EE 260 HW#1 - Linear MNIST Classifier

Submitted By: Pranshu Shrivastava

1. **Write the code for downloading and formatting the data.**

```
!curl -O http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
!curl -O http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
!curl -O http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
!curl -O http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
!gunzip t*-ubyte.gz
import numpy as np
import time
from mlxtend.data import loadlocal_mnist
X, y = loadlocal_mnist(
       images_path='/content/train-images-idx3-ubyte',
       labels_path='/content/train-labels-idx1-ubyte',)
X_test, y_test = loadlocal_mnist(
       images_path='/content/t10k-images-idx3-ubyte',
       labels_path='/content/t10k-labels-idx1-ubyte',)
X_train = np.asarray(X).astype(np.float32)
y_train = np.asarray(y).astype(np.float32)
X_test = np.asarray(X_test).astype(np.float32)
y_test = np.asarray(y_test).astype(np.float32)
X = X_train / 255.0
X_T = X_test / 255.0
```

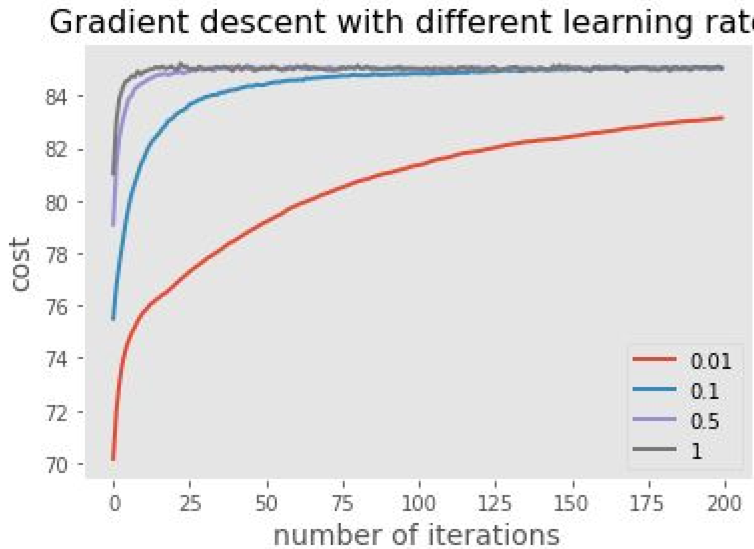
2. **Write the code for minibatch SGD implementation for your linear MNIST classifier.**

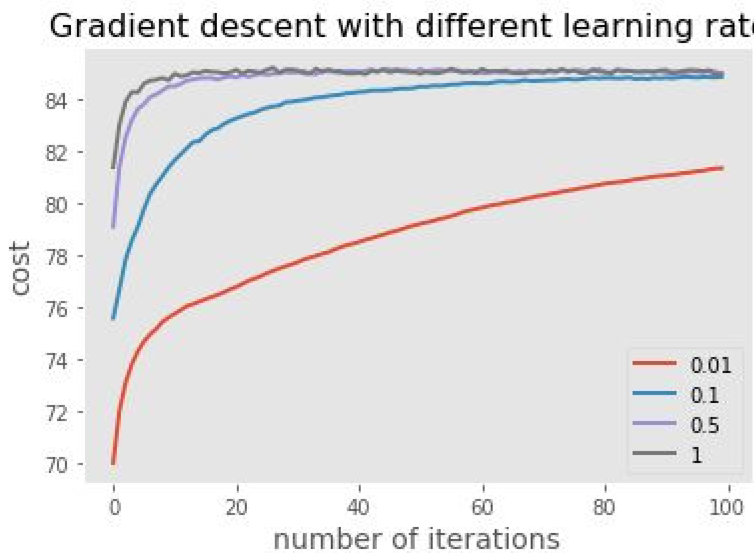
```
def
minibatch_gradient_descent(X,y,theta,learning_rate=0.01,iterations=10,batc
h_size =20):

   m = len(y)
   n = len(X)
   cost_history = np.zeros(iterations)
   accArr = np.zeros(iterations)
   n_batches = int(m/batch_size)
```

```
    for it in range(iterations):
        cost =0.0
        indices = np.random.permutation(m)
        X = X[indices]
        y = y[indices]
        for i in range(0,m,batch_size):
            X_i = X[i:i+batch_size]
            y_i = y[i:i+batch_size]

            prediction = np.dot(X_i,theta)
            theta = theta -(1/m)*learning_rate*( X_i.T.dot((prediction -
y_i)))
            cost += cal_cost(theta,X_i,y_i)

        cost_history[it]  = cost
        accuracy = acc(n, X, theta, y)
        print("iteration: {0} with accuracy: {1}".format(it, accuracy))
        accArr[it] = accuracy

    return theta, cost_history, accArr
```

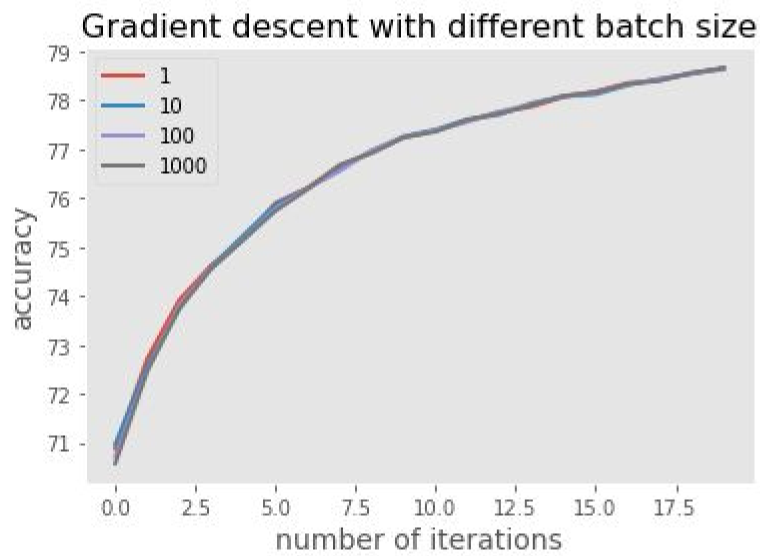
3. **The role of batch size: Run your code with batch sizes B = 1, 10, 100, 1000. For each batch size,**

    - **Determine a good choice of learning rate**
      By multiple iteration, we can determine 0.1 as an optimal learning rate.

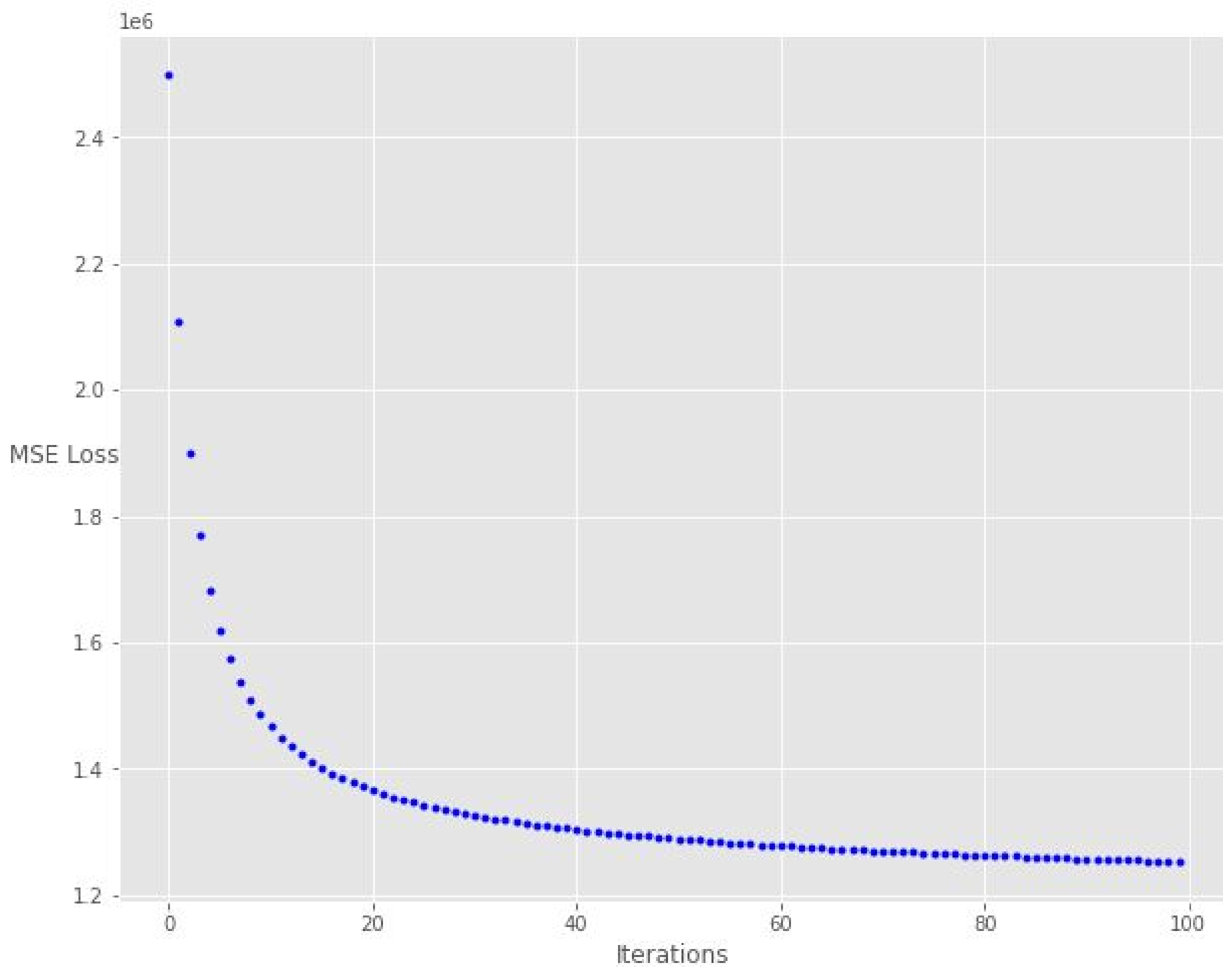




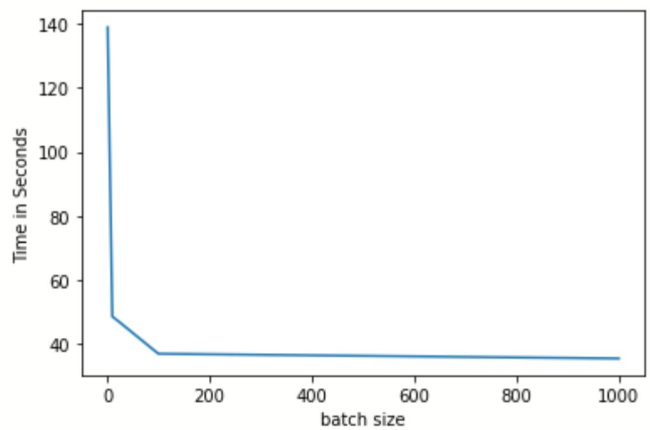
- **Pick ITR sufficiently large to ensure the (approximate) convergence of the training loss**
  Batch size = 100 gives optimal accuracy.



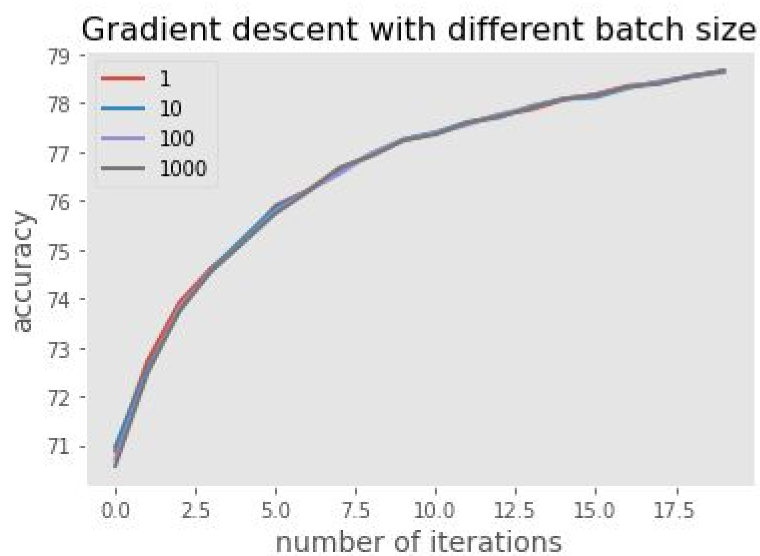
- **Plot the progress of training loss (y-axis) as a function of the iteration counter t (x-axis)**

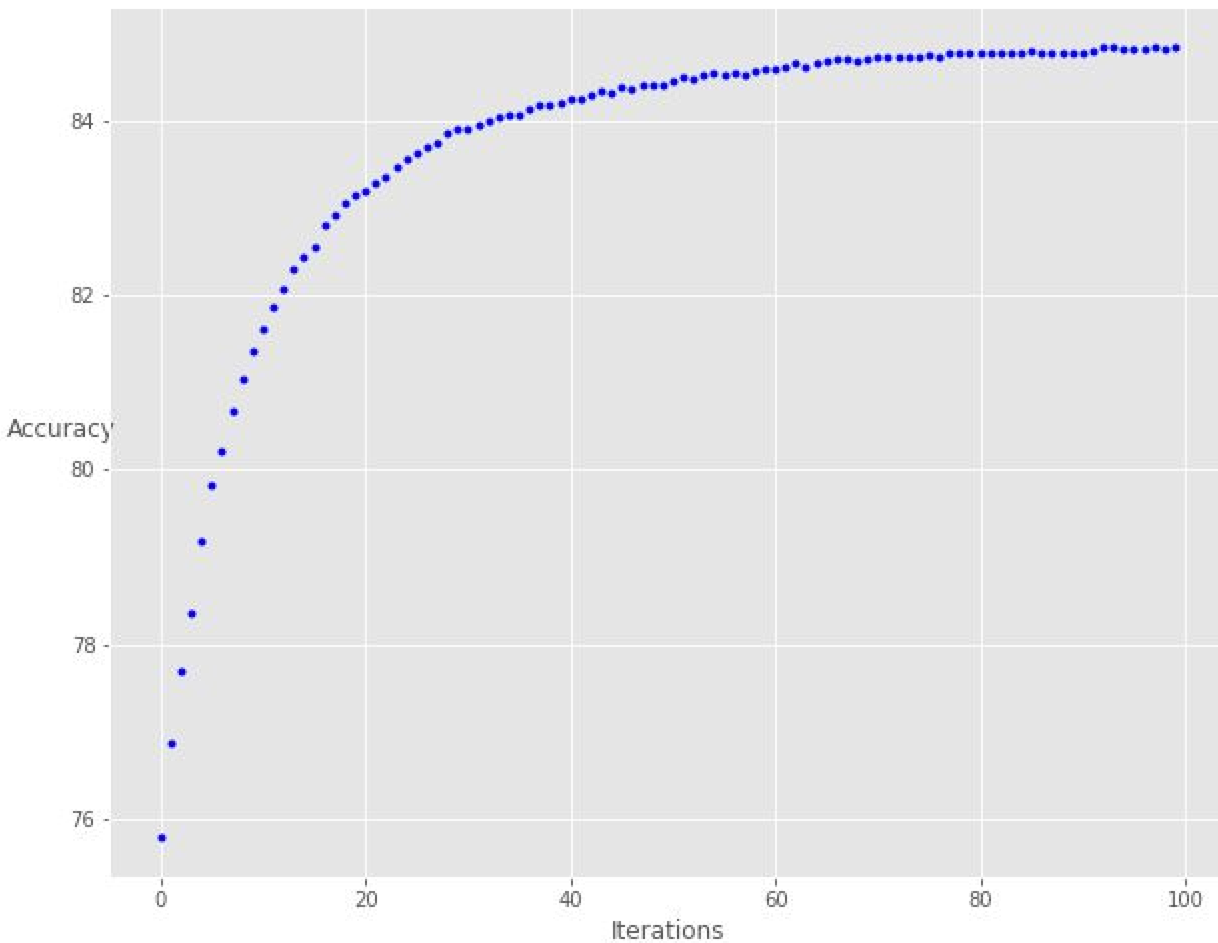


- **Report how long the training takes (in seconds).**
  For 100 batch size, the time taken is approximately 40 seconds.



- **Plot the progress of the test accuracy (y-axis) as a function of the iteration counter t (x-axis)**



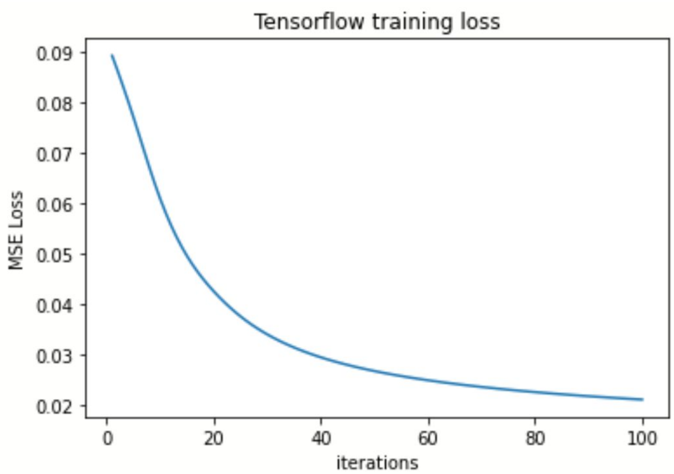


4. **Comment on the role of batch size.**
    a. Larger batch size improves training speed and reduces accuracy.
    b. Smaller batch sizes results in higher accuracy and larger training time.
    c. Accuracy falls with the increase of batch size.
5. **The role of training dataset size:**
   Accuracy increases with dataset size. Below is the data for 50 iterations, learning rate = 0.01 and batch size = 100

```
Accuracy with dataset size = 10000 82.0
Accuracy with dataset size = 1000 78.0
Accuracy with dataset size = 500 75.6
Accuracy with dataset size = 100 75.12
```

6. **Implementation using Pytorch**

```
from keras.models import Sequential
from keras.layers import Dense, Activation from keras import optimizers
sgd = optimizers.SGD(lr=0.5)
output_dim = nb_classes = 10
model = Sequential()
model.add(Dense(output_dim, input_dim=input_dim,
activation='softmax',kernel_initializer='random_uniform',bias_initializer= 'zeros'))
batch_size = 100 nb_epoch = 100
model.compile(optimizer='sgd', loss='mean_squared_error', metrics=['accuracy'])
history = model.fit(X_train, Y_train, batch_size=batch_size,
nb_epoch=nb_epoch,verbose=0)
score = model.evaluate(X_test, Y_test, verbose=0) print('Test accuracy:', score[1])
```



92% accuracy was obtained using tensorflow as opposed to the hand-written algorithm which gave 86% accuracy.