LAB 12

```
!pip install tensorflow-graphics
!pip install trimesh
```

```
import numpy as np
import tensorflow as tf
import trimesh

import tensorflow_graphics.geometry.transformation as tfg_transformation
from tensorflow_graphics.notebooks import threejs_visualization
```

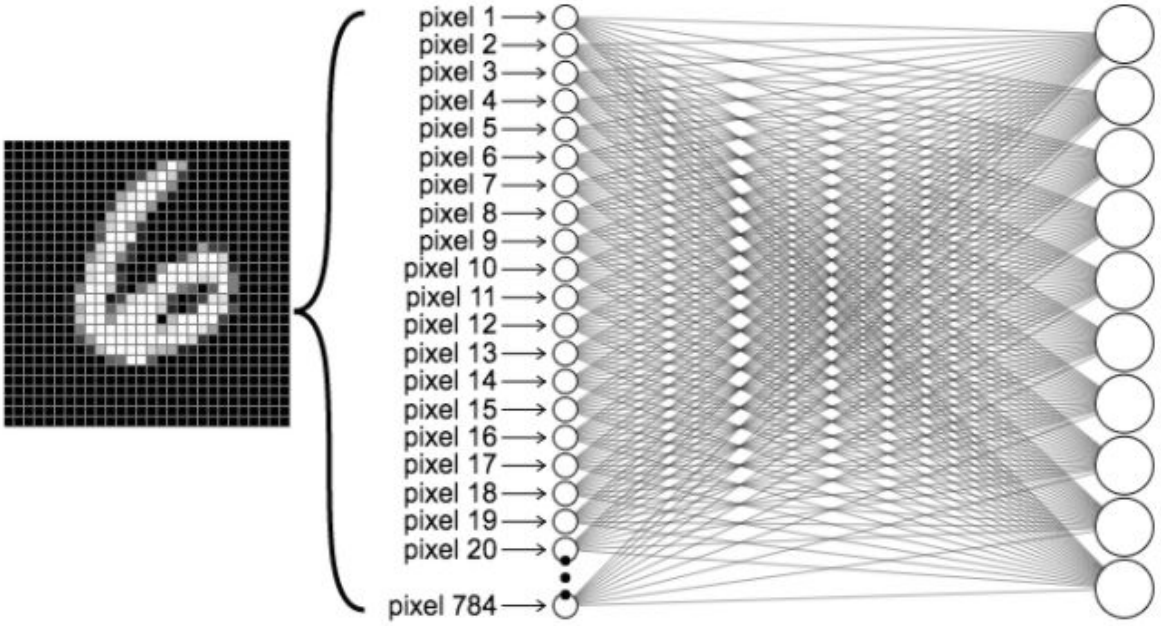
```
!wget -N https://storage.googleapis.com/tensorflow-graphics/notebooks/index/cow.obj
mesh = trimesh.load("cow.obj")
mesh = {"vertices": mesh.vertices, "faces": mesh.faces}
_ = threejs_visualization.triangular_mesh_renderer(mesh, width=400, height=400)
axis = np.array((0., 1., 0.))
angle = np.array((np.pi / 4.,))
mesh["vertices"] = tfg_transformation.axis_angle.rotate(mesh["vertices"], axis,
                                                        angle).numpy()
_ = threejs_visualization.triangular_mesh_renderer(mesh, width=400, height=400)
```

ZADANIE 2

Jednowarstwowa sieć neuronowa w Tensorflow do klasyfikacji cyfr z MNIST.
Rozpoznawanie cyfr MNIST jest jednym z najbardziej popularnych problemów w świecie uczenia maszynowego.
Zbirór MNIST i problem z nim związany czyli klasyfikacja odręcznie pisanych cyfr jest już kultowym zadaniem w dziedzinie uczenia maszynowego. Pierwsze prace Yana Lecun pojawiły się w roku 1998, później systematycznie od tego czasu kolejni badacze usiłowali pobić poprzedni rekord w rozpoznawaniu cyfr. Obecny rekord wynosi 99.79% (MNIST Test Error 0.21%).

Jednowarstwowa sieć neuronowa

Jest to stosunkowo prosta siec zawierająca wejście i wyjście. Na wejściu podajemy jeden obrazek 28x28px, rozciągnięty wiersz po wierszu jako wektor o 784 wymiarach (28*28=784), każdemu obrazkowi będzie odpowiadała etykieta 0…9, zakodowana jako 10-elementowy wektor (one-hot encoding). Stąd też wyjściem sieci jest 10-cio elementowy wektor zawierający prawdopodobieństwa określające na ile rozpatrywana cyfra na obrazie podobna jest do poszczególnych cyfr. Na i-tej pozycji mamy prawdopodobieństwo, że na wejściu podaliśmy liczbę i, czyli na pozycji 0 mamy prawdopodobieństwo tego, że podany obraz zawiera cyfrę zero, na pozycji 1 prawdopodobieństwo, że podana cyfra to 1 i tak dalej.



Sieć neuronowa jako operacje macierzowe

Zauważmy, że do każdego neuronu wyjściowego dochodzi 784 połączeń, każde takie połączenie ma stowarzyszoną wagę (liczbę rzeczywistą). Przemnażając 784 wartości z wejścia przez 784 wag, połączonych z wybranym węzłem wyjściowym, a następnie sumując je otrzymujemy jedną liczbę. Wartość ta, w węźle przepuszczana jest przez nieliniowy filtr (funkcję aktywacji), abyśmy na koniec otrzymali liczbę informującą nas na ile dane wejście przypomina cyfrę z wybranej pozycji.
Co mamy dane? Po pierwsze zbiór cyfr, w postaci obrazów, z których każda to wektor xi∈R^784. Po drugie, posiadamy zestaw wag. Na jeden z dziesięciu neuronów wyjściowych przypadają 784 wagi, w celu usystematyzowania pracy z nimi ułóżmy je w macierz W :

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,10} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,10} \\ \vdots & \vdots & \ddots & \vdots \\ w_{784,1} & w_{784,2} & \cdots & w_{784,10} \end{bmatrix}$$

Mamy 10 kolumn w macierzy, czyli tyle ile klas. Kolumna o numerze i przechowuje współczynniki (połączenia) dla i-tego neuronu wyjściowego. Chcąc obliczyć wartości na wyjściu sieci wystarczy dokonać mnożenia macierzy, przemnożyć wektor xi przez macierz W.

$$\begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,784} \end{bmatrix} * \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,10} \\ w_{2,1} & w_{2,2} & \dots & w_{2,10} \\ \vdots & \vdots & \ddots & \vdots \\ w_{784,1} & w_{784,2} & \dots & w_{784,10} \end{bmatrix} =$$
$$= \begin{bmatrix} out_{1,1} & out_{1,2} & \dots & out_{1,10} \end{bmatrix}$$

Podejście wsadowe, przetwarzamy wektory w paczkach (batch)

Powyższe podejście możemy jeszcze trochę ulepszyć. Zamiast w danym kroku rozpatrywać tylko jeden wektor wejściowy to można wziąć ich całą paczkę (batch) np. 100 i przemnożyć przez macierz W dzięki temu zabiegowi od razu otrzymamy przykładowe odpowiedzi sieci dla 100 obrazów. Pozwala to lepiej wykorzystać zasoby obliczeniowe i dokonać wielu optymalizacji podczas mnożenia. A także uczynić proces uczenia bardziej stabilnym (głównie chodzi o obliczenia gradientów w algorytmie wstecznej propagacji błędów).

$$\begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,784} \\ x_{2,1} & x_{2,2} & \dots & x_{2,784} \\ \vdots & \vdots & \ddots & \vdots \\ x_{100,1} & x_{100,2} & \dots & x_{100,784} \end{bmatrix} * \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,10} \\ w_{2,1} & w_{2,2} & \dots & w_{2,10} \\ \vdots & \vdots & \ddots & \vdots \\ w_{784,1} & w_{784,2} & \dots & w_{784,10} \end{bmatrix} =$$
$$= \begin{bmatrix} out_{1,1} & out_{1,2} & \dots & out_{1,10} \\ out_{2,1} & out_{2,2} & \dots & out_{2,10} \\ \vdots & \vdots & \ddots & \vdots \\ out_{100,1} & out_{100,2} & \dots & out_{100,10} \end{bmatrix}$$

Wyjściowe wartości out przepuszczane są przez funkcję aktywacji, a następnie cały wiersz jest normalizowany, tak aby określał prawdopodobieństwo. Suma wartości z wiersza powinna wynosić 1.

Trening sieci neuronowej

W naszym przykładzie, sieć będzie uczona z wykorzystaniem algorytmu wstecznej propagacji błędów wraz z wykorzystaniem algorytmu Gradient Descent do optymalizacji. Tak naprawdę to całością zajmie się Tensorflow, my skorzystamy z gotowych funkcji.

Importy, stałe oraz ściągnięcie zbioru MNIST

Na początku importujemy własną bibliotekę z kilkoma funkcjami do wizualizacji (znajduje się ona w udostępnionym projekcie), tensorflow oraz funkcję do ściągnięcia zbioru MNIST. Ustawiamy kilka stałych:

NUM_ITERS=5000 – liczba iteracji algorytmu, ile mini-batch'y ma przetworzyć, zwiększając dostaniemy lepsze wyniki, ale i dłużej się będzie liczył model
DISPLAY_STEP=100 – co ile iteracji obliczamy statysytki, które pomogą nam zwizualizować postęp uczenia (test error, loss function etc)
BATCH=100 – rozmiar paczki, ile obrazków bierzemy naraz pod uwagę (zazwyczaj 64,100,128, 200)
Następnie ściągamy zbiór MNIST, tak naprawdę ściągnięcie nastąpi przy pierwszym uruchominiu, później skrypt będzie korzystał z już sciągniętej wersji.

```
1.  import visualizations as vis
2.  import tensorflow as tf
3.  from tensorflow.contrib.learn.python.learn.datasets.mnist import read_data_sets
4.
5.
6.  NUM_ITERS=5000
7.  DISPLAY_STEP=100
8.  BATCH=100
9.  tf.set_random_seed(0)
10.
11.  # Download images and labels
12.  mnist = read_data_sets("MNISTdata", one_hot=True, reshape=False,
    validation_size=0)
13.  # mnist.test (10K images+labels) -> mnist.test.images, mnist.test.labels
14.  # mnist.train (60K images+labels) -> mnist.train.images, mnist.test.labels
```

Placeholders, variables – przygotowujemy pojemniki

Przygotujmy teraz zmienne oraz pojemniki(placeholders) na dane niezbędne dla grafu Tensorflow.
Określamy, że zmienna X będzie pojemnikiem na dane o rozmiarach [ilość obrazów w paczce, szerokość obrazu, wysokość obrazu, ilość kanałów]. Komentarza wymagają tylko pierwszy i ostatni wymiar, None informuje TF, że ma się sam dostosować do ilości danych, a 1 na końcu określa, że nasz obraz jest w odcieniach szarości. Gdybyśmy przetwarzali obraz kolorowy RGB to byśmy operowali 3 kanałami. Podobnie do X tworzymy pojemnik Y_ przechowujący etykiety związane z obrazami, Y_ zawiera wektory o 10 elementach zakodowane zgodnie z one-hot encoding.

Istotnym elementem jest macierz W, zauważcie że jest ona typu tf.Variable. Jest to specjalny typ w tensorflow służący do przechowywania parametrów modelu, które mają być

optymalizowane. TF sam będzie uaktualniał ich wartości w trakcie uczenia (wiem, it's magic). XX – jest naszym zbiorem X, z rozwiniętymi wierszami.

Na samym końcu definiujemy nasz model, przemnażamy (tf.matmul) wartości wejściowe z XX przez wagi z macierzy dodajemy przesunięcie i wszystko normalizujemy poprzez wykorzystnie funkcji tf.softmax

```
1.  # All the data will be stored in X - tensor, 4 dimensional matrix
2.  # The first dimension (None) will index the images in the mini-batch
3.  X = tf.placeholder(tf.float32, [None, 28, 28, 1])
4.
5.  # correct answers will go here
6.  Y_ = tf.placeholder(tf.float32, [None, 10])
7.
8.  # weights W[784, 10] - initialized with random values from normal distribution
    mean=0, stddev=0.1
9.  W = tf.Variable(tf.truncated_normal([784, 10],stddev=0.1))
10.
11. # biases b[10]
12. b = tf.Variable(tf.zeros([10]))
13.
14. # matplotlib visualisation
15. allweights = tf.reshape(W, [-1])
16. allbiases = tf.reshape(b, [-1])
17.
18. # flatten the images, unrole eacha image row by row, create vector[784]
19. # -1 in the shape definition means compute automatically the size of this dimension
20. XX = tf.reshape(X, [-1, 784])
21.
22. # Define model
23. Y = tf.nn.softmax(tf.matmul(XX, W) + b)
```

Mając zdefiniowany model, potrzebujemy funkcji oceny naszego modelu. Najczęściej nazywa się ją funkcją straty (loss function). Jest to funkcja dzięki, dzięki której wiemy czy model jest dobry czy zły, na jej podstawie dokonujemy optymalizacji. Informuje ona o ilości popełnianych błędów przez nasz model, jeżeli wartość jest wysoka to znaczy, że nasz model wymaga jeszcze uczenia, czyli dostosowania wag z macierzy W. W naszym przykładzie wykorzystam cross entropy.

Dodatkowo w celach pomocniczych definiujemy w jaki sposób będziemy obliczać accuracy. Na koniec tworzymy instancję klasy GradientDescentOptimizer i do jej metody minimize przekazujemy "przepis" na obliczenie cross_entropy

```
1.  # cross-entropy
2.  # log takes the log of each element, * multiplies the tensors element by element
3.  # reduce_mean will add all the components in the tensor
4.  # so here we end up with the total cross-entropy for all images in the batch
5.  cross_entropy = -tf.reduce_mean(Y_ * tf.log(Y)) * 1000.0  # normalized for batches of
    100 images,
6.                                                           # *10 because  "mean"
    included an unwanted division by 10
7.
8.  # accuracy of the trained model, between 0 (worst) and 1 (best)
9.  correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
10. accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
11.
12. # training, learning rate = 0.005
13. train_step = tf.train.GradientDescentOptimizer(0.005).minimize(cross_entropy)
```

Ostatnim etapem jest uruchomienie obliczeń w pętli. Najpier inicjalizujemy wszystkie zmienne i parametry z sieci, tworzymy pomocnicze listy na wyniki cząstkowe do wizualizacji. Tworzymy sesję tensorflow w pętli for kolejno uruchamiamy obliczenia przy pomocy sess.run. Funkcja ta jest wyjątkowa, to dzięki niej dzieje się cała magia, ona wprawia w ruch maszynerię działającą pod spodem. Wszelkie niezbędne dane do obliczeń przekazujemy z wykorzystaniem uprzednio stworzonych placeholders poprzez słownik feed_dict. Z punktu widzenia modelu najważniejsze jest wywołanie sess.run(train_step, feed_dict={X: batch_X, Y_: batch_Y}) to ono dokonuje wykonania kroku optymalizacyjnego, poprzednie wywołania służą tylko obliczeniu statystyk podczas uczenia takich jak (accuracy_trn, accuracy_tst, loss itp).

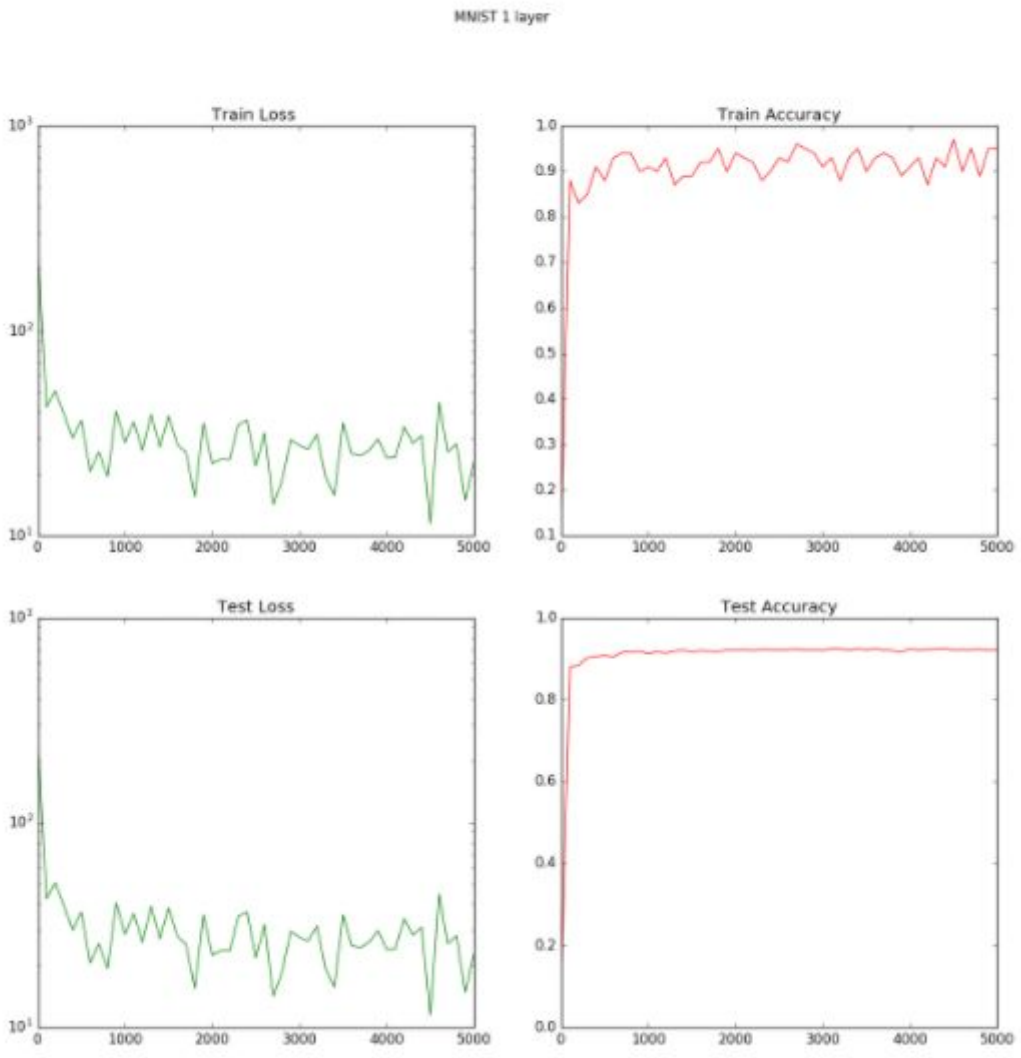
Ostatnia linia ma zadanie narysować wykresy przedstawiające jak wyglądał trening.

```
# Initializing the variables
init = tf.global_variables_initializer()

train_losses = list()
train_acc = list()
test_losses = list()
test_acc = list()

saver = tf.train.Saver()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)


    for i in range(NUM_ITERS+1):
        # training on batches of 100 images with 100 labels
        batch_X, batch_Y = mnist.train.next_batch(BATCH)

        if i%DISPLAY_STEP ==0:
            # compute training values for visualisation
            acc_trn, loss_trn, w, b = sess.run([accuracy, cross_entropy, allweights, allbiases], feed_dict={X: batch_X, Y_: batch_Y})


            acc_tst, loss_tst = sess.run([accuracy, cross_entropy], feed_dict={X: mnist.test.images, Y_: mnist.test.labels})

            print("#{} Trn acc={} , Trn loss={} Tst acc={} , Tst loss={}".format(i,acc_trn,loss_trn,acc_tst,loss_tst))

            train_losses.append(loss_trn)
            train_acc.append(acc_trn)
            test_losses.append(loss_trn)
            test_acc.append(acc_tst)

        # the backpropagationn training step
        sess.run(train_step, feed_dict={X: batch_X, Y_: batch_Y})



title = "MNIST 1 layer"
vis.losses_accuracies_plots(train_losses,train_acc,test_losses,test_acc,title,DISPLAY_STEP)
```

W wyniku działania skryptu dla 5k iteracji powinniście otrzymać wynik w konsoli:

```
#0 Trn acc=0.10000000149011612 , Trn loss=257.5960998535156 Tst acc=0.09359999746084213 , Tst
loss=262.1988525390625
#100 Trn acc=0.8799999952316284 , Trn loss=42.37456512451172 Tst acc=0.879800021648407 , Tst
loss=41.94294357299805
#200 Trn acc=0.8299999833106995 , Trn loss=50.943721771240234 Tst acc=0.883899986743927 , Tst
loss=39.74076843261719
#300 Trn acc=0.8500000238418579 , Trn loss=39.26817321777344 Tst acc=0.9021999835968018 , Tst
loss=33.9247932434082
# ....
#4800 Trn acc=0.9100000262260437 , Trn loss=32.18285369873047 Tst acc=0.9223999977111816 , Tst
loss=27.490428924560547
#4900 Trn acc=0.9399999976158142 , Trn loss=19.267147064208984 Tst acc=0.9240999817848206 , Tst
loss=27.271032333374023
#5000 Trn acc=0.949999988079071 , Trn loss=14.251474380493164 Tst acc=0.923799991607666 , Tst
loss=27.61037826538086
```

Dodatkowo zostaną wygenerowane wykresy:



Źrodło:
https://github.com/ksopyla/tensorflow-mnist-convnets
https://ksopyla.com/python/tensorflow/klasyfikacja-cyfr-mnist-tensorflow/
https://www.youtube.com/watch?v=Trc52FvMLEg&feature=emb_title&ab_channel=SoonHinKhor
https://ksopyla.com/category/python/tensorflow/