Assignment 4 - Classifying handwritten digits 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 regression.

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.

In [1300]:
```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
print(tf.__version__)

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

```
2.0.0
```

In [1301]:
```
# 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()
```


Ankle boot
T-shirt/top
T-shirt/top
Dress
T-shirt/top
Pullover
Sneaker
Pullover
Sandal
Sandal
T-shirt/top
Ankle boot
Sandal
Sandal
Sneaker
Ankle boot
Trouser
T-shirt/top
Shirt
Coat
Dress
Trouser
Coat
Bag
Coat

In [1302]:
```
# We will reshape (flatten) X arrays so that they become rank 2 array
s.
# 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

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

In [1303]:
```
# Toward this goal, let's check what np.arange(C) produces
np.arange(C)
```

Out[1303]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [1304]:
```
# Let's see again what Y is
Y
```

Out[1304]:
```
array([[9],
       [0],
       [0],
       ...,
       [3],
       [0],
       [5]], dtype=uint8)
```

In [1305]:
```
# What would broadcasting these arrays together would look like?
# Let's check.

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]]
```

In [1306]:
```
# 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 o
f 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 = (a == b)
print("Y_new=", Y_new)
Y = Y_new.astype(int)
print("Y=",Y)

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

Y_new_test = a == b
print("Y_new_test=", Y_new_test)
Y_test = Y_new_test.astype(int)
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 False]
 [ True False False ... False False False]
 [False False False ... False False False]
 ...
 [False False False ... False False False]
 [False False False ... False False False]
 [False False False ... False  True False]]
Y= [[0 0 0 ... 0 0 0]
 [1 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 1 0]]
Y_new_test= [[False False False ...  True False False]
 [False False  True ... False False False]
 [False  True False ... False False False]
 ...
 [False False False ... False False False]
 [False False False ... False False False]
 [False False False ... False False False]]
Y_test= [[0 0 0 ... 1 0 0]
 [0 0 1 ... 0 0 0]
 [0 1 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 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 $x \in \mathbb{R}^{m\times n}$

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

In [1307]:
```
def softmax(x):
    # This normalization is done to avoid number overflow.
    # See: https://stats.stackexchange.com/questions/304758/softmax-overflow
    x_normalized = x - np.max(x,axis=1,keepdims=True)

    # Create an array x_exp by applying np.exp() element-wise to x_normalized.
    # Put your code here (one line)
    x_exp = np.exp(x_normalized)
    # Create an array x_sum that contains the sum of each row of x_exp.
    # Use np.sum(..., axis = 1, keepdims = True).
    # Put your code here (one line)
    x_sum = np.sum(x_exp, axis = 1, keepdims = True)
    # Compute softmax(x) by dividing x_exp by x_sum.
    # It should automatically use numpy broadcasting.
    # Return this array.
    # Add a very small constant, 1e-15, to each matrix entry (using broadcasting)
    # in order to avoid any pure-zero entries.
    # This number doesn't make predictions much off, and it solves the log(0) issue
    # in the cross-entropy function.
    # Replace None with your code
    return (x_exp / x_sum) + 0.000000000000001


# Let's test
x = np.array([
    [1, 2, 3, 1, 2],
    [9, 5, 1, 0 ,0]])
print("softmax(x) = " + str(softmax(x)))
```

```
softmax(x) = [[6.74508059e-02 1.83350300e-01 4.98397788e-01 6.74508059e-02
  1.83350300e-01]
 [9.81452586e-01 1.79759312e-02 3.29240664e-04 1.21120871e-04
  1.21120871e-04]]
```

Expected output

```
softmax(x) = [[  6.74508059e-02   1.83350300e-01   4.98397788e-01   6.7450
8059e-02
    1.83350300e-01]
 [  9.81452586e-01   1.79759312e-02   3.29240664e-04   1.21120871e-04
    1.21120871e-04]]
```

Exercise 3 - Compute one semi-vectorized iteration for softmax

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

In [1308]:
```
# 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[0:1, : ]  #x is [1,784]
y = Y[0:1,:]
print("y", y)

z = x @ w + b
print("z", z)
a = softmax(1/((1+np.exp(-z))))
print("a", a)
J = -(y*np.log(a)+(1-y)*np.log(1-a))
print("J", J[0][9])

dz = a-y
print("dz", dz)
dw = (x * z.reshape(10,1)).T
print("dw", dw)
db = dz
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 2.3025850929940357
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]]
```

Expected output

```
y [[0 0 0 0 0 1 0 0 0 0]]
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 2.3025850929940455
dz [[ 0.1  0.1  0.1  0.1  0.1 -0.9  0.1  0.1  0.1  0.1]]
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 -0.9  0.1  0.1  0.1  0.1]]
```

In [1309]:
```
#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
    x = np.reshape(X[i], (1,n_x))
    y = Y[i,0]

    z = x @ w + b
    a = softmax(1/(1+np.exp(-z)))
    J += np.sum(-(y*np.log(a)+(1-y)*np.log(1-a)))

    dz = softmax(a-y)
    dw += (x*z.reshape(10,1)).T
    db += dz

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

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

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

```
J 3.25082973391109
Time needed:  6.454204797744751
```

Expected output

```
J =  2.3025850929954172
Time needed:  3.750260591506958
```

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.

In [1310]:
```
#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)
Z = softmax(X @ w + b)            # Z is  (55000, 10)
A = softmax(1/(1+np.exp(softmax(Z))))            # A is  (55000, 10)
J = -(1/m)*np.sum(Y*np.log(softmax(A))+(1-Y)*np.log(1-softmax(A)))

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


dw = (1/m) * X.T @ dZ             #dw is (784, 10)
db = (1/m) * np.sum(dZ)

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

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

```
J =  3.25082973391448
Time needed:  0.6512594223022461
```

Expected output

```
J =  2.3025850929940366
Time needed:  0.46091175079345703
```

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.

In [1311]:
```
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 = softmax(X @ w + b)                # Z is  (55000, 10)
    A = 1/(1 + np.exp(-Z))              # A is  (55000, 10)
    J = -(1/m)*np.sum(Y*np.log(softmax(A))+(1-Y)*np.log(1-softmax(A))
)
    print(iter, J)

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


    dw = (1/m) * X.T @ dZ             #dw is (784, 10)
    db = (1/m) * np.sum(dZ)

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

```
0 3.25082973391448
1 3.245873491996776
2 3.241068085163646
3 3.236339387347906
4 3.231622645480618
5 3.226893708741467
6 3.2221693268051514
7 3.217490166643499
8 3.2129023675047232
9 3.2084457410381377
```

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.

In [1312]:
```
# Replace None by your code

def accuracy(A, Y):
    return np.sum(np.argmax(A,1) == np.argmax(Y,1))/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  62.13999999999999
Accuracy on the test set is  61.309999999999995
```

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.

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.