Neural_Network_Classifier

April 10, 2022

1 Section 1 : Importing Dependencies

[137]:
```
import jax                                                                   #␣
 ↪Another version of numpy for Computation on GPU / TPU
import jax.numpy as jnp
from jax import random
from activations import *

import matplotlib.pyplot as plt                                              #␣
 ↪Library for Visualization


import tensorflow.keras.datasets.mnist as mnist                              #␣
 ↪Tensorflow library for deep learning computation
from tensorflow.keras.utils import to_categorical

from sklearn.metrics import classification_report, confusion_matrix          #␣
 ↪Sklearn for classifications metrics
```

2 Section 2 : Dataset

- We will use Mnist Dataset for classifying Hand-Written digits

[2]:
```
training_dataset , test_dataset = mnist.load_data()
# Extracting inputs and labels
X_train, y_train = training_dataset
X_test,  y_test  = test_dataset
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
```

[3]:
```
# Exploring dataset

m_train = X_train.shape[0]
num_px = X_train.shape[1:]
```

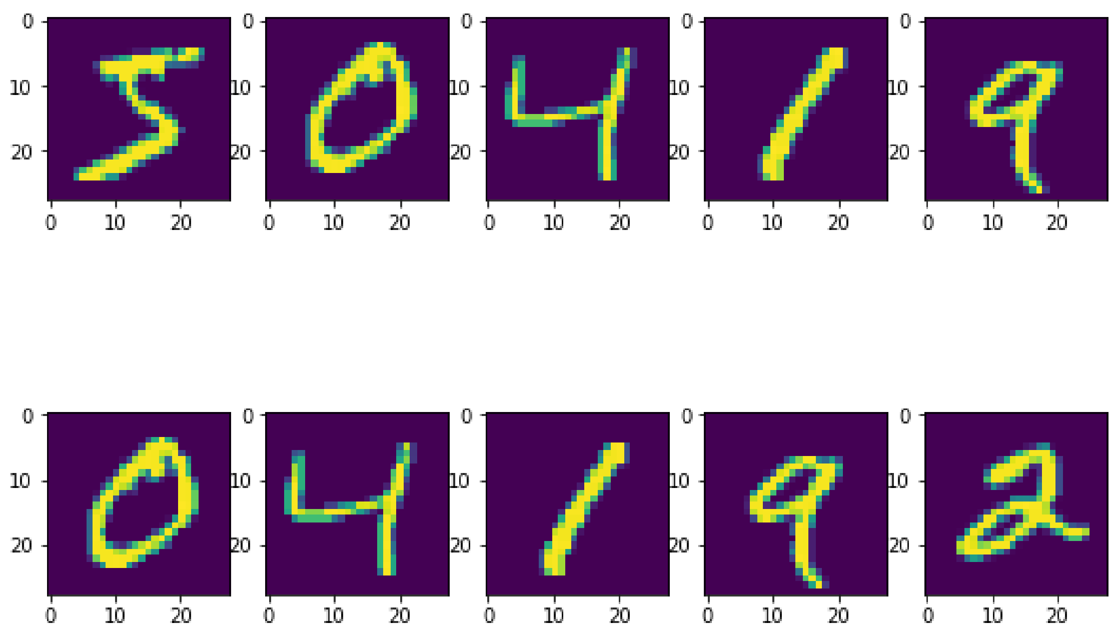
```
m_test = X_test.shape[0]

print ("Number of training examples: " + str(m_train))
print ("Number of testing examples: " + str(m_test))
print ("Each image is of size: ",num_px)
print ("Training X inputs  shape: " + str(X_train.shape))
print ("Training Y outputs shape: " + str(y_train.shape))
print ("Testing  X inputs  shape: " + str(X_test.shape))
print ("Testing  Y outputs shape: " + str(y_test.shape))
```

```
Number of training examples: 60000
Number of testing examples: 10000
Each image is of size:  (28, 28)
Training X inputs  shape: (60000, 28, 28)
Training Y outputs shape: (60000,)
Testing  X inputs  shape: (10000, 28, 28)
Testing  Y outputs shape: (10000,)
```

Below code will show sample Images of Handwritten digits

[4]:
```
igure, ax = plt.subplots(2,5,figsize=(10,7))
for i in range(2):
    for j in range(5):
        ax[i,j].imshow(X_train[i+j])
```



As usual, we need reshape and standardize the images before feeding them to the network.

```
[5]: # Reshape the training and test examples
X_train_flatten = X_train.reshape(X_train.shape[0], -1).T   # The "-1" makes␣
 ↪reshape flatten the remaining dimensions
X_test_flatten  = X_test.reshape(X_test.shape[0], -1).T

# Standardize data to have feature values between 0 and 1.
train_x = X_train_flatten/255
test_x = X_test_flatten/255

# One - hot encoding of real values for multiclass classification
y_train = to_categorical(y_train).T
y_test  = to_categorical(y_test).T

print ("train_x's shape: " + str(train_x.shape))
print ("test_x's shape: " + str(test_x.shape))
print ("train_y's shape: " + str(y_train.shape))
print ("test_y's shape: " + str(y_test.shape))
```

```
train_x's shape: (784, 60000)
test_x's shape: (784, 10000)
train_y's shape: (10, 60000)
test_y's shape: (10, 10000)
```

3 Section 3 : Deep Neural Network

- 1 input layer, 2 hidden layer, 3 hidden layer
- initializing defaul parameters
- use activation functions like relu, sigmoid, softmax
- we use cross-entropy loss function for loss
- Uses Jax pytree concept, for updating parameters

3.1 3.1 Implementing functions for Neural network

```
[122]: def initialize_parameters(layers_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of each layer␣
 ↪in our network

    Returns:
    parameters -- python list containing parameters "W0", "b0", ..., "WL", "bL":
                    Wl -- weight matrix of shape (layer_dims[l],␣
 ↪layer_dims[l-1])
                    bl -- bias vector of shape (layer_dims[l], 1)
    """
    key = random.PRNGKey(32)
    parameters = []
```

```
    for i in range(1,len(layers_dims)):
        W = random.normal(key, (layers_dims[i], layers_dims[i-1]))/jnp.
↪sqrt(layers_dims[i-1])
        b = jnp.zeros((layers_dims[i],1))
        parameters.append([W,b])

    return parameters
```

3.1.1 3.1.1 Cross Entropy Loss Function

[124]:
```
def cost_function(A,Y):
    """
    Arguments:
    A -- probability vector corresponding to label predictions, shape (10,␣
↪number of examples)
    Y -- true "label" vector , shape (10, number of examples)

    Returns:
    cost -- cross-entropy cost
    """
    cost = -jnp.mean(Y * jnp.log(A + 1e-8))
    return cost
```

3.1.2 3.1.2 Forward Propagation

[125]:
```
def linear_activation_forward(X,parameters,layers_size):
    """
    Arguments:
    X -- inputs , shape(features, num-of-example)
    parameters -- weights  + baises matrix:
    layers_size -- number of layers in network

    Returns:
    A -- the output of the neural network
    cache -- a python dictionary containing "linear_cache" and␣
↪"activation_cache";
             stored for computing the backward pass efficiently
    """
    caches = {}
    A = X
    for i in range(layers_size - 1):
        Z = parameters[i][0].dot(A)  + parameters[i][1]
        A = relu(Z)
        caches["A" + str(i + 1)] = A
        caches["W" + str(i + 1)] = parameters[i][0]
        caches["Z" + str(i + 1)] = Z
```

```python
    Z = parameters[layers_size-1][0].dot(A) + parameters[layers_size-1][1]
    A = softmax(Z)
    caches["A" + str(layers_size)] = A
    caches["W" + str(layers_size)] = parameters[layers_size-1][0]
    caches["Z" + str(layers_size)] = Z

    return A, caches
```

3.1.3 3.1.3 Backward Propagation

[126]:
```python
def linear_activation_backward(X, Y, caches, layers_size):
    """
    Arguments:
    X -- inputs , shape(features, num-of-example)
    Y -- true outputs, shape(classes, num_of_example)
    cache -- dictionary of values (linear_cache, activation_cache) we store for␣
 ↪computing backward propagation efficiently
    layers_size -- number of layers in network

    Returns:
    grads -- gradient of all weights and biases in network
    """
    grads = []
    m = X.shape[1]
    caches["A0"] = X
    A = caches["A" + str(layers_size)]

    dZ = A - Y
    dW = dZ.dot(caches["A" + str(layers_size - 1)].T)/m
    db = jnp.sum(dZ, axis=1, keepdims=True)/m
    dAprev = jnp.dot(caches["W" + str(layers_size)].T, dZ)

    grads.insert(0,[dW,db])

    for i in range(layers_size-1 , 0 , -1):
        # dZ = dAprev * sigmoid_derivative(caches["Z" + str(i)])
        dZ = relu_backward(dAprev,caches["Z" + str(i)])
        dW = dZ.dot(caches["A" + str(i - 1)].T)/m
        db = jnp.sum(dZ, axis=1, keepdims=True)/m
        if i > 1 :
            dAprev = jnp.dot(caches["W" + str(i)].T, dZ)

        grads.insert(0,[dW,db])

    return grads
```

3.1.4 3.1.4 Updating Parameters

[127]:
```
def update_parameters(parameters, grads, learning_rate, layers_size):
    """
    Uses Jax pytree Concept for updating gradient of parameters

    Arguments:
    parameters -- python dictionary containing parameters
    grads -- python dictionary containing gradients, output of␣
↪linear_activation_backward

    Returns:
    parameters -- python dictionary containing updated parameters
                  parameters["W" + str(l)] = ...
                  parameters["b" + str(l)] = ...
    """
    upd = lambda x,y : x - learning_rate * y

    parameters = jax.tree_map(upd,parameters,grads)

    return parameters
```

3.2 3.2 Model

- Model for training and fitting a given dataset

[128]:
```
def accuracy_measures(x, y, parameters):
    A,caches = linear_activation_forward(x, parameters, len(parameters)-1)

    y_hat = jnp.argmax(A,axis=0)
    y     = jnp.argmax(y,axis=0)

    accuracy = (y_hat == y).mean()

    return accuracy*100
```

[129]:
```
# Model which bind together a neural network and helps to learn
def model(X, Y, layers_dims, learning_rate=0.075, iterations=1000):
    layers_size = len(layers_dims) - 1
    costs = []
    parameters = initialize_parameters(layers_dims)

    for i in range(iterations):

        # forward Propagation
        A, caches = linear_activation_forward(X,parameters,layers_size)
```

```
        # cost
        cost = cost_function(A,Y)

        # backward propagation
        grads = linear_activation_backward(X, Y, caches, layers_size)

        parameters = update_parameters(parameters, grads, learning_rate,␣
↪layers_size)

        # accuracy = accuracy_measures(X,Y,parameters)
        if i%50 == 0:
            print("Iteration {} : ".format(i))
            print("Cost : {}".format(cost))

        if i%50 == 0:
            costs.append(cost)

    return parameters,costs
```

[129]:

```
```

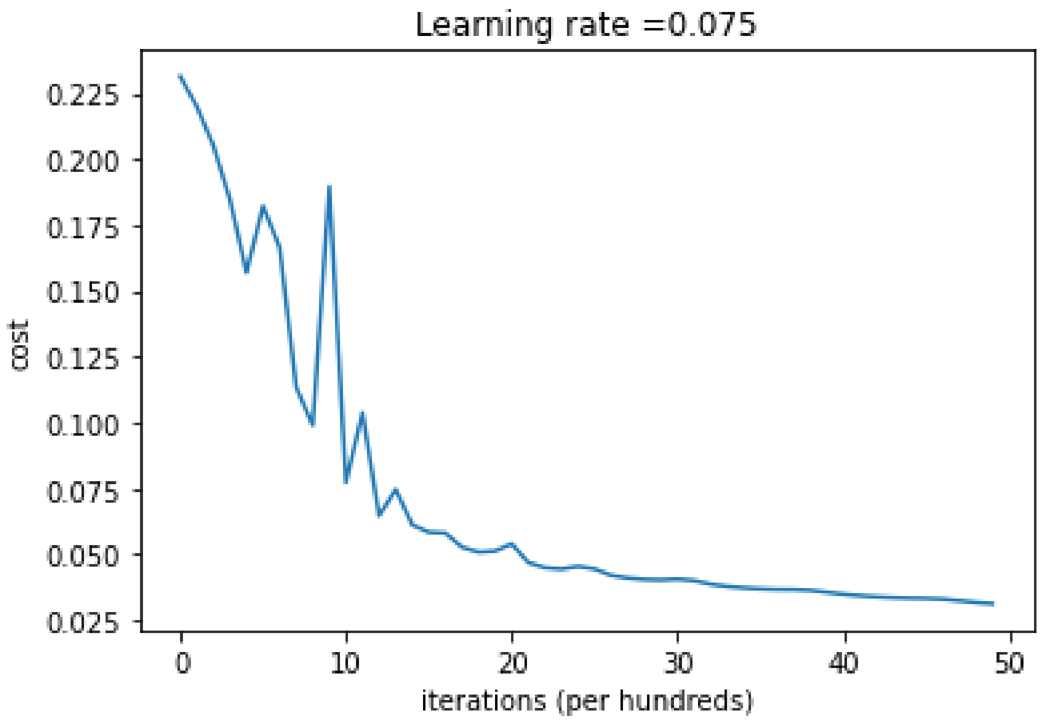
4 Section 4 : Validation and Classifications metrics

[132]:

```
layers_dim = [train_x.shape[0],24,12,24,10]
parameters,costs = model(train_x, y_train, layers_dim)
```

```
Iteration 0 :
Cost : 0.23136411607265472
Iteration 50 :
Cost : 0.18217206001281738
Iteration 100 :
Cost : 0.07739816606044769
Iteration 150 :
Cost : 0.05848364531993866
Iteration 200 :
Cost : 0.054091256111860275
Iteration 250 :
Cost : 0.044572994112968445
Iteration 300 :
Cost : 0.04068170487880707
Iteration 350 :
Cost : 0.036976125091314316
Iteration 400 :
Cost : 0.034946754574775696
Iteration 450 :
Cost : 0.03336076810956001
```

[133]:
```
# Plotting loss and no. of iteration graph
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
plt.title("Learning rate =" + str(0.075))
plt.show()
```



4.1 4.1 : Classification report and Confusion metrics

[160]:
```
y_pred,cache = linear_activation_forward(test_x,parameters,len(parameters))
```

[162]:
```
y_pred = jnp.argmax(y_pred,axis=0)
y_true = jnp.argmax(y_test,axis=0)
```

[171]:
```
print(classification_report(y_true.T, y_pred.T))
```

```
              precision    recall  f1-score   support

           0       0.94      0.97      0.95       980
           1       0.95      0.98      0.96      1135
           2       0.92      0.89      0.91      1032
           3       0.89      0.92      0.90      1010
           4       0.91      0.91      0.91       982
```

```
           5       0.91      0.80      0.85       892
           6       0.93      0.93      0.93       958
           7       0.92      0.92      0.92      1028
           8       0.88      0.89      0.88       974
           9       0.87      0.89      0.88      1009

    accuracy                           0.91     10000
   macro avg       0.91      0.91      0.91     10000
weighted avg       0.91      0.91      0.91     10000
```

[163]:

```
confusion_matrix(y_true.T,y_pred.T)
```

[163]:

```
array([[ 952,    0,    2,    2,    0,    9,    7,    5,    3,    0],
       [   0, 1107,    8,    4,    0,    1,    1,    1,   12,    1],
       [  14,   12,  923,   15,    9,    0,   13,   11,   32,    3],
       [   1,    2,   18,  932,    0,   17,    0,   17,   19,    4],
       [   2,    2,    4,    0,  897,    0,   23,    1,    7,   46],
       [  17,    2,   12,   65,   18,  715,    9,    5,   28,   21],
       [  17,    3,   13,    0,   10,   12,  894,    2,    5,    2],
       [   3,   19,   13,    2,    3,    1,    0,  945,    3,   39],
       [   6,   10,   12,   18,   11,   22,    7,    7,  866,   15],
       [   4,    4,    0,   13,   34,   13,    7,   31,   10,  893]])
```

[ ]: