Technische Universität Hamburg
Institut für Mathematik
Lehrstuhl Numerische Mathematik

Jens-Peter M. Zemke
Jonas Grams

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:

```
import numpy as np
from numpy.random import rand, randn

class RandnAverage:

  def wfun(self, ni, no):

    avarage = (ni + no) / 2.0
    return randn(no, ni) * np.sqrt(1.0/avarage)

class RandAverage:

  def wfun(self, ni, no):

```

```
    avarage = (ni + no) / 2.0
    return (rand(no, ni) - .5) * np.sqrt(12.0/avarage)


class RandnOrthonormal:

  def wfun(self, ni, no):

    A = randn(no, ni)
    U, _, V = np.linalg.svd(A, full_matrices=False)
    if ni >= no:
      return V
    else:
      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 1.3[1]). Use classes and subclasses to group activation functions. See the following listing for an example using the already implemented activation functions:

```
import numpy as np

# Heaviside family activation functions
class HeavisideLike:

  def __init__(self):

    self.factor = 1.0

"""
TODO Implement the following activation functions from the
     Heaviside activation function family:
      - Heaviside function
      - Modified Heaviside function
      - Logistic function
      - Exp function (as mentioned on the exercise sheet)
"""

# Sign family activation functions
class SignLike:

  def __init__(self):

    self.factor = 1.0

"""
TODO Implement the following activation functions from the
     Sign activation function family:
      - Sign function
      - TanH function
      - SoftSign function
```

[1] Version from October 20th, 2021

```
"""
# ReLU family activation functions
class ReLULike:

  def __init__(self):

    self.factor = np.sqrt(2)

class ReLU(ReLULike):

  def __init__(self):

    super().__init__()
    self.data = None
    self.name = 'ReLU'

  def evaluate(self, x):

    self.data = x
    return x.clip(min = 0)

  def backprop(self, delta):

    return (self.data >= 0) * delta

"""
TODO Add the following activation functions from the
     ReLU activation function family:
      - leaky ReLU function
      - ELU function
      - SoftPlus function
      - Swish function
"""
# Abs family activation functions
class AbsLike:

  def __init__(self):

    self.factor = 1.0

class Abs(AbsLike):

  def __init__(self, alpha=0.0):

    super().__init__()
    self.data  = None
    self.name  = 'Abs'
    self.alpha = alpha

  def evaluate(self, x):

    self.data = x
    if(self.alpha == 0.0):
      return np.abs(x)
    else:
      return np.sqrt(x*x + self.alpha**2) - self.alpha
```

```python

  def backprop(self, delta):

    if(self.alpha == 0.0):
      return np.sign(self.data)*delta
    else:
      return self.data/np.sqrt(self.data**2 + self.alpha**2) * delta

"""
TODO Add the following activation functions from the
     Abs activation function family:
      - LOCo function
      - Twist function
      - SoftAbs function
"""
```

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 $\mathsf{Exp}(x) := e^x$?

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,

```python
import numpy as np

# Heaviside family activation functions
class HeavisideLike():

  def __init__(self):

    self.factor = 1.0

class Heaviside(HeavisideLike):

    def __init__(self):

        super().__init__()
        self.data = None
        self.name = 'Heavyside'

    def evaluate(self, x):

        return (x >= 0) * 1.0

    def backprop(self, delta):

        return 0.0*delta

```

```python
class ModifiedHeaviside(HeavisideLike):

    def __init__(self):

        super().__init__()
        self.data = None
        self.name = 'Modified Heaviside'

    def evaluate(self, x):

        return (x > 0) * 1.0 + (x == 0) * 0.5

    def backprop(self, delta):

        return 0.0 * delta

class Logistic(HeavisideLike):

    def __init__(self, k=1.0):

        super().__init__()
        self.data = None
        self.name = 'Logistic'
        self.k    = k

    def evaluate(self, x):

        self.data = 1.0/(1.0 + np.exp(-self.k * x))
        return self.data

    def backprop(self, delta):

        return self.k * self.data * (1.0 - self.data) * delta

class SoftMax(HeavisideLike):

    def __init__(self):

        super().__init__()
        self.name = 'SoftMax'
        self.data = None # store data for faster execution

    def evaluate(self, x):
        x_exp = np.exp(x)
        scale = x_exp.sum(axis=1, keepdims=True)
        return x_exp / scale

    def backprop(self, delta):
        return delta

class Exp(HeavisideLike):

    def __init__(self):

        super().__init__()
        self.data = None
```

```
        self.name = 'Exp'

    def evaluate(self, x):

        self.data = np.exp(x)
        return self.data

    def backprop(self, delta):

        return self.data * delta

# Sign family activation functions
class SignLike:

    def __init__(self):

      self.factor = 1.0

class Sign(SignLike):

    def __init__(self):

        super().__init__()
        self.data = None
        self.name = 'Sign'

    def evaluate(self, x):

        return np.sign(x)

    def backprop(self, delta):

        return 0.0 * delta

class TanH(SignLike):

    def __init__(self, k=1.0):

        super().__init__()
        self.data = None
        self.name = 'TanH'
        self.k    = k

    def evaluate(self, x):

        self.data = np.tanh(self.k * x)

        return self.data

    def backprop(self, delta):

        return self.k * (1.0 - self.data**2) * delta

class SoftSign(SignLike):

    def __init__(self, k=1.0):
```

```python

        super().__init__()
        self.data = None
        self.name = 'SoftSign'
        self.k    = k

    def evaluate(self, x):

        self.data = (self.k * x)/(1.0 + np.abs(self.k * x))
        return self.data

    def backprop(self, delta):

        return self.k/(self.data**2) * delta

# ReLU family activation functions
class ReLULike:

    def __init__(self):

      self.factor = np.sqrt(2)

class ReLU(ReLULike):

    def __init__(self):

        super().__init__()
        self.data = None
        self.name = 'ReLU'

    def evaluate(self, x):

        self.data = x
        return x.clip(min = 0)

    def backprop(self, delta):

        return (self.data >= 0) * delta

class leakyReLU(ReLULike):

    def __init__(self, alpha=.01):

        super().__init__()
        self.data  = None
        self.name  = 'leaky ReLU'
        self.alpha = alpha

    def evaluate(self, x):

        self.data = x
        return x.clip(min=0) + self.alpha*x.clip(max=0)

    def backprop(self, delta):

        return   (self.data >= 0) * delta \
```

```
                + (self.data < 0) * self.alpha * delta

class ELU(ReLULike):

    def __init__(self, alpha=1.0):

        super().__init__()
        self.data  = None
        self.name  = 'ELU'
        self.alpha = alpha

    def evaluate(self, x):

        self.data = x
        return x.clip(min=0) + \
               (np.exp(x.clip(max=0)) - 1.0) * self.alpha

    def backprop(self, delta):

        return   (self.data >= 0) * delta \
               + (self.data < 0) * self.alpha * np.exp(self.data) *
                  delta

class SoftPlus(ReLULike):

    def __init__(self, k=1.0):

        super().__init__()
        self.data = None
        self.name = 'SoftPlus'
        self.k    = k

    def evaluate(self, x):

        self.data = np.log(np.exp(self.k * x) + 1.0)/self.k
        return self.data

    def backprop(self, delta):

        return 1.0 / (1.0 + np.exp(-self.data)) * delta

class Swish(ReLULike):

    def __init__(self, k=1.0):

        super().__init__()
        self.data = None
        self.name = 'Swish'
        self.k    = k

    def evaluate(self, x):

        self.data = self.k * x
        y = np.exp(self.data)
        return x * y/(1.0 + y)

```

```
    def backprop(self, delta):

        y = np.exp(self.data)/(1.0 + np.exp(self.data))
        return y * (1.0 + self.data * (1.0 - y)) * delta

# Abs family activation functions
class AbsLike:

    def __init__(self):

      self.factor = 1.0

class Abs(AbsLike):

    def __init__(self, alpha=0.0):

        super().__init__()
        self.data  = None
        self.name  = 'Abs'
        self.alpha = alpha

    def evaluate(self, x):

        self.data = x
        if(self.alpha == 0.0):
            return np.abs(x)
        else:
            return np.sqrt(x*x + self.alpha**2) - self.alpha

    def backprop(self, delta):

        if(self.alpha == 0.0):
            return np.sign(self.data)*delta
        else:
            return self.data/np.sqrt(self.data**2 + self.alpha**2) *
               delta

class LOCo(AbsLike):

    def __init__(self, k=1.0):

        super().__init__()
        self.data = None
        self.name = 'LOCo'
        self.k    = k

    def evaluate(self, x):

        self.data = self.k * x
        return np.log(np.cosh(self.data))/self.k

    def backprop(self,delta):

        return np.tanh(self.data) * delta

class Twist(AbsLike):
```

```

    def __init__(self, k=1.0):

        super().__init__()
        self.data = None
        self.name = 'Twist'
        self.k    = k

    def evaluate(self, x):

        self.data = self.k * x
        return x * np.tanh(self.data)

    def backprop(self, delta):

        y = np.tanh(self.data)
        return y * (1.0 + self.data * (1 - y * y)) * delta

class SoftAbs(AbsLike):

    def __init__(self, k=1.0):

        super().__init__()
        self.data = None
        self.name = 'SoftAbs'
        self.k    = k

    def evaluate(self, x):

        self.data = self.k * x
        return self.data * x / (1.0 + np.abs(self.data))

    def backprop(self, delta):

        y = 1.0 + np.abs(self.data)
        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.

```
import numpy              as np
from   random             import randrange
import matplotlib.pyplot as plt

from networks    import SequentialNet
from layers      import DenseLayer
from activations import *
from optimizers  import *

DATA = np.load('mnist.npz')
x_train, y_train = DATA['x_train'], DATA['y_train']
x_test, y_test   = DATA['x_test'], DATA['y_test']
x_train, x_test = x_train / 255.0, x_test / 255.0

x = x_train.reshape(60000, 784).T
```

```python
I = np.eye(10)
y = I[:, y_train]

bs, ep, eta = 10, 10, .01

functions1 = [Logistic(), TanH(), ReLU(), leakyReLU(),
              ELU(), SoftPlus(), Abs(), LOCo(), Exp()]
functions2 = [Logistic(), TanH(), ReLU(), leakyReLU(),
              ELU(), SoftPlus(), Abs(), LOCo(), Exp()]

for fun1, fun2 in zip(functions1, functions2):

    print('Activation function =', fun1.name)
    layers = [DenseLayer(784, 100, afun=fun1, optim=SGD(eta)),
              DenseLayer(100, 10 , afun=fun2, optim=SGD(eta))]
    netz = SequentialNet(784, layers)
    netz.train(x, y, bs, ep)

    y_tilde = netz.evaluate(x_test.reshape(10000, 784, 1))
    guess   = np.argmax(y_tilde, 1).T
    print('accuracy =', np.sum(guess == y_test)/100)

    for i in range(4):

        k = randrange(y_test.size)
        plt.title('Label is {lb}, guess is {gs}'.format(lb=y_test[k], gs=guess[0,k]))
        plt.imshow(x_test[k], cmap='gray')
        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)

a) Add the optimizer Adam (Lecture notes, section 2.5.7[2]) as class `Adam` to the `optimizers.py` script. It should work similar to the `SGD` class.

b) 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:

```python
import numpy as np

class SGD:

    def __init__(self, eta=.01, momentum=0.0):

        self.eta = eta
        self.momentum = momentum
        if self.momentum > 0.0:
            self.v = []
```

[2]Version from October 20th, 2021

```
            self.name = 'SGD({})␣with␣momentum␣{}'.format(eta, momentum)
        else:
            self.name = 'SGD({})'.format(eta)

    def update(self, data, ddata):

        if self.momentum > 0.0:

            if self.v == []:
                self.v = [np.zeros_like(p) for p in data]

            for v, p, dp in zip(self.v, data, ddata):

                v  = v * self.momentum + dp * self.eta
                p -= v

        else:

          for p, dp in zip(data, ddata):

            p -= self.eta * dp



class Adam:

    def __init__(self, eta =.001, beta1 =.9, beta2 =.999, eps=1e-8):

        self.eta   = eta
        self.beta1 = beta1
        self.beta2 = beta2
        self.eps   = eps

        self.w = []
        self.v = []
        self.k = 0

        self.name = 'Adam'

    def update(self, data, ddata):

        if self.v == []:
            self.v = [np.zeros_like(p) for p in data]
        if self.w == []:
            self.w = [np.zeros_like(p) for p in data]

        self.k += 1
        alpha   = self.eta*np.sqrt(1 - self.beta2**self.k)/(1 - self.beta1**
           self.k)
        for v, w, p, dp in zip(self.v, self.w, data, ddata):

            v = self.beta1*v + (1 - self.beta1)*dp
            w = self.beta2*w + (1 - self.beta2)*dp**2

            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.*

a) 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.
b) What differences do you observe?
c) 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:

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

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 `l2`.
   - It provides a method `update(self, p, dp)` which adds `p` scaled by the regularization parameter `l2` 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.

```
    def __init__(self,
                 ni, # Number of inputs
                 no, # Number of outputs
                 afun = None, # Activationfunction for the layer
                 optim = None,
                 initializer = None,
                 kernel_regularizer = None
    ):

        self.ni   = ni
        self.no   = no

        if afun is None:
            self.afun = ReLU()
        else:
            self.afun = afun

        if optim is None:
            self.optim = SGD()
        else:
            self.optim = optim

        self.kernel_regularizer = kernel_regularizer

        if initializer is None:
          self.initializer = RandnAverage()
        else:
          self.initializer = initializer

        self.W    = self.afun.factor * self.initializer.wfun(ni, no)
        self.b    = np.zeros((no,1))
        self.__z  = None
        self.__a  = None
        self.dW   = np.zeros_like(self.W)
        self.db   = np.zeros_like(self.b)
```

```
    def backprop(self, delta):

        delta   = self.afun.backprop(delta)
```

```
        self.dW = delta @ self.__a.T
        self.db = delta @ np.ones((delta.shape[1],1))

        if not self.kernel_regularizer is None:
            self.kernel_regularizer.update(self.W, self.dW)

        return self.W.T @ delta
```

```
class L2Regularizer:

    def __init__(self, l2):

        self.l2 = l2

    def update(self, p, dp):

        dp += self.l2 * p
```

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

```
import numpy             as np
from   random            import randrange
import matplotlib.pyplot as plt

from networks    import SequentialNet
from layers      import *
from activations import *
from optimizers  import *

DATA = np.load('mnist.npz')
x_train, y_train = DATA['x_train'], DATA['y_train']
x_test, y_test   = DATA['x_test'], DATA['y_test']

x_train, x_test = x_train / 255.0, x_test / 255.0

x = x_train.reshape(60000, 784).T
I = np.eye(10)
y = I[:, y_train]

bs, ep, eta = 10, 10, .01
l2 = .005

optim1 = [SGD(eta),
          SGD(eta, momentum=.9),
          Adam(eta)]

optim2 = [SGD(eta),
          SGD(eta, momentum=.9),
          Adam()]

afun1 = Logistic()
afun2 = Logistic()

for opt1, opt2 in zip(optim1, optim2):

    print('Test', opt1.name)
    layers = [DenseLayer(784, 100, afun=afun1, optim=opt1),
```

```
              DenseLayer(100, 10, afun=afun2, optim=opt2)]
    netz = SequentialNet(784, layers)
    netz.train(x, y, bs, ep)

    y_tilde = netz.evaluate(x_test.reshape(10000, 784, 1))
    guess   = np.argmax(y_tilde, 1).T
    print('accuracy =', np.sum(guess == y_test)/100)

    for i in range(4):

        k = randrange(y_test.size)
        plt.title('Label is {lb}, guess is {gs}'.format(lb=y_test[k], gs=guess[0,k]))
        plt.imshow(x_test[k], cmap='gray')
        plt.show()

    print('Test', opt1.name, 'with L2 regularization')
    layers = [DenseLayer(784, 100, afun=afun1, optim=opt1,
                         kernel_regularizer=L2Regularizer(l2)),
              DenseLayer(100, 10, afun=afun2, optim=opt2,
                         kernel_regularizer=L2Regularizer(l2))]
    netz = SequentialNet(784, layers)
    netz.train(x, y, bs, ep)

    y_tilde = netz.evaluate(x_test.reshape(10000, 784, 1))
    guess   = np.argmax(y_tilde, 1).T
    print('accuracy =', np.sum(guess == y_test)/100)

    for i in range(4):

        k = randrange(y_test.size)
        plt.title('Label is {lb}, guess is {gs}'.format(lb=y_test[k], gs=guess[0,k]))
        plt.imshow(x_test[k], cmap='gray')
        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.