Факултет инжењерских наука Универзитета у Крагујевцу

Назив студијског програма: Рачунарска техника и софтверско инжењерство

Ниво студија: Основне академске студије

Модул: Програмски преводиоци

Предмет: Програмски преводиоци

Број индекса: 579/2015

Ленка В. Милић

Предвиђање биљних болести биљке малине употребом конволуција у дубоким неуронским мрежама

Дипломски рад

Комисија за преглед и одбрану:

1. **Др. Владимир Миловановић - ментор**
2. _____ Датум одбране:_____
3. _____ Оцена:_____

У оквиру овог завршног рада кандидат треба да

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Препоручена литература:

[1] .....

[2] .....

[3] .....

...

Крагујевац, датум

ментор:

Владимир Миловановић Др.

_____

Резиме

Рад је посвећен малим фармерима на подручју централне и западне Србије који на својим парцелама поседују узгоје биљке малине. „Ова биљка је изложена патогенима као што су вируси, бактерије и гљивице због чега су потребне адекватне мере како би биљка била заштићена или излечена. Биљне болести узроковане патогенима узрокују значајне губитке приноса заседа у Србији која је на свету једна од највећих произвођача и извозника овог воћа (1).” Стога је сваки допринос овој пољопривредној грани од јаке важности. Сврха овог рада је приказ примене дубоког учења у класификацији биљних болести малине према сликама заражених листова, стабљика и плодова. Техником дубоког учења креира се неуронска мрежа чији се параметри тренирају до оптималних вредности. Резултат неуронске мреже на основу једног улазног податка је предвиђена биљна болест малине, односно здрава малина. Овакав закључак може бити искоришћен зарад примене одговарајућих мера као што је орезивање, коришћење хемијских препарата (и слично) како би се спречио даље развијање биљне болести.

Кључне речи

Неуронска мрежа, параметри, мини серија улазних и излазних података, конволуција, потпуно повезана мрежа, пропагација, активациона функција, линеарна функција трошка параметара, трошак параметара, тренирање параметара.

Summary

The paper is dedicated to small farmers in central and western Serbia who own raspberry plants on their plots. „This plant is exposed to pathogens such as viruses, bacteria and fungi, so adequate measures are needed to protect or cure the plant. Plant diseases caused by pathogens cause significant losses of yields in Serbia, which is one of the largest producers and exporters of this fruit in the world (1).” Therefore, any contribution to this agricultural branch is of great importance. The purpose of this paper is to demonstrate the application of deep learning in the classification of raspberry plant diseases according to images of infected leaves, stems and fruits. A deep learning technique creates a neural network whose parameters are trained to optimal values. The result of the neural network, based on one input, is the predicted herbal raspberry disease, that is, healthy plant raspberry. Such a conclusion can be used to implement appropriate measures such as pruning, the use of chemical preparations (and the like) to prevent the further development of plant disease.

Key words

Neural network, parameters, mini-series of input and output data, convolution, fully connected network, propagation, activation function, linear parameter cost function, parameter cost, parameter training.

Садржај

1. Увод

Задатак рада је да обухвати тринаест стања биљке малине од којих прво стање описује здраву биљку малине, а сва остала упућују на неку специфичну болест, вирус или стање које је узроковано малинином мушицом:

- Здрава малина
- Антракоза малине (*Anthracnose,* енгл.)
- Бактериозни рак - (*Agrobacterium tumefaciens*, лат.)
- Сива трулеж (*Gray mold*, енгл.)
- Бактериозна пламењача (*Erwinia amylovora,* лат.)
- Стање малине узроковано малинином мушицом
- Наранџаста рђа (*Orange rust,* енгл.)
- Пламењача корена малине (*Phytophthora,* енгл.)
- Љубичаста пегавост / Сушење изданка малине (*Didymella applanata* / *Cane blight,* енгл.)
- Вирус патуљасти жбун – RBDV (*Raspberry Bushy dwarf virus,* енгл.)
- Вирус тачкастог листа (*Leaf spot virus*, енгл.)
- Мозаик вирус (*Mosaic virus,* енгл.)
- Вирус коју узрокује коврџање листа малине (*Leaf curl virus,* енгл.)

Улазне параметре у мрежи (*атрибути*, слика 1), чине вредности пиксела слика заражених листова, стабљика и плодова. Укупан број атрибута је 120 хиљада за сваку од инстанци на улазу. Слике су тродимензионе, RGB слике облика (200, 200, 3).
Број инстанци који дефинише укупан број слика у проблему *предвиђања биљних болести малине* је 2341.
Излаз неуронске мреже су бинарне вредности према чему је једна од вредности излазног вектора 1 и дефинише установљену биљну болест малине или здраву биљку малине, а преосталих 12 вредности на излазу су нуле и негирају постојање преосталих стања. Класификација одређеног стања регулисана је нумерички и односи се на позицију неурона у последњем слоју неуронске мреже.

Слика 1 :
Опис скупа података

2. Потребни софтвери и библиотеке

Софтвер:

– Anaconda (платформа за Python науку о подацима)

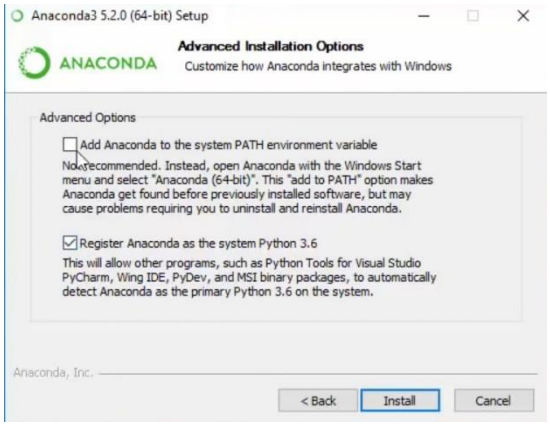
Библиотеке:

– numpy
- scikit-learn
- matplotlib
- PIL
- math
- io
- jupyter notebook

Техника дубоког учења (или хијерархијског учења) базира се на учењу репрезентације података. Величина улазног скупа података неуронске мреже може достићи велике размере при чему време извршавања програма треба остати ефективно. Управо из тог разлога за проблем *предвиђања биљних болести малине* коришћена је Python библиотека *numpy*. Ова библиотека служи за брзе компутације између структура података које садрже велики број елемената. Улазни скуп података неуронске мреже представљен је као једна таква структура, односно као *n*-димензиони *numpy* вектор.
Компутације се на нивоу векторских структура имплементирају помоћу векторизације и *broadcast*-инга коју омогућава ова библиотека. Уметност векторизације у Python-у омогућава процесуирање *n*-димензионих *numpy* вектора без коришћења *foor* петљи. Коришћење споре *foor* петље у области дубоког учења је јако непожељно јер успорава извршавање програма односно тренирање параметара неуронске мреже. *Numpy* операције над *n*-димензионим *numpy* векторима омогућавају и до 300 пута мање време које је потребно за извршавања програма од оног времена извршавања када би се уместо операција ове библиотеке користила *foor* петља зарад истог резултата.

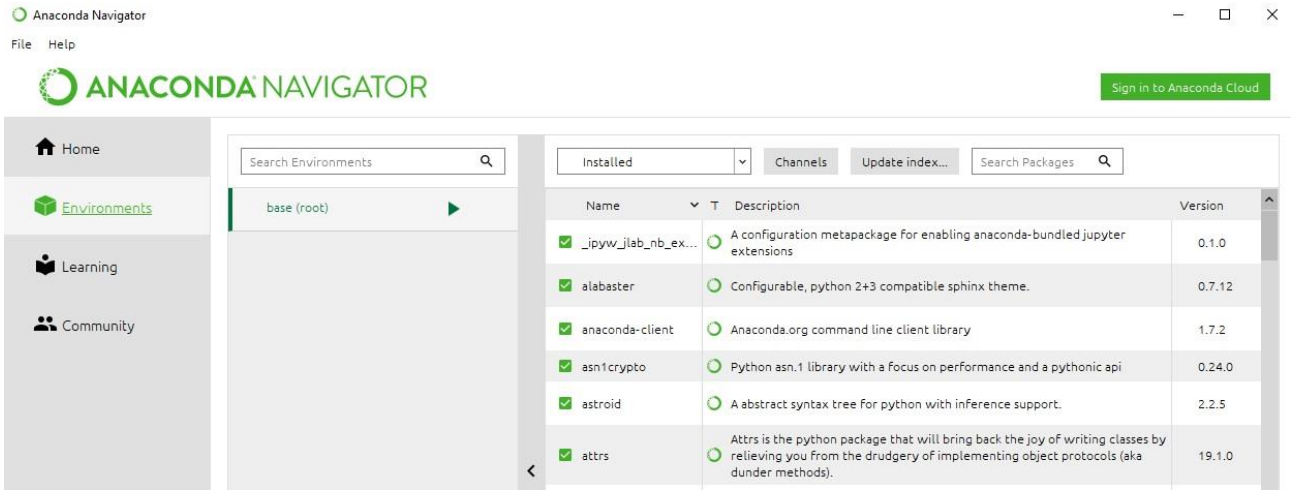
Проблем *предвиђања биљних болести малине* имплементиран је коришћењем Anaconda 2019 платформе. Ова платформа се користи за развијање, тестирање и тренинг и омогућава брзо преузимање python библиотека, анализу података, визуализацију резултата, итд.
На следећем веб линку могуће је преузети једну од Anaconda дистрибуција :
https://www.anaconda.com/distribution/

У току инсталације квадрат поред описа *«Add Anaconda to the system PATH environment variable»* се не селектује уколико на рачунару већ постоји инсталиран Python (слика 2).



Слика 2:
Инсталација Anaconda 2019 дистрибуције

Након инсталације софтвера у Windows старт менију избором Anaconda navigator-а приказује се прозор као на слици 3 и дифолт окружење *base (root).*



Слика 3 :
Преглед окружења Anaconda navigator

Библиотеке је могуће инсталирати у оквиру окружења Anaconda navigator (преглед инсталираних библиотека: Environments → base(root) окружење или виртуално окружење уколико је креирано → Installed) или у Anaconda конзоли.

Инсталирање пакета у оквиру Anaconda конзоле :

- `conda install -c anaconda numpy`
- `conda install -c anaconda scikit-learn`
- `conda update scikit-learn`
- `conda install -c conda-forge matplotlib`
- `conda install -c anaconda pil`
- `conda install -c anaconda jupyter`

Радно окружење јупитерова свеска (jupyter notebook, енгл.) је веб апликација отвореног кода која омогућава интерактиван код, једначине, визуализацију и наративан текст на једном месту.

Покретање радног окружења, у оквиру локалног сервера на рачунару, у Anaconda конзоли :

- `jupyter notebook`

Потребне библиотеке су учитане на почетку свеске као на слици 4

Покретање програма *"raspberries_proj.ipynb"* се извршава :

- кликом на опцију `Run` (*једна по једна линија*) или
- `Kernel` → `Restart and Run All` (*све линије*)

```
In [1]: import os
        import math
        from matplotlib import image
        from PIL import Image
        from sklearn.model_selection import train_test_split
        import matplotlib.pyplot as plt
        import numpy as np

        from activations import *
        from helpers1 import solve_fc, solve_conv, solve_all, solve_all_grads
        from helpers2 import *
        from helpers_conv import *
        from helpers_fc import *


        %matplotlib inline
        plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
        plt.rcParams['image.interpolation'] = 'nearest'
        plt.rcParams['image.cmap'] = 'gray'

        %load_ext autoreload
        %autoreload 2

        np.random.seed(1)
```

Слика 4 :
Учитавање свих библиотека у оквиру јупитерове свеске и подешавање параметара за штампање графика

3. Обрада и припрема података

Улазни подаци неуронске мреже су вредности пиксела слика у опсегу од 0 до 255. Преузимање вредности пиксела слика из свих под-директоријума директоријума *„pics”* омогућава функција *readAllPixels* која преузима укупан број под-директоријума, *num_of_dirs* и имена свих под-директоријума дефинисаних у листи *dir_names* (слика 5). Ова функција враћа укупан број успешно учитаних слика *m* и листу *pixels* која садржи тачно тринаест под-листа. Ових тринаест под-листа појединачно садрже вредности пиксела слика које се односе на одређену биљну болест малине (слика 6).

```
def readAllPixels(num_of_dirs,dir_names):
    pixels_all = list()
    m = 0
    for i in range(num_of_dirs):
        pixels_onefolder = list()
        for filename in listdir(dir_names[i]):
            try:
                image = Image.open(dir_names[i] + '/' + filename)
                data = asarray(image) # convert image to numpy array
                pixels_onefolder.append(data)
                m = m + 1
            except IOError:
                print("error opening an image: %s",filename)
                pass
        pixels_all.append(pixels_onefolder)
    return pixels_all, m
```

Слика 5 :

Учитавање вредности пиксела и пребројавање укупног броја слика

```
In [2]: dir_names = [
        "pics/Zdrava_malina/",
        "pics/Antrakoza_maline_Anthracnose/",
        "pics/Bakteriozni_rak_Crown_Gall_and_Cane_Gall/",
        "pics/Siva_trulez_Cane_botrytis_Botrytis_cinerea_Gray_Mold_Powdery/",
        "pics/Bakteriozna_plamenjaca_Erwinia_amylovora/",
        "pics/Malinina_musica_galica_Lasioptera_rubi/",
        "pics/Narandzasta_rdja_Orange_Rust_Phragmidium/",
        "pics/Plamenjača_korena_maline_Fitoftora_Fragariae_var_rubi_Phytophthora/",
        "pics/Ljubicasta_pegavost_Sušenje_izdanaka_maline_Didymella_applanata_Spur_blight_and_Cane_blight/",
        "pics/virus/Raspberry_bushy_dwarf_RBDV/",
        "pics/virus/Tackasti_list_Leaf_spot/",
        "pics/virus/uzrok_vasi/Mozaik_Raspberry_mosaic/",
        "pics/virus/uzrok_vasi/Kovrdzanje_listova_Raspberry_leaf_curl/"
        ]

In [3]: # Get all pixel values
        num_of_dirs = len(dir_names)
        pixels, m = readAllPixels(num_of_dirs,dir_names)
```

Слика 6:

Складиштење вредности пиксела и укупног броја слика m

Numpy вектор *x* садржи све улазне податке у облику вектора димензије (m, 200, 200, 3). Излазни подаци представљени су вектором *y* облика (m, num_of_dirs) и креирани су према распореду вредности пиксела у листи *pixels* (слика 7).

Излазни вектор за скупове вредности пиксела који се класификирају као
прво стање биљке малине (стање здраве малине) је :
`[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.],`
друго стање биљке малине (биљна болест Антракоза) је :
`[0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.],`
...
последње, тринаесто стање биљке малине (вирус коврџања листа малине) је :
`[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]`

```
In [6]: # Create dataset 'x' - store all pixel values from 'pixels' in one vector 'x'
        # Create dataset 'y'
        # Images: 200x200x3, data x: mxdim(image), data y: mx#output_classes

        x = np.zeros((m, pixels[0][0].shape[0],pixels[0][0].shape[1], pixels[0][0].shape[2]))
        y = np.zeros((m, num_of_dirs))
        c1, c2 = 0, 0

        for i in range(num_of_dirs):
            for j in range(len(pixels[i])):
                x[c1] = pixels[i][j]
                y[c1][c2] = 1
                c1 = c1 + 1
            c2 = c2 + 1

        print("Number of all examples: " + str(x.shape[0]) + " images")
        print("Dimension of dataset x: " + str(x.shape))
        print("Dimension of dataset y: " + str(y.shape))
        print()
        print("x, y data type is: "+ str(type(x[0][0][0][0]))+ ", " + str(type(y[0][0])) )
        print()

        print("y values for pictures from the first subdirectory:\n " + str(y[len(pixels[0])-1]))
        print("y values for pictures from the second subdirectory:\n " + str(y[len(pixels[0])]))
        print("y values for pictures from the third subdirectory:\n " + str(y[len(pixels[0])+len(pixels[1])]))
        print("...")
        print("y values for pictures from the last subdirectory:\n " + str(y[m-1]))

        # Free memory immediately (I)
        pixels = None
```

```
Number of all examples: 2341 images
Dimension of dataset x: (2341, 200, 200, 3)
Dimension of dataset y: (2341, 13)

x, y data type is: <class 'numpy.float64'>, <class 'numpy.float64'>

y values for pictures from the first subdirectory:
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
y values for pictures from the second subdirectory:
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
y values for pictures from the third subdirectory:
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
...
y values for pictures from the last subdirectory:
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
```

Слика 7 :

Креирање питру вектора за сет свих података, x и y

За тренинг параметара неуронске мреже из скупа *x* и *y* изабрано је 80 посто података, а за тестирање параметара неуронске мреже изабрано је преосталих 20 посто. Функција *train_test_split* која омогућава раздвајање података на тренинг и тест скуп налази се у пакету *sklearn.model_selection* (слика 4).

Тренинг подаци су *numpy* вектори *x_train* и *y_train.*
Тест подаци су *numpy* вектори *x_test* и *y_test*.
Преглед броја тренинг/тест примера, димензија ових вектора и укупног броја улазних података мреже налазе се на слици 8.

```
In [10]: # Split train and test data
x_train, x_test, y_train, y_test = train_test_split(x,y,test_size=0.2)

# Free memory immediately (II)
x, y = None, None

# Standardize data
x_train = x_train/255.
x_test = x_test/255.
```

```
In [11]: # Inspect data
m_train = x_train.shape[0]
m_test = x_test.shape[0]
num_px = x_train.shape[1]
depth = x_train.shape[3]

print ("Number of training examples: m_train = " + str(m_train))
print ("Number of testing examples: m_test = " + str(m_test))
print ("Height/width of each square image: num_px = " + str(num_px))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) + ", " + str(depth) + ")\n")
print ("x_train shape: " + str(x_train.shape))
print ("y_train shape: " + str(y_train.shape))
print ("x_test shape: " + str(x_test.shape))
print ("y_test shape: " + str(y_test.shape) + "\n")
print ("Number of all examples/images: m = " + str(m))
```

```
Number of training examples: m_train = 1872
Number of testing examples: m_test = 469
Height/width of each square image: num_px = 200
Each image is of size: (200, 200, 3)

x_train shape: (1872, 200, 200, 3)
y_train shape: (1872, 13)
x_test shape: (469, 200, 200, 3)
y_test shape: (469, 13)

Number of all examples/images: m = 2341
```

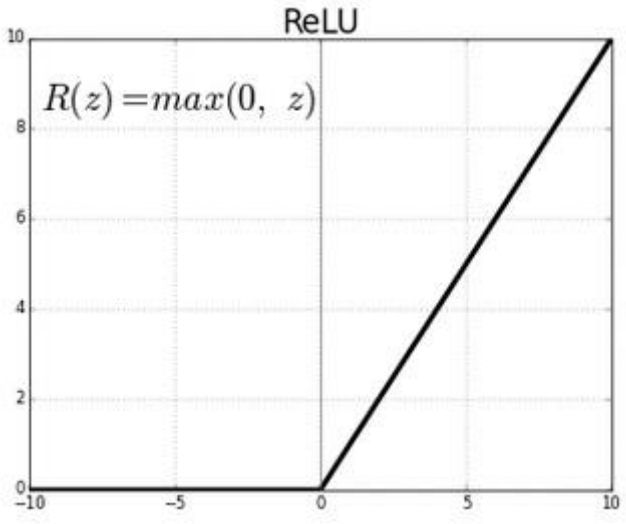
Слика 8 :

Преглед података који ће се користити за тренинг и тест неуронске мреже

4. Архитектура неуронске мреже

Модел неуронске мреже чине два конволуцијска слоја (*Conv -> ReLu -> MaxPooL*), десет потпуно повезана слоја (*Fc*) и један излазни слој.

Активациона функција која је коришћена у скривеним слојевима је ***ReLu*** *(Rectified linear unit)* нелинеарна функција (слика 9 и 10). У излазном слоју активациона функција је ***Softmax*** нелинеарна функција (слика 11 и 12).



Слика 9 :

ReLu активациона функција

```python
def relu(Z):
    """
    Implement the RELU function.
    Arguments:
    Z -- Output of the linear layer, of any shape
    Returns:
    A -- Post-activation parameter, of the same shape as Z
    cache -- a python dictionary containing "A" ;
    stored for computing the backward pass efficiently
    """
    A = np.maximum(0.0,Z)

    assert(A.shape == Z.shape)

    cache = Z
    return A, cache
```

Слика 10 :

Имплементација ReLu активационе функције скривених слојева

$$softmax(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

Слика 11 :

Softmax активациона функција

```python
def stable_softmax(Z):
    """
    Implement the SOFTMAX function.
    Arguments:
    Z -- Output of the linear layer, of any shape
    Returns:
    A -- Post-activation parameter, of the same shape as Z
    cache -- a python dictionary containing "A" ;
    stored for computing the backward pass efficiently
    """
    stabilize = Z - np.max(Z)  # shifts all of elements in the vector to: negative to zero
    ex = np.exp(stabilize) # negatives with large exponents saturate to zero rather than the infinity
    suma = np.sum(ex)
    result = np.divide(ex, suma) # avoiding: overflowing and resulting in nan
    catch = Z
    assert(result.shape == Z.shape)
    return result, catch
```

Слика 12 :

Имплементација Softmax активационе функције излазног слоја

Зашто су избор управо ове активационе функције слојева?

4. 1. Активационе функције и функција предвиђања

У скривеним слојевима ***ReLu*** функција има бољу перформансу конвергенције у односу на сигмоидалну функцију скривених слојева, $sigmoid(\mathrm{Y}) = \frac{1}{1+e^{-Y}}$.
Такође временски је ефективнија јер само одређује максимум вредност између улазне вредности *X* и нуле. Главна предност функције ***ReLu*** је мања вероватноћа да дође до нестајања градијената (нула вредности градијената) или експлодирања градијената (јако велике вредности градијената) у једном од слојева неуронске мреже.

Линеарна компутација пре нелинеарне активације у једном слоју израчунава се према формули: $Z = W * X + b$
„ Међутим уколико је активациона функција линеарна $g(Z) = Z$, а вектор *b* изједначен са нула вектором онда је предвиђање на излазу неуронске мреже
$y\hat{} = g(Z^{[\mathrm{L}]})$ једнако следећем изразу:

$y\hat{} = W^{[L]} * W^{[L-1]} * W^{[L-2]} * \ldots * W^{[2]} * W^{[1]} * X$
где је L нумеричка вредност последњег слоја и
$W^{[1]} * X$ је $Z^{[1]}$ или активациона вредност $g(Z^{[1]})$
$W^{[2]} * W^{[1]} * X$ је $Z^{[2]}$ или активациона вредност $g(Z^{[2]})$, итд.

Уколико је сваки вектор $W^{[l]}$, за l из опсега $[1, L]$, изједначена са вектором

$A =$ | *0.5* | *0* |
|---|---|
| *0* | *0.5* |

важи следеће :

$$y\hat{} = W^{[L]} * A^{[L-1]} * X \qquad (2)$$ ”

Израз доказује чињеницу да ће предвиђање $y\hat{}$ имати степен опадања $0.5^{[L]}$.
Уколико су параметри $W < I$ (јединична матрица) за дубоке неуронске мреже вредност активационе функције ће се експоненцијално смањивати, односно расти за 0.5 → 1.5 ($1.5^{[L]}$ степен раста и $W > I$). У првом случају вредности активационих функција ће тежити нестајању тј. нули, а у другом случају експлодирању тј. бесконачно. Самим тим и њихови градијенти ће бити мањи, односно већи, како учење напредује дубље унутар неуронске мреже. Како вредности ових параметара зависе од дубине мреже на иницијализацију вектора W у слоју l треба да утиче број неурона у слоју $l - 1$. Стога уколико је варијанса параметара W грубо око нуле, дефинисана изразом $\frac{2}{n^{[l-1]}}$, где је $n^{[l-1]}$ број неурона у слоју $l - 1$, а варијанса вредности ***ReLu*** активационе функције подразумевано 1, онда ће и $y\hat{}$ предвиђање бити у том опсегу. На овај начин могуће је редуковање проблема нестајања/експлодирања градијента, али не и искорење овог проблема. У поглављу *5.1* имплементирана је иницијализација параметара за потпуно повезану мрежу на начин који је представљен овде.

У логистичком регресијском моделу ***Softmax*** функција се користи за мулти-класификацију вредности излазног слоја неуронске мреже. Збир вероватноћа вредности на које је примењен ***Softmax*** је један. Максимална вредност у колони излазног вектора облика (m, 13), која припада опсегу (0, 1), за једну инстанцу на улазу представља предвиђену биљну болест за ту инстанцу и биће претворена у вредност 1. Упоређивањем вредности које је неуронска мрежа предвидела за тренинг/тест скуп података y^ и реалних вредности излазног скупа y одређује се прецизност неуронске мреже на овим скуповима. Предвиђање у неуронској мрежи имплементира се као један пролазак кроз све слојеве мреже (слика 13), док је тренинг заснован на узастопним проласцима у циљу оптимизовања параметара слојева неуронске мреже. Тренинг параметара прати и назадан пролазак (пропагација уназад) о коме ће речи бити касније.

```
def predict(X, y, parameters, layers_dims_conv, channels, nc):
    """
    X -- train/test data of shape (#examples x dim(image))
    y -- train/test data of shape (#examples x 1)
    parameters -- parameters of the trained model
    Returns:
    p -- predictions for the given dataset X of shape (#examples x 1)
    """
    m = X.shape[0]
    hparameters1 = {"pad" : 2, "stride": 3}
    hparameters2 = {"stride" : 2, "f": 2}
    hparameters3 = {"pad" : 1, "stride": 3}
    hparameters4 = {"stride" : 2, "f": 2}

    parameters_conv = solve_conv(parameters, layers_dims_conv, channels, nc)
    # Forward propagation conv
    P2, cache_pool2, cache_pool1, cache_conv1, cache_conv2, c1, c2 = Convolutional_Forward(X, parameters_conv, hparameters1,
hparameters2, hparameters3, hparameters4)

    # Forward propagation FC
    FC1 = P2.reshape(P2.shape[0], -1).T

    # Get parameters for FC network, parameters_fc
    parameters_fc = solve_fc(parameters, layers_dims_conv)
    # Forward propagation FC
    A3_fc, cache_fc = L_model_forward(FC1, parameters_fc)

    a, b = A3_fc.shape
    p = np.zeros((a, b))
    # convert probas to 0/1 predictions
    tmp = np.argmax(A3_fc, axis=0)
    for i in range(0, b):
        p[tmp[i],i] = 1
    p = p.T
    print("Accuracy: "  + str(np.mean((p[:,:] == y[:,:]))))
    return p
```

Слика 13 :
Предвиђање на тренинг и тест скупу података и одређивање тачности мреже на ова два скупа

4. 2. Параметри неуронске мреже

Слојеви неуронске мреже садрже велики број параметара који учествују у креирању комплексне границе одлучивања. Граница одлучивања класификује улазне податке према излазима система тј. стањима биљке малине. Параметри неуронске мреже који припадају слоју l преузимају информације од слоја $l-1$ и учествују у линераној компутацији и нелинеарној активацији према чему граница одлучивања добија комплекснији изглед.

Број слојева у конволуцијском делу неуронске мреже је два према чему постоје два филтера при операцији конволуције. Филтери у конволуцијском делу неуронске мреже представљају *numpy n*-димензионе векторе. Овакви вектори постоје и у потпуно повезаном делу неуронске мреже. Димензије филтера при операцији конволуције су редом (5, 5, 3, 4) и (3, 3, 4, 8). Прве три димензије се односе на ширину, висину и дубину и филтера, а трећа димензија на то колико је оваквих филтера употребљено у конволуцијској операцији.
Број слојева у потпуно повезаном делу мреже као и број параметара који припадају овим слојевима (без излазног слоја) је десет. Број неурона у скривеним слојевима потпуно повезане мреже је редом 447, 774, 1094, 631, 315, 181, 127, 73, 36, 20. Њихове димензије зависе од броја неурона у суседним слојевима. Због овакве конфигурације мреже параметар W који припада првом скривеном слоју је облика (447, 200), други је облика (774, 447), трећи је облика (1094, 774), … , и последњи је облика (20, 36). Излаз конволуцијске мреже је излаз последњег *pooling* слоја димензије (m, 5, 5, 8). Овај 4-димензиони вектор обликован је у вектор димензије (200, m) и он представља улазни вектор у потпуно повезану мрежу. На овај начин је установљен број улазних веза које припадају првом параметру W потпуно повезане мреже (→200). Унутар конволуцијске мреже параметри су 4-димензиони, док у потпуно повезаној мрежи 2-димензиони, због чега је било потребно обликовање ових вектора при преласку са конволуцијске на потпуно повезану мрежу.

Параметар последњег излазног слоја је облика (13,20) и једино његове вредности учествују у ***Softmax*** активационој функцији.

5. Модел неуронске мреже

Генерална методологија мреже :

1. Иницијализација параметара / Дефинисање параметара
2. Иницијализација оптимизатора Адам алгоритма
3. Петља кроз број епоха
    3. 1. Избор мини-серија улазних и излазних података
    3. 2. Петља за сваку мини-серију података
    3. 3. Пропагација унапред
        3. 3. 1. Конволуцијска мрежа
        3. 3. 2. Креирање 2D улаза у потпуно повезану мрежу
        3. 3. 3. Потпуно повезана мрежа
    3. 4. Рачунање трошка параметара
    3. 5. Пропагација уназад
        3. 5. 1. Потпуно повезана мрежа
        3. 5. 2. Конволуцијска мрежа
    3. 6. Update-овање параметара (користећи параметре и градијенте из пропагације уназад)
4. Штампање трошка параметара при свакој епохи
5. Коришћење истренираних параметара за предвиђање

5. 1. Иницијализација параметара / Дефинисање параметара

На слици 14 и 15 приказане су функције иницијализације параметара W за конволуцијски део и потпуно повезани део неуронске мреже.
Слика 14 приказује рандом иницијализацију параметара конволуцијске мреже. Слика 15 приказују *«he»* иницијализацију параметара која се показала најбољом за потпуно повезане, дубоке мреже. Ова иницијализациона техника омогућава иницијализацију параметара тако да вредности параметара у једном слоју зависе од броја неурона претходног слоја. Разлог томе је што величина мреже (број неурона у скривеним слојевима) утиче на експлодирање/нестајање градијената у одређеном слоју тј. када градијенти постану нула или јако велики.

Иницијализациона техника *«he»* је имплементирана као рандом техника са додатним чланом који износи $\sqrt{\frac{2}{layer\_dims[l-1]}}$, а који представља варијансу вредности параметара W (поглавље 4.1)

```python
def initialize_parameters_rand_CONV(layers_dims, channels, nc):
    """
    Arguments:
    layer_dims -- python array (list) containing the size of each layer.
    channels -- python array (list) containing the number of channels of each filter.
    nc -- python array (list) containing the number of filters in each convolution.

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ..., "WL", "bL":
                  Wl+1 -- weight matrix of shape (layers_dims[l], layers_dims[l], channels[l], nc[l])
                  bl+1 -- bias vector of zeros of shape (1, 1, 1, nc[l])
    """

    np.random.seed(3)
    parameters = {}
    L = len(layers_dims) # integer representing the number of layers

    for l in range(L):
        parameters['W' + str(l+1)] = np.random.randn(layers_dims[l], layers_dims[l], channels[l], nc[l])
        parameters['b' + str(l+1)] = np.zeros((1, 1, 1, nc[l]))

    return parameters
```

Слика 14 :

Random иницијализација параметара конволуцијске мреже

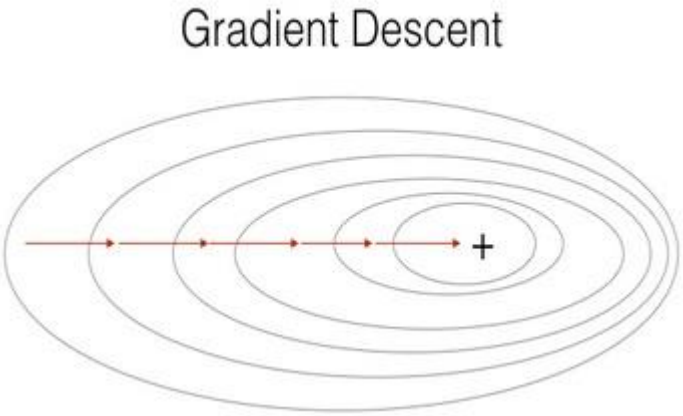
```python
def initialize_parameters_he_FC(layer_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of each layer in our network

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ..., "WL", "bL":
                  Wl -- weight matrix of shape (layer_dims[l], layer_dims[l-1])
                  bl -- bias vector of zeros of shape (layer_dims[l], 1)

    Tips:
    - For example: the layer_dims for the "Planar Data classification model" would have been [2,2,1].
    This means W1's shape was (2,2), b1 was (1,2), W2 was (2,1) and b2 was (1,1). Now you have to generalize it!
    - In the for loop, use parameters['W' + str(l)] to access Wl, where l is the iterative integer.
    """

    np.random.seed(3)
    parameters = {}
    L = len(layer_dims) # number of layers in the network

    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1]) * np.sqrt(2 / layer_dims[l-1])
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

        assert(parameters['W' + str(l)].shape == layer_dims[l], layer_dims[l-1])
        assert(parameters['W' + str(l)].shape == layer_dims[l], 1)

    return parameters
```

Слика 15 :

«he» иницијализација параметара потпуно повезане мреже

5. 2. Иницијализација оптимизатора Адам алгоритма

Оптимизатори су компоненте алгоритма који налази локални минимум функције трошка параметара (алгоритам „Градијентног спуста”, слика 16). Оптимизатори служе да убрзају алгоритам односно учење параметара неуронске мреже. „Адам” оптимизациони алгоритам је настао спајањем „Момент”алгоритма (*Momentum*, енгл.) и алгоритма „Квадратни корен средњег квадрата” (*Root Mean Square prop*, енгл.) који редукују осцилације функције трошка параметара. Осцилације трошка параметара се јављају приликом процесуирања серија података (у односу на целе скупове података) у току једне епохе (поглавље 5.3).



Слика 16 :
Алгоритам Градијентног спуста (Gradient Descent, енгл.)

Момент алгоритам

Овај алгоритам је настао спајањем алгоритма „Мини серије Градијентног спуста” (*Mini-Batch Gradient Descent*, енгл., поглавље 5.3) и алгоритма „Експоненцијално отежано усредњавање” (*Exponentially weighted averages*, енгл.). Идеја је редуковати осцилације трошка параметара које алгоритам „Мини серије Градијентног спуста” прави у циљу достизања оптималне вредности функције трошка параметара. На нову вредност функције трошка параметара утиче оптимизатор v при чему су вредности разлика хоризонталних корака близу нули. Разлике по вертикалној оси остају непромењене.

„Интуиција: параметар v се сматра као «брзина» лопте која се котрља низбрдо ка оптималном трошку и развија брзину (и нагиб - момент) према смеру градијента тј. «нагиба брда» (2).”

Алгоритам Квадратни корен средњег квадрата

Алгоритам који такође убрзава градијентни спуст, јер успорава учење у вертикалном смеру, а убрзава у хоризонталном смеру, употребљава се због недостатка „Момент" алгоритма који не редукује разлике у вертикалним корацима. Када ова два алгоритма раде заједно (→Адам) учење је брже за било коју структуру неуронске мреже.

Адам алгоритам

„Како ради?

1. Рачуна експоненцијално отежане просеке претходних градијената и складишти их у променљиву v (складиштење пре корекције која користи одступање b), и израчунава $vcorrected$ (са корекцијом која користи одступање b)
2. Рачуна експоненцијално отежане просеке квадрата претходних градијената и складишти их у променљиву s (складиштење пре корекције са одступањем b), и израчунава $scorrected$ (са корекцијом која користи одступање b)
3. Update-ује параметре у правцу заснованом на комбиновању информација из 1. и 2

(2)."

Иницијализација Адам оптимизатора v и s је нула-иницијализација *numpy* вектора који припадају Python-овом речнику оптимизатора. Вредности оптимизатора приказане су кроз пример (слика 17), а имплементација иницијализације ових речника приказана је на слици 18.

```
parameters = initialize_adam_test_case()

v, s = initialize_adam(parameters)
print("v[\"dW1\"] = " + str(v["dW1"]))
print("v[\"db1\"] = " + str(v["db1"]))
print("v[\"dW2\"] = " + str(v["dW2"]))
print("v[\"db2\"] = " + str(v["db2"]))
print("s[\"dW1\"] = " + str(s["dW1"]))
print("s[\"db1\"] = " + str(s["db1"]))
print("s[\"dW2\"] = " + str(s["dW2"]))
print("s[\"db2\"] = " + str(s["db2"]))
```

```
v["dW1"] = [[0. 0. 0.]
 [0. 0. 0.]]
v["db1"] = [[0.]
 [0.]]
v["dW2"] = [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
v["db2"] = [[0.]
 [0.]
 [0.]]
s["dW1"] = [[0. 0. 0.]
 [0. 0. 0.]]
s["db1"] = [[0.]
 [0.]]
s["dW2"] = [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
s["db2"] = [[0.]
 [0.]
 [0.]]
```

Слика 17 :

Иницијализација оптимизатора v и s

```
def initialize_adam(parameters) :
    """
    Initializes v and s as two python dictionaries with:
                - keys: "dW1", "db1", ..., "dWL", "dbL"
                - values: numpy arrays of zeros of the same shape as the corresponding gradients/parameters.

    Arguments:
    parameters -- python dictionary containing your parameters.
                    parameters["W" + str(l)] = Wl
                    parameters["b" + str(l)] = bl

    Returns:
    v -- python dictionary that will contain the exponentially weighted average of the gradient.
                    v["dW" + str(l)] = ...
                    v["db" + str(l)] = ...
    s -- python dictionary that will contain the exponentially weighted average of the squared gradient.
                    s["dW" + str(l)] = ...
                    s["db" + str(l)] = ...

    """
    L = len(parameters) // 2 # number of layers in the neural networks
    v = {}
    s = {}

    # Initialize v, s. Input: "parameters". Outputs: "v, s".
    for l in range(L):
        v["dW" + str(l+1)] = np.zeros((parameters["W" + str(l+1)].shape[0],parameters["W" + str(l+1)].shape[1]))
        v["db" + str(l+1)] = np.zeros((parameters["b" + str(l+1)].shape[0],parameters["b" + str(l+1)].shape[1]))
        s["dW" + str(l+1)] = np.zeros((parameters["W" + str(l+1)].shape[0],parameters["W" + str(l+1)].shape[1]))
        s["db" + str(l+1)] = np.zeros((parameters["b" + str(l+1)].shape[0],parameters["b" + str(l+1)].shape[1]))

    return v, s
```

Слика 18 :
Иницијализација Адам оптимизатора

5. 3. Петља кроз број епоха

Епоха је време које је потребно да неуронска мрежа процесуира цео сет улазних података било да су ону процесуирани одједном (*«hole batch»*) или у серијама (*«mini batch»*). Пролазак кроз епохе омогућен је неизбежном *foor* петљом.

5. 3. 1. Избор мини-серија улазних и излазних података

Мини-серија улазних и излазних података односи се на серије парова улазних и излазних тренинг података *minibatch_X u minibatch_Y* (слика 20).

Мини-серије Градијентног спуста алгоритам:

Разлика алгоритма Мини-серија Градијентног спуста у односу на стандардни алгоритам Градијентног спуста је што алгоритам Градијентног спуста даје резултат вредности трошка параметара (*cost,* слика 20) тек након што процесуира цео сет података.

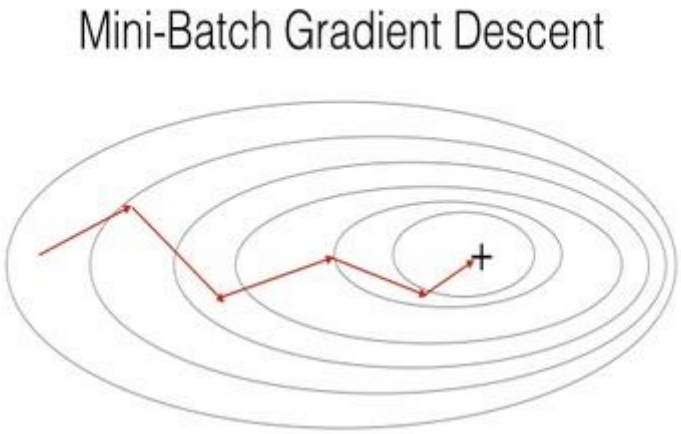
Уколико је *m* укупан број тренинг примера, односно цео сет тренинг података, а *num_iter* број итерација, односно број епоха, овај алгоритам ће након једне итерације процесуирати цео сет тренинг података од *m* примера и направити мали корак у оптимизацији трошка (трошак ће бити оптимизован само једном).

Да би оптимизовао укупан трошак овај алгоритам ће процесуирати цео сет тренинг података (*«hole batch»*) онолико пута колики је дефинисани, максимални број итерација.

5. 3. 2. Петља за сваку мини-серију података

Уколико је *k+1* једнако укупном броју тренинг података подељеним са величином серије података, да би оптимизовао укупан трошак овај алгоритам ће процесуирати серије тренинг података *k+1* пута и понављаће ово онолико пута колики је дефинисани, максимални број итерација. Алгоритам садржи додатну *foor* петљу за процесуирање свих мини-серија тренинг података (слика 20).

Функција трошка алгоритма Градијентног спуста у времену стално опада. Функција трошка Мини-серија Градијентног спуста у времену није константно опадајућа функција, а разлог томе је што приликом рачунања функције трошка сваки пут алгоритам ради са другачијим сетом података. Трошак се update-ује након процесуирања сваке мини-серије, а не након процесуирања целог сета тренинг података. Из овог разлога функција трошка параметара садржи шум (слика 19).



Слика 19:
Алгоритам Мини-серије Градијентног спуста (Mini-batch Gradient Descent, енгл.)

За поделу скупа тренинг података, *X* и *Y* на мини-серије података креирана је функција *random_mini_batches.*

Вектори *X* и *Y* су вектори димензија редом (m, Hi, Wi, Ci) и (m, n_y), где је *m* укупан број тренинг инстанци, *(Hi, Wi, Ci)* - димензија сваке инстанце и *n_y* број излазних вредности за сваку од инстанци (слика 20).

```
def random_mini_batches(X, Y, mini_batch_size = 64, seed = 0):
    """
    Creates a list of random minibatches from (X, Y)

    Arguments:
    X -- input data, of shape (input size, number of examples) (m, Hi, Wi, Ci)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1, number
of examples) (m, n_y)
    mini_batch_size - size of the mini-batches, integer
    seed -- this is only for the purpose of grading, so that you're "random minibatches
are the same as ours.

    Returns:
    mini_batches -- list of synchronous (mini_batch_X, mini_batch_Y)
    """

    m = X.shape[0]                  # number of training examples
    mini_batches = []
    np.random.seed(seed)

    # Step 1: Shuffle (X, Y)
    permutation = list(np.random.permutation(m))
    shuffled_X = X[permutation,:,:,:]
    shuffled_Y = Y[permutation,:]

    # Step 2: Partition (shuffled_X, shuffled_Y). Minus the end case.
    num_complete_minibatches = math.floor(m/mini_batch_size) # number of mini batches
of size mini_batch_size in your partitionning
    for k in range(0, num_complete_minibatches):
        mini_batch_X = shuffled_X[k * mini_batch_size : k * mini_batch_size +
mini_batch_size,:,:,:]
        mini_batch_Y = shuffled_Y[k * mini_batch_size : k * mini_batch_size +
mini_batch_size,:]
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    # Handling the end case (last mini-batch < mini_batch_size)
    if m % mini_batch_size != 0:
        mini_batch_X = shuffled_X[num_complete_minibatches * mini_batch_size : m,:,:,:]
        mini_batch_Y = shuffled_Y[num_complete_minibatches * mini_batch_size : m,:]
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    return mini_batches
```

Слика 20 :

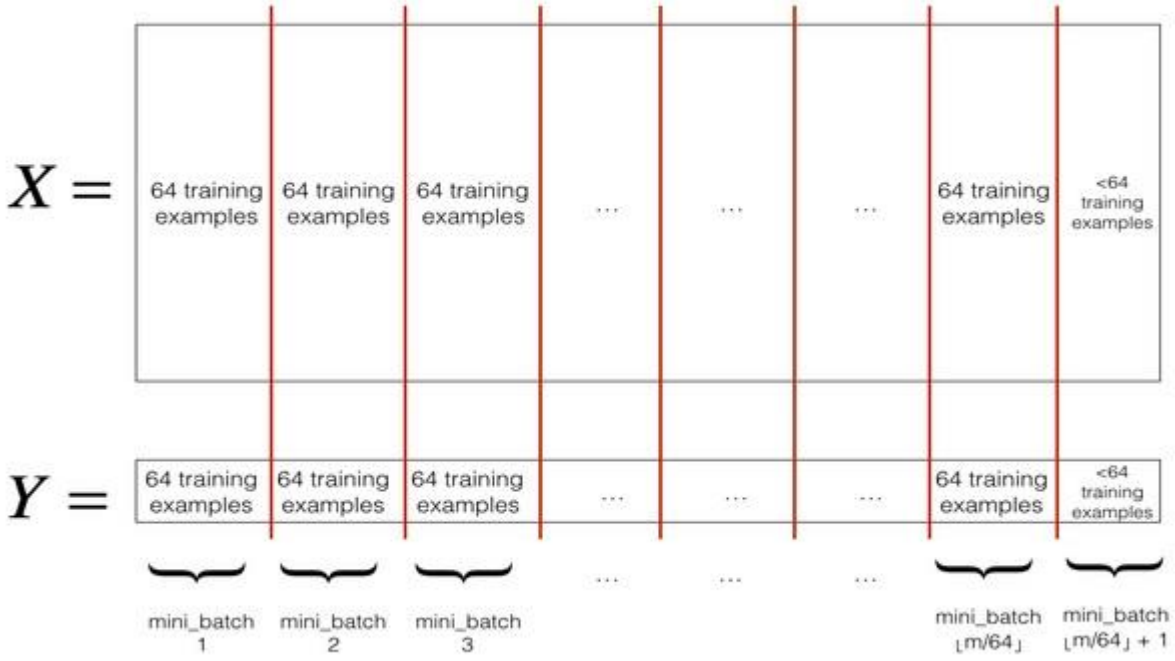
Имплементација функције која креира random серије тренинг података

Како би се на рандом начин из датог скупа података изабрале мини-серије тренинг података ова функција креира листу *permutation*. Листа садржи пермутоване, целе бројеве из опсега (0, m). Пермутација је обезбеђена *numpy* функцијом *np.random.permutation* чија је примена приказана на слици 21. Векторизација у Python-у је омогућила избегавање петљи и пермутацију података у само једном реду.

```
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.permutation(arr)
array([[6, 7, 8],
       [0, 1, 2],
       [3, 4, 5]])
```

Слика 21 :
Приказ функције np.random.permutation

Пермутоване вредности се односе на њима одговарајући тренинг пример. Структуре које дефинишу пермутоване податке су *shuffled*_X и *shuffled*_Y (слика 20).
У односу на изабрану величину мини-серију података (колико тренинг примера садржи једна серија) креира се број који описује колико таквих серија постоји. Уколико се испостави да укупна количина тренинг података *m* није дељива са величином једне мини-серије *mini_batch_size* онда ће све мини-серије (сем последње) имати исти број парова података, а последња серија онолико парова података колико је потребно да би се допунио број до укупног броја тренинг података (слика 22).



Слика 22
Серије података тренинг примера

5. 3. 3. Пропагација унапред

Како функционише пропагација унапред заправо тако функционише и неуронска мрежа када треба да креира предвиђање за одређени улазни податак.

Активационе функције које се користе у слојевима неуронске мреже у области неуронских мрежа не треба да буду линеарне.

“Композиција сваке две линеарне функције је линеарна функција”

Повећање дубине неуронске мреже тј. броја скривених слојева једна је од метода за увећање комплексности границе одлучивања. Уколико су као активационе функције изабране линеарне функције неуронска мрежа ће израчунати линеарну активациону функцију улаза и понашаће се као да нема скривене слојеве.

За границу одлучивања ово значи да се неће добро прилагодити подацима јер ће имати линеаран облик. Да би се добро прилагодила улазним подацима граница одлучивања може бити јако комплексна.

Ово је још један разлог због чега су за активационе функције изабране нелинеарне функције ***ReLu*** и ***Softmax***. Ове функције се врло добро могу прилагодити било ком сету података.

5. 3. 3. 1. Конволуцијска мрежа

Пропагација унапред кроз конволуцијску мрежу односи се на математику операцију конволуције између улазног вектора у конволуцијски слој и филтера тог слоја (слика 24). Резултат конволуције неуронске мреже је вектор чији је параметар који се односи на висину и ширину појединачне слике исти као параметар висине и ширине слике која припада улазном вектору. Трећи параметар резултујућег вектора односи се на дубину ове запремине и једнак је броју филтера који су употребљени у конволуцијској операцији. Такође број канала који дефинише дубину филтера se мора слагати са бројем канала улазног вектора.

Метод нула-подметања (*zero-padding*)

Нула-подметање слике је техника која обезбеђује да се при конволуцији сви пиксели слике користе равномеран број пута. Неравномеран број коришћења пиксела је последица тога што се за конволуцију вредности пиксела једне слике користи вишеструка конволуција.

Једна конволуција над целом сликом могућа је уколико би филтер био истих димензија као цела слика. Тада би филтер имао параметар за сваки пиксел слике. На тај начин сваки пиксел слике у операцији конволуције би се користио само једном са одговарајућим параметром филтера.

Вишеструка конволуција односи се на ситуацију када је улазни сет пиксела целе слике подељен на делове чије су димензије (обавезно) једнаке димензији филтера. Скупови оваквих пиксела појединачно учествују у конволуцији са истим филтером у једном слоју неуронске мреже. Разлог неравномерног коришћења пиксела у конволуцији је последица тога што пиксели који припадају једном скупу могу припадати и другом скупу пиксела (исти пиксели учествују у више конволуција). Такође они пиксели који се налазе угловима слике могу се наћи само у једном скупу пиксела (пиксели по ивицама слике учествују у само једној конволуцији). Ова појава се односи на начин како су пиксели раздвајани у посебне скупове (слика 25).

Метод *нула-подметања* омогућава додавање оквира нула-пиксела (по ивицама слике). На тај начин омогућено је равномерно коришћење пиксела слика у конволуцијским операцијама. У односу на величину параметра ове методе *pad* зависи са колико оквира нула је слика проширена (слика 23). Све слике су проширене тако да након конволуције параметар висине и ширине резултујуће слике остане исти као одговарајући параметар слике пре конволуције (*«same* техника*»*).

```
def zero_pad(X, pad):
    """
    Pad with zeros all images of the dataset X. The padding is applied to the height
and width of an image,
    as illustrated in Figure 1.

    Argument:
    X -- python numpy array of shape (m, n_H, n_W, n_C) representing a batch of m
images
    pad -- integer, amount of padding around each image on vertical and horizontal
dimensions

    Returns:
    X_pad -- padded image of shape (m, n_H + 2*pad, n_W + 2*pad, n_C)
    """

    X_pad = np.pad(X,((0,0),(pad,pad),(pad,pad),(0,0)), 'constant')
    return X_pad
```

Слика 23:
Метод Нула-подметања (zero-padding)

Један корак конволуцијске операције приказан је на слици 24 и представља множење вредности пиксела једног дела слике и вредности елемената одабраног филтера. Множење матрица се односи на "*element-wise*" матрично множење. Резултат једне конволуције је сума свих производа тј скаларна вредност.

```
def conv_single_step(a_slice_prev, W, b):
    """
    Apply one filter defined by parameters W on a single slice (a_slice_prev) of the
output activation
    of the previous layer.

    Arguments:
    a_slice_prev -- slice of input data of shape (f, f, n_C_prev)
    W -- Weight parameters contained in a window - matrix of shape (f, f, n_C_prev)
    b -- Bias parameters contained in a window - matrix of shape (1, 1, 1)

    Returns:
    Z -- a scalar value, result of convolving the sliding window (W, b) on a slice x of
the input data
    """
    # Element-wise product between a_slice and W. Do not add the bias yet.
    s = np.multiply(a_slice_prev,W)
    # Sum over all entries of the volume s.
    Z = np.sum(s)
    # Add bias b to Z. Cast b to a float() so that Z results in a scalar value.
    Z = float(Z + b)

    return Z
```

Слика 24 :

Конволуцијска операција дела слике и филтера из одговарајућег конволуцијској слоја

Функција која омогућава пропагацију унапред кроз један слој конволуцијске мреже прати следеће кораке:

1. Одређивање висине/ширине дела излазног вектора
2. Нула-подметање вектора
3. Петља за сваку мини-серију тренинг података
4. Селекција једног тренинг податка
5. Петља кроз број пиксела по вертикали и хоризонтали и број канала резултујућег вектора након конволуције
6. Израчунавање вредности позиције за сва четири краја тренутног дела слике
7. Креирање 3D дела слике на основу позиција сва четири краја дела слике
8. Конволуција 3D дела слике и одговарајућег филтера

Колико ће пиксела бити прескочено (на десно или доле) у креирању следећег скупа пиксела зависи од параметра ове методе *stride*. Ово је 2. параметар разматране методе.

Пропагација унапред кроз конволуцијску мрежу приказана је на слици 25.

```
def conv_forward(A_prev, W, b, hparameters):
    """
    Implements the forward propagation for a convolution function

    Arguments:
    A_prev -- output activations of the previous layer, numpy array of shape (m, n_H_prev, n_W_prev, n_C_prev)
    W -- Weights, numpy array of shape (f, f, n_C_prev, n_C)
    b -- Biases, numpy array of shape (1, 1, 1, n_C)
    hparameters -- python dictionary containing "stride" and "pad"

    Returns:
    Z -- conv output, numpy array of shape (m, n_H, n_W, n_C)
    cache -- cache of values needed for the conv_backward() function
    """
    # Retrieve dimensions from A_prev's shape (≈1 line)
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # Retrieve dimensions from W's shape (≈1 line)
    (f, f, n_C_prev, n_C) = W.shape

    # Retrieve information from "hparameters" (≈2 lines)
    stride = hparameters['stride']
    pad = hparameters['pad']

    # Compute the dimensions of the CONV output volume using the formula given above. Hint: use int() to floor. (≈2 lines)
    n_H = int((n_H_prev - f + 2 * pad) / stride) + 1
    n_W = int((n_W_prev - f + 2 * pad) / stride) + 1

    # Initialize the output volume Z with zeros. (≈1 line)
    Z = np.zeros((m, n_H, n_W, n_C))

    # Create A_prev_pad by padding A_prev
    A_prev_pad = zero_pad(A_prev, pad)

    for i in range(m):                                # loop over the batch of training examples
        a_prev_pad = A_prev_pad[i]                    # Select ith training example's padded activation
        for h in range(n_H):                          # loop over vertical axis of the output volume
            for w in range(n_W):                      # loop over horizontal axis of the output volume
                for c in range(n_C):                  # loop over channels (= #filters) of the output volume
                    # Find the corners of the current "slice" (≈4 lines)
                    vert_start = h * stride
                    vert_end = vert_start + f
                    horiz_start = w * stride
                    horiz_end = horiz_start + f
                    # Use the corners to define the (3D) slice of a_prev_pad (See Hint above the cell). (≈1 line)
                    a_slice_prev = a_prev_pad[vert_start:vert_end, horiz_start:horiz_end, :]
                    # Convolve the (3D) slice with the correct filter W and bias b, to get back one output neuron. (≈1
line)
                    Z[i, h, w, c] = conv_single_step(a_slice_prev, W[...,c], b[...,c])

    # Making sure your output shape is correct
    assert(Z.shape == (m, n_H, n_W, n_C))

    # Save information in "cache" for the backprop
    cache = (A_prev, W, b, hparameters)

    return Z, cache
```

Слика 25 :

Пропагација унапред кроз конволуцијски слој

Pooling техника

Ова техника се користи када је потребно смањење димензија вектора унутар неуронске мреже. Заправо вектор над којим се врши *pooling* техника је резултујући вектор конволуције.

Слојем неуронске мреже се сматра онај слој који у себи садржи параметре чије вредности могу да се тренирају до оптималних вредности. *Pooling техника* не пружа ову могућност тренирања. Она не представља посебан слој, већ се операција придружује претходном конволуцијском слоју. Стога се сматра да једном слоју конволуцијске мреже припада операција конволуције и операција коју описује *pooling техника*.

Оба конволуцијска слоја неуронске мреже користе *max-pooling технику* са параметрима *f* = *#филтера* = 2 и *s* = *stride* = 2 (постоји и *average-pooling*).
Овакви параметри осигуравају да ће прве две димензије улазног вектора, тј. висина и ширина, бити дупло мање након операције *max-pooling-a*. Такође дубина улазног вектора након *max-pooling-a* остаје иста за *stride* = 2. Пропагација унапред за *pooling технику* имплементирана је на сличан начин као и пропагација при конволуцији.
Операција која се користи у техници *max-pooling*-а је селекција максималног елемента у скупу пиксела једног дела слике (слика 26).

```
def pool_forward(A_prev, hparameters, mode = "max"):
    """
    Implements the forward pass of the pooling layer

    Arguments:
    A_prev -- Input data, numpy array of shape (m, n_H_prev, n_W_prev, n_C_prev)
    hparameters -- python dictionary containing "f" and "stride"
    mode -- the pooling mode you would like to use, defined as a string ("max" or "average")

    Returns:
    A -- output of the pool layer, a numpy array of shape (m, n_H, n_W, n_C)
    cache -- cache used in the backward pass of the pooling layer, contains the input and hparameters
    """
    # Retrieve dimensions from the input shape
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # Retrieve hyperparameters from "hparameters"
    f = hparameters["f"]
    stride = hparameters["stride"]

    # Define the dimensions of the output
    n_H = int(1 + (n_H_prev - f) / stride)
    n_W = int(1 + (n_W_prev - f) / stride)
    n_C = n_C_prev

    # Initialize output matrix A
    A = np.zeros((m, n_H, n_W, n_C))

    for i in range(m):                         # loop over the training examples
        for h in range(n_H):                     # loop on the vertical axis of the output volume
            for w in range(n_W):                 # loop on the horizontal axis of the output volume
                for c in range (n_C):            # loop over the channels of the output volume

                    # Find the corners of the current "slice" (≈4 lines)
                    vert_start = h * stride
                    vert_end = vert_start + f
                    horiz_start = w * stride
                    horiz_end = horiz_start + f

                    # Use the corners to define the current slice on the ith training example of A_prev, channel c. (≈1
line)
                    a_prev_slice =  A_prev[i,vert_start:vert_end, horiz_start:horiz_end, c]

                    # Compute the pooling operation on the slice. Use an if statment to differentiate the modes. Use
np.max/np.mean.
                    if mode == "max":
                        A[i, h, w, c] = np.max(a_prev_slice)
                    elif mode == "average":
                        A[i, h, w, c] = np.mean(a_prev_slice)

    # Store the input and hparameters in "cache" for pool_backward()
    cache = (A_prev, hparameters)

    # Making sure your output shape is correct
    assert(A.shape == (m, n_H, n_W, n_C))

    return A, cache
```

Слика 26 :
Пропагација унапред и pooling техника

```
def Convolutional_Forward(minibatch_X, parameters_conv, hparameters1, hparameters2, hparameters3, hparameters4):
    Z1, cache_conv1 = conv_forward(minibatch_X, parameters_conv["W1"], parameters_conv["b1"], hparameters1 )
    A1, c1 = relu(Z1)
    P1, cache_pool1 = pool_forward(A1, hparameters2)

    Z2, cache_conv2 = conv_forward(P1, parameters_conv["W2"], parameters_conv["b2"], hparameters3 )
    A2, c2 = relu(Z2)
    P2, cache_pool2 = pool_forward(A2, hparameters4)
    return P2, cache_pool2, cache_pool1, cache_conv1, cache_conv2, c1, c2
```

Слика 27 :
Функција која омогућава пропагацију унапред кроз два конволуцијска слоја (и pooling техника)

5. 3. 3. 2. Креирање 2D улаза у потпуно повезану мрежу

Излаз последњег конволуцијској слоја *P2* је излазни вектор након последње примене *pooling технике*. Овај вектор је димензије (m, 5, 5, 8), где је *m* величина једне мини-серије података. Параметри потпуно повезане мреже су 2-димензиони вектори, стога је овај 4-димензиони вектор обликован у вектор димензије (200, m) :
FC1 = P2.reshape(P2.shape[0], -1).T (слика 39)

5. 3. 3. 3. Потпуно повезана мрежа

У једном неурону слоја потпуно повезане неуронске мреже се најпре извршава линеарна компутација *z = w*a + b* као на слици 30. Линеарном комутацијом обезбеђује се зависност резултата мреже од параметара потпуно повезане мреже и улазног скупа података. Различите вредности параметара *w* и *b* резултирају другачијом границом одлучивања. Најоптималније вредности параметара *w* и *b* креирају границу одлучивања која се најбоље прилагођава улазним подацима и управо ове параметре алгоритам Градијентног спуста покушава да оптимизује.

За линеарну једначину чији резултат зависи од параметара *w* и *b* може се искористити интуиција описана једначином *l = h1*m + h2* или
„ Дужина растегљивог канапа *l(→z)* о који је обешена лоптица масе *m(→a)*, за *m* = 0 увек је нека константна вредност *h2*(→*b*). Дужина растегљивог канапа *l(→z)* зависи од масе *m(→a)* и градијента *h1(→w),* што значи да за свако *m(→a)* дужина канапа *l(→z)* се повећава h1(→*w*) пута. “

Линеарна репрезентација се затим шаље активационим функцијама ***ReLu*** или ***sigmoid.*** Активационим функцијама се обезбеђује нелинеарност модела неуронске мреже да би се избегла последица да скривени слојеви да немају утицај на креирање границе одлучивања. Пропагација унапред кроз потпуно повезану мрежу имплементирана је помоћу функција са слика 28, 29 и 30.

Имплементација активационе функције ***Softmax***, која је коришћена у пропагацији кроз потпуно повезану мрежу, изведена је на начин да одговара реалним подацима које неуронске мрежа процесуира. Тиме је обезбеђено да подаци не достигну *null* вредности (слика 12) и да не дође до прекорачења опсега вредности које припадају float64 типу података.

```
def L_model_forward(X, parameters):
    """
    Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID computation

    Arguments:
    X -- data, numpy array of shape (input size, number of examples)
    parameters -- output of initialize_parameters_deep()

    Returns:
    AL -- last post-activation value
    caches -- list of caches containing:
                every cache of linear_relu_forward() (there are L-1 of them, indexed from 0 to L-2)
                the cache of linear_sigmoid_forward() (there is one, indexed L-1)
    list_A_prev_prev list of A's and minibatch X from conv network. This is used for Residual network.
    """

    caches = []
    A = X
    L = len(parameters) // 2                  # number of layers in the neural network

    # Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches" list.
    for l in range(1, L):
        A_prev = A
        A, cache = linear_activation_forward(A_prev, parameters['W' + str(l)], parameters['b' + str(l)],
                                             activation = "relu")
        caches.append(cache)

    # Implement LINEAR -> SOFTMAX. Add "cache" to the "caches" list.
    AL, cache = linear_activation_forward(A, parameters['W' + str(L)], parameters['b' + str(L)],
                                          activation = "softmax")
    caches.append(cache)

    assert(AL.shape == (13,X.shape[1]))

    return AL, caches
```

Слика 28 :

Функција која омогућава пролаз кроз свих L слојева потпуно повезане мреже

```
def linear_activation_forward(A_prev, W, b, activation):
    """
    Implement the forward propagation for the LINEAR->ACTIVATION layer

    Arguments:
    A_prev -- activations from previous layer (or input data): (size of previous layer,
number of examples)
    W -- weights matrix: numpy array of shape (size of current layer, size of previous
layer)
    b -- bias vector, numpy array of shape (size of the current layer, 1)
    activation -- the activation to be used in this layer, stored as a text string:
"softmax" or "relu"

    Returns:
    A -- the output of the activation function, also called the post-activation value
    cache -- a python dictionary containing "linear_cache" and "activation_cache";
             stored for computing the backward pass efficiently
    """

    if activation == "softmax":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = stable_softmax(Z)

    elif activation == "relu":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)

    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache
```

Слика 29 :

Функција која омогућава избор активационе функције у слојевима и рачунање вредности активационе функције у скривеним слојевима и на излазу

```
def linear_forward(A_prev, W, b):
    """
    Implement the linear part of a layer's forward propagation.

    Arguments:
    A -- activations from previous layer (or input data): (size of previous layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)
    b -- bias vector, numpy array of shape (size of the current layer, 1)

    Returns:
    Z -- the input of the activation function, also called pre-activation parameter
    cache -- a python dictionary containing "A", "W" and "b" ; stored for computing the backward pass efficiently
    """
    Z = W.dot(A_prev) + b

    assert(Z.shape == (W.shape[0], A_prev.shape[1]))
    cache = (A_prev, W, b)

    return Z, cache
```

Слика 30:

Функција која израчунава вредност линеарне функције пре активације

5. 3. 4. Рачунање трошка параметара

Функција трошка параметра се користи за процену успешности модела. Она је мера колико је модел погрешан у погледу његове способности да процени однос између улазних и излазних података.
У односу на очекиване вредности излазног вектора избран је модел који примењује логистичку регресију. Логистичка регресија уноси зависну променљиву која се представља бинарним вредностима. То значи да би исход могао бити у било којем од два облика (0 или 1). Обзиром на вишеструки излаз (који зависи од броја стања биљке малине) примењена је ***Softmax*** функција. Функција трошка за проблем логистичке регресије са ***Softmax*** функцијом се назива и *"Cross-validation" или"Cross-entropy-Loss"* функција. Функција којом се израчунава трошак логистичке регресије приказана је на слици 31.

```
def cross_entropy_loss(a3, Y):

    """
    Implement the cost function

    Arguments:
    a3 -- post-activation, output of forward propagation size of [num_of_atributes x num_of_examples ]
    Y -- "true" labels vector, same shape as a3

    Returns:
    cost - value of the cost function
    """
    m = Y.shape[1]
    return 1./m * -1 * np.sum(Y * (a3 + (-np.max(a3) - np.log(np.sum(np.exp(a3-np.max(a3)))))))
```

Слика 31 :
Функција трошка параметара логистичке регресије са Softmax функцијом

5. 3. 5. Пропагација уназад

Пропагација уназад се користи за израчунавање градијената свих параметара у мрежи. Градијенти параметара су потребни како би се извршио update свих параметара у току оптимизације трошка параметара када се укључи један од алгоритма за оптимизацију.

5. 3. 5. 1. Потпуно повезана мрежа

Функције које омогућавају рачунање градијената параметара кроз потпуно повезану мрежу налазе се на сликама 32, 33, 34. За рачунање градијената потпуно повезане мреже коришћени су и градијенти функција ***ReLu*** и ***Softmax*** (слика 35 и 36) и градијент функције трошка параметара *dZ* одређен као на слици 32.

```
def L_model_backward(AL, Y, caches, X):
    """
    Implement the backward propagation for the [LINEAR->RELU] * (L-1) -> LINEAR -> SOFTMAX group

    Arguments:
    AL -- probability vector, output of the forward propagation (L_model_forward())
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat)
    caches -- list of caches containing:
                every cache of linear_activation_forward() with "relu" (there are (L-1) or them, indexes from 0 to L-2)
                the cache of linear_activation_forward() with "SOFTMAX" (there is one, index L-1)

    Returns:
    grads -- A dictionary with the gradients
             grads["dA" + str(l)] = ...
             grads["dW" + str(l)] = ...
             grads["db" + str(l)] = ...
    """
    grads = {}
    L = len(caches) # the number of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL

    # Initializing the backpropagation
    dZ = AL - Y # Derivative of Cross Entropy Loss with Softmax

    # Lth layer (SOFTMAX -> LINEAR) gradients. Inputs: "AL, Y, caches". Outputs: "grads["dAL"], grads["dWL"], grads["dbL"]
    current_cache = caches[L-1]
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] = linear_activation_backward(dZ, AL, current_cache, activation =
"softmax")

    for l in reversed(range(L-1)):
        # lth layer: (RELU -> LINEAR) gradients.
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA" + str(l + 2)], AL, current_cache, activation = "relu")
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads
```

Слика 32 :

Функција која израчунава градијент активационе функције и градијент параметара на излазу и омогућава пролаз кроз свих L-1 слојева мреже

```
def linear_activation_backward(dA, AL, cache, activation):
    """
    Implement the backward propagation for the LINEAR->ACTIVATION layer.

    Arguments:
    dA -- post-activation gradient for current layer l
    cache -- tuple of values (linear_cache, activation_cache) we store for computing backward propagation efficiently
    activation -- the activation to be used in this layer, stored as a text string: "softmax" or "relu"

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l), same shape as W
    db -- Gradient of the cost with respect to b (current layer l), same shape as b
    """
    linear_cache, activation_cache = cache

    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    elif activation == "softmax":
        dZ = softmax_backward(dA, AL, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db
```

Слика 33 :

Функција која омогућава подешавање активационе функције у скривеним слојевима и рачунање градијената активационих функција

```
def linear_backward(dZ, cache):
    """
    Implement the linear portion of backward propagation for a single layer (layer l)

    Arguments:
    dZ -- Gradient of the cost with respect to the linear output (of current layer l)
    cache -- tuple of values (A_prev, W, b) coming from the forward propagation in the
current layer

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the previous
layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l), same shape as W
    db -- Gradient of the cost with respect to b (current layer l), same shape as b
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = 1./m * np.dot(dZ,A_prev.T)
    db = 1./m * np.sum(dZ, axis = 1, keepdims = True)
    dA_prev = np.dot(W.T,dZ)

    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db
```

Слика 34 :

Функција која израчунава градијент активационе функције претходног слоја помоћу линеарних компутација

```
def relu_backward(dA, cache):
    """
    Implement the backward propagation for a single RELU unit.

    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation efficiently

    Returns:
    dZ -- Gradient of the cost with respect to Z
    """

    Z = cache
    dZ = np.array(dA, copy=True) # just converting dz to a correct object.

    # When z <= 0, you should set dz to 0 as well.
    dZ[Z <= 0] = 0

    assert (dZ.shape == Z.shape)

    return dZ
```

Слика 35 :

Функција која израчунава градијент ReLu активационе функције

```
def softmax_backward(dA, AL, cache):
    Z = cache
    m, n = AL.shape
    ones =  np.ones((n, m))
    matrix = np.matmul(AL, ones) * (np.identity(m) - np.matmul(np.ones((m, n)), AL.T))
    dZ = np.matmul(matrix, dA)
    assert (dZ.shape == Z.shape)
    return dZ
```

Слика 36 :

Функција која израчунава градијент softmax активационе функције

5. 3. 5. 2. Конволуцијска мрежа

Пропагација уназад је имплементирана као на сликама 37, 38 и 39.

```
def conv_backward(dZ, cache):
    """
    Implement the backward propagation for a convolution function

    Arguments:
    dZ -- gradient of the cost with respect to the output of the conv layer (Z), numpy array of shape (m, n_H, n_W, n_C)
    cache -- cache of values needed for the conv_backward(), output of conv_forward()

    Returns:
    dA_prev -- gradient of the cost with respect to the input of the conv layer (A_prev),
               numpy array of shape (m, n_H_prev, n_W_prev, n_C_prev)
    dW -- gradient of the cost with respect to the weights of the conv layer (W)
          numpy array of shape (f, f, n_C_prev, n_C)
    db -- gradient of the cost with respect to the biases of the conv layer (b)
          numpy array of shape (1, 1, 1, n_C)
    """

    # Retrieve information from "cache"
    (A_prev, W, b, hparameters) = cache

    # Retrieve dimensions from A_prev's shape
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # Retrieve dimensions from W's shape
    (f, f, n_C_prev, n_C) = W.shape

    # Retrieve information from "hparameters"
    stride = hparameters["stride"]
    pad = hparameters["pad"]

    # Retrieve dimensions from dZ's shape
    (m, n_H, n_W, n_C) = dZ.shape

    # Initialize dA_prev, dW, db with the correct shapes
    dA_prev = np.zeros((m, n_H_prev, n_W_prev, n_C_prev))
    dW = np.zeros((f, f, n_C_prev, n_C))
    db = np.zeros((1, 1, 1, n_C))

    # Pad A_prev and dA_prev
    A_prev_pad = zero_pad(A_prev, pad)
    dA_prev_pad = zero_pad(dA_prev, pad)

    for i in range(m):                       # loop over the training examples

        # select ith training example from A_prev_pad and dA_prev_pad
        a_prev_pad = A_prev_pad[i]
        da_prev_pad = dA_prev_pad[i]

        for h in range(n_H):                   # loop over vertical axis of the output volume
            for w in range(n_W):               # loop over horizontal axis of the output volume
                for c in range(n_C):           # loop over the channels of the output volume

                    # Find the corners of the current "slice"
                    vert_start = h
                    vert_end = vert_start + f
                    horiz_start = w
                    horiz_end = horiz_start + f

                    # Use the corners to define the slice from a_prev_pad
                    a_slice = a_prev_pad[vert_start:vert_end, horiz_start:horiz_end, :]

                    # Update gradients for the window and the filter's parameters using the code formulas given above
                    da_prev_pad[vert_start:vert_end, horiz_start:horiz_end, :] += W[:,:,:,c] * dZ[i, h, w, c]
                    dW[:,:,:,c] += a_slice * dZ[i, h, w, c]
                    db[:,:,:,c] += dZ[i, h, w, c]

        # Set the ith training example's dA_prev to the unpaded da_prev_pad (Hint: use X[pad:-pad, pad:-pad, :])
        dA_prev[i, :, :, :] = da_prev_pad[pad:-pad, pad:-pad, :]

    # Making sure your output shape is correct
    assert(dA_prev.shape == (m, n_H_prev, n_W_prev, n_C_prev))

    return dA_prev, dW, db
```

Слика 37 :

Функција која израчунава градијент параметара филтера и градијент улазних података у једном слоју конволуцијске мреже

```
def create_mask_from_window(x):
    """
    Creates a mask from an input matrix x, to identify the max entry of x.

    Arguments:
    x -- Array of shape (f, f)

    Returns:
    mask -- Array of the same shape as window, contains a True at the position corresponding to the max entry of x.
    """
    mask = x == np.max(x)

    return mask
```

Слика 38 :

Помоћна функција за пропагацију уназад кроз pooling слој која прати максимум вредност параметара слоја након технике pooling-а

```
def pool_backward(dA, cache, mode = "max"):
    """
    Implements the backward pass of the pooling layer

    Arguments:
    dA -- gradient of cost with respect to the output of the pooling layer, same shape as A
    cache -- cache output from the forward pass of the pooling layer, contains the layer's input and hparameters
    mode -- the pooling mode you would like to use, defined as a string ("max" or "average")

    Returns:
    dA_prev -- gradient of cost with respect to the input of the pooling layer, same shape as A_prev
    """

    # Retrieve information from cache (≈1 line)
    (A_prev, hparameters) = cache

    # Retrieve hyperparameters from "hparameters" (≈2 lines)
    stride = hparameters["stride"]
    f = hparameters["f"]

    # Retrieve dimensions from A_prev's shape and dA's shape (≈2 lines)
    m, n_H_prev, n_W_prev, n_C_prev = A_prev.shape
    m, n_H, n_W, n_C = dA.shape

    # Initialize dA_prev with zeros (≈1 line)
    dA_prev = np.zeros(A_prev.shape)

    for i in range(m):                       # loop over the training examples
        # select training example from A_prev (≈1 line)
        a_prev = A_prev[i]
        for h in range(n_H):                   # loop on the vertical axis
            for w in range(n_W):               # loop on the horizontal axis
                for c in range(n_C):           # loop over the channels (depth)
                    # Find the corners of the current "slice" (≈4 lines)
                    vert_start = h
                    vert_end = vert_start + f
                    horiz_start = w
                    horiz_end = horiz_start + f

                    # Compute the backward propagation in both modes.
                    if mode == "max":
                        # Use the corners and "c" to define the current slice from a_prev (≈1 line)
                        a_prev_slice = a_prev[vert_start:vert_end, horiz_start:horiz_end, c]
                        # Create the mask from a_prev_slice (≈1 line)
                        mask = create_mask_from_window(a_prev_slice)
                        # Set dA_prev to be dA_prev + (the mask multiplied by the correct entry of dA) (≈1 line)
                        dA_prev[i, vert_start:vert_end, horiz_start:horiz_end, c] += np.multiply(mask, dA[i, h, w, c])

                    elif mode == "average":
                        # Get the value a from dA (≈1 line)
                        da = dA[i, h, w, c]
                        # Define the shape of the filter as fxf (≈1 line)
                        shape = (f, f)
                        # Distribute it to get the correct slice of dA_prev. i.e. Add the distributed value of da. (≈1
line)
                        dA_prev[i, vert_start:vert_end, horiz_start:horiz_end, c] += distribute_value(da, shape)

    # Making sure your output shape is correct
    assert(dA_prev.shape == A_prev.shape)

    return dA_prev
```

Слика 39 :

Функција која израчунава градијент улазних података pooling слоја

5. 3. 6. Update-овање параметара

Update-овање параметара се врши како би се пронашле оптималне вредности за све параметре мреже који дефиниши границу одлучивања. Помоћу вредности најоптималнијих параметара W и b трошак параметара (cost) је минималан.

Градијент опадања

Правило градијента опадања : $W^{[l]} = W^{[l]} - \alpha\, dW^{[l]}$, где је α стопа учења.
$b^{[l]} = b^{[l]} - \alpha\, db^{[l]}$

Вредност стопе учења која је коришћена није константна вредност већ се смањује чиме је учење убрзано (слика 41).

Како би update-овао параметре градијент опадања користи градијенте параметара који су претходно израчунати.
Уколико је градијент параметра W позитиван број следи да ће вредност претходног параметра W бити већа од вредности update-ованог параметра W , па ће смер градијента опадања одржати правилан смер ка минимуму.
Уколико је градијент параметра W негативан број следи да ће вредност претходног параметра W бити мања од вредности update-ованог параметра W па ће смер градијента опадања одржати правилан смер ка минимуму.

Стога за било који знак градијента параметара W градијент опадања ће тежити да минимизује параметре W и b.

Адам алгоритам

“Правило Адама за l = 1,…, L:

$$\begin{cases} v_{W^{[l]}} = \beta_1 v_{W^{[l]}} + (1-\beta_1)\frac{\partial J}{\partial W^{[l]}} \\ v^{corrected}_{W^{[l]}} = \frac{v_{W^{[l]}}}{1-(\beta_1)^t} \\ s_{W^{[l]}} = \beta_2 s_{W^{[l]}} + (1-\beta_2)(\frac{\partial J}{\partial W^{[l]}})^2 \\ s^{corrected}_{W^{[l]}} = \frac{s_{W^{[l]}}}{1-(\beta_2)^t} \\ W^{[l]} = W^{[l]} - \alpha \frac{v^{corrected}_{W^{[l]}}}{\sqrt{s^{corrected}_{W^{[l]}}}+\varepsilon} \end{cases}$$

, где је :
t броји број корака Адам алгоритма,
l број слојева,
$\beta 1$ и $\beta 2$ су хипер параметри који контролишу експоненцијално отежане просеке,
стопа учења и
ε јако мали број да би се избегло дељење нулом

(2).”

Како би update-овао параметре алгоритам Адам користи градијенте параметара који су претходно израчунати (слика 40).

```
def update_parameters_with_adam(parameters, grads, v, s, t, learning_rate = 0.01,
                                beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8):
    """
    Update parameters using Adam

    Arguments:
    parameters -- python dictionary containing your parameters:
                    parameters['W' + str(l)] = Wl
                    parameters['b' + str(l)] = bl
    grads -- python dictionary containing your gradients for each parameters:
                    grads['dW' + str(l)] = dWl
                    grads['db' + str(l)] = dbl
    v -- Adam variable, moving average of the first gradient, python dictionary
    s -- Adam variable, moving average of the squared gradient, python dictionary
    learning_rate -- the learning rate, scalar.
    beta1 -- Exponential decay hyperparameter for the first moment estimates
    beta2 -- Exponential decay hyperparameter for the second moment estimates
    epsilon -- hyperparameter preventing division by zero in Adam updates

    Returns:
    parameters -- python dictionary containing your updated parameters
    v -- Adam variable, moving average of the first gradient, python dictionary
    s -- Adam variable, moving average of the squared gradient, python dictionary
    """
    L = len(parameters) // 2                 # number of layers in the neural networks
    v_corrected = {}                         # Initializing first moment estimate, python dictionary
    s_corrected = {}                         # Initializing second moment estimate, python dictionary

    # Perform Adam update on all parameters
    for l in range(L):
        # Moving average of the gradients. Inputs: "v, grads, beta1". Output: "v".
        v["dW" + str(l+1)] = beta1*v["dW" + str(l+1)]+(1-beta1)*grads['dW'+str(l+1)]
        v["db" + str(l+1)] = beta1*v["db" + str(l+1)]+(1-beta1)*grads['db'+str(l+1)]

        # Compute bias-corrected first moment estimate. Inputs: "v, beta1, t". Output: "v_corrected".
        v_corrected["dW" + str(l+1)] = v["dW" + str(l+1)]/(1-pow(beta1,t))
        v_corrected["db" + str(l+1)] = v["db" + str(l+1)]/(1-pow(beta1,t))

        # Moving average of the squared gradients. Inputs: "s, grads, beta2". Output: "s".
        s["dW" + str(l+1)] = beta2*s["dW" + str(l+1)]+(1-beta2)*np.power(grads['dW'+str(l+1)],2)
        s["db" + str(l+1)] = beta2*s["db" + str(l+1)]+(1-beta2)*np.power(grads['db'+str(l+1)],2)

        # Compute bias-corrected second raw moment estimate. Inputs: "s, beta2, t". Output: "s_corrected".
        s_corrected["dW" + str(l+1)] = s["dW" + str(l+1)]/(1-pow(beta2,t))
        s_corrected["db" + str(l+1)] = s["db" + str(l+1)]/(1-pow(beta2,t))

        # Update parameters. Inputs: "parameters, learning_rate, v_corrected, s_corrected, epsilon". Output:
"parameters".
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)]-learning_rate*np.divide(v_corrected["dW" +
str(l+1)],np.sqrt(s_corrected["dW" + str(l+1)])+epsilon)
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)]-learning_rate*np.divide(v_corrected["db" +
str(l+1)],np.sqrt(s_corrected["db" + str(l+1)])+epsilon)

    return parameters, v, s
```

Слика 40 :
Update-овање параметара користећи алгоритам Адам

6. Функција модела неуронске мреже

За тренирање параметара неуронске мреже све функције које су претходно приказане у моделу неуронске мреже су повезане у оквиру функције модела са слике 41, 42 и 43.

Функција модела је дефинисана на следећи начин :

model(X, Y, layers_dims_conv, channels, nc, layers_dims_fc, learning_rate = 0.0007, mini_batch_size = 64, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8, num_epochs = 100, print_cost = True), где је

- X - вредност свих улазних тренинг података
- Y - вредност свих излазних тренинг података
- layers_dims_conv - речник који садржи димензије филтера у конволуцијском делу мреже
- channels - број канала за сваки од филтера
- nc - број филтера за сваку од конволуција
- layers_dims_fc - речник који садржи димензије параметара у скривеним слојевима потпуно повезане мреже
- learning_rate = 0.0007 - дифолт вредност стопе учења
- mini_batch_size = 64 дифолт вредност величине серије података
- beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8 - дифолт вредности хипер параметара Адам алгоритма
- num_epochs = 100 - дифолт вредност броја епоха
- print_cost = False дифолт вредност параметра који одлучује о цртању функције трошка

```
def model(X, Y, layers_dims_conv, channels, nc, layers_dims_fc,
          learning_rate_init = 0.0007, mini_batch_size = 64, beta1 = 0.9, beta2 = 0.999,
          epsilon = 1e-8, num_epochs = 100, print_cost = True):
    costs = []                        # to keep track of the cost
    bcost = []
    t = 0                             # initializing the counter required for Adam update
    seed = 10                         # For grading purposes, so that your "random" minibatches are the same as ours

    hparameters1 = {"pad" : 2, "stride": 3}
    hparameters2 = {"stride" : 2, "f": 2}
    hparameters3 = {"pad" : 1, "stride": 3}
    hparameters4 = {"stride" : 2, "f": 2}

    # Initialize parameters CONV
    parameters_conv = initialize_parameters_rand_CONV(layers_dims_conv,channels,nc)
    # Initialize parameters FC
    parameters_fc = initialize_parameters_he_FC(layers_dims_fc)
    # Gather parameters in python dict
    parameters_all = solve_all(parameters_conv, parameters_fc)
    L = len(parameters_fc) // 2                 # number of layers in the neural network
    # Initialize the optimizer
    v, s = initialize_adam(parameters_all)
    learning_param1 = learning_rate_init / 40
    train_num = X.shape[0]
    learning_param2 = int( train_num / mini_batch_size )
    for i in range(num_epochs):
            learning_rate = learning_rate_init * (1/(1+learning_param1*(i*learning_param2)))
        # Define the random minibatches. We increment the seed to reshuffle differently the dataset after each epoch
            seed = seed + 1
            minibatches = random_mini_batches(X, Y, mini_batch_size, seed)
            real = i + 1
            if print_cost:
                print ("Running Epoch %i..." %(real))
                print ("Learning rate = " + str(learning_rate))

            for minibatch in minibatches:

                # Select a minibatch
                (minibatch_X, minibatch_Y) = minibatch
```

Слика 41 :

Модел оптимизоване неуронске мреже, I део

```
                # Get parameters for CONV network, parameters_conv
                parameters_conv = solve_conv(parameters_all, layers_dims_conv, channels, nc)
                # Forward propagation CONV
                P2, cache_pool2, cache_pool1, cache_conv1, cache_conv2, c1, c2 = Convolutional_Forward(
                    minibatch_X, parameters_conv, hparameters1, hparameters2, hparameters3, hparameters4)

                # Create 2D input for FC nerwork, FC1
                FC1 = P2.reshape(P2.shape[0], -1).T

                # Get parameters for FC network, parameters_fc
                parameters_fc = solve_fc(parameters_all, layers_dims_conv)
                # Forward propagation FC
                AL, cache_fc = L_model_forward(FC1, parameters_fc)

                # Compute cost
                cost = cross_entropy_loss(AL, minibatch_Y.T)
                if print_cost:
                    bcost.append(cost)

                # Backward propagation FC
                grads_fc = L_model_backward(AL, minibatch_Y.T, cache_fc, minibatch_X.T)

                # Backward propagation CONV, I
                dP2 = pool_backward(P2, cache_pool2, mode = "max")
                dA2 = relu_backward(dP2, c2)
                dZ2, dW2, db2 = conv_backward(dA2, cache_conv2)
                # Backward propagation CONV, II
                dP1 = pool_backward(dZ2, cache_pool1, mode = "max")
                dA1 = relu_backward(dP1, c1)
                dZ1, dW1, db1 = conv_backward(dA1, cache_conv1)
```

Слика 42 :

Модел оптимизоване неуронске мреже, II део

```
            # Gather data
            parameters_all = solve_all(parameters_conv, parameters_fc)
            grads_all = solve_all_grads(grads_fc, dA1, dW1, db1, dA2, dW2, db2 )

            # Update parameters with Adam optimizer
            t = t + 1 # Adam counter
            parameters_all, v, s = update_parameters_with_adam(parameters_all, grads_all, v, s, t,
                                                               learning_rate, beta1, beta2,  epsilon)

        # Print the cost after every epoch & save
        if print_cost:
            print ("Cost after epoch %i: %f" %(real, cost))
            costs.append(cost)
            print("\n\n\n")

# plot the cost
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('epochs')
plt.title("Learning rate = " + str(learning_rate))
plt.show()

# plot the bcost
plt.plot(bcost)
plt.ylabel('batch cost')
plt.xlabel('batch epochs')
plt.title("Learning rate = " + str(learning_rate))
plt.show()

return parameters_all, cost, bcost
```

Слика 43 :

Модел оптимизоване неуронске мреже, III део

7. Тренирање и резултати неуронске мреже

За тренирање параметара неуронске мреже позива са функција модела којој су прослеђени одговарајући параметри који фигуришу унутар конволуцијске (*layers_dims_conv, channels, ns*) и потпуно повезане мреже (*layers_dims_fc*) (слика 44).

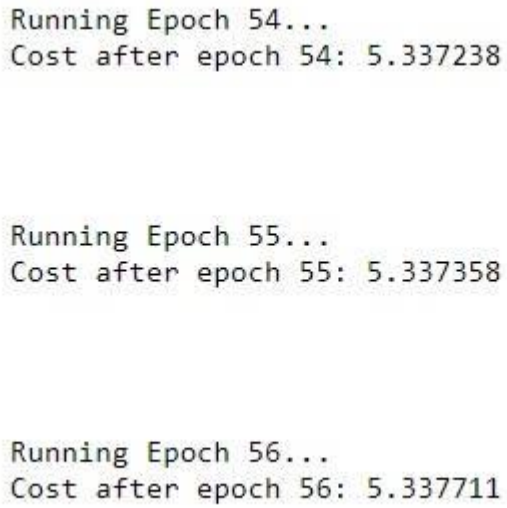
```
In [ ]: layers_dims_conv = [5, 3]                    # In first conv layer dim(filter)=(fxfxc)=5x5xc1
                                                     # where c1 is a number of channels for the first filter
                                                     # In second  conv layer dim(filter)=(fxfxc)=3x3xc2
                                                     # where c2 is a number of channels for the second filter

        channels = [3, 4]                            # c1 = 3, c2 = 4
        nc = [4, 8]                                  # Number of filters in first conv layer = 4, second = 8
        layers_dims_fc = [200, 447, 774, 1094, 631, 315, 181, 127, 73, 36, 20, num_of_dirs]


        # train 2-layer convolutional & 3-layer fully-connected model
        parameters, cost, bcost = model(x_train, y_train, layers_dims_conv, channels, nc, layers_dims_fc)
```

Слика 44 :

Позив функције модела и тренирање параметара на тренинг скупу података

Резултати неуронске мреже, односно вредности функције трошка параметра у току 10 епоха су приказани на сликама 45, 46 и 47.

Слика 45 :
Резултат после првих 4 епоха



Слика 46 :
Резултат после 56 епоха

```
Cost after epoch 97: 5.336540

Running Epoch 98...
Cost after epoch 98: 5.335908

Running Epoch 99...
Cost after epoch 99: 5.335480

Running Epoch 100...
Cost after epoch 100: 5.336910
```

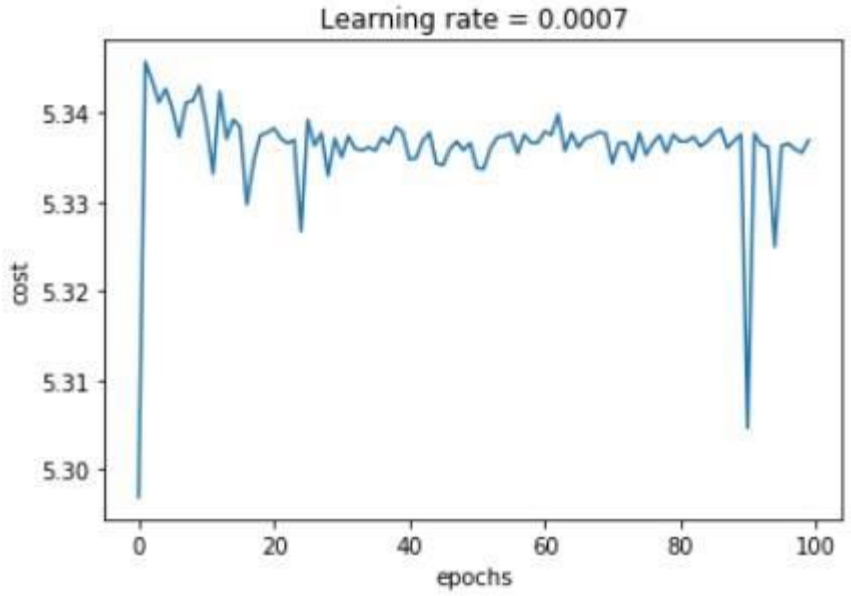
Слика 47 :
Резултат после 100 епоха

Тачност неуронске мреже на тренинг и тест скупу односно разлика предвиђених вредности излазног вектора у односу на стварне вредности излазног вектора представљена је на слици 48.

```
In [11]: # Predict
         print ("On the train set:")
         pred_train = predict(x_train, y_train, parameters)
         print ("On the test set:")
         pred_test = predict(x_test, y_test, parameters)

         On the train set:
         Accuracy: 0.8698224852071006
         On the test set:
         Accuracy: 0.871740200098409
```

Слика 48 :
Одређивање тачности алгоритма на тренинг и тест скупу

Вредност функције трошка није константно опадајућа функција јер Адам алгоритам оптимизације у току епоха изабира серије улазних података на рандом начин тј. при рачунању трошка и update-овања параметара увек ради са различитим скупом серије података. Графички приказ представљен је на слици 49.



Слика 49 :
Графички приказ вредности функције трошка у зависности од броја епоха

8. Закључак

Грешка коју овај класификациони алгоритам прави на :

- тренинг скупу податка је : 0,130, грешка је 13%
- тест скупу података је : 0,128, грешка је 12,8%

Ова грешка се може сматрати грешком високог одступања/бијаса (*bias*, engl.). Тренинг сет даје објашњење о високом бијасу. Грешка услед бијаса је количина колико се предвиђање очекиваног модела разликује од праве вредности тренинг података. Начини на који се ова грешка може смањити су:

- већа мрежа / већи број слојева мреже
- дужа итерација / већи број епоха
- другачија архитектура мреже

9. Литература

1. М. Николић, М. Ивановић, С. Миленковић, Ј. Миливојевић, М. Милутиновић. The state and prospects of raspberry production in Serbia. Acta horticulturae, 2008; 777(777):243-250, доступно са: https://www.researchgate.net/publication/284707670_The_state_and_prospects_of_raspberry_production_in_Serbia

2. A. Ng, Ванредни професор са Универзитета Станфорд и оснивач курсева deeplearning.ai и Coursera Inc., Deep Learning Specialization [Видео материјали], доступно са: https://www.coursera.org/specializations/deep-learning