Implementing a Gradient Descent Algorithm.

In this assignment, you'll be learning about one of the hottest topics in all of science and technology today - machine learning. Computers tend to think like we do; through association and recall. Humans don't ever have to ask, how do I train myself to think? We are born with the inate ability to make associations, take in data, record observations, self-correct, etc... But even though we are born with the ability, we are not initially very good at it. It takes years of data gathering to get good at making decisions when presented with new data, hence, we all touch a hot stove at least once. Therefore, it shouldn't be a surprise that if a computer is going to learn to think, it has to do a fair amount of training first. The process governing the "training" requires an amount of data that may be unfathomable to the human memory and is often represented mathematically as a series of complicated linear algebra techniques. That is where you come in!

In this notebook, I have programmed a gradient descent algorithm. As you work through the notebook you will learn a little bit about machine learning. Unfortunately, my code was a linear algebra black box. I had packages do a lot for me - dot products, transposes, sums, etc. But you know how to do all of those things by now! So your assignment will be to work your way through this notebook, evaluating cells as you go. Hopefully, it teaches you about how we as scientists can teach computers to think. But you will come across linear algebra as you go and currently, this notebook has no way of doing many of the operations.

You should:

1.) Write a matrix library containing functions called transpose, sum, dot, and multiply. Each of these should take in a matrix and perform the operations listed in their name. A skeleton code has been set up for you in matrix_library_student.py.

For example, matrix_library_student.transpose(A) should return a matrix that is the transpose of A. matrix_library_student.dot(A, B) should take two matrix objects A and B, and return the dot product of the two.

2.) Validate each of the functions you've written. This can be done simply by comparing the output of your function with the output of the numpy function that is meant to do the same operation. A skeleton code has been set up for you in lin_alg_test_student.py.

3.) Run the gradient descent code and try to come up with some real world examples that you think gradient descent would be applicable for based on what you've learned.

Note that if a particular cell lacks comments, it is because an operation is done that is not important for you to understand in this context. Or, if you see some numpy commands floating around - THEY ARE THERE ON PURPOSE, LEAVE THEM BE!

In [ ]:
```
# import the necessary packages
import matplotlib.pyplot as plt
from sklearn.datasets.samples_generator import make_blobs
import numpy as np
import time
import matrix_library_solutions as ml
import lin_alg_test_solutions as lat
from IPython import display

epochs = 200
alpha = 0.01
```

We can now test your matrix library before we proceed too far.

In [ ]:
```
lat.test_sum_matrix()
lat.test_sum_list()
lat.test_transpose()
lat.test_dot()
lat.test_multiply()
```

Generate some data

Machine learning is a great tool for classifying data. We are going to work in a "feature" space, meaning the axes are features of the data themselves. Imagine we are trying to train a computer to tell the difference between cars and trucks. How do you tell the difference when you see them? The question almost seems obvious - we intuitively know a car is a car and a truck is a truck. But if we expect a computer to know the difference, we better teach it how to differentiate. Two features that would be immensely useful would be the height of the vehicle and the width of the vehicle. Trucks are on average taller and wider than cars. If we plotted their features on a 2D feature space - the populations would be clustered in different areas. These features are represented as matrices.

Below I've generated some data on a feature space. You can think up your own physical example for what these features might represent but for now we will just consider them nameless-features.

In [ ]:
```
# generate a 2-class classification problem with 250 data points,
# where each data point is a 2D feature vector
(X, y) = make_blobs(n_samples=250, n_features=2, centers=2,cluster_std=1.05
plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=plt.cm.Spectral)

## Insert a column of 1's - the reason will be apparent later.
X = np.c_[np.ones((X.shape[0])), X]
```

We want to give the computer the tools it needs to find the line dividing these two classes. Once that is done, we could in theory give it a point anywhere on the feature space and it should be able to tell us if an object with those features belongs in the purple population or the red population.

It is clear at this stage that a line dividing these populations will likely be linear. It will take the form:

$$h(x^{(i)}) = \theta_0 + \theta_i x^{(i)}$$

Notice that this term looks like a neat linear equation but both θ_i and $x^{(i)}$ are matrices or vectors containing many points each. The θ's are referred to as weights. These are the values we are trying to find with the help of gradient descent. They will determine where our best fit line is to divide the populations. $h(x^{(i)})$ represents a category - in this case, a 0 (for red) or a 1 for (purple).

When we talk about the optimal weights, what we are really saying is the weights that minimize the error function given by:

$$J(\theta_i) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right)^2$$

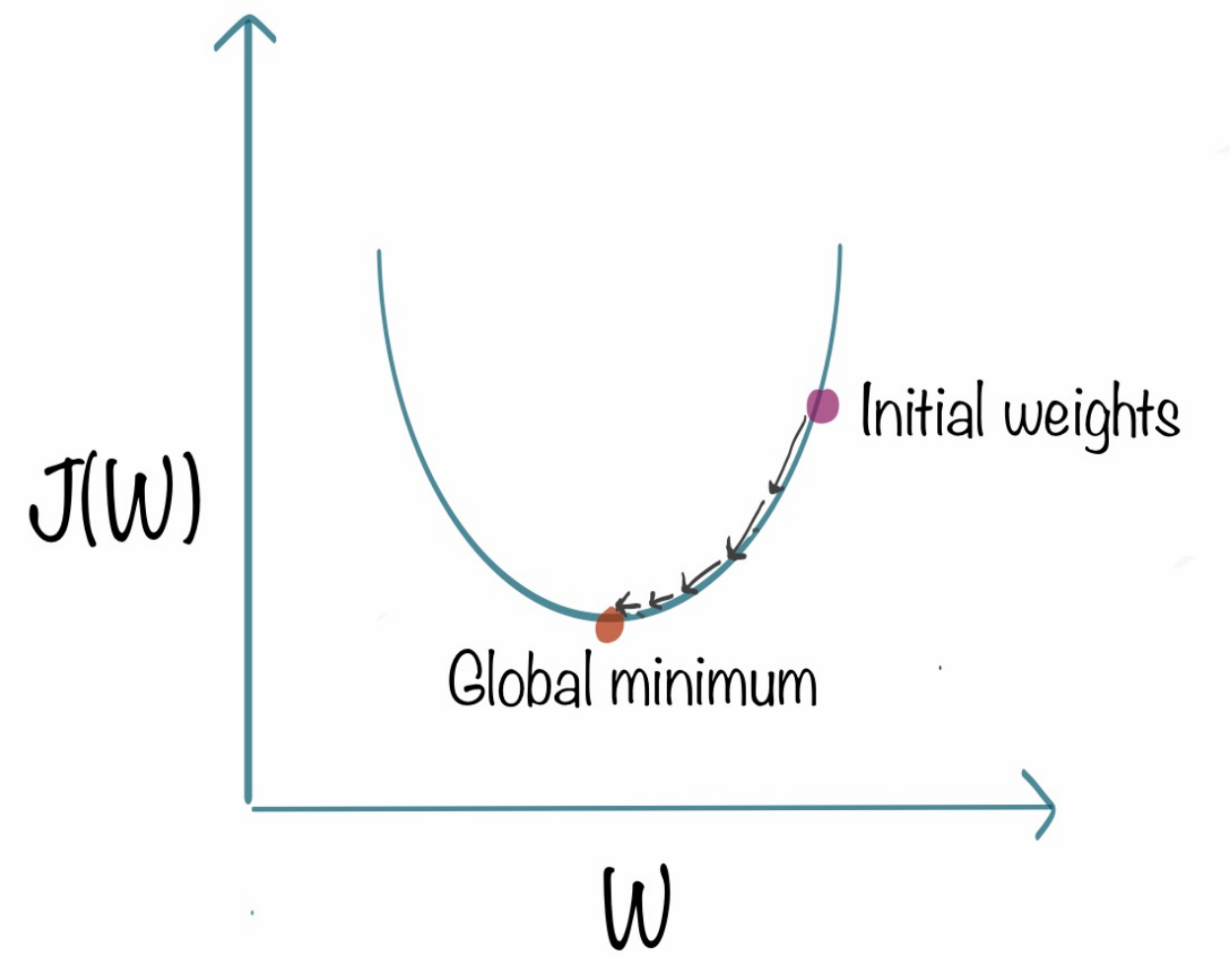
where $y^{(i)}$ represents the true classification and is a matrix. This is not the only error or "loss" function that is used in machine learning. T. Dietterich excellently summarizes their use and functionality in his review article. Minimizing this should remind you of your calculus class: we just need to take the derivative of the function with respect to the θ_i.

$$\frac{\partial}{\partial \theta_i} J(\theta_i) = \frac{1}{m} \sum_{i=1}^{m} \left(h_{\theta_i} \left(x^{(i)} \right) - y^{(i)} \right) \cdot x^{(i)}$$

So what the network is doing is asking the question: for a given weight vector, $[\theta_0, \theta_1]$, is $h(x^{(i)}) - y^{(i)}$ close to 0? If not, we need to try a different weight vector, $[\theta_0, \theta_1]$, until $h(x^{(i)}) - y^{(i)}$ gets closer to 0. Mathematically, this is expressed in the form of an update equation that we update as we descend into our global minimum.

$$\theta_i := \theta_i - \alpha \frac{\partial}{\partial \theta_i} J(\theta_i)$$

α is the learning rate and is not important for this assignment. Pictorially, you could imagine the weights of best fit are at the global minimum of the error function, $J(\theta_i)$. This update equation allows us to progress down the gradient and will converge when we reach the bottom. In practice, this can get very complicated as the datasets become less pristine. But we won't worry about that for now.



There is one more important point that I skipped over - how do we compute h_{θ_i}? We use what is called an activation function. You can find many in the literature (Agostinelli et. al.) but in essence, we want a function that given any real input, we can get an output that is essentially binary. For this work, we use the sigmoid function defined below. Activation functions help us determine what class an input belongs to.

At this point, you should have an appreciation for the library you built. It will be immensely powerful for handling these calculations and it is only basic linear algebra! The unavoidable fact is, real data is large and messy and linear algebra gives you all the tools you need to work with the largest datasets you can imagine. Your library is only being used on ~100s of datapoints in this assignment, but it could easily handle ~100,000 or larger.

In [ ]:
```
def sigmoid_activation(x):
    # compute and return the sigmoid activation value for a
    # given input value
    return 1.0 / (1 + np.exp(-x))
print(sigmoid_activation(5))
print(sigmoid_activation(-5))
```

If all of your functions were built properly, this should execute just fine! The plot will display how the decision boundary changes as we compute gradient descent.

In [ ]:

```
# initialize our weight matrix such it has the same number of
# columns as our input features
theta = np.random.uniform(size=(X.shape[1],))

# initialize a list to store the loss value for each epoch
lossHistory = []

print(np.shape(X), np.shape(theta))

# loop over the desired number of epochs
for epoch in np.arange(0, epochs):
    # take the dot product between our features `X` and the
    # weight matrix `W`, then pass this value through the
    # sigmoid activation function, thereby giving us our
    # predictions on the dataset
    preds = sigmoid_activation(ml.multiply(X,theta))

    # now that we have our predictions, we need to determine
    # our `error`, which is the difference between our predictions
    # and the true values
    error = preds - y

    # given our `error`, we can compute the total loss value as
    # the sum of squared loss -- ideally, our loss should
    # decrease as we continue training
    loss = ml.list_sum(error ** 2)
    lossHistory.append(loss)
    #print(epoch + 1, loss)

    # the gradient update is therefore the dot product between
    # the transpose of `X` and our error, scaled by the total
    # number of data points in `X`
    gradient = ml.multiply(ml.transpose(X),error) / X.shape[0]

    # in the update stage, all we need to do is nudge our weight
    # matrix in the opposite direction of the gradient (hence the
    # term "gradient descent" by taking a small step towards a
    # set of "more optimal" parameters
    theta += -alpha * gradient

    # compute the line of best fit by setting the sigmoid function
    # to 0 and solving for X2 in terms of X1
    Y = (-theta[0] - (theta[1] * X)) / theta[2]

    # plot the original data along with our line of best fit
    if epoch % 20:
        plt.plot(X, Y, "r-",linewidth=0.3)
        plt.scatter(X[:, 1], X[:, 2], marker=".", c=y)
        plt.xlim(-2.5,15)
        plt.ylim(-3,15)
        display.display(plt.gcf())
        display.clear_output(wait=True)
```

Now that you have everything working, you could use the infrastructure on a new data set of your choice. Data is plentiful in today's world. I'm sure you could find a dataset you'd like to classify at

https://data.world/datasets/classification (https://data.world/datasets/classification).

Also, if you're going to do machine learning seriously, it helps to use packages. Sure, we vilified black box packages earlier, but if you truly understand what is going on under the hood, they can be immensely helpful and efficient. For this sort of task, Tensorflow is an incredible tool and I encourage the reader to check it out (Adadi. et. al.).

References

Dietterich, Thomas. (2002). Machine Learning for Sequential Data: A Review. 15-30. 10.1007/3-540-70659-3_2.

Agostinelli, Forest & Hoffman, Matthew & Sadowski, Peter & Baldi, Pierre. (2014). Learning Activation Functions to Improve Deep Neural Networks.

Abadi, Martin & Agarwal, Ashish & Barham, Paul & Brevdo, Eugene & Chen, Zhifeng & Citro, Craig & Corrado, G.s & Davis, Andy & Dean, Jeffrey & Devin, Matthieu & Ghemawat, Sanjay & Goodfellow, Ian & Harp, Andrew & Irving, Geoffrey & Isard, Michael & Jia, Yangqing & Kaiser, Lukasz & Kudlur, Manjunath & Levenberg, Josh & Zheng, Xiaoqiang. (2015). TensorFlow : Large-Scale Machine Learning on Heterogeneous Distributed Systems.

https://data.world/datasets/classification (https://data.world/datasets/classification)

In [ ]: