

Homework 1 and 2

Question 1. 实现正向传播和反向传播的推导

在进行证明前，需要先定义一些符号：

表 1. 符号说明

符号含义	含义
$a^{(l)}$	第 1 层经过激活函数激活后的输出
$a^{(L)}$	最后一层输出，这时 $y = a^{(L)}$
x	输入数据，并且 $x = a^{(1)}$
$w_{ij}^{(l)}$	权重，l 代表第 l 层，i 代表第 l 层的第 i 个神经元，j 代表第 l-1 层的第 j 个神经元
$b^{(L)}$	第 1 层的偏置
$z^{(l)}$	对第 l-1 层的输出进行仿射变化得到结果，也就是 $z^{(l)} = w^{(l)}a^{(l-1)} + b^{(l)}$
$\sigma^{(l)}$	第 1 层的激活函数，并且有 $a^{(l)} = \sigma^{(l)}(z^{(l)})$
$\delta_j^{(l)}$	损失函数对于第 l 层第 j 个加权输出的偏导，也就是 $\delta_j^{(l)} = \frac{\partial E}{\partial z_j^{(l)}}$
t	输出的监督值
$E(y, t)$	损失函数，这里才用 L2 范数，也就是 $E(y, t) = \frac{\sum_k (y_k - t_k)^2}{2}$
θ	权重，就是 w 的总称
L	最后一层的神经网络的层数

正向传播和反向传播都是为了求解得到损失函数关于权重以及偏置的梯度，采用的都是链式法则。

(a) 反向传播公式推导：

由链式法则，可以得到：

$$(1a) \quad \left\{ \begin{array}{l} \frac{\partial E}{\partial w_{ij}^{(l)}} = \frac{\partial E}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial w_{ij}^{(l)}} \\ \frac{\partial E}{\partial b_j^{(l)}} = \frac{\partial E}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial b_j^{(l)}} \end{array} \right.$$

对于 (1a) 和 (1b) 式子，可以化为：

$$(2a) \quad \left\{ \begin{array}{l} \frac{\partial E}{\partial w_{ij}^{(l)}} = \delta^{(l)} \frac{\partial z^{(l)}}{\partial w_{ij}^{(l)}} \\ \frac{\partial E}{\partial b_j^{(l)}} = \delta^{(l)} \frac{\partial z^{(l)}}{\partial b_j^{(l)}} \end{array} \right.$$

所以，也就是求解 $\delta^{(l)}, \frac{\partial z^{(l)}}{\partial w_{ij}^{(l)}}, \frac{\partial z^{(l)}}{\partial b_j^{(l)}}$ 这三个变量。

对于该神经网络，假设我们使用的激活函数都为同一个激活函数，也就是 $\sigma = \sigma^{(l)}$ ，为了方便起见，后文中将激活函数写为： σ 。

首先先求解 $\delta^{(l)}$ ：

- 求解 $\delta^{(l)}$:

对于最后一层神经网络，我们从定义中知道：

$$a^{(L)} = \sigma(z^{(L)})$$

那么对于最后一层：

$$\delta^{(L)} = \frac{\partial E}{\partial z^{(L)}} = \frac{\partial \sum_k \frac{(y_k - a_k^{(L)})^2}{2}}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}}$$

也就是

$$\delta^{(L)} = \sum_k (a^{(L)} - y_k) \sigma'(z^{(L)})$$

当 $1 \leq l \leq L-1$ 时，我们这里采用链式法则来计算，由于是反向传播，所以我们选择用 $l+1$ 层的偏导来表示，也就是：

$$\delta^{(l)} = \frac{\partial E}{\partial z^{(l)}} = \frac{\partial E}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial z^{(l)}} = \delta^{(l+1)} \frac{\partial z^{(l+1)}}{\partial z^{(l)}}$$

这是一个递推关系式，为了进一步求解，我们需要求解 $\frac{\partial z^{(l+1)}}{\partial z^{(l)}}$ 从定义中可以知道：

$$z^{(l+1)} = w^{(l+1)} a^{(l)} + b^{(l+1)} = w^{(l+1)} \sigma(z^{(l)}) + b^{(l+1)}$$

所以有：

$$\frac{\partial z^{(l+1)}}{\partial z^{(l)}} = w^{(l+1)} \sigma'(z^{(l)})$$

那么这样我们便可以得到 $\delta^{(l)}$ 的表达式为：

$$\delta^{(l)} = \sigma'(z^{(l)}) (w^{(l+1)})^T \delta^{(l+1)}$$

至此我们完成了第一步，也就是求解 $\delta^{(l)}$ 然后我们求解 $\frac{\partial z^{(l)}}{\partial w_{ij}^{(l)}}$, $\frac{\partial z^{(l)}}{\partial b_j^{(l)}}$ 这两个即可，便可以完成反向传播的推导。

- 求解 $\frac{\partial z^{(l)}}{\partial w_{ij}^{(l)}}$

从定义中可以知道：

$$z^{(l+1)} = w^{(l+1)} a^{(l)} + b^{(l+1)}$$

所以有：

$$\frac{\partial z^{(l)}}{\partial w_{ij}^{(l)}} = a^{(l-1)}$$

- 求解 $\frac{\partial z^{(l)}}{\partial b_j^{(l)}}$

从定义中可以知道：

$$z^{(l+1)} = w^{(l+1)} a^{(l)} + b^{(l+1)}$$

所以有：

$$\frac{\partial z^{(l)}}{\partial b_j^{(l)}} = 1$$

至此，我们便完成了反向传播的推导，也就是：

$$\begin{aligned}
 (3a) \quad & \frac{\partial E}{\partial w_{ij}^{(l)}} = \delta^{(l)} a^{(l-1)} \\
 (3b) \quad & \frac{\partial E}{\partial b_j^{(l)}} = \delta^{(l)} \\
 (3c) \quad & \text{其中：} \delta^{(l)} = \begin{cases} \sigma'(z^{(l)})(w^{(l+1)})^T \delta^{(l+1)}, & l = 1 \dots L-1 \\ \sum_k (a^{(L)} - y_k) \sigma'(z^{(L)}), & l = L \end{cases}
 \end{aligned}$$

(b) 正向传播公式推导：

所谓正向传播，与上面不同的地方仅仅存在于关于 $\delta^{(l)}$ 求解过程中的链式法则使用上，这里将使用 l-1 层的偏导来表示，所以被称为正向传播。所以我们只需要改写其中关于 $\delta^{(l)}$ 的求解部分即可，其他部分与反向传播相同。

- 对于第一层神经网络，我们从定义中知道：

$$a^{(1)} = x$$

那么对于第一层：

$$\delta^{(L)} = \frac{\partial E}{\partial z^{(L)}} = \frac{\partial E}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} = \frac{\partial E}{\partial x} \frac{\partial x}{\partial z^{(L)}}$$

由于 x 是常数，所以 $\frac{\partial x}{\partial z^{(L)}} = 0$ 也就是：

$$\delta^{(L)} = 0$$

- 当 $2 \leq l \leq L$ 时，我们这里采用链式法则来计算，由于是正向传播，所以我们选择用 l-1 层的偏导来表示，也就是：

$$\delta^{(l)} = \frac{\partial E}{\partial z^{(l)}} = \frac{\partial E}{\partial z^{(l-1)}} \frac{\partial z^{(l-1)}}{\partial z^{(l)}} = \delta^{(l-1)} \frac{\partial z^{(l-1)}}{\partial z^{(l)}}$$

这是一个递推关系式，为了进一步求解，我们需要求解 $\frac{\partial z^{(l+1)}}{\partial z^{(l)}}$ 从定义中可以知道：

$$z^{(l+1)} = w^{(l+1)} a^{(l)} + b^{(l+1)} = w^{(l+1)} \sigma(z^{(l)}) + b^{(l+1)}$$

所以有：

$$z^{(l-1)} = \sigma^{-1}\left(\frac{z^{(l)} - b^{(l)}}{w^{(l)}}\right)$$

那么可以得到：

$$\frac{\partial z^{(l-1)}}{\partial z^{(l)}} = \left(\sigma^{-1}\left(\frac{z^{(l)} - b^{(l)}}{w^{(l)}}\right)\right)'$$

那么这样我们便可以得到 $\delta^{(l)}$ 的表达式为：

$$\delta^{(l)} = \delta^{(l-1)} \left(\sigma^{-1}\left(\frac{z^{(l)} - b^{(l)}}{w^{(l)}}\right)\right)'$$

至此，我们便完成了正向传播的推导，也就是：

$$\begin{aligned}
 (4a) \quad & \frac{\partial E}{\partial w_{ij}^{(l)}} = \delta^{(l)} a^{(l-1)} \\
 (4b) \quad & \frac{\partial E}{\partial b_j^{(l)}} = \delta^{(l)} \\
 (4c) \quad & \text{其中：} \delta^{(l)} = \begin{cases} \delta^{(l-1)} (\sigma^{-1}(\frac{z^{(l)} - b^{(l)}}{w^{(l)}}))', & l = 2 \dots L \\ 0, & l = 1 \end{cases}
 \end{aligned}$$

Question 2. 对于一个神经网络 $y = DNN(x, \theta)$, 试求解 $\frac{\partial y}{\partial x}$

从上面，推导过程，不难看出，对于一个 n 层的神经网络，如果其输出为 y ，那么有：

$$y = DNN(x, \theta) = \sigma(w^{(n)} \sigma(w^{(n-1)} \sigma(\dots) + b^{(n-1)}) + b^{(n)})$$

所以一个神经网络的输出对于其输入的偏导可以表示为：

$$\frac{\partial y}{\partial x} = \frac{\partial DNN(x, \theta)}{\partial x} = \sigma'(z^{(n)} + b) \times w^{(n)} \times \sigma'(z^{(n-1)} + b) \times w^{(n-1)} \times \dots \times \sigma'(w^{(1)}x + b^{(1)}) \times w^{(1)}$$

或者采用数值求导的方法，令 h 为一个较小的值，那么：

$$\frac{\partial y}{\partial x} = \frac{\partial DNN(x, \theta)}{\partial x} = \frac{DNN(x + h, \theta) - DNN(x - h, \theta)}{2h}$$

Question 3. 对于以下实现以下几种算法，并画出其对应的收敛情况。

- 1) 1:gradient descend
- 2) 2:Ada grad
- 3) 3:RMS prop
- 4) 4:Momentum
- 5) 5:Nesterov
- 6:Adam

对于各个算法的实现如下：

```

1  """
2  This project will include the normal optimization algorithms for the DeepLearning :
3  1:gradient descend
4  2:Ada grad
5  3:RMS prop
6  4:Momentum
7  5:Nesterov
8  6:Adam
9  """
10 import numpy as np
11 import matplotlib.pyplot as plt
12
13
14 class OptimizationAlgorithm:
15     def __init__(self, obj_func, num_var, lower_bound, upper_bound, epoch, analytical_grad=None, iter_hist=False):
16         """
17         This is the initialization of this problem , and it is usually defined as a minimizing problem .
18         Parameters
19         -----

```

```

20     obj_func: Objective function
21     num_var: number of variables for this function
22     lower_bound: the lower bound for the variables . Obviously , this is a list .
23     upper_bound: the upper bound for the variables . Obviously , this is a list .
24     epoch: the number of the iteration.
25     analytical_grad: the analytical grad for the function if given .
26     iter_hist: whether store the value for each step .
27     """
28
29     self.obj_func = obj_func
30     self.num_var = num_var
31     self.lower_bound = lower_bound
32     self.upper_bound = upper_bound
33     self.epoch = epoch
34     if analytical_grad:
35         self.analytical_grad = analytical_grad
36         self.has_analytical_grad = True
37     else:
38         self.has_analytical_grad = False
39     if iter_hist:
40         self.has_iter_hist = True
41         self.iter_hist = np.array(np.zeros(shape=(self.num_var,)))
42     else :
43         self.has_iter_hist = False
44
45     @staticmethod
46     def numerical_gradient(f, x, h=1e-8):
47         """
48
49         Parameters
50         -----
51         f:objective function
52         x:at where the grad is obtained
53         h:the step
54
55         Returns
56         -----
57         return the corresponding gradient
58         """
59         grad = np.zeros_like(x)
60
61         it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
62         while not it.finished:
63             idx = it.multi_index
64             tmp_val = x[idx]
65             x[idx] = float(tmp_val) + h
66             fxh1 = f(x) # f(x+h)
67             x[idx] = tmp_val - h
68             fxh2 = f(x) # f(x-h)
69             grad[idx] = (fxh1 - fxh2) / (2 * h)
70             x[idx] = tmp_val # 还原值
71             it.iternext()
72
73         return grad
74
75     def gradient_descend(self, grad_method, lr=1e-3, display_process=False):
76         """
77         Use the general gradient descend method to obtain the optimal solution.
78         Parameters
79         -----
80         grad_method: The method to obtain gradient , by analytical or numerical

```

```

81     lr: The learning rate .
82     display_process: whether display the iteration process.
83
84     Returns
85     -----
86     Return the x , where the objective function gets the minimum.
87     """
88     x = np.random.uniform(low=self.lower_bound, high=self.upper_bound)
89     for i in range(self.epoch):
90         if grad_method == "analytical" and self.has_analytical_grad:
91             # use the analytical method to obtain the gradient
92             grad = self.analytical_grad(x)
93         if grad_method == "numerical":
94             # use the numerical method to obtain the gradient
95             grad = self.numerical_gradient(self.obj_func, x)
96         if self.has_iter_hist:
97             self.iter_hist = np.vstack((self.iter_hist, x))
98         x -= lr*grad
99         if display_process:
100             if i % 50 == 0:
101                 print(f"epoch: {i}/{self.epoch}, x={x}, f={self.obj_func(x)}")
102
103     return x
104
105 def ada_grad(self, grad_method, lr=1e-1, display_process=False):
106     """
107     Use the adaptive gradient descend method to obtain the optimal solution.
108     For the summation of the grad will accumulate , so the step for each iteration
109     will approach the 0 . In the end , the iteration will terminate .
110     Parameters
111     -----
112     grad_method: The method to obtain gradient , by analytical or numerical
113     lr: The learning rate .
114     display_process: whether display the iteration process.
115
116     Returns
117     -----
118     Return the x , where the objective function gets the minimum.
119     """
120     # for the ada grad algorithm , a list to record the history should be maintained
121     gt = 0
122     x = np.random.uniform(low=self.lower_bound, high=self.upper_bound)
123     for i in range(self.epoch):
124         if grad_method == "analytical" and self.has_analytical_grad:
125             # use the analytical method to obtain the gradient
126             grad = self.analytical_grad(x)
127         if grad_method == "numerical":
128             # use the numerical method to obtain the gradient
129             grad = self.numerical_gradient(self.obj_func, x)
130         if self.has_iter_hist:
131             self.iter_hist = np.vstack((self.iter_hist, x))
132         # get the accumulative grad
133         gt += np.sum(np.square(grad))
134         # The Ada grad algorithm is implemented!
135         x -= lr*grad/np.sqrt(gt+1e-9)
136         if display_process:
137             if i % 50 == 0:
138                 print(f"epoch: {i}/{self.epoch}, x={x}, f={self.obj_func(x)}")
139
140     return x
141

```

```

142 def rms_prop(self, grad_method, lr=1e-2, display_process=False):
143     """
144     Use the rms prop method to obtain the optimal solution.
145     For the sumation of the grad will accumulate , so the step for each iteration
146     will approach the 0 . In the end , the iteration will terminate .
147     Parameters
148     -----
149     grad_method: The method to obtain gradient , by analytical or numerical
150     lr: The learning rate .
151     display_process: whether display the iteration process.
152
153     Returns
154     -----
155     Return the x , where the objective function gets the minimum.
156     """
157     # for the rms prop algorithm , a sumation of the grad should be maintained .
158     gt = 0
159     beta = 0.9
160
161     x = np.random.uniform(low=self.lower_bound, high=self.upper_bound)
162     # self.w = np.array((x[0], x[1]))
163     for i in range(self.epoch):
164         if grad_method == "analytical" and self.has_analytical_grad:
165             # use the analytical method to obtain the gradient
166             grad = self.analytical_grad(x)
167         if grad_method == "numerical":
168             # use the numerical method to obtain the gradient
169             grad = self.numerical_gradient(self.obj_func, x)
170         if self.has_iter_hist:
171             self.iter_hist = np.vstack((self.iter_hist, x))
172         # get the accumulative grad
173         gt = beta*gt+(1-beta)*np.sum(np.square(grad))
174         # The Ada grad algorithm is implemented!
175         x -= lr * grad / (np.sqrt(gt) + 1e-9)
176         if display_process:
177             if i % 50 == 0:
178                 print(f"epoch: {i}/{self.epoch}, x={x}, f={self.obj_func(x)}")
179
180     return x
181
182 def momentum(self, grad_method, lr=1e-3, display_process=False):
183     """
184     Use the momentum modified general gradient descend method to obtain the optimal solution.
185     The momentum method will use the momentum to avoid the local optimal
186     Parameters
187     -----
188     grad_method: The method to obtain gradient , by analytical or numerical
189     lr: The learning rate .
190     display_process: whether display the iteration process.
191
192     Returns
193     -----
194     Return the x , where the objective function gets the minimum.
195     """
196     x = np.random.uniform(low=self.lower_bound, high=self.upper_bound)
197     # the initial velocity
198     velocity = np.zeros_like(x)
199     momentum_factor = 0.1
200     for i in range(self.epoch):
201         if grad_method == "analytical" and self.has_analytical_grad:
202             # use the analytical method to obtain the gradient

```

```

203         grad = self.analytical_grad(x)
204     if grad_method == "numerical":
205         # use the numerical method to obtain the gradient
206         grad = self.numerical_gradient(self.obj_func, x)
207     if self.has_iter_hist:
208         self.iter_hist = np.vstack((self.iter_hist, x))
209     # update the velocity
210     velocity = velocity*momentum_factor-lr*grad
211     x = x+velocity
212     if display_process:
213         if i % 50 == 0:
214             print(f"epoch: {i}/{self.epoch}, x={x}, f={self.obj_func(x)}")
215
216     return x
217
218 def nesterov(self, grad_method, lr=1e-3, display_process=False):
219     """
220     Use the momentum modified general gradient descend method to obtain the optimal solution.
221     The momentum method will use the momentum to avoid the local optimal.
222     Nesterov method is a kind of acceleration method that step ahead a bit before
223     the gradient descend .
224     Parameters
225     -----
226     grad_method: The method to obtain gradient , by analytical or numerical
227     lr: The learning rate .
228     display_process: whether display the iteration process.
229
230     Returns
231     -----
232     Return the x , where the objective function gets the minimum.
233     """
234     x = np.random.uniform(low=self.lower_bound, high=self.upper_bound)
235     # the initial velocity
236     velocity = np.zeros_like(x)
237     momentum_factor = 0.1
238
239     # then all of the grads should be acquired in the accelerated_x rather than x
240     for i in range(self.epoch):
241         # accelerate the x first before obtaining the grad
242         accelerated_x = x + velocity * momentum_factor
243         if grad_method == "analytical" and self.has_analytical_grad:
244             # use the analytical method to obtain the gradient
245             grad = self.analytical_grad(accelerated_x)
246         if grad_method == "numerical":
247             # use the numerical method to obtain the gradient
248             grad = self.numerical_gradient(self.obj_func, accelerated_x)
249         if self.has_iter_hist:
250             self.iter_hist = np.vstack((self.iter_hist, x))
251         # update the velocity , this is the same as the momentum method
252         velocity = velocity * momentum_factor - lr * grad
253         x = x + velocity
254         if display_process:
255             if i % 50 == 0:
256                 print(f"epoch: {i}/{self.epoch}, x={x}, f={self.obj_func(x)}")
257
258     return x
259
260 def adam(self, grad_method, lr=1e-1, display_process=False):
261     """
262     This is the implementation for the adam algorithm , which
263     is the combination of the ada grad and momentum algorithm.

```



```

264     Parameters
265     -----
266     grad_method: The method to obtain gradient , by analytical or numerical
267     lr: The learning rate .
268     display_process: whether display the iteration process.
269
270     Returns
271     -----
272     Return the x , where the objective function gets the minimum.
273
274     """
275     # the declining rate for the moment estimation
276     rho1, rho2 = 0.9, 0.999
277     delta = 1e-8
278     gt = 0
279     x = np.random.uniform(low=self.lower_bound, high=self.upper_bound)
280     # initialize the first moment and the second moment
281     s = np.zeros_like(x)
282     r = np.zeros_like(x)
283     for i in range(self.epoch):
284         if grad_method == "analytical" and self.has_analytical_grad:
285             # use the analytical method to obtain the gradient
286             grad = self.analytical_grad(x)
287         if grad_method == "numerical":
288             # use the numerical method to obtain the gradient
289             grad = self.numerical_gradient(self.obj_func, x)
290         if self.has_iter_hist:
291             self.iter_hist = np.vstack((self.iter_hist, x))
292         # update the first and the second moment
293         s = rho1*s+(1-rho1)*grad
294         r = rho2*r+(1-rho2)*np.square(grad)
295         # get the partial first and second moment
296         s_hat = s/(1-rho1**(i+1))
297         r_hat = r/(1-rho1**(i+1))
298         # use the partial first and second moment to modify the GD method
299         x -= lr * s_hat / (np.sqrt(r_hat)+delta)
300         if display_process:
301             if i % 50 == 0:
302                 print(f"epoch: {i}/{self.epoch}, x={x}, f={self.obj_func(x)}")
303
304     return x
305
306     def draw_process(self):
307         x = np.arange(self.lower_bound[0], self.upper_bound[0], 0.1)
308         y = np.arange(self.lower_bound[1], self.upper_bound[1], 0.1)
309         [x, y] = np.meshgrid(x, y)
310         plt.contour(x, y, x**2/20+y**2, 20)
311         w = self.iter_hist[1:]
312         plt.plot(w[:, 0], w[:, 1], 'g*', w[:, 0], w[:, 1])
313         plt.show()
314
315     def clear_hist(self):
316         self.iter_hist = np.array(np.zeros(shape=(self.num_var,)))
317
318
319 if __name__ == '__main__':
320     obj_func = lambda x: x[0]**2/20+x[1]**2
321     # solver = OptimizationAlgorithm(obj_func=obj_func, num_var=2, lower_bound=[-100, -100], upper_bound=[100, 100],
322     #                               epoch=1000000)
323     # optimal_x = solver.gradient_descent(grad_method="numerical", display_process=True)
324     # save the data for each algorithm

```

```
324 solver = OptimizationAlgorithm(obj_func=obj_func, num_var=2,
325                               lower_bound=[-100, -100], upper_bound=[100, 100],
326                               epoch=20000, analytical_grad=lambda x: np.array([x[0]/10, 2*x[1]]), iter_hist=
    True)
327 optimal_x = solver.adam(grad_method="numerical", display_process=True)
328 solver.draw_process()
329 print(f"optimal_x:{optimal_x}")
330 print(f"minimum function value {obj_func(optimal_x)}")
331 # draw_process(solver.w)
```

LISTING 1. 基于 python3 对于 6 中优化算法的实现