
Mathematics of Neural Networks
winter semester 2021/2022
exercise sheet 3, solutions

Exercise 1: (4 points)

- a) Add different weight initialization schemes to `initializers.py` as treated in the lecture:
- (i) A class `RandnAverage` that provides the method `wfun(self, m, n)` that returns an $m \times n$ random matrix with normally distributed entries and variance $2/(m+n)$.
Hint: Use `numpy.random.randn` and multiply the result by the square root of the desired variance.
 - (ii) A class `RandAverage` that provides the method `wfun(self, m, n)` that returns an $m \times n$ random matrix with uniformly distributed entries on the interval $[r, r]$, where the radius is given by $r := \sqrt{6/(m+n)}$.
Hint: Use `numpy.random.rand`. The method `rand` generates random numpy arrays with entries uniformly distributed in the interval $[0, 1]$. Shift and scale the results in an appropriate way.
 - (iii) A class `RandnOrthonormal` that provides the method `wfun(self, m, n)` that returns an $m \times n$ random orthonormal matrix.
Hint: Use `numpy.linalg.svd` or `numpy.linalg.qr` as sketched in the lecture to obtain a random orthonormal matrix from a random matrix.
- b) On initialization of a `DenseLayer` add the attribute `initializer`, which is given as argument on initialization and should be `RandnAverage()` by default.
The weight matrix should be initialized by `wfun` from the initializer and scaled by a factor depending on the activation function. This factor can be obtained from the attribute `factor` of the activation function `afun`.

Solution:

- a) A possible implementation of the three initialization methods can be seen in the following listing:

```
1 import numpy as np
2 from numpy.random import rand, randn
3
4 class RandnAverage:
5
6     def wfun(self, ni, no):
7
8         avarage = (ni + no) / 2.0
9         return randn(no, ni) * np.sqrt(1.0/avarage)
10
11 class RandAverage:
12
13     def wfun(self, ni, no):
14
```

```

15     avarage = (ni + no) / 2.0
16     return (rand(no, ni) - .5) * np.sqrt(12.0/avarage)
17
18
19 class RandnOrthonormal:
20
21     def wfun(self, ni, no):
22
23         A = randn(no, ni)
24         U, _, V = np.linalg.svd(A, full_matrices=False)
25         if ni >= no:
26             return V
27         else:
28             return U

```

b) The modifications to `DenseLayer` are visible in the solution of exercise 4.

Exercise 2: (4 points)

- a) Add as many other activation functions to `activations.py` as you like. e.g., those mentioned in the lecture (See lecture notes, section 1.3¹). Use classes and subclasses to group activation functions. See the following listing for an example using the already implemented activation functions:

```

1 import numpy as np
2
3 # Heaviside family activation functions
4 class HeavisideLike:
5
6     def __init__(self):
7
8         self.factor = 1.0
9
10    """
11    TODO Implement the following activation functions from the
12    Heaviside activation function family:
13        - Heaviside function
14        - Modified Heaviside function
15        - Logistic function
16        - Exp function (as mentioned on the exercise sheet)
17    """
18
19 # Sign family activation functions
20 class SignLike:
21
22     def __init__(self):
23
24         self.factor = 1.0
25
26    """
27    TODO Implement the following activation functions from the
28    Sign activation function family:
29        - Sign function
30        - TanH function
31        - SoftSign function

```

¹Version from October 20th, 2021

```
32 """
33 # ReLU family activation functions
34 class ReLULike:
35
36     def __init__(self):
37
38         self.factor = np.sqrt(2)
39
40 class ReLU(ReLULike):
41
42     def __init__(self):
43
44         super().__init__()
45         self.data = None
46         self.name = 'ReLU'
47
48     def evaluate(self, x):
49
50         self.data = x
51         return x.clip(min = 0)
52
53     def backprop(self, delta):
54
55         return (self.data >= 0) * delta
56
57 """
58 TODO Add the following activation functions from the
59 ReLU activation function family:
60     - leaky ReLU function
61     - ELU function
62     - SoftPlus function
63     - Swish function
64 """
65 # Abs family activation functions
66 class AbsLike:
67
68     def __init__(self):
69
70         self.factor = 1.0
71
72 class Abs(AbsLike):
73
74     def __init__(self, alpha=0.0):
75
76         super().__init__()
77         self.data = None
78         self.name = 'Abs'
79         self.alpha = alpha
80
81     def evaluate(self, x):
82
83         self.data = x
84         if(self.alpha == 0.0):
85             return np.abs(x)
86         else:
87             return np.sqrt(x*x + self.alpha**2) - self.alpha
```

```

88
89     def backprop(self, delta):
90
91         if(self.alpha == 0.0):
92             return np.sign(self.data)*delta
93         else:
94             return self.data/np.sqrt(self.data**2 + self.alpha**2) * delta
95
96     """
97     TODO Add the following activation functions from the
98         Abs activation function family:
99         - L0Co function
100         - Twist function
101         - SoftAbs function
102     """

```

Do you have ideas for activation functions different from the ones mentioned in lecture? You could search the internet or take a look at those gathered in Wikipedia.

Hint: The initialization factor for activation functions of one family can be set by calling `super().__init__()`

- b) Expand example 5c) of exercise sheet 2 to use different activation functions. What do you observe? What happens when you use a function that is generally not used as an activation function like $\text{Exp}(x) := e^x$?

Solution:

- a) We implemented a selection of activation functions that are at least differentiable almost everywhere and nonzero for most inputs mentioned in the first lecture. We also added the exponential function which is used as activation with derivative given by the function itself. These functions are defined in the following listing,

```

1  import numpy as np
2
3  # Heaviside family activation functions
4  class HeavisideLike():
5
6      def __init__(self):
7
8          self.factor = 1.0
9
10 class Heaviside(HeavisideLike):
11
12     def __init__(self):
13
14         super().__init__()
15         self.data = None
16         self.name = 'Heavyside'
17
18     def evaluate(self, x):
19
20         return (x >= 0) * 1.0
21
22     def backprop(self, delta):
23
24         return 0.0*delta
25

```

```
26 class ModifiedHeaviside(HeavisideLike):
27
28     def __init__(self):
29
30         super().__init__()
31         self.data = None
32         self.name = 'Modified_Heaviside'
33
34     def evaluate(self, x):
35
36         return (x > 0) * 1.0 + (x == 0) * 0.5
37
38     def backprop(self, delta):
39
40         return 0.0 * delta
41
42 class Logistic(HeavisideLike):
43
44     def __init__(self, k=1.0):
45
46         super().__init__()
47         self.data = None
48         self.name = 'Logistic'
49         self.k = k
50
51     def evaluate(self, x):
52
53         self.data = 1.0/(1.0 + np.exp(-self.k * x))
54         return self.data
55
56     def backprop(self, delta):
57
58         return self.k * self.data * (1.0 - self.data) * delta
59
60 class SoftMax(HeavisideLike):
61
62     def __init__(self):
63
64         super().__init__()
65         self.name = 'SoftMax'
66         self.data = None # store data for faster execution
67
68     def evaluate(self, x):
69         x_exp = np.exp(x)
70         scale = x_exp.sum(axis=1, keepdims=True)
71         return x_exp / scale
72
73     def backprop(self, delta):
74         return delta
75
76 class Exp(HeavisideLike):
77
78     def __init__(self):
79
80         super().__init__()
81         self.data = None
```

```

82         self.name = 'Exp'
83
84     def evaluate(self, x):
85
86         self.data = np.exp(x)
87         return self.data
88
89     def backprop(self, delta):
90
91         return self.data * delta
92
93 # Sign family activation functions
94 class SignLike:
95
96     def __init__(self):
97
98         self.factor = 1.0
99
100 class Sign(SignLike):
101
102     def __init__(self):
103
104         super().__init__()
105         self.data = None
106         self.name = 'Sign'
107
108     def evaluate(self, x):
109
110         return np.sign(x)
111
112     def backprop(self, delta):
113
114         return 0.0 * delta
115
116 class TanH(SignLike):
117
118     def __init__(self, k=1.0):
119
120         super().__init__()
121         self.data = None
122         self.name = 'TanH'
123         self.k = k
124
125     def evaluate(self, x):
126
127         self.data = np.tanh(self.k * x)
128
129         return self.data
130
131     def backprop(self, delta):
132
133         return self.k * (1.0 - self.data**2) * delta
134
135 class SoftSign(SignLike):
136
137     def __init__(self, k=1.0):

```

```

138
139         super().__init__()
140         self.data = None
141         self.name = 'SoftSign'
142         self.k = k
143
144     def evaluate(self, x):
145
146         self.data = (self.k * x)/(1.0 + np.abs(self.k * x))
147         return self.data
148
149     def backprop(self, delta):
150
151         return self.k/(self.data**2) * delta
152
153 # ReLU family activation functions
154 class ReLULike:
155
156     def __init__(self):
157
158         self.factor = np.sqrt(2)
159
160 class ReLU(ReLULike):
161
162     def __init__(self):
163
164         super().__init__()
165         self.data = None
166         self.name = 'ReLU'
167
168     def evaluate(self, x):
169
170         self.data = x
171         return x.clip(min = 0)
172
173     def backprop(self, delta):
174
175         return (self.data >= 0) * delta
176
177 class leakyReLU(ReLULike):
178
179     def __init__(self, alpha=.01):
180
181         super().__init__()
182         self.data = None
183         self.name = 'leaky ReLU'
184         self.alpha = alpha
185
186     def evaluate(self, x):
187
188         self.data = x
189         return x.clip(min=0) + self.alpha*x.clip(max=0)
190
191     def backprop(self, delta):
192
193         return (self.data >= 0) * delta \

```

```

194         + (self.data < 0) * self.alpha * delta
195
196 class ELU(ReLULike):
197
198     def __init__(self, alpha=1.0):
199
200         super().__init__()
201         self.data = None
202         self.name = 'ELU'
203         self.alpha = alpha
204
205     def evaluate(self, x):
206
207         self.data = x
208         return x.clip(min=0) + \
209             (np.exp(x.clip(max=0)) - 1.0) * self.alpha
210
211     def backprop(self, delta):
212
213         return (self.data >= 0) * delta \
214             + (self.data < 0) * self.alpha * np.exp(self.data) *
                delta
215
216 class SoftPlus(ReLULike):
217
218     def __init__(self, k=1.0):
219
220         super().__init__()
221         self.data = None
222         self.name = 'SoftPlus'
223         self.k = k
224
225     def evaluate(self, x):
226
227         self.data = np.log(np.exp(self.k * x) + 1.0)/self.k
228         return self.data
229
230     def backprop(self, delta):
231
232         return 1.0 / (1.0 + np.exp(-self.data)) * delta
233
234 class Swish(ReLULike):
235
236     def __init__(self, k=1.0):
237
238         super().__init__()
239         self.data = None
240         self.name = 'Swish'
241         self.k = k
242
243     def evaluate(self, x):
244
245         self.data = self.k * x
246         y = np.exp(self.data)
247         return x * y/(1.0 + y)
248

```



```

249     def backprop(self, delta):
250
251         y = np.exp(self.data)/(1.0 + np.exp(self.data))
252         return y * (1.0 + self.data * (1.0 - y)) * delta
253
254 # Abs family activation functions
255 class AbsLike:
256
257     def __init__(self):
258
259         self.factor = 1.0
260
261 class Abs(AbsLike):
262
263     def __init__(self, alpha=0.0):
264
265         super().__init__()
266         self.data = None
267         self.name = 'Abs'
268         self.alpha = alpha
269
270     def evaluate(self, x):
271
272         self.data = x
273         if(self.alpha == 0.0):
274             return np.abs(x)
275         else:
276             return np.sqrt(x*x + self.alpha**2) - self.alpha
277
278     def backprop(self, delta):
279
280         if(self.alpha == 0.0):
281             return np.sign(self.data)*delta
282         else:
283             return self.data/np.sqrt(self.data**2 + self.alpha**2) *
                delta
284
285 class LCo(AbsLike):
286
287     def __init__(self, k=1.0):
288
289         super().__init__()
290         self.data = None
291         self.name = 'LCo'
292         self.k = k
293
294     def evaluate(self, x):
295
296         self.data = self.k * x
297         return np.log(np.cosh(self.data))/self.k
298
299     def backprop(self, delta):
300
301         return np.tanh(self.data) * delta
302
303 class Twist(AbsLike):

```

```

304
305     def __init__(self, k=1.0):
306
307         super().__init__()
308         self.data = None
309         self.name = 'Twist'
310         self.k = k
311
312     def evaluate(self, x):
313
314         self.data = self.k * x
315         return x * np.tanh(self.data)
316
317     def backprop(self, delta):
318
319         y = np.tanh(self.data)
320         return y * (1.0 + self.data * (1 - y * y)) * delta
321
322 class SoftAbs(AbsLike):
323
324     def __init__(self, k=1.0):
325
326         super().__init__()
327         self.data = None
328         self.name = 'SoftAbs'
329         self.k = k
330
331     def evaluate(self, x):
332
333         self.data = self.k * x
334         return self.data * x / (1.0 + np.abs(self.data))
335
336     def backprop(self, delta):
337
338         y = 1.0 + np.abs(self.data)
339         return (1.0 + y)/(y**2) * self.data * delta

```

- b) We use the various activation functions implemented in a). See the following listing for a possible implementation that applies all these to the MNIST example of exercise 5c) of exercise sheet 2.

```

1 import numpy as np
2 from random import randrange
3 import matplotlib.pyplot as plt
4
5 from networks import SequentialNet
6 from layers import DenseLayer
7 from activations import *
8 from optimizers import *
9
10 DATA = np.load('mnist.npz')
11 x_train, y_train = DATA['x_train'], DATA['y_train']
12 x_test, y_test = DATA['x_test'], DATA['y_test']
13 x_train, x_test = x_train / 255.0, x_test / 255.0
14
15 x = x_train.reshape(60000, 784).T

```

```

16 I = np.eye(10)
17 y = I[:, y_train]
18
19 bs, ep, eta = 10, 10, .01
20
21 functions1 = [Logistic(), TanH(), ReLU(), leakyReLU(),
22               ELU(), SoftPlus(), Abs(), L0Co(), Exp()]
23 functions2 = [Logistic(), TanH(), ReLU(), leakyReLU(),
24               ELU(), SoftPlus(), Abs(), L0Co(), Exp()]
25
26 for fun1, fun2 in zip(functions1, functions2):
27
28     print('Activation_function=', fun1.name)
29     layers = [DenseLayer(784, 100, afun=fun1, optim=SGD(eta)),
30               DenseLayer(100, 10, afun=fun2, optim=SGD(eta))]
31     netz = SequentialNet(784, layers)
32     netz.train(x, y, bs, ep)
33
34     y_tilde = netz.evaluate(x_test.reshape(10000, 784, 1))
35     guess = np.argmax(y_tilde, 1).T
36     print('accuracy=', np.sum(guess == y_test)/100)
37
38     for i in range(4):
39
40         k = randrange(y_test.size)
41         plt.title('Label_is_{lb}, guess_is_{gs}'.format(lb=y_test[k], gs
42               =guess[0,k]))
43         plt.imshow(x_test[k], cmap='gray')
44         plt.show()

```

We observe that the first eight activation functions work satisfactorily, the exponential fails with `nan` (not a number). The first three activation functions give results that lie for this learning rate, minibatch, and number of epochs close to 85-88% accuracy on the test set, whereas the others give results around 95%.

Exercise 3: (4 points)

- Add the optimizer Adam (Lecture notes, section 2.5.7²) as class `Adam` to the `optimizers.py` script. It should work similar to the `SGD` class.
- Add momentum to the `SGD` class (See lecture notes, section 2.5.2).

Solution: We implemented two classes, `SGD` and `Adam`. The code for these including the solution of exercise 4 can be found in the next listing:

```

1 import numpy as np
2
3 class SGD:
4
5     def __init__(self, eta=.01, momentum=0.0):
6
7         self.eta = eta
8         self.momentum = momentum
9         if self.momentum > 0.0:
10             self.v = []

```

²Version from October 20th, 2021

```

11         self.name = 'SGD({})_with_momentum_{}'.format(eta, momentum)
12     else:
13         self.name = 'SGD({})'.format(eta)
14
15     def update(self, data, ddata):
16
17         if self.momentum > 0.0:
18
19             if self.v == []:
20                 self.v = [np.zeros_like(p) for p in data]
21
22             for v, p, dp in zip(self.v, data, ddata):
23
24                 v = v * self.momentum + dp * self.eta
25                 p -= v
26
27         else:
28
29             for p, dp in zip(data, ddata):
30
31                 p -= self.eta * dp
32
33
34
35     class Adam:
36
37         def __init__(self, eta = .001, beta1 = .9, beta2 = .999, eps=1e-8):
38
39             self.eta = eta
40             self.beta1 = beta1
41             self.beta2 = beta2
42             self.eps = eps
43
44             self.w = []
45             self.v = []
46             self.k = 0
47
48             self.name = 'Adam'
49
50         def update(self, data, ddata):
51
52             if self.v == []:
53                 self.v = [np.zeros_like(p) for p in data]
54             if self.w == []:
55                 self.w = [np.zeros_like(p) for p in data]
56
57             self.k += 1
58             alpha = self.eta*np.sqrt(1 - self.beta2**self.k)/(1 - self.beta1**
                    self.k)
59             for v, w, p, dp in zip(self.v, self.w, data, ddata):
60
61                 v = self.beta1*v + (1 - self.beta1)*dp
62                 w = self.beta2*w + (1 - self.beta2)*dp**2
63
64                 p -= v*alpha/np.sqrt(w + self.eps)

```

Exercise 4: (4 points) You can add a penalty term for the size of weights by using L^2 regularization to change the cost function. The resulting effect is often called *weight decay*.

- Compare plain SGD using L^2 regularization with SGD using momentum by calculating the first three respective updates of a single weight matrix by hand.
- What differences do you observe?
- Without directly calculating it, what do you expect for other optimizers like Adam?

Solution: We omit the layer index and denote the weight matrix by \mathbf{W} . We start by calculating the first three updates without using momentum or regularization to better study the effects of both

$$\begin{aligned}
 \mathbf{W}_1 &= \mathbf{W}_0 - \eta \frac{\partial C}{\partial \mathbf{W}_0} \\
 \mathbf{W}_2 &= \mathbf{W}_1 - \eta \frac{\partial C}{\partial \mathbf{W}_1} \\
 &= \mathbf{W}_0 - \eta \frac{\partial C}{\partial \mathbf{W}_0} - \eta \frac{\partial C}{\partial \mathbf{W}_1} \\
 \mathbf{W}_3 &= \mathbf{W}_2 - \eta \frac{\partial C}{\partial \mathbf{W}_2} \\
 &= \mathbf{W}_0 - \eta \frac{\partial C}{\partial \mathbf{W}_0} - \eta \frac{\partial C}{\partial \mathbf{W}_1} - \eta \frac{\partial C}{\partial \mathbf{W}_2}.
 \end{aligned}$$

Using regularization leads to the following updates:

$$\begin{aligned}
 \mathbf{W}_1 &= \mathbf{W}_0(1 - \lambda\eta) - \eta \frac{\partial C}{\partial \mathbf{W}_0} \\
 \mathbf{W}_2 &= \mathbf{W}_1(1 - \lambda\eta) - \eta \frac{\partial C}{\partial \mathbf{W}_1} \\
 &= (\mathbf{W}_0(1 - \lambda\eta) - \eta \frac{\partial C}{\partial \mathbf{W}_0})(1 - \lambda\eta) - \eta \frac{\partial C}{\partial \mathbf{W}_1} \\
 &= \mathbf{W}_0(1 - \lambda\eta)^2 - \eta(1 - \lambda\eta) \frac{\partial C}{\partial \mathbf{W}_0} - \eta \frac{\partial C}{\partial \mathbf{W}_1} \\
 \mathbf{W}_3 &= (\mathbf{W}_0(1 - \lambda\eta)^2 - \eta(1 - \lambda\eta) \frac{\partial C}{\partial \mathbf{W}_0} - \eta \frac{\partial C}{\partial \mathbf{W}_1})(1 - \lambda\eta) - \eta \frac{\partial C}{\partial \mathbf{W}_2} \\
 &= \mathbf{W}_0(1 - \lambda\eta)^3 - \eta(1 - \lambda\eta)^2 \frac{\partial C}{\partial \mathbf{W}_0} - \eta(1 - \lambda\eta) \frac{\partial C}{\partial \mathbf{W}_1} - \eta \frac{\partial C}{\partial \mathbf{W}_2}.
 \end{aligned}$$

For Momentum initialised by $\mathbf{V}_0 = 0$ we get

$$\begin{aligned}
\mathbf{V}_1 &= \gamma \mathbf{V}_0 + \frac{\partial C}{\partial \mathbf{W}_0} = \frac{\partial C}{\partial \mathbf{W}_0} \\
\mathbf{W}_1 &= \mathbf{W}_0 - \eta \mathbf{V} = \mathbf{W}_0 - \eta \frac{\partial C}{\partial \mathbf{W}_0} \\
\mathbf{V}_2 &= \gamma \mathbf{V}_1 + \frac{\partial C}{\partial \mathbf{W}_1} \\
&= \gamma \frac{\partial C}{\partial \mathbf{W}_0} + \frac{\partial C}{\partial \mathbf{W}_1} \\
\mathbf{W}_2 &= \mathbf{W}_1 - \eta \mathbf{V}_2 \\
&= \mathbf{W}_0 - \eta \frac{\partial C}{\partial \mathbf{W}_0} - \eta (\gamma \frac{\partial C}{\partial \mathbf{W}_0} - \eta \frac{\partial C}{\partial \mathbf{W}_1}) \\
&= \mathbf{W}_0 - \eta(1 + \gamma) \frac{\partial C}{\partial \mathbf{W}_0} - \eta \frac{\partial C}{\partial \mathbf{W}_1} \\
\mathbf{V}_3 &= \gamma \mathbf{V}_2 + \frac{\partial C}{\partial \mathbf{W}_2} \\
&= \gamma (\gamma \frac{\partial C}{\partial \mathbf{W}_0} + \frac{\partial C}{\partial \mathbf{W}_1}) + \frac{\partial C}{\partial \mathbf{W}_2} \\
\mathbf{W}_3 &= \mathbf{W}_2 - \mathbf{V}_3 \\
&= (\mathbf{W}_0 - \eta(1 + \gamma) \frac{\partial C}{\partial \mathbf{W}_0} - \eta \frac{\partial C}{\partial \mathbf{W}_1}) - \gamma^2 \eta \frac{\partial C}{\partial \mathbf{W}_0} - \gamma \eta \frac{\partial C}{\partial \mathbf{W}_1} - \eta \frac{\partial C}{\partial \mathbf{W}_2} \\
&= \mathbf{W}_0 - \eta(1 + \gamma + \gamma^2) \frac{\partial C}{\partial \mathbf{W}_0} - \eta(1 + \gamma) \frac{\partial C}{\partial \mathbf{W}_1} - \eta \frac{\partial C}{\partial \mathbf{W}_2}.
\end{aligned}$$

Now compare the third update:

$$\begin{array}{llll}
\mathbf{W}_0 & -\frac{\partial C}{\partial \mathbf{W}_0} \eta & -\frac{\partial C}{\partial \mathbf{W}_1} \eta & -\frac{\partial C}{\partial \mathbf{W}_2} \eta \\
\mathbf{W}_0(1 - \lambda \eta)^3 & -\frac{\partial C}{\partial \mathbf{W}_0} \eta(1 - \lambda \eta)^2 & -\frac{\partial C}{\partial \mathbf{W}_1} \eta(1 - \lambda \eta) & -\frac{\partial C}{\partial \mathbf{W}_2} \eta \\
\mathbf{W}_0 & -\frac{\partial C}{\partial \mathbf{W}_0} \eta(1 + \gamma + \gamma^2) & -\frac{\partial C}{\partial \mathbf{W}_1} \eta(1 + \gamma) & -\frac{\partial C}{\partial \mathbf{W}_2} \eta
\end{array}$$

Note that the gradients are respective to the same cost functions but $\frac{\partial C}{\partial \mathbf{W}_1}$ using momentum is different to $\frac{\partial C}{\partial \mathbf{W}_1}$ using regularization because both are evaluated at slightly different places due to the difference in the first update. This extends to $\frac{\partial C}{\partial \mathbf{W}_2}$ as well. Using weight decay leads to smaller weights by gradually reducing the influence of past gradients. Using momentum however has the opposite effect. The effect of a past gradient increases with every step.

But although both ideas influence how the weight matrix changes, their motivation is quite different. Regularization changes the cost function whereas applying momentum changes how you navigate on the cost function. Adam is similar in this regard to using momentum, but not as easy to write down. The influence of the past gradients depends additionally on an exponentially decaying average of past squared gradients.

Exercise 5: (4 points)a) Implement L^2 regularization in our code:(i) Add a class `L2Regularizer` to `layers.py`:

- It is initialized with a regularization parameter λ .
- It provides a method `update(self, p, dp)` which adds p scaled by the regularization parameter λ to the derivatives computed with backpropagation.

(ii) Add an (optional) attribute `kernel_regularizer` to the `DenseLayer` class which is initialized as `None` by default.(iii) Add the regularization update as additional step in the backpropagation method `backprop` of `DenseLayer`.

b) Compare the effect of the regularization on plain SGD, SGD with momentum, and Adam by expanding example 5c) of exercise sheet 2. What do you observe?

Solution: The implementation of the class `L2Regularizer` and the changes to `DenseLayer` can be found in the following listings.

```

1  def __init__(self,
2      ni, # Number of inputs
3      no, # Number of outputs
4      afun = None, # Activationfunction for the layer
5      optim = None,
6      initializer = None,
7      kernel_regularizer = None
8  ):
9
10     self.ni = ni
11     self.no = no
12
13     if afun is None:
14         self.afun = ReLU()
15     else:
16         self.afun = afun
17
18     if optim is None:
19         self.optim = SGD()
20     else:
21         self.optim = optim
22
23     self.kernel_regularizer = kernel_regularizer
24
25     if initializer is None:
26         self.initializer = RandnAverage()
27     else:
28         self.initializer = initializer
29
30     self.W = self.afun.factor * self.initializer.wfun(ni, no)
31     self.b = np.zeros((no,1))
32     self.__z = None
33     self.__a = None
34     self.dW = np.zeros_like(self.W)
35     self.db = np.zeros_like(self.b)

```

```

1  def backprop(self, delta):
2
3      delta = self.afun.backprop(delta)

```

```

4         self.dW = delta @ self.__a.T
5         self.db = delta @ np.ones((delta.shape[1],1))
6
7         if not self.kernel_regularizer is None:
8             self.kernel_regularizer.update(self.W, self.dW)
9
10        return self.W.T @ delta

```

```

1 class L2Regularizer:
2
3     def __init__(self, l2):
4
5         self.l2 = l2
6
7     def update(self, p, dp):
8
9         dp += self.l2 * p

```

A solution to part b) can be found in the following listing.

```

1 import numpy as np
2 from random import randrange
3 import matplotlib.pyplot as plt
4
5 from networks import SequentialNet
6 from layers import *
7 from activations import *
8 from optimizers import *
9
10 DATA = np.load('mnist.npz')
11 x_train, y_train = DATA['x_train'], DATA['y_train']
12 x_test, y_test = DATA['x_test'], DATA['y_test']
13
14 x_train, x_test = x_train / 255.0, x_test / 255.0
15
16 x = x_train.reshape(60000, 784).T
17 I = np.eye(10)
18 y = I[:, y_train]
19
20 bs, ep, eta = 10, 10, .01
21 l2 = .005
22
23 optim1 = [SGD(eta),
24           SGD(eta, momentum=.9),
25           Adam(eta)]
26
27 optim2 = [SGD(eta),
28           SGD(eta, momentum=.9),
29           Adam()]
30
31 afun1 = Logistic()
32 afun2 = Logistic()
33
34 for opt1, opt2 in zip(optim1, optim2):
35
36     print('Test', opt1.name)
37     layers = [DenseLayer(784, 100, afun=afun1, optim=opt1),

```



```

38         DenseLayer(100, 10, afun=afun2, optim=opt2)]
39     netz = SequentialNet(784, layers)
40     netz.train(x, y, bs, ep)
41
42     y_tilde = netz.evaluate(x_test.reshape(10000, 784, 1))
43     guess    = np.argmax(y_tilde, 1).T
44     print('accuracy =', np.sum(guess == y_test)/100)
45
46     for i in range(4):
47
48         k = randrange(y_test.size)
49         plt.title('Label is {lb}, guess is {gs}'.format(lb=y_test[k], gs=
50             guess[0,k]))
51         plt.imshow(x_test[k], cmap='gray')
52         plt.show()
53
54     print('Test', opt1.name, 'with L2 regularization')
55     layers = [DenseLayer(784, 100, afun=afun1, optim=opt1,
56         kernel_regularizer=L2Regularizer(12)),
57         DenseLayer(100, 10, afun=afun2, optim=opt2,
58         kernel_regularizer=L2Regularizer(12))]
59     netz = SequentialNet(784, layers)
60     netz.train(x, y, bs, ep)
61
62     y_tilde = netz.evaluate(x_test.reshape(10000, 784, 1))
63     guess    = np.argmax(y_tilde, 1).T
64     print('accuracy =', np.sum(guess == y_test)/100)
65
66     for i in range(4):
67
68         k = randrange(y_test.size)
69         plt.title('Label is {lb}, guess is {gs}'.format(lb=y_test[k], gs=
70             guess[0,k]))
71         plt.imshow(x_test[k], cmap='gray')
72         plt.show()

```

L^2 regularization to the Adam optimizers slows the training down, because of the additional operations. The accuracy also decreases. For the logistic activation function and $\lambda = 0.005$ we only have about 70 percent accuracy compared to 97 percent without regularization.