# Objectives

* Implement Stochastic Gradient Descent
* Stochastic Gradient Descent in Sickit-Learn
* Implement Mini-batch Gradient Descent

# Lab Tool(s)

<https://www.kaggle.com/>

# Lab Deliverables

Submit a pdf document to Blackboard containing your solution to the lab assessment at the end of this document.

# Recall: Gradient Descent

Gradient Descent is a very generic optimization algorithm capable of finding optimal

solutions to a wide range of problems. The general idea of Gradient Descent is to

tweak parameters iteratively in order to minimize a cost function.

Suppose you are lost in the mountains in a dense fog; you can only feel the slope of

the ground below your feet. A good strategy to get to the bottom of the valley quickly

is to go downhill in the direction of the steepest slope. This is exactly what Gradient

Descent does: it measures the local gradient of the error function with regards to the

parameter vector θ, and it goes in the direction of descending gradient. Once the gradient

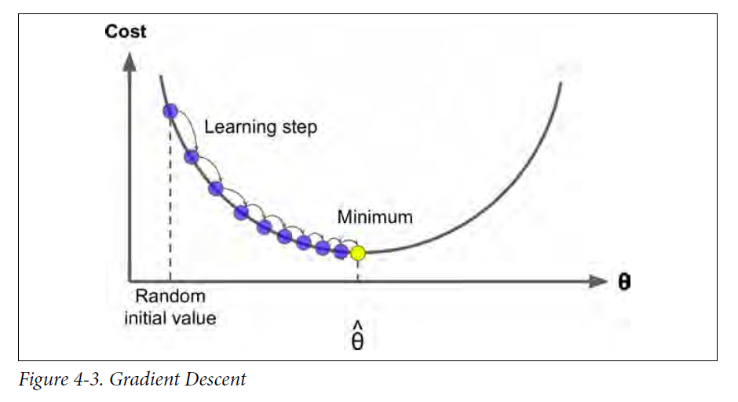
is zero, you have reached a minimum!

Concretely, you start by filling θ with random values (this is called random initialization),

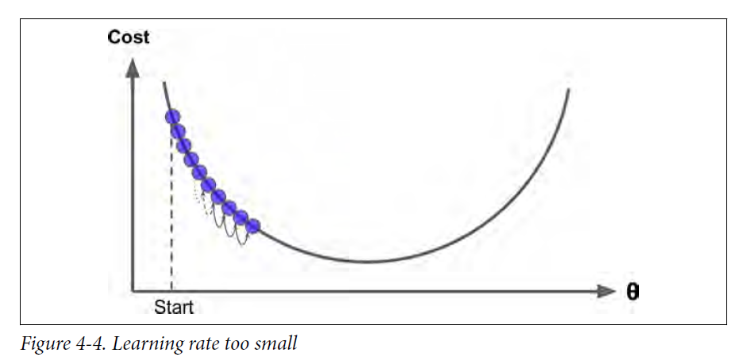
and then you improve it gradually, taking one baby step at a time, each step

attempting to decrease the cost function (e.g., the MSE), until the algorithm converges

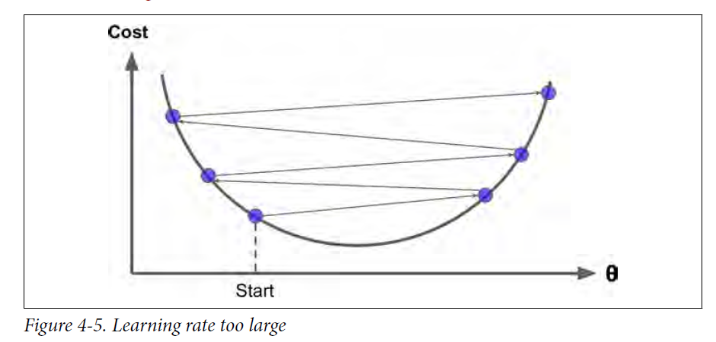
to a minimum.



An important parameter in Gradient Descent is the size of the steps, determined by the learning rate hyperparameter. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time.



On the other hand, if the learning rate is too high, you might jump across the valley and end up on the other side, possibly even higher up than you were before. This might make the algorithm diverge, with larger and larger values, failing to find a good solution.



Finally, not all cost functions look like nice regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum very difficult.

**Note:** When using Gradient Descent, you should ensure that all features have a **similar scale** (e.g., using Scikit-Learn’s StandardScaler class), or else it will take much longer to converge.

# Implementing ‘Stochastic’ Gradient Descent

The main problem with **Batch Gradient Descent** is the fact that it uses the whole training set to compute the gradients at every step, which makes it **very slow when the training set is large**.

At the opposite extreme, **Stochastic Gradient Descent just picks a random instance** in the training set at every step and computes the gradients based only on that single instance. Obviously this makes the algorithm much faster since it has very little data to manipulate at every iteration. It also **makes it possible to train on huge training sets**, since only one instance needs to be in memory at each iteration.

But, due to **its stochastic (i.e., random) nature**, this algorithm is much **less regular than Batch Gradient Descent**: instead of gently decreasing until it reaches the minimum, the **cost function will bounce up** and **down**, decreasing only on average.

On the positive side, Stochastic Gradient Descent has **a better chance of finding the global minimum** than Batch Gradient Descent does.

Therefore randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum.

Therefore randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum. One solution to this dilemma is to **gradually reduce the learning rate**. The steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum. This process is called simulated anneal.

**Step1:** Generate data

**import** **numpy** **as** **np**

m = 100

X = 2 \* np.random.rand(m, 1)

y = 4 + 3 \* X + np.random.randn(m, 1)

**Step1:** Implement Stochastic Gradient Descent

n\_epochs = 50

t0, t1 = 5, 50 *# learning schedule hyperparameters*

**def** learning\_schedule(t):

**return** t0 / (t + t1)

theta = np.random.randn(2,1) *# random initialization*

**for** epoch **in** range(n\_epochs):

**for** i **in** range(m):

random\_index = np.random.randint(m)

xi = X\_b[random\_index:random\_index+1]

yi = y[random\_index:random\_index+1]

gradients = 2 \* xi.T.dot(xi.dot(theta) - yi)

eta = learning\_schedule(epoch \* m + i)

theta = theta - eta \* gradients

Note that since instances are picked randomly, some instances may be picked several times per epoch while others may not be picked at all.

**Step2:** Check theta and compare it to the original (remember, *y* = 4 + 3*x*1 + Gaussian noise)

theta

The first value represents theta\_0 and the second is for theta\_1. Remember that the true ones are 4 and 3 respectively. How did the learned ones compare to the true ones?

# ‘Stochastic’ Gradient Descent in Sickit-learn

To perform Linear Regression using SGD with Scikit-Learn, you can use the SGDRe gressor class, which defaults to optimizing the squared error cost function. The following code runs 50 epochs, starting with a learning rate of 0.1 (eta0=0.1), using the default learning schedule (different from the preceding one), and it does not use any regularization (penalty=None; more details on this shortly).

**from** **sklearn.linear\_model** **import** SGDRegressor

sgd\_reg = SGDRegressor(max\_iter=50, tol=-np.infty, penalty=**None**, eta0=0.1, random\_state=42)

sgd\_reg.fit(X, y.ravel())

To check the learned parameters;

sgd\_reg.intercept\_, sgd\_reg.coef\_

# Implementing the mini-batch Gradient Descent

At each step, instead of computing the gradients based on the full training set (as in Batch GD) or based on just one instance (as in Stochastic GD), Mini-batch GD computes the gradients on small random sets of instances called minibatches.

# Lab Assessment

**Step1:** Create a new notebook and name it “CCAI312\_YOURSTUDENTID\_Lab6”

**Step2:** Generate the following data

X = 2 \* np.random.rand(100, 1)

y = 4 + 3 \* X + np.random.randn(100, 1)

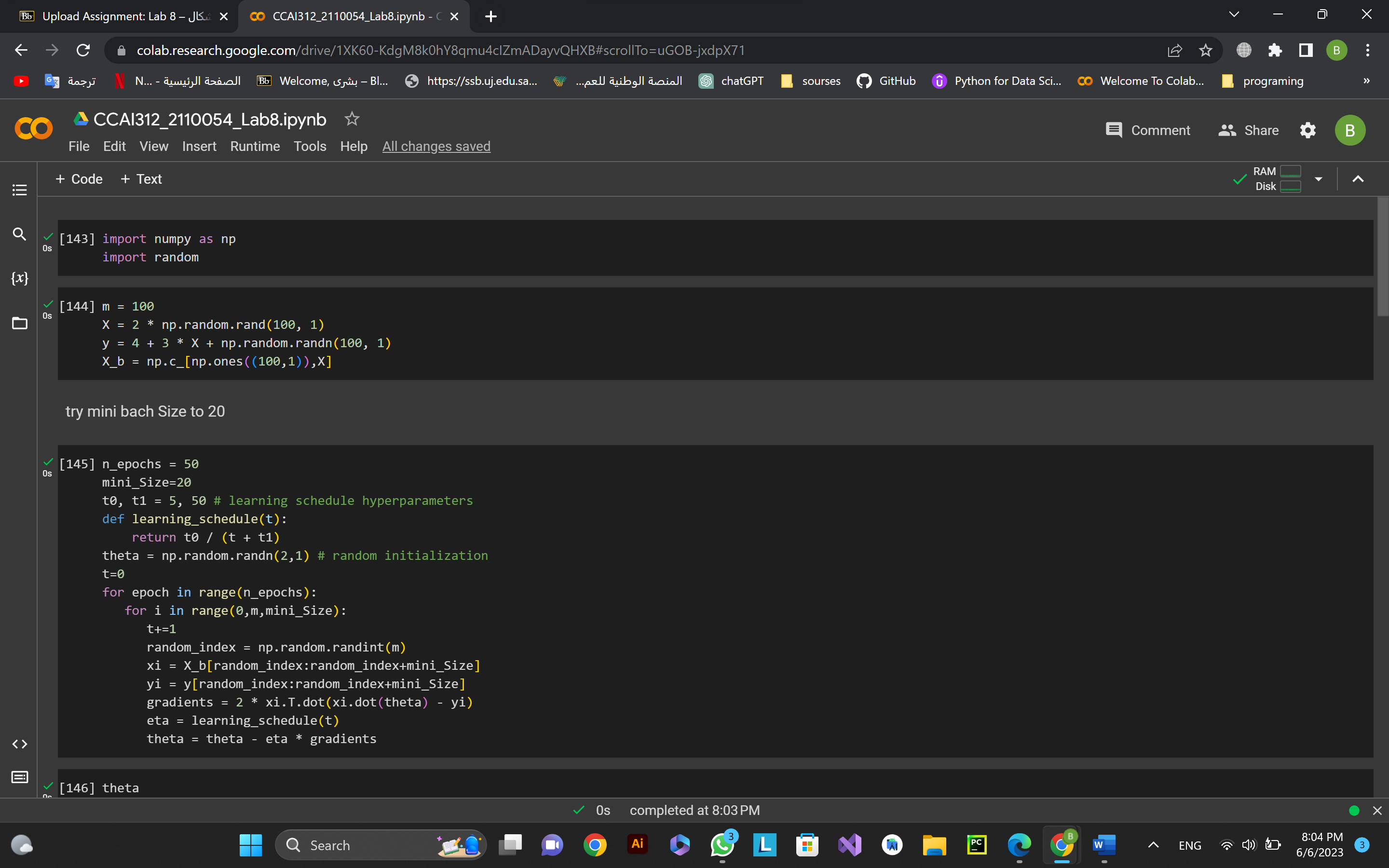
**Step3:** Implement the mini-batch gradient descent algorithm,

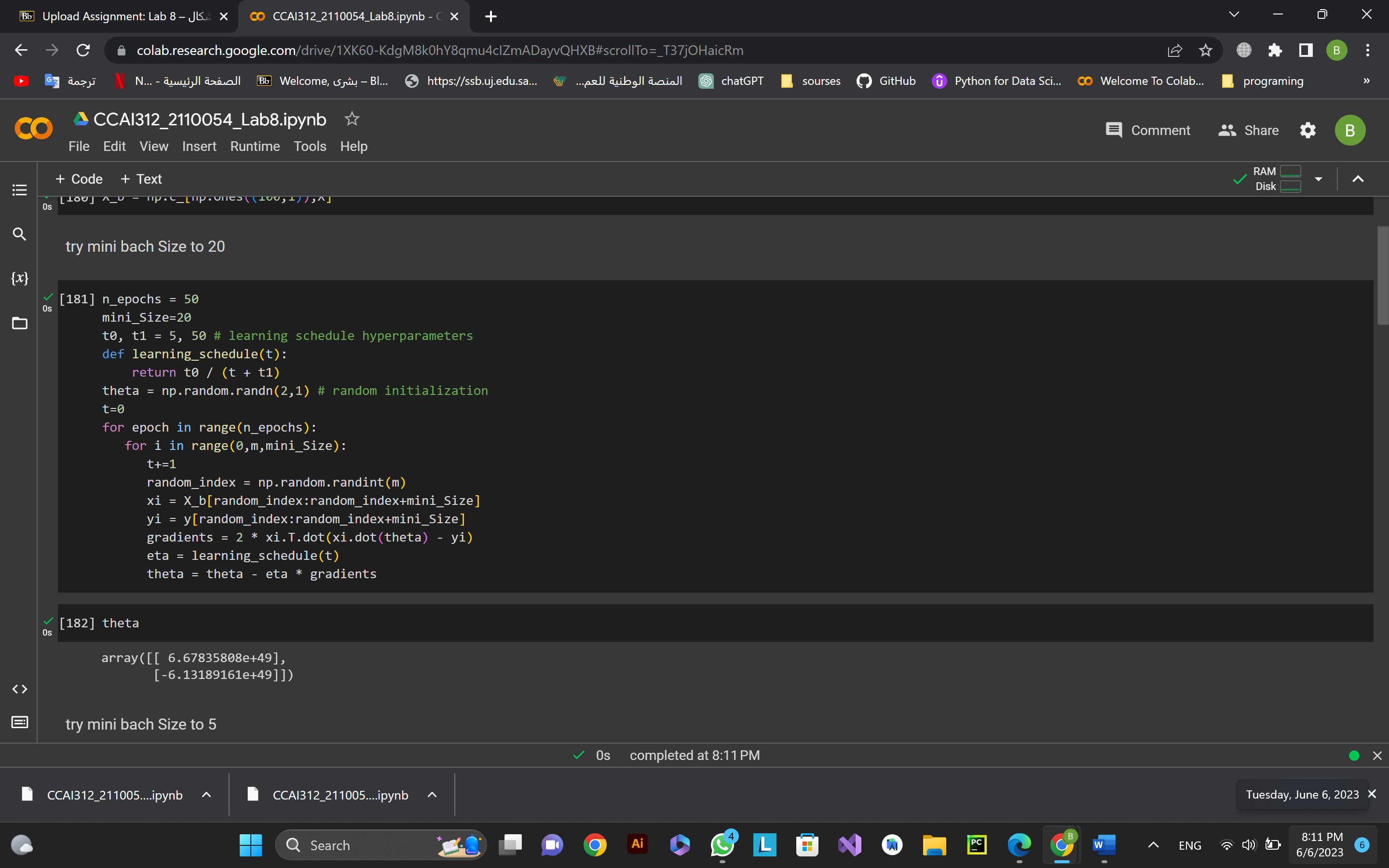
**Step4:** Report the learned parameters when the mini-batch size is 20. Repeat this step with 2 different mini-batch sizes and report the learned parameters.

**Step6:** Submit a pdf document containing **your code and answers** to Blackboard, name the file as: CCAI312\_YOURSTUDENTID\_Lab6.pdf.

References:

Hands-on machine learning with sickit-learn and tensor flow, Chapter 4





A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated