Kiana Mills

Psych 186B – Hw 6

11/16/19

*In this model you will create a neural network simulation that performs a parity (odd/even) judgment on the input units: if the # of input units turned on is even then the output is 1, otherwise the output is zero.*

1. **Set up the parameters of the model:** learning rate (lrate): for this problem, a learning rate of around 1 should be fine; number of input units: 8; number of hidden units: 3; number of output units: 1

lrate = 1

input\_units = 8

hidden\_units = 3

output\_units = 1

num\_patterns = 8

2. **Set up the patterns that the network will be trained on.** This should be a matrix of ones and zeros, with the number of rows equal to the number of input units. The number of columns represents the number of patterns that the network will be trained on; for this exercise, 8 patterns should be sufficient. You can create these patterns by rounding random uniform variates created using rand().

train = np.round(randInitialWeights(input\_units, num\_patterns) + 0.5)

patterns = []

for i in range(train.shape[0]):

patterns.append(train[:,i])

3**. Determine the desired output for each pattern**, which is 1 if the number of ones in the input is even and zero if the number is odd. Hint: you can use the mod() function to determine whether the number of on-elements in odd or even.

desired\_output = targetIs(train)

4. **Create two matrices for the weights** that connect the input to hidden units (w\_fg), and hidden units to the output unit (w\_gh). Fill these matrices with uniform random numbers between –0.5 and 0.5.

w\_fg = randInitialWeights(hidden\_units, input\_units)

w\_gh = randInitialWeights(output\_units, hidden\_units)

5. **Create a loop that will continue to iterate the model until one of two conditions is met.**

while epoch <= num\_epochs:

# For each input pattern

for p in range(num\_patterns):

pattern = train[:,p]

# a-e

input\_to\_hidden = w\_fg.dot(pattern) # pass the activation from the input units to the hidden units

input\_to\_hidden.reshape((1,3))

hidden\_activation = sigmoid(input\_to\_hidden ) # determine hidden unit activation

input\_to\_output = np.dot(w\_gh, hidden\_activation) # pass activation from hidden to output

output\_activation = sigmoid(input\_to\_output) # determine output activation

error = desired\_output - output\_activation

# Shorthands for convenience

t = desired\_output[p]

h = output\_activation

g = hidden\_activation

Wgh\_g = input\_to\_output

Wfg\_f = input\_to\_hidden

**# Determine and Apply the weight changes (dw) \*\***

dw\_fg = lrate \* np.dot(np.diag(f\_prime(Wfg\_f)),w\_gh.T) \* (t-h) \* f\_prime(Wgh\_g) \* pattern

w\_fg = w\_fg + dw\_fg

g = g.reshape(1,3)

dw\_gh = lrate \* np.dot(np.diag(f\_prime(Wgh\_g))\*(t-h), g)

w\_gh = w\_gh + dw\_gh

epoch += 1

# Compute SSE over all patterns

# first compute a-e again with the new w\_fg and w\_gh

input\_to\_hidden = w\_fg.dot(train) # pass the activation from the input units to the hidden units

hidden\_activation = activation\_fn(input\_to\_hidden, 3, 8) # determine hidden unit activation

input\_to\_output = np.dot(w\_gh, hidden\_activation) # pass activation from hidden to output

output\_activation = activation\_fn(input\_to\_output, 1, 8) # determine output activation

output\_error = desired\_output - output\_activation

SSE = np.trace(np.dot(output\_error.T,output\_error))

SSE\_vec.append(SSE)

if epoch % 10 == 0:

print("Epoch: ", epoch)

print("SSE: ", SSE)

if SSE <= 0.01:

break

6. **Determine the weight changes (dw) for** each layer of connections. Apply the weight changes:

**This is shown above \*\***

**The Sigmoid Function and its Gradient (f\_prime())**

# The Sigmoid Function ( a smooth, monotonically increasing, differentiable function )

def sigmoid(A):

return 1./(1+np.exp(-A))

# Applies the sigmoid function element-wise

def activation\_fn(input\_pattern, dim, pattern):

output = np.zeros((dim, pattern))

for i in range(dim):

for j in range(dim):

output[i,j] = sigmoid(input\_pattern[i][j])

return output

# Derivative of the sigmoid function

def f\_prime(A):

return (np.exp(-A) / np.square((1+np.exp(-A))))

**7. Plotting SSE**

I ran into some issues plotting SSE and found that my SSE was converging, but not to zero initially. But I managed to get an appropriate looking graph eventually ☺

A screenshot of a cell phone

Description automatically generated