**Assignment 2**

**조선해양공학과 2017-35978 이성준**

문제: XOR을 배우는 2-layer perceptron을 학습시켜라. (python과 numpy만 이용)

Loss function, optimization 방법 등을 바꿔가면서 총 두 가지 방법으로 풀이해 보았다.

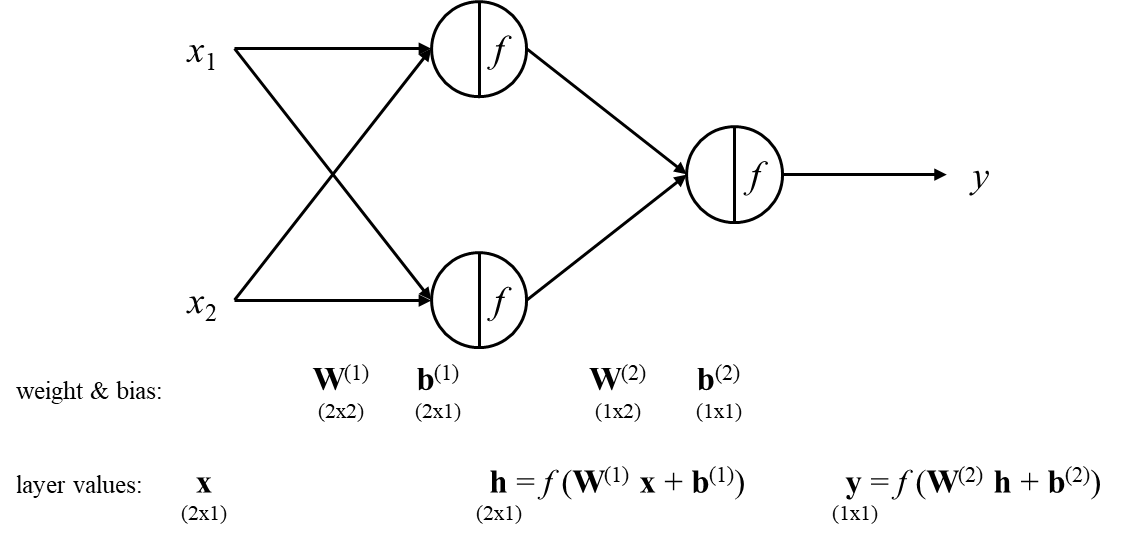
**1. 풀이 #1**

* 1st layer’s activation function: **sigmoid**
* 2nd layer’s activation function (output function): **sigmoid**
* Loss function: **mean square error**
* Optimization method: **gradient descent method**

학습에 앞서 아래는 numpy를 import 하고, 데이터를 준비하는 코드이다. Sigmoid와 ReLU 함수도 여기서 정의하였다.

|  |
| --- |
| import numpy as np  import matplotlib.pyplot as plt  %matplotlib inline  # data for learning XOR [x1, x2, y]  d1 = [0, 0, 0]  d2 = [1, 0, 1]  d3 = [0, 1, 1]  d4 = [1, 1, 0]  ndata = 4  # input data  X = np.array([d1[0:2], d2[0:2], d3[0:2], d4[0:2]])  X = X.reshape(4, 2, 1)  # ground truth data  Y = np.array([d1[2], d2[2], d3[2], d4[2]])  Y = Y.reshape(4, 1, 1)  # activation function  sigmoid = lambda x: 1 / (1+ np.exp(-x))  sigmoid\_dot = lambda y: y \* (1-y)  relu = lambda x: np.maximum(x, 0)  relu\_dot = lambda x: x >= 0 |

Multi-layer perceptron의 구조는 아래와 같이 하였다. Layer는 2개, 첫째 layer의 note 수는 2개, 둘째 layer (즉, output layer)의 node 수는 1개이다.



아래는 weight와 bias들을 초기화 한 것이다. Xavier initialization을 적용하였다.

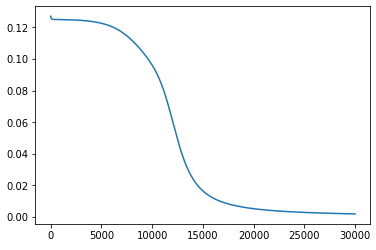
|  |
| --- |
| ####################################  # initialize weight and bias  W1 = np.random.randn(2, 2) / np.sqrt(2) # xavier initialization  b1 = np.zeros((2, 1))  W2 = np.random.randn(1, 2) / np.sqrt(2) # xavier initialization  b2 = np.zeros((1, 1)) |

다음은 학습 코드이다. Learning rate 0.1로 하여 gradient descent 방법을 사용하였고, epoch를 30000까지 진행하였다. 손실함수는 mean square error를 사용하였다.

|  |
| --- |
| ####################################  # training  learning\_rate = 0.1  epoch = 30000  loss\_hist = []  for i in range(epoch):    # hidden layer (1st PERCEPTRON layer)  # shape = (ndata, nNode=2, 1)  H = sigmoid( np.matmul(W1, X) + b1 )    # output layer (2nd PERCEPTRON layer)  # shape = (ndata, nNode=1, 1)  Y\_ = sigmoid( np.matmul(W2, H) + b2 )    # loss function (MSE)  loss = ( 0.5 \* (Y - Y\_) \*\* 2 ).mean()  loss\_hist.append(loss)    # backpropagation  dLdW2 = np.matmul((Y\_ - Y) \* sigmoid\_dot(Y\_), H.transpose(0, 2, 1)) # shape = (ndata, 1, 2)  dLdb2 = (Y\_ - Y) \* sigmoid\_dot(Y\_) # shape = (ndata, 1, 1)    dLdH = np.matmul(W2.transpose(), (Y\_ - Y) \* sigmoid\_dot(Y\_)) # shape = (ndata, 2, 1)  dLdW1 = np.matmul(dLdH \* sigmoid\_dot(H), X.transpose(0, 2, 1)) # shape = (ndata, 2, 2)  dLdb1 = dLdH \* sigmoid\_dot(H) # shape = (ndata, 2, 1)    # gradient  gradient = np.array([dLdW1.mean(axis=0), dLdW2.mean(axis=0), dLdb1.mean(axis=0), dLdb2.mean(axis=0)])    # gradient descent  W1, W2, b1, b2 = np.array([W1, W2, b1, b2]) - learning\_rate \* gradient |

학습과정에서의 loss 변화를 아래 코드를 통해 그래프로 표시하였다.

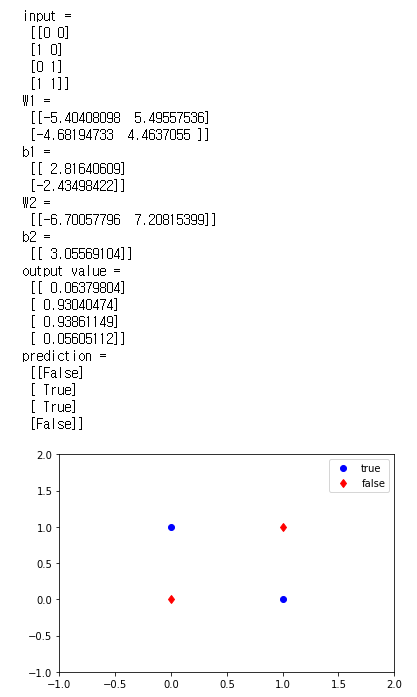
|  |
| --- |
| loss\_plot = plt.plot(loss\_hist)  plt.show(loss\_plot) |



Gradient descent를 사용한 현재 case에서는 약 20,000번 정도의 epoch에서 0 근처로 수렴하는 것을 볼 수 있다.

아래는 학습이 완료된 모델을 사용하여 XOR 예측결과를 나타내는 코드이다. 모델의 최종 weight, bias 값도 함께 출력하였다.

|  |
| --- |
| ####################################  # prediction test after training  H = sigmoid( np.matmul(W1, X) + b1 )  Y\_ = sigmoid( np.matmul(W2, H) + b2 )  print('input = \n', X.reshape(4, 2))  print('W1 = \n', W1)  print('b1 = \n', b1)  print('W2 = \n', W2)  print('b2 = \n', b2)  print('output value = \n', Y\_.reshape(4, 1))  print('prediction = \n', Y\_.reshape(4, 1) > 0.5)  Y\_p = Y\_.reshape(4) > 0.5  plt.plot(X[Y\_p, 0, 0], X[Y\_p, 1, 0], 'bo')  plt.plot(X[Y\_p == False, 0, 0], X[Y\_p == False, 1, 0], 'rd')  plt.xlim([-1, 2])  plt.ylim([-1, 2])  plt.legend(['true', 'false'])  plt.show() |



x1, x2 중 하나만 1일 때에는 참(파란색 점), 둘 다 0이거나 1일 때는 거짓(빨간색 점)으로 나타나고 있다. 즉, XOR 연산이 성공적으로 수행되는 것을 볼 수 있다.

**2. 풀이 #2**

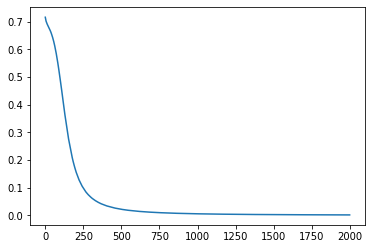
* 1st layer’s activation function: **ReLU**
* 2nd layer’s activation function (output function): **sigmoid**
* Loss function: **Cross Entropy Cost Function (CECF)**
* Optimization method: **ADAM optimization method**

위와 같이 방법을 달리하여 수행해 보았다. 모델에서의 변화는 첫 번째 layer의 활성화 함수를 ReLU로 바꾸었다. 출력 함수는 0~1 사이로 예측해야 하므로 sigmoid를 그대로 사용하였다. 손실 함수는 sigmoid output에서의 convexity를 보장하는 cross entropy cost function을 이용하였으며, 최적화 방법으로는 수렴속도가 빠른 ADAM 방법을 사용하였다.

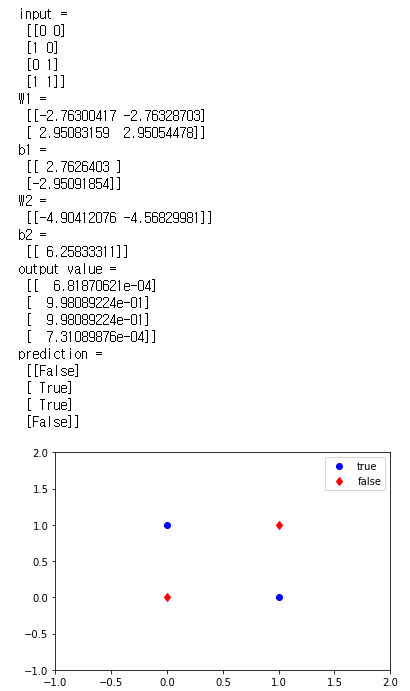
아래는 이를 적용하여 학습하는 코드이며, 앞선 “풀이#1”과 동일한 초기화 코드 및 결과 가시화 코드는 생략하였다.

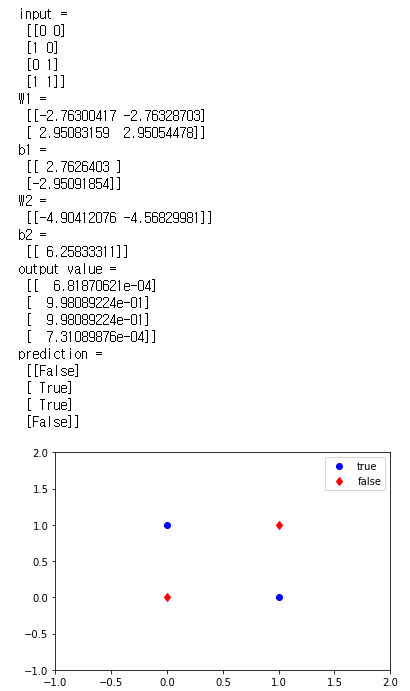
|  |
| --- |
| ####################################  # training  learning\_rate = 0.01  epoch = 2000  loss\_hist = []  # ADAM hyper-parameters  beta1 = 0.9  beta2 = 0.999  e = 1E-8  m = np.array([np.zeros(W1.shape), np.zeros(W2.shape), np.zeros(b1.shape), np.zeros(b2.shape)])  v = np.array([np.zeros(W1.shape), np.zeros(W2.shape), np.zeros(b1.shape), np.zeros(b2.shape)])  for i in range(epoch):    # hidden layer (1st PERCEPTRON layer)  # shape = (ndata, nNode=2, 1)  Hin = np.matmul(W1, X) + b1  H = relu( Hin )    # output layer (2nd PERCEPTRON layer)  # shape = (ndata, nNode=1, 1)  Y\_in = np.matmul(W2, H) + b2  Y\_ = sigmoid( Y\_in )    # loss function (Cross Entropy)  loss = -(Y \* np.log(Y\_) + (1 - Y) \* np.log(1 - Y\_)).mean()  loss\_hist.append(loss)    # backpropagation  dLdW2 = np.matmul((Y\_ - Y), H.transpose(0, 2, 1)) # shape = (ndata, 1, 2)  dLdb2 = (Y\_ - Y) # shape = (ndata, 1, 1)    dLdH = np.matmul(W2.transpose(), (Y\_ - Y)) # shape = (ndata, 2, 1)  dLdW1 = np.matmul(dLdH \* relu\_dot(Hin), X.transpose(0, 2, 1)) # shape = (ndata, 2, 2)  dLdb1 = dLdH \* relu\_dot(Hin) # shape = (ndata, 2, 1)    # gradient  gradient = np.array([dLdW1.mean(axis=0), dLdW2.mean(axis=0), dLdb1.mean(axis=0), dLdb2.mean(axis=0)])    #####################  # ADAM optimization  # update momentum  m = beta1 \* m + (1 - beta1) \* gradient    # update RMSProp  v = beta2 \* v + (1 - beta2) \* gradient\*\*2    # unbias  t = i + 1  m\_hat = m / (1 - beta1\*\*t)  v\_hat = v / (1 - beta2\*\*t)    # update weights  W1, W2, b1, b2 = np.array([W1, W2, b1, b2]) - learning\_rate / (v\_hat + e)\*\*0.5 \* m\_hat |

아래의 그림은 학습을 진행하는 과정에서 loss의 변화를 그래프로 나타낸 것이다. 약 500 번째 epoch 근처에서 0과 매우 근접하게 수렴하는 것을 확인할 수 있다. 앞서 “풀이#1”보다 작은 learning rate를 사용했음에도(0.1 vs 0.01) 훨씬 빠르게 수렴하는 것을 알 수 있다. (20,000 vs 500)



학습이 완료된 모델의 최종 weight, bias 값과, 이 모델을 사용한 예측 결과를 출력하면 아래와 같다. Output value를 보면 매우 높은 confidence로 true와 false를 예측하고 있다.





마찬가지로 예측 결과를 가시화하면 위의 그래프와 같고, XOR 연산이 정확하게 수행되는 것을 볼 수 있다.

**3. 결론**

2-layer perceptron을 사용하면 XOR 문제도 성공적으로 수행할 수 있다는 것을 확인하였다. 다만 방법에 따라 학습의 효율성은 달라질 수 있는데, ReLU 활성화 함수, cross entropy 손실 함수, ADAM 최적화 방법을 사용한 “풀이 #2”가 훨씬 빠르고 정확하게 학습하는 것을 확인해 볼 수 있었다.