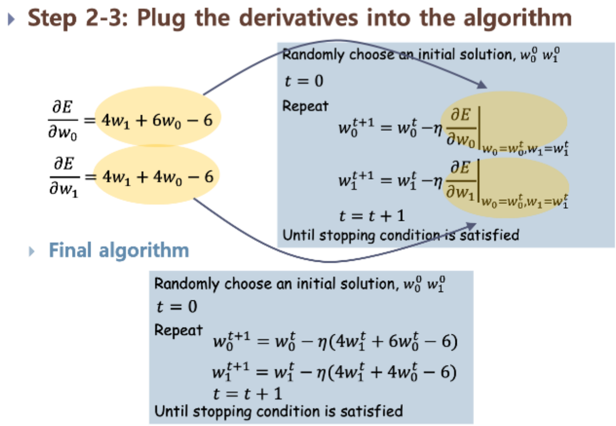
2020310083 손형준

1. Make a python program of final algorithm



- hyperparameters
  - learning rate: 0.01
  - iteration: 1000

```python
import numpy as np

# Gradient of E with respect to w0 and w1
def calc_gradient(w0, w1):
    grad_w0 = 4 * w1 + 6 * w0 - 6
    grad_w1 = 4 * w1 + 4 * w0 - 6
    return np.array([grad_w0, grad_w1])

# Gradient Descent Algorithm
def gradient_descent_ver1(learning_rate, iteration=1000):
    # Randomly initialize w0 and w1
    w = np.random.rand(2)

    t = 0
    while t < iteration:

        # Calculate the gradient at the current w
        grad = calc_gradient(w[0], w[1])

        # Update w
        new_w = w - learning_rate * grad
        w = new_w
        t += 1

    return w, t

# Run the gradient descent algorithm
learning_rate = 0.01
solution, iterations = gradient_descent_ver1(learning_rate)
print(f"""Solution: w0 = {solution[0]}, w1 = {solution[1]}""")
```

```
Solution: w0 = 9.50964156643118e-05, w1 = 1.4998782027544835
```

2. Make Python Program

$$D = \{(x,t) | (-1,1), (0,1), (1,1), (1,0)\}$$

Qudaratic function optimization

At the point which minimize $E(w) = \sum_{i=1}(f(x_i) - t_i)^2$

a) $f(x) = w_1 x + w_0$

b) $f(x) = w_1 cos(\pi x) + w_0$

the process are following:

```python
from sympy import symbols, solve

# given data points (x, t)
given_data = np.array([(-1, 1), (0, 1), (1, 1), (1, 0)])

w0, w1= symbols('w0 w1')
```

```
Error_function = sum([(w0 + w1*x - t)**2 for x, t in given_data])
Error_function = Error_function.expand()

# redefine calc_gradient function
# differentiate Error_function with respect to w0 and w1
def calc_gradient(w0, w1):
    grad_w0 = 8*w0 + 2*w1 - 6
    grad_w1 = 2*w0 + 6*w1
    return np.array([grad_w0, grad_w1])


print(f"""(a)
E(w0, w1) = {Error_function}
dE(w0, w1)/dw0 = {calc_gradient(w0, w1)[0]}
dE(w0, w1)/dw1 = {calc_gradient(w0, w1)[1]}\n""")

# Gradient Descent Algorithm
gradient_descent_ver1(learning_rate)
solution, iterations = gradient_descent_ver1(learning_rate)
print(f"""Solution: w0 = {solution[0]}, w1 = {solution[1]}
best fit f(x) = ({solution[0]}) + ({solution[1]}) * x\n""")

Error_function = sum([(w0 + w1*np.cos(np.pi*x) - t)**2 for x, t in given_data])
Error_function = Error_function.expand()

# redefine calc_gradient function
# differentiate Error_function with respect to w0 and w1
def calc_gradient(w0, w1):
    grad_w0 = 8*w0 - 4.0*w1 - 6
    grad_w1 = -4.0*w0 + 8.0*w1 + 2.0
    return np.array([grad_w0, grad_w1])

print(f"""\n(b)
E(w0, w1) = {Error_function}
dE(w0, w1)/dw0 = {calc_gradient(w0, w1)[0]}
dE(w0, w1)/dw1 = {calc_gradient(w0, w1)[1]}\n""")

# Gradient Descent Algorithm
gradient_descent_ver1(learning_rate)
solution, iterations = gradient_descent_ver1(learning_rate)
print(f"""Solution: w0 = {solution[0]}, w1 = {solution[1]}
best fit f(x) = ({solution[0]}) + ({solution[1]}) * cos(πx)\n""")
```

```
(a)
E(w0, w1) = 4*w0**2 + 2*w0*w1 - 6*w0 + 3*w1**2 + 3
dE(w0, w1)/dw0 = 8*w0 + 2*w1 - 6
dE(w0, w1)/dw1 = 2*w0 + 6*w1

Solution: w0 = 0.8181818181818173, w1 = -0.27272727272727204
best fit f(x) = (0.8181818181818173) + (-0.27272727272727204) * x


(b)
E(w0, w1) = 4*w0**2 - 4.0*w0*w1 - 6*w0 + 4.0*w1**2 + 2.0*w1 + 3
dE(w0, w1)/dw0 = 8*w0 - 4.0*w1 - 6
dE(w0, w1)/dw1 = -4.0*w0 + 8.0*w1 + 2.0

Solution: w0 = 0.8333333333333323, w1 = 0.16666666666666596
best fit f(x) = (0.8333333333333323) + (0.16666666666666596) * cos(πx)
```

---

3. Gradient

the given f(x) is $f(x) = w_2x + cos w_1 x + w_0$

Error function is $E(w) = \sum_{i=1} Error_i$

which $Error_i = (f(x_i) - t_i)^2 = (w_2x_i + cos w_1 x_i + w_0 - t_i)^2$

the process is following:

In [ ]:
```
print(f"""Error_i = (w0 + cos(w1*x) + w2*x - t_i)^2
dError_i/dw0 = 2*(w0 + cos(w1*x) + w2*x - t_i)
dError_i/dw1 = -2*x*sin(w1*x)*(w0 + cos(w1*x) + w2*x - t_i)
dError_i/dw2 = 2*x*(w0 + cos(w1*x) + w2*x - t_i)""")

def calc_gradient(w0=1, w1=1, w2=1):
    grad_w0 = sum([2*(w0 + np.cos(w1*x) + w2*x - t_i) for x, t_i in given_data])
    grad_w1 = sum([-2*x*np.sin(w1*x)*(w0 + np.cos(w1*x) + w2*x - t_i) for x, t_i in given_data])
    grad_w2 = sum([2*x*(w0 + np.cos(w1*x) + w2*x - t_i) for x, t_i in given_data])
    return np.array([grad_w0, grad_w1, grad_w2])

print(f"""
(a)
when w0 = 1, w1 = 1, w2 = 1,
sum(dError_i/dw0) = dE/dw0 = {calc_gradient()[0]}
sum(dError_i/dw1) = dE/dw1 = {calc_gradient()[1]}
sum(dError_i/dw2) = dE/dw2 = {calc_gradient()[2]}\n""")

print(f"""
```

```
(b)
when w0 =2, w1 = 2, w2 = 2,
sum(dError_i/dw0) = dE/dw0 = {calc_gradient(2, 2, 2)[0]}
sum(dError_i/dw1) = dE/dw1 = {calc_gradient(2, 2, 2)[1]}
sum(dError_i/dw2) = dE/dw2 = {calc_gradient(2, 2, 2)[2]}""")
```

```
Error_i = (w0 + cos(w1*x) + w2*x - t_i)^2
dError_i/dw0 = 2*(w0 + cos(w1*x) + w2*x - t_i)
dError_i/dw1 = -2*x*sin(w1*x)*(w0 + cos(w1*x) + w2*x - t_i)
dError_i/dw2 = 2*x*(w0 + cos(w1*x) + w2*x - t_i)

(a)
when w0 = 1, w1 = 1, w2 = 1,
sum(dError_i/dw0) = dE/dw0 = 9.241813835208838
sum(dError_i/dw1) = dE/dw1 = -6.093776219708632
sum(dError_i/dw2) = dE/dw2 = 9.08060461173628


(b)
when w0 =2, w1 = 2, w2 = 2,
sum(dError_i/dw0) = dE/dw0 = 13.503118980717147
sum(dError_i/dw1) = dE/dw1 = -8.641161635984396
sum(dError_i/dw2) = dE/dw2 = 15.167706326905716
```

---

4. Python program for enw algorithm

Randomly choose an initial solution, w_0^0 w_1^0

$t = 0$

Repeat

$g_0^t = 0;\ g_1^t = 0$

for all $(\mathbf{x}_i, t_i) \in Data$

$$g_0^t = g_0^t + \left.\frac{\partial E_i}{\partial w_0}\right|_{w_0 = w_0^t, w_1 = w_1^t}$$

$$g_1^t = g_1^t + \left.\frac{\partial E_i}{\partial w_0}\right|_{w_0 = w_0^t, w_1 = w_1^t}$$

$$w_0^{t+1} = w_0^t - \eta g_0^t$$

$$w_1^{t+1} = w_1^t - \eta g_1^t$$

$$t = t + 1$$

Until stopping condition is satisfied

In [ ]:

```
print("Error_i = (w0 + cos(w1*x) + w2*x - t_i)^2")

def calc_gradient_ver2(w0, w1, w2, given_data=given_data):
    total_dE_dw0, total_dE_dw1, total_dE_dw2 = 0, 0, 0
    for x, t in given_data:
        error = w2 * x + np.cos(w1 * x) + w0 - t
        dEi_dw0 = 2 * error
        dEi_dw1 = 2 * error * (-x * np.sin(w1 * x))
        dEi_dw2 = 2 * error * x
        total_dE_dw0 += dEi_dw0
        total_dE_dw1 += dEi_dw1
        total_dE_dw2 += dEi_dw2
    return np.array([total_dE_dw0, total_dE_dw1, total_dE_dw2])

#
def gradient_descent_ver2(learning_rate, max_iter=1000):
    w = np.random.rand(3)
    t = 0
    while t < max_iter:
        grad = calc_gradient_ver2(w[0], w[1], w[2])
        new_w = w - learning_rate * grad
        w = new_w
        t += 1
    return w, t

learning_rate = 0.01
solution, iterations = gradient_descent_ver2(learning_rate)
print(f"""Solution: w0 = {solution[0]}, w1 = {solution[1]}, w2 = {solution[2]}""")
print(f"""best fit f(x) = {solution[0]} + cos({solution[1]} * x) + ({solution[2]}) * x""")
```

```
Error_i = (w0 + cos(w1*x) + w2*x - t_i)^2
Solution: w0 = 0.0009170157064287247, w1 = 0.7245106494902539, w2 = -0.24990559574069385
best fit f(x) = 0.0009170157064287247 + cos(0.7245106494902539 * x) + (-0.24990559574069385) * x
```