Assignment 4 - Classifying images using Numpy and Keras.

For this assignment, you should complete the exercises in this notebook. It is similar to the notebook posted for binary logistic neuron.

Look for requests containing the text "**your code**". E.g. "put your code here", "replace None by your code", etc. If there is no such request in a cell, just run the cell.

```
import tensorflow as tf
from tensorflow import keras
print(tf.__version__)

import numpy as np
import matplotlib.pyplot as plt
import time
```

```
2.9.2
```

```
# Let's load the dataset of handwritten digits.

(X, Y), (X_test, Y_test) = keras.datasets.fashion_mnist.load_data()

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(X[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[Y[i]])
plt.show()
```


```
# We will reshape (flatten) X arrays so that they become rank 2 arrays.
# We will reshape the Y arrays so that they are not rank 1 arrays but rank 2 arrays.
# They should be rank 2 arrays.

X = X.reshape(X.shape[0], X.shape[1]*X.shape[2])
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1]*X_test.shape[2])
```

```
Y = Y.reshape((Y.shape[0],1))
Y_test = Y_test.reshape((Y_test.shape[0],1))

print("Train dataset shape", X.shape, Y.shape)
print("Test dataset shape", X_test.shape, Y_test.shape)

print("Y =", Y)

m   = X.shape[0]
n_x = X.shape[1]

C = 10
```

```
Train dataset shape (60000, 784) (60000, 1)
Test dataset shape (10000, 784) (10000, 1)
Y = [[9]
 [0]
 [0]
 ...
 [3]
 [0]
 [5]]
```

Exercise 1 - One Hot Encoding

Convert Y to "one-hot" encoding. E.g. if the original Y is

$$Y = \begin{bmatrix} 1 \\ 5 \\ 9 \end{bmatrix}$$

the new Y should be

$$Y = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

```
# Toward this goal, let's check what np.arange(C) produces
np.arange(C)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
# Let's see again what Y is
Y
```

```
array([[9],
       [0],
       [0],
       ...,
       [3],
       [0],
       [5]], dtype=uint8)
```

What would broadcasting these arrays together would look like?

Let's check.

```
# This is formatted as code



a,b = np.broadcast_arrays(np.arange(C), Y)

print("broadcasted np.arange(C) = \n", a)
print("broadcasted Y = \n", b)
```

```
broadcasted np.arange(C) =
 [[0 1 2 ... 7 8 9]
 [0 1 2 ... 7 8 9]
 [0 1 2 ... 7 8 9]
 ...
 [0 1 2 ... 7 8 9]
 [0 1 2 ... 7 8 9]
```

```
 [0 1 2 ... 7 8 9]]
broadcasted Y =
 [[9 9 9 ... 9 9 9]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [3 3 3 ... 3 3 3]
 [0 0 0 ... 0 0 0]
 [5 5 5 ... 5 5 5]]
```

```
# If we compare np.arange(C) with Y using the equality sign ==,
# the numpy broadcasting will do its magic to give us what we want.
# Try it out. Then assign the result to a new variable Y_new.
# Don't worry for the "True" and "False" values looking like strings.
# They behave in fact like numbers, i.e. True is 1, False is 0.
# Finally, assign Y_new to Y so that we have to deal with Y in rest of the notebook.
# Cast the boolean values of Y_new to integer by calling Y_new.astype(int)

# Put your code in place of None objects

Y_new = np.arange(C) == Y
print("Y_new=", Y_new)
Y = Y_new.astype(int)
print("Y=",Y)

# create diagonal 1's matrix
diag = np.eye(C)
index_val = Y_test.reshape(-1)
Y_new_test = diag[index_val]
print("Y_new_test=", Y_new_test.astype(bool))
Y_test = Y_new_test
print("Y_test=",Y_test)
```

```
Y_new= [[False False False ... False False  True]
 [ True False False ... False False False]
 [ True False False ... False False False]
 ...
 [False False False ... False False False]
 [ True False False ... False False False]
 [False False False ... False False False]]
Y= [[0 0 0 ... 0 0 1]
 [1 0 0 ... 0 0 0]
 [1 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [1 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
Y_new_test= [[False False False ... False False  True]
 [False False  True ... False False False]
 [False  True False ... False False False]
 ...
 [False False False ... False  True False]
 [False  True False ... False False False]
 [False False False ... False False False]]
Y_test= [[0. 0. 0. ... 0. 0. 1.]
 [0. 0. 1. ... 0. 0. 0.]
 [0. 1. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 1. 0.]
 [0. 1. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

Expected output

```
Y_new= [[False False False ... False False  True]
 [ True False False ... False False False]
 [ True False False ... False False False]
 ...
 [False False False ... False False False]
 [ True False False ... False False False]
 [False False False ... False False False]]
Y= [[0 0 0 ... 0 0 1]
 [1 0 0 ... 0 0 0]
 [1 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
```

```
  [1 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]]
 Y_new_test= [[False False False ... False False  True]
  [False False  True ... False False False]
  [False  True False ... False False False]
  ...
  [False False False ... False  True False]
  [False  True False ... False False False]
  [False False False ... False False False]]
 Y_test= [[0 0 0 ... 0 0 1]
  [0 0 1 ... 0 0 0]
  [0 1 0 ... 0 0 0]
  ...
  [0 0 0 ... 0 1 0]
  [0 1 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]]
```

Exercise 2 - The Softmax Function

(Adapted from Andrew Ng's exercise in Coursera, deeplearning.ai)

Implement a softmax function using numpy. Softmax is a normalizing function used when the algorithm needs to classify two or more classes.

Instructions:

- for $x \in \mathbb{R}^{1\times n}$

$$\text{softmax}(x) = \text{softmax}\left(\begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}\right) = \begin{bmatrix} \frac{e^{x_1}}{\sum_j e^{x_j}} & \frac{e^{x_2}}{\sum_j e^{x_j}} & \cdots & \frac{e^{x_n}}{\sum_j e^{x_j}} \end{bmatrix}$$

- for a matrix $x \in \mathbb{R}^{m\times n}$

$$\text{softmax}(x) = \text{softmax}\begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix} = \begin{bmatrix} \frac{e^{x_{11}}}{\sum_j e^{x_{1j}}} & \frac{e^{x_{12}}}{\sum_j e^{x_{1j}}} & \frac{e^{x_{13}}}{\sum_j e^{x_{1j}}} & \cdots & \frac{e^{x_{1n}}}{\sum_j e^{x_{1j}}} \\ \frac{e^{x_{21}}}{\sum_j e^{x_{2j}}} & \frac{e^{x_{22}}}{\sum_j e^{x_{2j}}} & \frac{e^{x_{23}}}{\sum_j e^{x_{2j}}} & \cdots & \frac{e^{x_{2n}}}{\sum_j e^{x_{2j}}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{e^{x_{m1}}}{\sum_j e^{x_{mj}}} & \frac{e^{x_{m2}}}{\sum_j e^{x_{mj}}} & \frac{e^{x_{m3}}}{\sum_j e^{x_{mj}}} & \cdots & \frac{e^{x_{mn}}}{\sum_j e^{x_{mj}}} \end{bmatrix}$$

$$= \begin{pmatrix} \text{softmax(first row of x)} \\ \text{softmax(second row of x)} \\ \dots \\ \text{softmax(last row of x)} \end{pmatrix}$$

[ ] ↳ *2 cells hidden*

Exercise 3 - Compute one semi-vectorized iteration for softmax

Perform one semi-vectorized iteration of gradient descent for the softmax classification.

```
# First do this for only one training example (the first one, i=0).
# Print out everything you compute, e.g. print("z", z), print("a", a), etc.

J = 0
w = np.zeros((n_x,C))
b = np.zeros((1,C))

dw = np.zeros((n_x,C));
db = np.zeros((1,C));

i = 0;


# Replace None objects by your code

x = X[i:i+1, : ]  #x is [1,784]
y = Y[i:i+1, :]
print("y", y)
```

```
# take the dot product using np.dot()
dot_prod = np.dot(x[i], w)
# add matrices
z = dot_prod + b
print("z", z)

# calling softmax function will add 1e-15 to it
a = softmax(z)
print("a", a)
J = -np.sum(np.log(a) * y)
print("J", J)

dz = a - y[i]
print("dz", dz)

# give new shape to x and dz matrices with n_x and C constants
# newX reshaped with 784 because thats the size of matrix
# newDz reshaped with 10 because thats the size of the matrix
newX = x[i].reshape(n_x, 1)
newDz = dz.reshape(1, C)
# take their dot product
dw = np.dot(newX, newDz)/m
print("dw", dw)
db = a - y[i]
print("db", db)
     y [[0 0 0 0 0 0 0 0 0 1]]
     z [[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
     a [[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]]
     J 2.302585092994046
     dz [[ 0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1 -0.9]]
     dw [[0. 0. 0. ... 0. 0. 0.]
      [0. 0. 0. ... 0. 0. 0.]
      [0. 0. 0. ... 0. 0. 0.]
      ...
      [0. 0. 0. ... 0. 0. 0.]
      [0. 0. 0. ... 0. 0. 0.]
      [0. 0. 0. ... 0. 0. 0.]]
     db [[ 0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1 -0.9]]
```

Expected output

```
 y [[0 0 0 0 0 0 0 0 0 1]]
 z [[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
 a [[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]]
 J 2.3025850929940357
 dz [[ 0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1 -0.9]]
 dw [[0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  ...
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]]
 db [[ 0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1 -0.9]]
```

```
#one iteration, semi-vectorized

J = 0
w = np.zeros((n_x,C))
b = np.zeros((1,C))

dw = np.zeros((n_x,C));
db = np.zeros((1,C));

alpha = 0.000001

start_time = time.time()

for i in range(m):
    # Put your code here
    # find dot product and add matrices, both matrices obey dot product rules (dimensions)
    # add column matrice b
    z = np.dot(X[i], w) + b
```

```
        # get softmax from z
        a = softmax(z)
        J -= np.sum(np.log(a) * y)

        # set gradient with respect to linear output
        dz = a - Y[i]
        # set gradients with respect to w and b
        dw += np.dot(X[i].reshape(n_x,1),dz.reshape(1,C))
        db += a - Y[i]

    J /= m; dw /= m; db /= m

    w -= alpha*dw
    b -= alpha*db

    print("J", J)
    print("Time needed: ", time.time() - start_time)
```

```
J 2.3025850929954172
Time needed:  3.8441343307495117
```

Expected output

```
J =  2.302585092995416
Time needed:  5.705857038497925
```

Of course, your running time will be different.

Exercise 4 - Compute one fully-vectorized iteration for softmax

Perform one fully-vectorized iteration of gradient descent for the softmax classification.

```
#one iteration, fully vectorized, no for loop

J = 0
w = np.zeros((n_x,C))
b = np.zeros((1,C))

alpha = 0.000001

start_time = time.time()


#Replace the None objects and alpha*0 by your code.

# Convention:
# Use capital letter when the variable is for the whole dataset of m train examples.

# X is (55000,784), Y is (55000,10), w is (784,10), b is (1,10)
#   (55000,784) x (784,10)
Z = np.dot(X, w) + b             # Z is  (55000, 10)
A = softmax(Z)            # A is  (55000, 10)
J = np.sum(-np.sum(np.log(A) * Y, axis=1))/m

dz = A - Y               # dZ is (55000, 10)
# take transpose of X (55000,784) -> (784,55000)
# dot prod with dz (784,55000)x(55000,10), then divide by 60000
dw = np.dot(X.T, dz)/m           #dw is (784, 10)
db = np.sum(A - Y)/m

w -= alpha*0
b -= alpha*0

print("J = ", J)
print("Time needed: ", time.time() - start_time)
```

```
J =  2.3025850929940463
Time needed:  0.5206742286682129
```

Expected output

```
J =  2.3025850929940366
Time needed:  0.6775269508361816
```

We observe that the time of the fully vectorized version is almost one order of magnitude smaller.

Exercise 5 - Compute several fully-vectorized iterations for softmax

Perform 100 fully-vectorized iterations of gradient descent for the softmax classification. Start with doing 10 iterations first, check the accuracy you achieve, then try for 100 iterations. Print out the cost after each iteration.

```
J = 0
w = np.zeros((n_x,C))
b = np.zeros((1,C))

alpha = 0.000001

#Replace the None objects and alpha*0 by your code.

# Convention:
# Use capital letter when the variable is for the whole dataset of m train examples.

for iter in range(10):
    # X is (55000,784), Y is (55000,10), w is (784,10), b is (1,10)
    Z = np.dot(X, w) + b             # Z is  (55000, 10)
    A = softmax(Z)              # A is  (55000, 10)
    J = np.sum(-np.sum(np.log(A) * Y, axis=1))/m
    print(iter, J)

    dz = A - Y               # dZ is (55000, 10)


    dw = np.dot(X.T, dz)/m                #dw is (784, 10)
    db = np.sum(A - Y)/m

    # make sure to modify w, b every iteration
    w -= alpha*dw
    b -= alpha*db
```

```
 0 2.3025850929940463
 1 2.1462055484811278
 2 2.0308194022674777
 3 1.9311838264975678
 4 1.8436117489915664
 5 1.7662246129412005
 6 1.6975659901982603
 7 1.636412814864482
 8 1.5817258425589016
 9 1.5326222061991983
```

Expected output

```
0 2.3025850929940366
1 2.146205548481119
2 2.030819399467471
3 1.9311838198434277
4 1.8436117384575559
5 1.7662245987459277
6 1.6975659726233228
7 1.636412794191129
8 1.581725819043199
9 1.5326221800639164
```

Exercise 6 - Compute the accuracy

Compute the accuracy of softmax classification on the train and test datasets.

Use np.argmax(A, 1) and np.argmax(Y, 1) to find the predicted and real class for each example. They return an array of numbers each, e.g. [7 3 9 ..., 8 0 8] and [7 3 4 ..., 5 6 8]. Compare them using ==. You will get an array of booleans, e.g. [True True False ..., False False True]. Sum up the latter using np.sum(). True values will be considered 1, False values will be considered 0, so the sum will tell us how many True values we got. Then divide by Y.shape[0] and multiply by 100 to get the accuracy in percentage.

```
# Replace None by your code

def accuracy(A, Y):
  # find max value element index of A and Y
    A_index = np.argmax(A, 1)
    Y_index = np.argmax(Y, 1)
    return (np.sum(A_index == Y_index)/Y.shape[0])*100

Z = X @ w + b
A = softmax(Z)

Z_test = X_test @ w + b
A_test = softmax(Z_test)

print("Accuracy on the train set is ", accuracy(A,Y))
print("Accuracy on the test set is ", accuracy(A_test,Y_test))
```

```
Accuracy on the train set is  66.35166666666666
Accuracy on the test set is  65.38000000000001
```

Expected output

```
Accuracy on the train set is  66.35166666666667
Accuracy on the test set is  65.38
```

Remark. These numbers are for 10 iterations. When you perform 100 iterations, the numbers will be much better.

Exercise 7 - Implement the Softmax classifier using Keras

Implement the Softmax classifier using Keras. This is similar to examples shown in class. Use categorical crossentropy for loss function. Use stochastic optimization (rmsprop or adam) with 5 epochs. Produce the accuracy on the test set in the end.

```
network = keras.Sequential([keras.layers.Dense(10, activation='softmax')])

# use stochastic optimization
network.compile(
    optimizer="adam",
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

# with 5 epocs
network.fit(X, Y, epochs=5)

loss, accuracy = network.evaluate(X_test, Y_test)
print("Produced test accuracy", accuracy)
```

```
Epoch 1/5
1875/1875 [==============================] - 4s 2ms/step - loss: 16.6016 - accuracy: 0.7454
Epoch 2/5
1875/1875 [==============================] - 3s 2ms/step - loss: 11.9266 - accuracy: 0.7905
Epoch 3/5
1875/1875 [==============================] - 4s 2ms/step - loss: 10.9183 - accuracy: 0.7959
Epoch 4/5
1875/1875 [==============================] - 3s 2ms/step - loss: 10.9585 - accuracy: 0.8008
Epoch 5/5
1875/1875 [==============================] - 4s 2ms/step - loss: 10.6250 - accuracy: 0.8013
313/313 [==============================] - 1s 1ms/step - loss: 13.3364 - accuracy: 0.8022
Produced test accuracy 0.8022000193595886
```

Exercise 8 - Implement a neural network with one hidden layer using Keras.

Turn the previous exercise into a 1-hidden layer neural network with rectified linear units and 15 hidden nodes. The output layer should continue to be softmax.

The hidden layer should be fully connected to the input layer and to the output layer.

Check the accuracy on the test set. If it is lower than 80%, increase the number of hidden nodes until you reach or exceed this level accuracy.

```
loss, accuracy = 0, 0
hidden_nodes = 15
while(accuracy < 0.80):
  network = keras.Sequential([
      keras.layers.Dense(hidden_nodes, activation='relu', input_shape=(784,)),
      keras.layers.Dense(10, activation='softmax')
  ])


  # use stochastic optimization
  network.compile(
      optimizer="adam",
      loss="categorical_crossentropy",
      metrics=["accuracy"]
  )

  # with 5 epocs
  network.fit(X, Y, epochs=10)

  loss, accuracy = network.evaluate(X_test, Y_test)
  print("Produced test accuracy:", accuracy, "\nHidden nodes:",hidden_nodes)
  hidden_nodes += 30
```

```
Epoch 1/10
1875/1875 [==============================] - 4s 2ms/step - loss: 2.8170 - accuracy: 0.1192
Epoch 2/10
1875/1875 [==============================] - 4s 2ms/step - loss: 2.2172 - accuracy: 0.1372
Epoch 3/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.8437 - accuracy: 0.2098
Epoch 4/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.7070 - accuracy: 0.2653
Epoch 5/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.6662 - accuracy: 0.2769
Epoch 6/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.6384 - accuracy: 0.2794
Epoch 7/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.6251 - accuracy: 0.2800
Epoch 8/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.6168 - accuracy: 0.2811
Epoch 9/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.6160 - accuracy: 0.2791
Epoch 10/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.6186 - accuracy: 0.2752
313/313 [==============================] - 1s 1ms/step - loss: 1.6063 - accuracy: 0.2828
Produced test accuracy: 0.28279998898506165
Nodes: 15
Epoch 1/10
1875/1875 [==============================] - 4s 2ms/step - loss: 2.7965 - accuracy: 0.5263
Epoch 2/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.0675 - accuracy: 0.5846
Epoch 3/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.9452 - accuracy: 0.6170
Epoch 4/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.8221 - accuracy: 0.6708
Epoch 5/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.7000 - accuracy: 0.7235
Epoch 6/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6596 - accuracy: 0.7342
Epoch 7/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6466 - accuracy: 0.7388
Epoch 8/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6376 - accuracy: 0.7428
Epoch 9/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6356 - accuracy: 0.7444
Epoch 10/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6318 - accuracy: 0.7429
313/313 [==============================] - 1s 2ms/step - loss: 0.6694 - accuracy: 0.7280
Produced test accuracy: 0.7279999852180481
Nodes: 45
Epoch 1/10
1875/1875 [==============================] - 5s 2ms/step - loss: 2.3156 - accuracy: 0.6435
Epoch 2/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.7341 - accuracy: 0.7268
Epoch 3/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6335 - accuracy: 0.7593
Epoch 4/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.5714 - accuracy: 0.7950
```

```
Epoch 5/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.5366 - accuracy: 0.8119
Epoch 6/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.5160 - accuracy: 0.8195
```

✓ 2m 4s completed at 8:37 PM