np.sum(np.exp(temp), axis=0)  
 **return** y\_predict.T  
 temp = ScoreMatrix - np.max(ScoreMatrix, axis=0)  
 expX = np.exp(temp)  
 y\_predict = expX / np.sum(expX)  
 **return** y\_predict # 사진마다의 y\_predict의 모든 데이터의 합은 1임  
  
 **def** LossFunction(self, y\_predict, Y): # Loss Function을 구하십시오 -> 직접 작성  
 # 3.3  
 """  
 **:param** y\_predict:  
 Softmax 함수를 이용해서 얻은 결과값임.  
 처음에 ScoreFuction을 이용해 ScoreMatrix를 계산했는데, 이 값들을 0~1 사이의 숫자, 즉 확률로 표현한 값임.  
 **:param** Y: 이미지 데이터의 정답 레이블임. one-hot 배열로, 정답인 인덱스에만 1이고 나머지는 0임  
 **:return**: 사진의 정답값과 y\_predict에서 가장 큰 값이 있는 인덱스가 틀릴 경우, 이 정답을 얼마나 제대로 맞추지 못했는지를 알려주는 척도임  
 즉, 리턴값인 loss가 작을 수록 잘 추측한 것임  
  
 Cross-Entropy를 사용해 loss 계산하는 공식은 -1 \* np.sum(Y \* np.log(y\_predict))임  
 Y에서 답이 아닌 인덱스의 값은 모두 0이므로 곱셈을 하면 결국 0임.  
 그래서 실질적으로 np.sum을 해서 나오는 값은 이미지의 실제 정답이 3이라 하면 np.log(y\_precict(3))일 것임  
 """  
  
 epsilon = 1e-7  
 y\_predict += epsilon  
 # y\_predict 내 어떤 값이 너무너무 작은 값일 경우에(0에 수렴하는 값) 컴퓨터는 그냥 0으로 인식을 하기 때문에 log 연산을 하면  
 # 'RuntimeWarning: divide by zero encountered in log' 라는 경고가 생김. 이를 방지하기 위해 아주 작은 수를 더해주었음  
  
 **if** Y.ndim == 2: # 두 장 이상의 이미지를 매개변수로 받았을 때 Y와 y\_predict의 shape는 (개수, 10)임  
 loss\_arr = -1 \* np.sum(Y \* np.log(y\_predict), axis=1)  
 loss = np.average(loss\_arr) # 이미지가 여러 개일 경우, 이미지마다 나온 loss의 평균을 내야 함  
 **else**: # 단 한 장의 이미지만 매개변수로 받았을 때 Y와 y\_predict의 shape는 (10, )임  
 loss = -1 \* np.sum(Y \* np.log(y\_predict))  
  
 **return** loss  
  
 **def** Forward(self, X, Y): # ScoreFunction과 Softmax, LossFunction를 적절히 활용해 y\_predict 와 loss를 리턴시키는 함수. -> 직접 작성  
 # 3.4  
 """  
 위에서 작성한 함수들을 사용하여 y\_predict와 loss를 리턴함  
 이 Forward 함수는 Optimization 함수에서 불러지는데,  
 이 때 main.py에서 SN.Optimization(x\_train, y\_train, x\_test, y\_test)가 불러짐  
 그러므로 이 Forward 함수의 매개변수 X, Y는 Optimization 함수에서 받았던 매개변수(이미지, 정답레이블)임  
 """  
 tmp = self.ScoreFunction(X)  
 y\_predict = self.Softmax(tmp)  
 loss = self.LossFunction(y\_predict, Y)  
  
 **return** y\_predict, loss  
  
 **def** delta\_Loss\_Scorefunction(self, y\_predict, Y): # 제공.dL/dScoreFunction  
 delta\_Score = y\_predict - Y  
 **return** delta\_Score  
  
 **def** delta\_Score\_weight(self, delta\_Score, X): # 제공. dScoreFunction / dw .  
 delta\_W = np.dot(X.T, delta\_Score) / X[0].shape  
 **return** delta\_W  
  
 **def** delta\_Score\_bias(self, delta\_Score, X): # 제공. dScoreFunction / db .  
 delta\_B = np.sum(delta\_Score) / X[0].shape  
 **return** delta\_B  
  
 # delta 함수를 적절히 써서 delta\_w, delta\_b 를 return 하십시오.  
 **def** BackPropagation(self, X, y\_predict, Y):  
 # 3.5  
 """  
 analytic gradient 방법에 입각해 dL/dw, dL/db를 계산하는 함수  
 제공된 delta\_함수들을 이용하여 계산함  
 이 리턴값들은 이후에 loss값이 작아지도록 Weight값과 Bias값을 수정할 때 필요함  
 """  
 delta\_score = self.delta\_Loss\_Scorefunction(y\_predict, Y) # [이미지 개수, 10]의 형태  
 **if** X.ndim == 1: # 한개의 이미지를 대상으로 계산할 경우, 배열이 [784,]의 형태이므로 [1, 784]로 바꾸어 계산할 수 있도록 함  
 X = X.reshape(1, 784)  
 delta\_W = self.delta\_Score\_weight(delta\_score.reshape([X.shape[0], 10]), X)  
 # 한 개의 이미지를 대상으로 계산할 땐 delta\_score의 shape가 [10, ]임.  
 # 그래서 한 개의 이미지일 때도 dot 연산이 가능하도록 매개변수로 delta\_score.reshape([X.shape[0], 10])를 전달했음. X.shape[0]는 이미지 개수임  
 delta\_B = self.delta\_Score\_bias(delta\_score, X)  
  
 **return** delta\_W, delta\_B  
  
 # 정확도를 체크하는 Accuracy 제공  
 **def** Accuracy(self, X, Y):  
 y\_score = self.ScoreFunction(X)  
 y\_score\_argmax = np.argmax(y\_score, axis=1)  
 **if** Y.ndim != 1: Y = np.argmax(Y, axis=1)  
 accuracy = 100 \* np.sum(y\_score\_argmax == Y) / X.shape[0]  
 **return** accuracy  
  
 # Forward와 BackPropagationAndTraining, Accuracy를 사용하여서 Training을 epoch만큼 시키고, 10번째 트레이닝마다  
 # Training Set의 Accuracy 값과 Test Set의 Accuracy를 print 하십시오  
  
 **def** Optimization(self, X\_train, Y\_train, X\_test, Y\_test, learning\_rate=0.01, epoch=100):  
 """  
 **:param** X\_train: 훈련 이미지  
 **:param** Y\_train: 훈련 이미지의 정답 레이블  
 **:param** X\_test: 시험 이미지  
 **:param** Y\_test: 시험 이미지의 정답 레이블  
 **:param** learning\_rate: loss의 값을 줄이기 위해 최적의 W와 b의 값을 조정할 때 얼마만큼 변화량을 주어 조정할 건지에 대한 수치임  
 **:param** epoch: 훈련을 epoch번 시킴. 전체 데이터셋을 한번 돈 것을 1epoch이라 함  
 이 함수에서는 그동안 작성했던 함수들을 이용하여 이미지 트레이닝을 시키고,  
 10번의 epoch마다 loss와 Training\_set의 Accuracy, Test\_set의 Accuracy를 출력  
 """  
 **for** i **in** range(epoch):  
 # 3.6  
 y\_predict, loss = self.Forward(X\_train, Y\_train)  
 del\_w, del\_b = self.BackPropagation(X\_train, y\_predict, Y\_train)  
 change\_w = self.W + (-1 \* learning\_rate \* del\_w)  
 change\_b = self.B + (-1 \* learning\_rate \* del\_b)  
 self.SetParams(change\_w, change\_b) # weight와 bias를 새롭게 조정함  
  
 # 함수 작성  
 **if** i % 10 == 0:  
 # 3.6 Accuracy 함수 사용  
 # 훈련 이미지 & 시험 이미지에 대해 얼마나 답을 잘 맞추는지 정확도 각각 계산  
 train\_accuracy = self.Accuracy(X\_train, Y\_train)  
 test\_accuracy = self.Accuracy(X\_test, Y\_test)  
 print(i, **"번째 트레이닝"**)  
 print(**'현재 Loss(Cost)의 값 : '**, loss)  
 print(**"Train Set의 Accuracy의 값 : "**, train\_accuracy)  
 print(**"Test Set의 Accuracy의 값 :"**, test\_accuracy)

* 퀴즈

1. np.max(temp, asix=0) 은 각각의 이미지에서 가장 큰 점수값을 말한다. 만약 이 값을 빼주지 않는다면, 지수함수인 exp() 연산을 했을 때 너무나 큰 값이 반환되기 때문에 오버플로우가 발생할 수 있음. 이를 방지하기 위해 np.max(temp, asix=0)를 빼준 것임.
2. **def** one\_hot\_encoding(i):  
    one\_hot = np.zeros(4)  
    one\_hot[i] = 1  
    **return** one\_hot
3. learning rate를 100배 하면 정확도가 꾸준히 올라가는 것이 아니라, 높아졌다가 낮아졌다가 하면서 들쑥날쑥한 정확도를 보인다. 이와 같은 결과를 보이는 이유는 최적의 W값으로 가기 위해서 loss값이 작은 쪽으로 경사를 따라 내려가야 하는데 learning rate를 100배 하면 너무 많이 이동하여 loss가 큰 쪽으로 다시 넘어가게 될 수도 있기 때문이다.
4. **def** Numerical\_Gradient(self, X, Y, h=0.000001):  
    dW = np.zeros(self.W.shape) # W와 같은 크기의 배열을 생성하는데, 우선 다 0으로 채움.  
     
    loss = self.Forward(X, Y)[1] # 원래 W를 이용한 loss 값  
     
    it = np.nditer(self.W, flags=[**'multi\_index'**], op\_flags=[**'readwrite'**])  
    # 다차원 배열 순회에 이용하는 iterator. 다차원 배열을 접근할 것이고, 읽고 쓰는 행위를 수행할 것이라는 의미  
     
    **while not** it.finished:  
    ix = it.multi\_index  
    old\_W = self.W[ix] # 이후에 원래 W값으로 되돌려 놓기 위해 미리 저장해 두었음  
    self.W[ix] = old\_W + h # 다차원배열을 순서대로 접근하면서 h를 더해줌  
    loss\_h = self.Forward(X, Y)[1] # h 더한 배열을 토대로 loss 계산  
    self.W[ix] = old\_W # W값 다시 원래대로 되돌려 놓음  
    dW[ix] = (loss\_h - loss) / h # 변화량을 계산해서 dW에 저장  
    it.iternext() # 다음 인덱스로 이동~  
     
    **return** dW