Dubai Campus

F21BC

Biologically Inspired Computation

Coursework 1a

Implementing Gradient Descent

Author:
Heba El-Shimy

Student Number:
H00280277

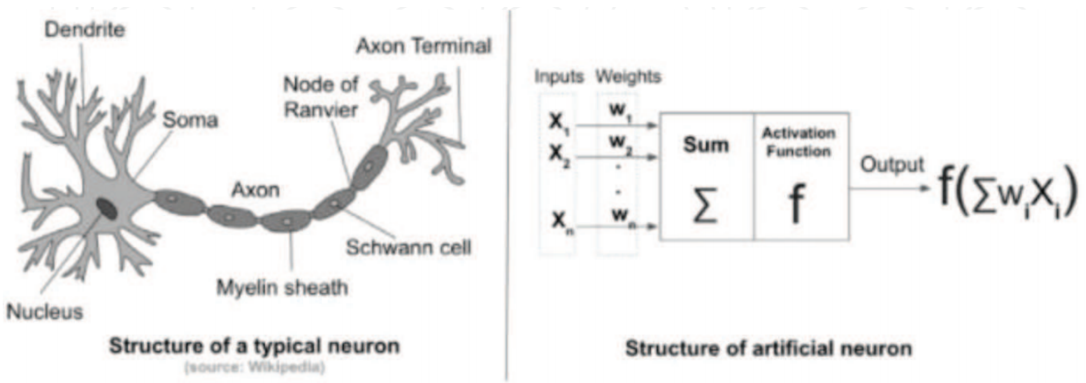
Contents

1 Introduction

This report briefly discusses the concept and implementation of one of the common neural networks learning algorithm, that is, gradient descent. The program accompanying this report represents how gradient descent was used to train a single neuron to identify cat images from non-cat images.

2 General Description

An artificial neuron is a concept derived from biology, particularly, the neurons in the human brain. The brain's neurons are the smallest computational units that accept multiple inputs from other neurons, make a calculation over the inputs and then produce an output that's conducted or passed on to another neuron, as one of its multiple inputs.



Human neuron structure versus artificial neuron [2]

The focus in this report will be the learning process of a single artificial neuron. The neuron itself is just a storage unit, that holds a computed value. As for the learning process, it is divided into two main parts, the feedforward part and the backpropagation.

The feedforward part is responsible for collecting all input values, adding different weights to them according to their importance in relation to the output, adding a bias value and finally summing all these values up and passing them through an activation function which defines a threshold that if exceeded by the calculated sum, then the neuron will fire, i.e. return true/1 as output and false otherwise.

The first step of the feedforward process can be written as follows:

$$z = \Sigma w_i x_i + b$$

where w_i is the ith item in the weights vector, x_i is the ith item in the input vector (for a single sample) and b is the bias vector.

The activation function used in this coursework is the *Sigmoid* function, which outputs a value of [0,1]. It can be written as follows:

$$a(z) = \frac{1}{1 + e^{-z}}$$

After finishing the feedforward step, we will calculate the loss, i.e., how far away was the algorithm from the true value. This is crucial for the model to adjust and find out better values for the weights and bias that enhance the algorithm's accuracy.

The loss function used in this coursework is the *Cross-Entropy* loss. It suits the problem we have in classifying images to cat and non-cat as it is a *binary classification* problem.

$$L(y, a) = -(ylog(a) + (1 - y)log(1 - a))$$

where a is the predicted value and y is the actual value. As we are dealing with m number of samples, we will need to take the sum of all losses across all samples.

$$J(w, b) = -\frac{1}{m} \sum_{i=1}^{m} L(y^i, a^i)$$

The second step in the training process is the backpropagation, and it's where the actual "learning" happens. The intuition behind this step is that calculated cost is sent back to the neuron to readjust its weights. The way this is handled is as an optimization problem, where the algorithm will try to minimize the cost and find the optimal weights and bias that gives the best performance.

The algorithm used for this learning process is *Gradient Descent*, which finds the derivative of the cost function and keeps taking steps iteratively towards the a minimum. *Gradient Descent* is a good algorithm to use with *Cross-Entropy* loss function as it is a perfect convex function with only one minimum, and since *Gradient Descent* is a local search algorithm, we can be sure that it will find the right solution and not be stuck into a local minimum while there is possible other global minimum as can be seen in other loss functions.

3 Choice of model parameters

The selection of the algorithm's parameters have been done after multiple experiments with different sets of parameters. Below is a table listing the different experiments that have been done along with the corresponding parameters and the final result represented as the accuracy.

- Weights initialization:
I tried different methods for initializing the weights vector by creating random values for the weights. Some of methods I followed were suggested by research and some were random trials by combining those suggestions. The suggested methods by research were:
a) Random values between -0.5 and 0.5[4]
b) Random values between -0.05 and 0.05[3]
c) Random values between 0 and 1 multiplied by a value of 0.01, but I changed that to 0.1[1]
d) Random values between 0 and 1 divided by the square root of the number of inputs, but I changed the range of values to be between [-0.5, 0.5] or [-0.05, 0.05][1]

- Learning Rate:
The initial learning rate was chosen randomly, starting by a value of 1 and by using trial and error reducing that to 0.1, 0.01 and 0.001. Bigger values, i.e, 1, tended to give errors when calculating the cost, where it was calculated as nan, so I excluded 1 as a learning rate after the first experiment.

- Learning Rate Decay:
There were several methods suggested by research for choosing a value of learning rate decay[1], I chose two of these methods:
a) Fixed decay by a value of 0.5 every 5 or 100 iterations

b) Constant decay by a value of 0.5 if cost does not improve after 50 or 100 iterations.

- Epochs:
The number of iterations the algorithm has to go through to optimize the weights and bias was chosen at random. I switched between 100 and 1000 epochs. I tried larger values of 10000, 20000 epochs but the improvement of model performance was negligible and did not justify the compute power used to run the iterations.

Experiment Number	Initial weights	Initial learning Rate	LR decay*	Number of Epochs	Cost at last epoch	Accuracy
01	$W = [-0.5, 0.5]$ [4]	1	lr /=2, epochs %5	100	nan	64%
02	$W = [-0.5, 0.5]$ [4]	0.1	lr /=2, epochs %5	100	2.04	56%
03	$W = [-0.5, 0.5]$ [4]	0.01	lr /=2, epochs %5	1000	2.80	40%
04	$W = [-0.05, 0.05]$ [3]	0.1	lr /=2, epochs %5	100	0.58	64%
05	$W = [-0.05, 0.05]$ [3]	0.1	lr /=2, epochs %5	1000	0.58	64%
06	$W = [-0.05, 0.05]/sqrt(n)$ [3][1]	0.1	lr /=2, epochs %5	100	0.56	58%
07	$W = [-0.5, 0.5]/sqrt(n)$ [4][1]	0.1	lr /=2, epochs %5	100	0.50	60%
08	$W = [-0.05, 0.05]/sqrt(n)$ [3][1]	0.01	lr /=2, epochs %50**	1000	0.41	64%
09	$W = [0, 1]/sqrt(n)$[1]	0.1	lr /=2, epochs %100***	1000	nan	70%
10	$W = [0, 1]/sqrt(n)$[1]	0.01	lr /=2, epochs %100***	1000	0.12	72%
11	$W = 0.1 * [0, 1]$[1]	0.01	lr /=2, epochs %50****	1000	0.14	76%
12	$W = [-0.5, 0.5]/sqrt(n)$ [4][1]	0.1	lr /=2, epochs %100	1000	0.14	78%

* Learning rate decays every few iterations[1]
** If cost does not improve by 0.5 for 50 iterations
*** If cost does not improve by 0.001 for 100 iterations
**** If cost does not improve by 0.025 for 50 iterations

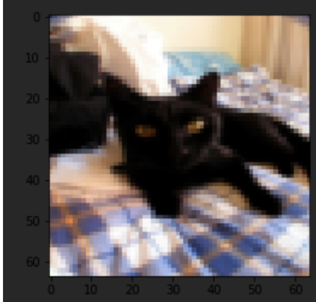
4 Results of the algorithm on the test dataset

Using the modified weights and bias from the previous training process to predict the output of a new "test" dataset achieved satisfactory results. I also collected a few pictures that were not in the dataset, then I preprocessed them as done previously with the train and test sets, i.e., I flattened then normalized them. I used them for testing the modified weights.

Giving the model more iterations to learn and train on images improved the overall performance as well as adjusting the learning rate to be adaptive, which allowed the model to take smaller steps when nearing the optimal/minimum cost, which aided in increasing the algorithm's accuracy and not overshooting and missing the optima.

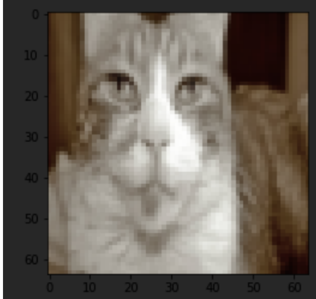
Below are some screenshots of the output of testing on new images.

```
# Example of a picture
index = int(np.random.randint(low=0, high=49))
plt.imshow(testSetX[index])
plt.show()

class_name = [0, 0]

if round(test_pred.ravel()[index]) == 0:
    class_name[0] = 'non-cat'
elif round(test_pred.ravel()[index]) == 1:
    class_name[1] = 'cat'

print ("y = " + str(testSetY[:, index]) +
       ", predicted class= " + str(round(test_pred.ravel()[index])) +
       ", it's a '" + class_name[int(round(test_pred.ravel()[index]))] +  "' picture.")
```



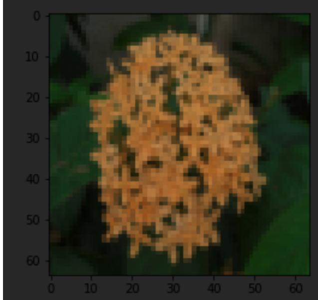

```
y = [1], predicted class= 1.0, it's a 'cat' picture.
```

```python
# Example of a picture
index = int(np.random.randint(low=0, high=49))
plt.imshow(testSetX[index])
plt.show()

class_name = [0, 0]

if round(test_pred.ravel()[index]) == 0:
    class_name[0] = 'non-cat'
elif round(test_pred.ravel()[index]) == 1:
    class_name[1] = 'cat'

print ("y = " + str(testSetY[:, index]) +
       ", predicted class= " + str(round(test_pred.ravel()[index])) +
       ", it's a '" + class_name[int(round(test_pred.ravel()[index]))] +  "' picture.")
```



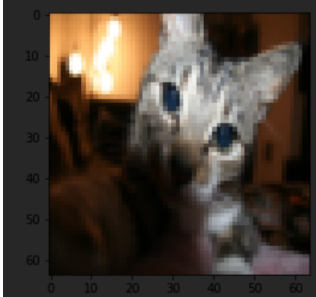

```
y = [1], predicted class= 1.0, it's a 'cat' picture.
```

```python
# Example of a picture
index = int(np.random.randint(low=0, high=49))
plt.imshow(testSetX[index])
plt.show()

class_name = [0, 0]

if round(test_pred.ravel()[index]) == 0:
    class_name[0] = 'non-cat'
elif round(test_pred.ravel()[index]) == 1:
    class_name[1] = 'cat'

print ("y = " + str(testSetY[:, index]) +
       ", predicted class= " + str(round(test_pred.ravel()[index])) +
       ", it's a '" + class_name[int(round(test_pred.ravel()[index]))] +  "' picture.")
```




```
y = [0], predicted class= 0.0, it's a 'non-cat' picture.
```

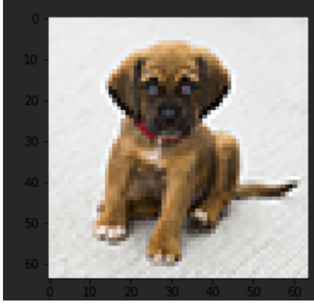
```python
# Example of a picture
index = int(np.random.randint(low=0, high=49))
plt.imshow(testSetX[index])
plt.show()

class_name = [0, 0]

if round(test_pred.ravel()[index]) == 0:
    class_name[0] = 'non-cat'
elif round(test_pred.ravel()[index]) == 1:
    class_name[1] = 'cat'

print ("y = " + str(testSetY[:, index]) +
       ", predicted class= " + str(round(test_pred.ravel()[index])) +
       ", it's a '" + class_name[int(round(test_pred.ravel()[index]))] +  "' picture.")
```



```
y = [1], predicted class= 1.0, it's a 'cat' picture.
```

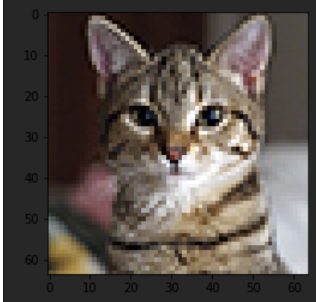
```python
image_1 = plt.imread(imgs[0])
plt.imshow(image_1)

# Flatten and normalize image
image_1FN = image_1.reshape(-1).T / 255
print(image_1FN.shape)

# Use optimized weights for predicting the ouput
test_1 = sigmoid(np.dot(W_mod.T, image_1FN) + b_mod)

class_test_1 = [0, 0]

if round(test_1[0]) == 0:
    class_test_1[0] = 'non-cat'
elif round(test_1[0]) == 1:
    class_test_1[1] = 'cat'

print ("y = [0]"  +
       ", predicted class= " + str(round(test_1[0])) +
       ", it's a '" + class_test_1[int(round(test_1[0]))] +  "' picture.")
```

```
(12288,)
y = [0], predicted class= 0.0, it's a 'non-cat' picture.
```



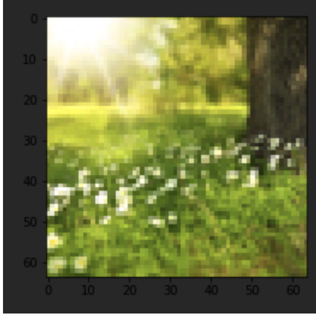
```
image_2 = plt.imread(imgs[1])
plt.imshow(image_2)

# Flatten and normalize image
image_2FN = image_2.reshape(-1).T / 255
print(image_2FN.shape)

# Use optimized weights for predicting the ouput
test_2 = sigmoid(np.dot(W_mod.T, image_2FN) + b_mod)

class_test_2 = [0, 0]

if round(test_2[0]) == 0:
    class_test_2[0] = 'non-cat'
elif round(test_2[0]) == 1:
    class_test_2[1] = 'cat'

print ("y = [0]"  +
       ", predicted class= " + str(round(test_2[0])) +
       ", it's a '" + class_test_2[int(round(test_2[0]))] +  "' picture.")
```

```
(12288,)
y = [0], predicted class= 1.0, it's a 'cat' picture.
```



```
image_3 = plt.imread(imgs[2])
plt.imshow(image_3)

# Flatten and normalize image
image_3FN = image_3.reshape(-1).T / 255
print(image_3FN.shape)

# Use optimized weights for predicting the ouput
test_3 = sigmoid(np.dot(W_mod.T, image_3FN) + b_mod)

class_test_3 = [0, 0]

if round(test_3[0]) == 0:
    class_test_3[0] = 'non-cat'
elif round(test_3[0]) == 1:
    class_test_3[1] = 'cat'

print ("y = [0]"  +
       ", predicted class= " + str(round(test_3[0])) +
       ", it's a '" + class_test_3[int(round(test_3[0]))] +  "' picture.")
```

```
(12288,)
y = [0], predicted class= 0.0, it's a 'non-cat' picture.
```



5 Marking Scheme

COURSEWORK 1a			
%1	QUESTIONS	S_MARK	FINAL
1	Did you code the entire algorithm?	x	
2	Did you vectorise your code?	x	
3	Is your code runnable?	x	
4	Did you collect data to measure convergence?	x	
5	Did you create a report explaining what you did?	x	
Total =			

6 Conclusions

Using the biological concept of a neuron and training it using backpropagation allowed for solving complex problems that were not easy to solve using rule-based systems, nor simple machine learning statistical functions. Artificial neural networks perform well in problems that relate to computer vision where a huge input is expected in the form of pixel data. I discussed in this report the building block of ANNs which is a single neuron and how to train it to classify cat images.

7 Appendices

7.1 Program Code

```python

# coding: utf-8

# # Gradient Descent
#
# **F21BC Coursework 1**
#
# <sub>Name: **Heba El-Shimy**</sub>
# <br>
# <sub>Based on code obtained from coursework specfication report, written by **Dr. Marta Vallejo**</sub>

# In[1]:


# Import libraries
import numpy as np
import matplotlib.pyplot as plt
import h5py
import os
from math import sqrt


# In[2]:


np.set_printoptions(precision=2)


# In[3]:


# Loading the dataset
os.getcwd()
train_dataset = h5py.File('trainCats.h5', "r")
trainSetX = np.array(train_dataset["train_set_x"][:]) # your train set features
trainSetY = np.array(train_dataset["train_set_y"][:]) # your train set labels
trainSetY = trainSetY.reshape((1, trainSetY.shape[0]))

test_dataset = h5py.File('testCats.h5', "r")
testSetX = np.array(test_dataset["test_set_x"][:]) # your test set features
```

```
testSetY = np.array(test_dataset["test_set_y"][:]) # your test set labels
testSetY = testSetY.reshape((1, testSetY.shape[0]))

classes = np.array(test_dataset["list_classes"][:]) # the list of classes


# In[4]:


# Example of a picture
index = 20
plt.imshow(trainSetX[index])
plt.show()
print ("y = " + str(trainSetY[:, index]) + ", it's a '" + classes[np.squeeze(trainSetY[:, index])].decode("utf-8") +  "' picture.")


# In[5]:


# Images dimensions
print(trainSetX.shape)
print(testSetX.shape)

print('Image dimensions: {}px x {}px '.format(trainSetX.shape[1], trainSetX.shape[2]))
print('Image channels: {}'.format(trainSetX.shape[-1]))
print('Number of training examples: {} images'.format(trainSetX.shape[0]))
print('Number of test examples: {} images'.format(testSetX.shape[0]))


# In[6]:


# Flatten the pictures
# Applying (num_pixel x num_pixel x num_channels)
trainSetXF= trainSetX.reshape(trainSetX.shape[0], -1).T
testSetXF = testSetX.reshape(testSetX.shape[0], -1).T

print('Shape of training data after flattening: {}'.format(trainSetXF.shape))
print('Shape of test data after flattening: {}'.format(testSetXF.shape))


# In[7]:
```

```python


# Normalize images
# Applying (pixel_value/255)
trainSetXFN = trainSetXF / 255
testSetXFN = testSetXF / 255

print('Shape of training data after normalizing: {}'.format(trainSetXFN
    .shape))
print('Shape of test data after normalizing: {}'.format(testSetXFN.
    shape))

print('First row of training data before normalizing: \n{}\n'.format(
    trainSetXF[0]))
print('First row of training data after normalizing: \n{}\n'.format(
    trainSetXFN[0]))

print('First row of test data before normalizing: \n{}\n'.format(
    testSetXF[0]))
print('First row of test data after normalizing: \n{}\n'.format(
    testSetXFN[0]))


# In[8]:


# Network Topology
print('Number of input units: {}'.format(trainSetXFN.shape[0]))
print('Number of outputs: {}'.format(classes.shape[0]))


# In[336]:


# Initialize weights
W = np.random.uniform(low=-0.5, high=0.5, size=(trainSetXFN.shape[0],
    1)) / sqrt(trainSetXFN.shape[1])

print('Shape of weights matrix: {}'.format(W.shape))
print('Range of values in weights matrix = [{} - {}]'.format(W.min(), W
    .max()))
print('First (only) column in weights matrix: \n{}'.format(W))


# In[26]:


# Initialize biases
b = np.zeros([1, ])
```

```

#print('Shape of bias vector: {}'.format(b.shape))
print('First value in bias vector: \n{}'.format(b))


# In[21]:


# Activation function
# Sigmoid

def sigmoid(z):
    """
    Compute sigmoid function
    @param z: value to compute sigmoid for (WX + b)
    """

    a = np.zeros([1, 1])
    a = 1 / (1 + np.exp(-z))

    return a


# In[22]:


# Cost calculation
# Cross-Entropy as the loss function

def cost(a, y):
    """
    Compute loss function
    @param a: predicted label
    @param y: actual label
    """

    L = np.sum((y * np.log(a)) + ((1 - y) * np.log(1 - a)))

    J = (-1 / y.shape[1]) * L

    return J


# In[338]:


# Training the neuron

W_mod = np.copy(W) # modified weights matrix
```

```
b_mod = np.copy(b) # modidied bias vector
lr = 0.1 # learning rate
epochs = 1000 # number of iterations
costs = [] # store all calculated costs

# Training iterations
for i in range(epochs):
    J = 0
    dW = np.zeros(W_mod.shape)
    db = b_mod

    z = np.dot(W_mod.T, trainSetXFN) + b_mod
    predicted_labels = sigmoid(z)

    J = cost(predicted_labels, trainSetY)
    dW = (1 / trainSetY.shape[1]) * np.dot(trainSetXFN, (predicted_labels - trainSetY).T)
    db = (1 / trainSetY.shape[1]) * np.sum((predicted_labels - trainSetY), axis=1)

    costs.append(J)
    # learning rate decay
    if i > 0:
        if i % 100 == 0: #and abs(costs[i-50] - J) <= 0.5:
            lr /= 2

    W_mod = W_mod - (lr * dW)
    b_mod = b_mod - (lr * db)
    print('Learning rate: {}'.format(lr))
    print('Iteration {}\t ======> \t Cost: {:.2f}\n'.format(i, J))


# In[339]:


# Test the model on the test set
test_pred = sigmoid(np.dot(W_mod.T, testSetXFN) + b_mod)

# Measure model accuracy
count = 0
for pred, label in zip(test_pred.ravel(), testSetY.ravel()):
    if round(pred) == label:
        count += 1

accuracy = float(count) / testSetY.shape[1] * 100
print('Model\'s accuracy = {:.2f}%'.format(accuracy))


# In[343]:
```

```
# Example of a picture
index = int(np.random.randint(low=0, high=49))
plt.imshow(testSetX[index])
plt.show()

class_name = [0, 0]

if round(test_pred.ravel()[index]) == 0:
    class_name[0] = 'non-cat'
elif round(test_pred.ravel()[index]) == 1:
    class_name[1] = 'cat'

print ("y = " + str(testSetY[:, index]) +
       ", predicted class= " + str(round(test_pred.ravel()[index])) +
       ", it's a '" + class_name[int(round(test_pred.ravel()[index]))]
    + "' picture.")


# In[405]:


import glob
import cv2
import os
import matplotlib.image as mpimg

# Experimenting on collected images

img_dir = "./images"
img_files = glob.glob(os.path.join(img_dir,'*.jpg'))
imgs = []

for img in img_files:
    imgs.append(img)


# In[406]:


image_1 = plt.imread(imgs[0])
plt.imshow(image_1)

# Flatten and normalize image
image_1FN = image_1.reshape(-1).T / 255
print(image_1FN.shape)

# Use optimized weights for predicting the ouput
```

```
test_1 = sigmoid(np.dot(W_mod.T, image_1FN) + b_mod)

class_test_1 = [0, 0]

if round(test_1[0]) == 0:
    class_test_1[0] = 'non-cat'
elif round(test_1[0]) == 1:
    class_test_1[1] = 'cat'

print ("y = [0]" +
       ", predicted class= " + str(round(test_1[0])) +
       ", it's a '" + class_test_1[int(round(test_1[0]))] + "' picture
   .")


# In[407]:


image_2 = plt.imread(imgs[1])
plt.imshow(image_2)

# Flatten and normalize image
image_2FN = image_2.reshape(-1).T / 255
print(image_2FN.shape)

# Use optimized weights for predicting the ouput
test_2 = sigmoid(np.dot(W_mod.T, image_2FN) + b_mod)

class_test_2 = [0, 0]

if round(test_2[0]) == 0:
    class_test_2[0] = 'non-cat'
elif round(test_2[0]) == 1:
    class_test_2[1] = 'cat'

print ("y = [0]" +
       ", predicted class= " + str(round(test_2[0])) +
       ", it's a '" + class_test_2[int(round(test_2[0]))] + "' picture
   .")


# In[408]:


image_3 = plt.imread(imgs[2])
plt.imshow(image_3)

# Flatten and normalize image
image_3FN = image_3.reshape(-1).T / 255
```

```
print(image_3FN.shape)

# Use optimized weights for predicting the ouput
test_3 = sigmoid(np.dot(W_mod.T, image_3FN) + b_mod)

class_test_3 = [0, 0]

if round(test_3[0]) == 0:
    class_test_3[0] = 'non-cat'
elif round(test_3[0]) == 1:
    class_test_3[1] = 'cat'

print ("y = [0]" +
       ", predicted class= " + str(round(test_3[0])) +
       ", it's a '" + class_test_3[int(round(test_3[0]))] + "' picture.")
```

7.2 How to run the code

The code was written in a Jupyter Notebook as it is the best medium for viewing the results of fragments of code and iterate and test. To view the notebook, there should be installed on your computer a scientific python package, for example Anaconda.

Download the notebook accompanied by this report and in a terminal window (unix) or cmd (windows), navigate to the folder that contains the recently downloaded `.ipynb` file and type `jupyter notebook`. This will spin up a local server on port 8888. You can view the notebook from your browser by navigating to http://localhost:8888 and clicking on the notebook name `GradientDescent.ipynb`.

Also, a copy of the notebook with the outputs has been saved as a `.html` file with the name `GradientDescent.html` and has been uploaded with this report. It can be viewed it in any browser.

References

[1] Cs231n convolutional neural networks for visual recognition, 2018.

[2] Adel El-Shahat. Introductory chapter: Artificial neural networks. In Adel El-Shahat, editor, *Advanced Applications for Artificial Neural Networks*, chapter 1. IntechOpen, Rijeka, 2018.

[3] Mercedes Fernández-Redondo and Carlos Hernandez Espinosa. Weight initialization methods for multilayer feedforward, 01 2001.

[4] Marta Vallejo. Biologically inspired computation, 2018.