GradientDescent

August 14, 2018

Gradient descent is another extremely important concept in machine learning that many algorithms rely on in some way or another. The main goal of a parametric machine learning algorithm is to 'learn' model parameters from training data in order to make predictions about future input data. Gradient descent provides a method for fitting these model parameters and relies upon an objective function that quantifies the fit of the model to the data. One common objective function is the maximum likelihood estimate, which we covered in the previous notebook.

To demonstrate how gradient descent works we shall again use simple linear regression as a motivating example. Although the parameters for linear regression can be solved in closed form (via Normal Equations) it provides a nice example of how gradient descent operates. The intuitions we develop in this notebook will apply to many other machine learning algoriths, from logistic regression to neural networks. As a word of warning, this notebook assumes that you are familiar with basic calculus and partial derivatives.

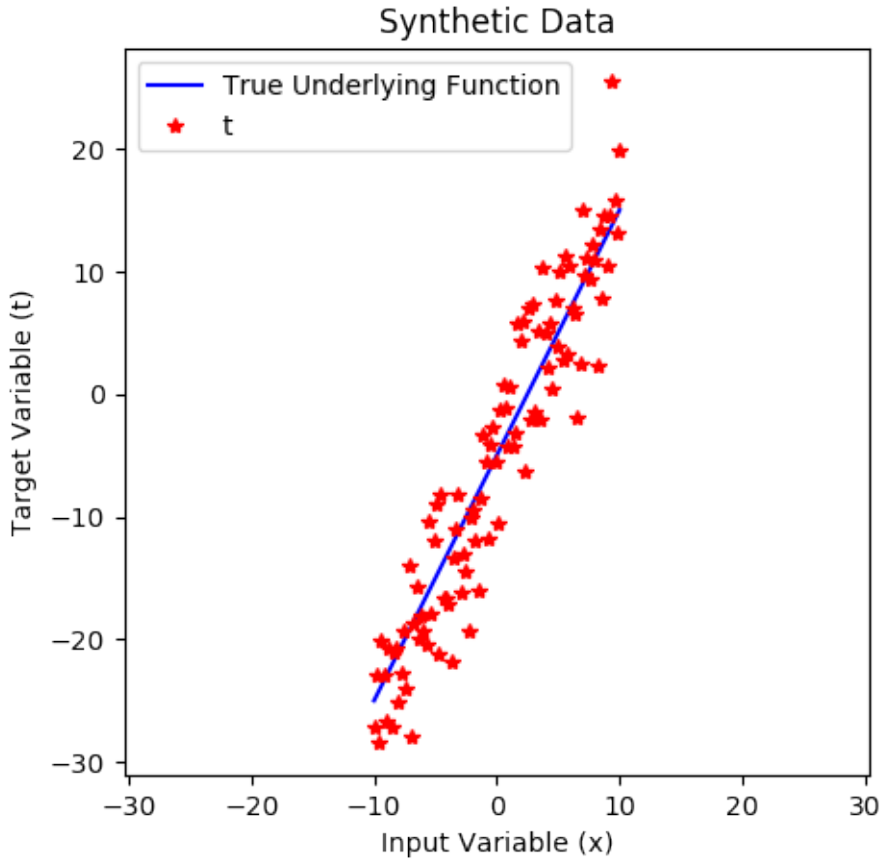
Lets start by generating a synthetic dataset using the following parameter values and gaussian noise distribution:

$$\theta = \begin{bmatrix}\theta_0 & \theta_1\end{bmatrix} = \begin{bmatrix}-5 & 2\end{bmatrix}$$
$$\epsilon_i \sim \mathcal{N}(0,\, 4^2)$$
$$t_i = \theta_0 + \theta_1 x_i + \epsilon_i$$

Where ϵ_i is the noise value, x_i is the input variable and t_i is the output/predicted variable. The following code uses these values to generate a training dataset for us to use:

```
In [42]: %matplotlib inline
         import numpy as np
         import matplotlib.pyplot as plt
         import matplotlib.mlab as mlab
         import matplotlib as mpl
         mpl.rcParams['figure.dpi']= 100

         num_points = 100
         sigma = 4
         theta = np.array([-5, 2])
```

```
X = np.linspace(-10, 10, num_points)
y = theta[0] + (theta[1] * X)
t = np.expand_dims(y + np.random.normal(0, sigma, num_points),
                  axis=-1)

fig = plt.figure(figsize=(5,5))
plt.plot(X, y, 'b', label='True Underlying Function')
plt.plot(X, t, 'r*', label='t')
plt.axis('equal')
plt.legend(loc='upper left')
plt.xlabel('Input Variable (x)')
plt.ylabel('Target Variable (t)')
plt.title('Synthetic Data')
plt.show()
```



So our goal is to fit a linear regression model to this datset using gradient descent. As mentioned previously, gradient descent relies on an objective function, which quantifies how well our parameter values fit the training data. For our purposes we shall use the **negative** log likelihood, which for linear regression is the sum of squared errors (SSE). Remember that minimising the SSE

in this case is equivalent to finding the maximum likelilood solution i.e. the parameter values that make the training data most probable. The negative log-likelihood / SSE is defined as:

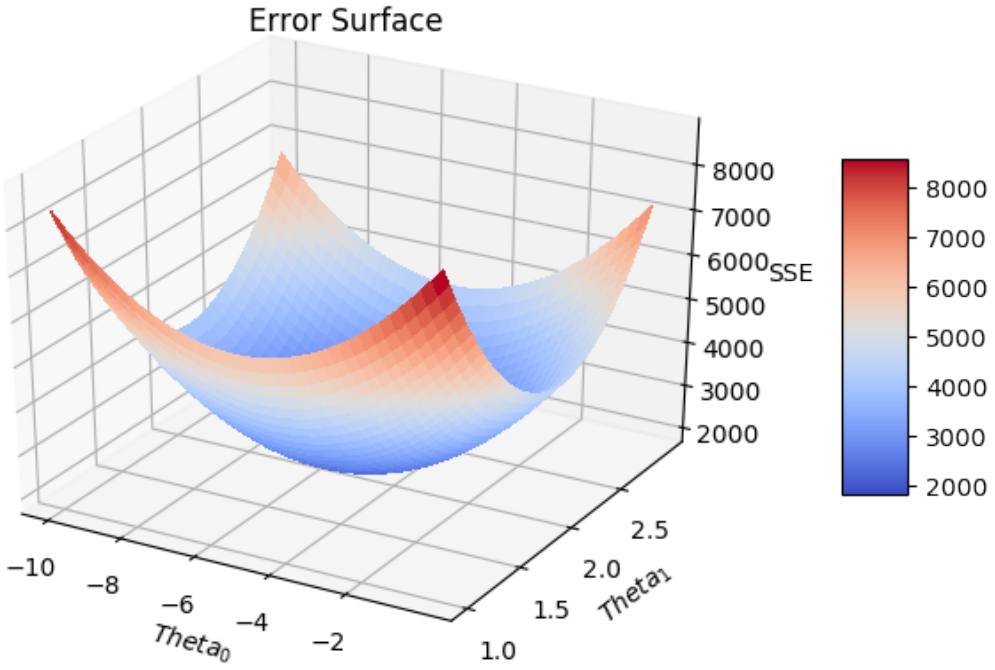
$$J(\theta) = \sum_{i=1}^{N} (t_i - y(x_i, \theta))^2$$

Where $y(x_i, \theta)$ is the output of our linear regression model:

$$y(x_i, \theta) = \theta_0 + \theta_1 x_i$$

Note that if we just used the normal log likelihood then we would want to maximise the value. In machine learning, functions that we want to minimise are often called cost functions because they represent 'costs' that we want to minimise. Lets use our objective function (the SSE) to look at the fit for a range of parameter values. The following code creates a surface plot for a range of parameter values and is sometimes called an 'error surface':

```
In [43]: from mpl_toolkits.mplot3d import Axes3D
         from matplotlib import cm

         num_per_parameter = 40
         theta_0 = np.linspace(-10, 0, num_per_parameter)
         theta_1 = np.linspace(1, 3, num_per_parameter)
         theta_0, theta_1 = np.meshgrid(theta_0, theta_1)

         J = np.zeros((num_per_parameter, num_per_parameter))
         for i in range(num_points):
             J += np.square(t[i] - (theta_0 + (theta_1 * X[i])))

         fig = plt.figure(figsize=(8,5))
         ax = fig.gca(projection='3d')
         surf = ax.plot_surface(theta_0, theta_1, J,
                                cmap=cm.coolwarm,
                                linewidth=0, antialiased=False)

         fig.colorbar(surf, shrink=0.5, aspect=5)
         plt.title('Error Surface')
         ax.set_xlabel('$Theta_0$')
         ax.set_ylabel('$Theta_1$')
         ax.set_zlabel('SSE')
         ax.set_xticks(np.arange(-10, 0, step=2))
         ax.set_yticks(np.arange(1, 3, step=.5))

         plt.show()
```



Since we want to minimise $J(\theta)$ we effectively want to find the parameter values that lie at the very bottom of this surface plot. Note how the surface is an elliptical parabola, this means that any minima we find will be a global minima. A parameter sweep like this gives us a rough idea of where the minimum is but it is not exact and becomes prohibitively expensive when the number of parameters increases! Luckily this is where gradient descent comes in. Gradient descent provides us with a method to take small steps across the surface in a direction that **locally** decreases the SSE. These local steps eventually lead us to a minima. We will now work through gradient descent step by step so that you can develop an intuition for how it works.

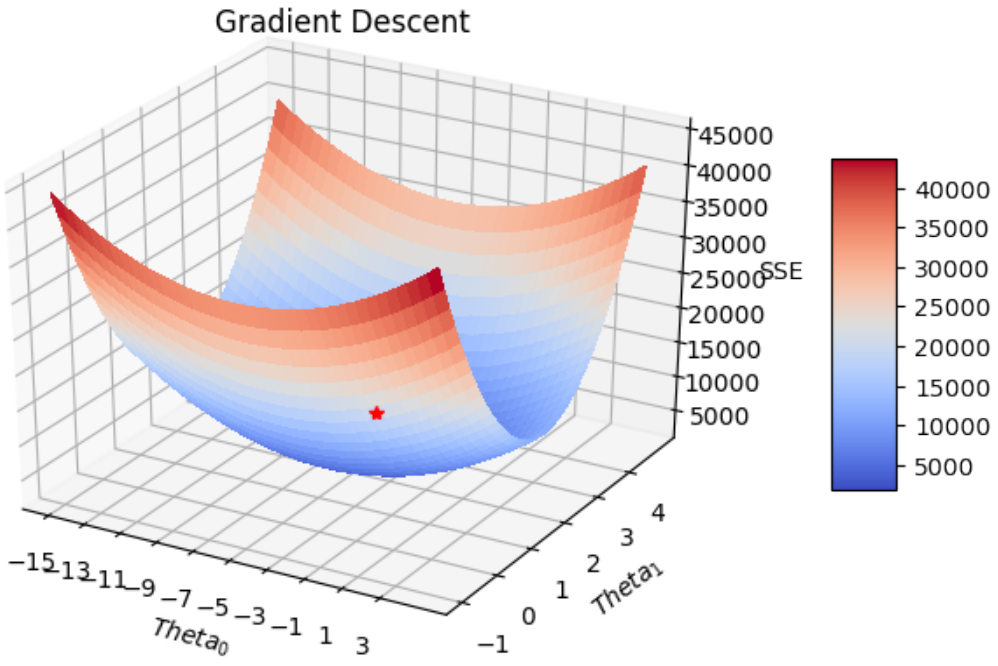
We can start by initialising the parameters of our model to zero:

$$\theta = \begin{bmatrix}\theta_0 & \theta_1\end{bmatrix} = \begin{bmatrix}0 & 0\end{bmatrix}$$

The code below evaluates the SSE for our intialised parameter values and indicates its location on the surface plot:

```
In [44]: theta = np.array([[0], [0]])
         pred = np.expand_dims(theta[0] + (theta[1] * X), axis=-1)
         j = np.squeeze(np.matmul((t - pred).transpose(),
                                  t - pred)) # vectorised SSE

         theta_0 = np.linspace(-15, 5, num_per_parameter)
         theta_1 = np.linspace(-1, 5, num_per_parameter)
         theta_0, theta_1 = np.meshgrid(theta_0, theta_1)
```

```
J = np.zeros((num_per_parameter, num_per_parameter))
for i in range(num_points):
    J += np.square(t[i] - (theta_0 + (theta_1 * X[i])))

fig = plt.figure(figsize=(8,5))
ax = fig.gca(projection='3d')
surf = ax.plot_surface(theta_0, theta_1, J,
                       cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
plt.plot([theta[0]], [theta[1]], [j], 'r*')

fig.colorbar(surf, shrink=0.5, aspect=5)
plt.title('Gradient Descent')
ax.set_xlabel('$Theta_0$')
ax.set_ylabel('$Theta_1$')
ax.set_zlabel('SSE')
ax.set_xticks(np.arange(-15, 5, step=2))
ax.set_yticks(np.arange(-1, 5, step=1))

plt.show()
```



The task of gradient descent is now to move our parameter values through the parameter space towards the minimum of our objective function. It does this by calculating the partial derivatives of our objective function $J(\theta)$ with respect to our current parameter values. The partial derivative of $J(\theta)$ with respect to θ_0 tells us how $J(\theta)$ changes as we increase θ_0 but keep θ_1 constant. Simi-

larly, the partial derivative of $J(\theta)$ with respect to θ_1 tells us how $J(\theta)$ changes as we increase θ_1 but keep θ_0 constant. These partial derivates are simple to calculate in the case of linear regression:

$$\begin{aligned}
\frac{\partial J}{\partial \theta_0} &= \frac{\partial}{\partial \theta_0} \sum_{i=1}^{N} (t_i - y(x_i, \theta))^2 \\
&= \frac{\partial}{\partial \theta_0} \sum_{i=1}^{N} (t_i - y(x_i, \theta))^2 \\
&= \frac{\partial}{\partial \theta_0} \sum_{i=1}^{N} (t_i - (\theta_0 + \theta_1 x_i))^2 \\
&= \sum_{i=1}^{N} \frac{\partial}{\partial \theta_0} (t_i - (\theta_0 + \theta_1 x_i))^2 \\
&= \sum_{i=1}^{N} 2(t_i - (\theta_0 + \theta_1 x_i)) \frac{\partial}{\partial \theta_0} (t_i - (\theta_0 + \theta_1 x_i)) \qquad \text{using the chain rule} \\
&= -2 \sum_{i=1}^{N} (t_i - (\theta_0 + \theta_1 x_i))
\end{aligned}$$

$$\begin{aligned}
\frac{\partial J}{\partial \theta_1} &= \frac{\partial}{\partial \theta_1} \sum_{i=1}^{N} (t_i - y(x_i, \theta))^2 \\
&= \frac{\partial}{\partial \theta_1} \sum_{i=1}^{N} (t_i - y(x_i, \theta))^2 \\
&= \frac{\partial}{\partial \theta_1} \sum_{i=1}^{N} (t_i - (\theta_0 + \theta_1 x_i))^2 \\
&= \sum_{i=1}^{N} \frac{\partial}{\partial \theta_1} (t_i - (\theta_0 + \theta_1 x_i))^2 \\
&= \sum_{i=1}^{N} 2(t_i - (\theta_0 + \theta_1 x_i)) \frac{\partial}{\partial \theta_1} (t_i - (\theta_0 + \theta_1 x_i)) \qquad \text{using the chain rule} \\
&= -2 \sum_{i=1}^{N} (t_i - (\theta_0 + \theta_1 x_i)) x_i
\end{aligned}$$

These partial derivatives really are the key to gradient descent. If for example, the partial derivate $\frac{\partial}{\partial \theta_0}$ is positive then increasing the value of θ_0 will also increase the value of $J(\theta)$ i.e. the SSE. In such circumstances we want to decrease the value of θ_0 so that the value of $J(\theta)$ also decreases. Conversely, if $\frac{\partial}{\partial \theta_0}$ is negative then we want to increase the value of θ_0 so that the value of $J(\theta)$ decreases. This basic intuition leads to the following update rule:

$$\theta_j = \theta_j - \alpha \frac{\partial J}{\partial \theta_j}$$

In effect this update rule is saying that we want to move our values in the opposite direction to the gradient i.e. 'downhill'. Note that if we were maximising our objective function (i.e. we were using the log likelihood rather than the negative log likelihood) then we would want to move in the direction of the gradient or 'uphill'. The update rule would therefore be:

$$\theta_j = \theta_j + \alpha \frac{\partial J}{\partial \theta_j}$$

The free parameter α is called the learning rate and tuning its value is particularly important if you want gradient descent to work effectively. It scales the partial derivates by some small amount (e.g. 0.01) so that we only make small changes to our parameter values. We only want to make small changes because the partial derivate tells us the gradient for a single point in parameter space and is a local estimate of how the objective function is changing with respect to the parameters. If we take too large a step we could actually end up in a region of the parameter space where the SSE is larger than before.

To apply gradient descent we simply employ an iterative procedure whereby we repeatedly calculate the partial derivates and apply the aforementioned update rule. Lets now apply gradient descent to our synthetic dataset using our SSE objective function. We have already worked out what form the partial derivates will take and so we know what our update rules will be:

$$\begin{aligned}
\theta_0 &= \theta_0 - \alpha \frac{\partial J}{\partial \theta_0} \\
&= \theta_0 - \alpha(-2 \sum_{i=1}^{N} (t_i - (\theta_0 + \theta_1 x_i))) \\
&= \theta_0 + 2\alpha \sum_{i=1}^{N} (t_i - (\theta_0 + \theta_1 x_i))) \\
\theta_1 &= \theta_1 - \alpha \frac{\partial J}{\partial \theta_1} \\
&= \theta_1 - \alpha(-2 \sum_{i=1}^{N} (t_i - (\theta_0 + \theta_1 x_i)) x_i) \\
&= \theta_1 + 2\alpha \sum_{i=1}^{N} (t_i - (\theta_0 + \theta_1 x_i)) x_i
\end{aligned}$$

There are a few alterations we can make to these update rules to make our life easier. Firstly, we can drop the 2 in both update rules because it can be absorbed into the learning rate α which is a free parameter. Secondly, we can take the mean over the training points rather than just the sum. If you follow the proofs then this is the same as using the mean squared error (MSE) rather than the sum of squared error (SSE). Dividing by a constant (N) doesn't change the location of the minimum of our objective function and so it will still give us the maximum likelihood estimate. The advantage of using the MSE however, is that now α is independent of the number of training points. If we find a suitable value for α for our data then using the MSE will ensure that the value works well regardless of our training set size. The final alteration we can make is more of an

implementational note. It is common in linear models to create a new input variable x_0 for each training point with a value of 1 so that our linear model becomes:

$$\begin{aligned} y(\mathbf{x}^i, \theta) &= \theta_0 x_0^i + \theta_1 x_1^i \\ &= \theta_0(1) + \theta_1 x_1^i \\ &= \theta_0 + \theta_1 x_1^i \end{aligned}$$

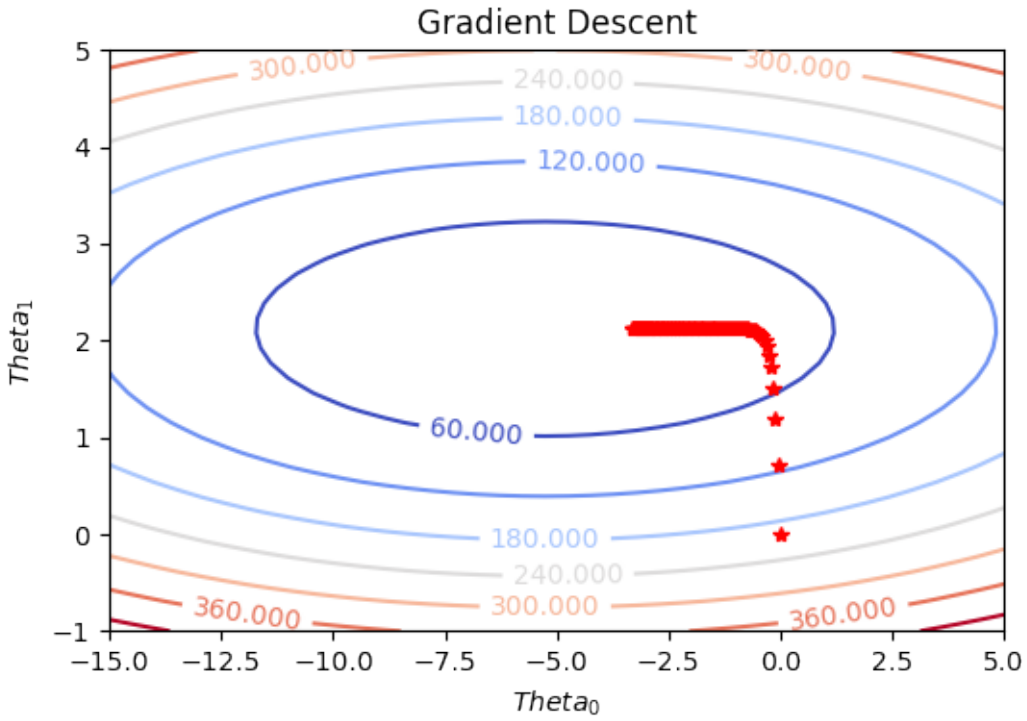
Where $\mathbf{x}^i$ is now a vector $\begin{bmatrix} x_0^i & x_1^i \end{bmatrix}$ containing the input values for datapoint i. The good thing about this is because $x_0^i = 1$ we can now just apply one update rule for all parameters, which with the other alterations, is as follows:

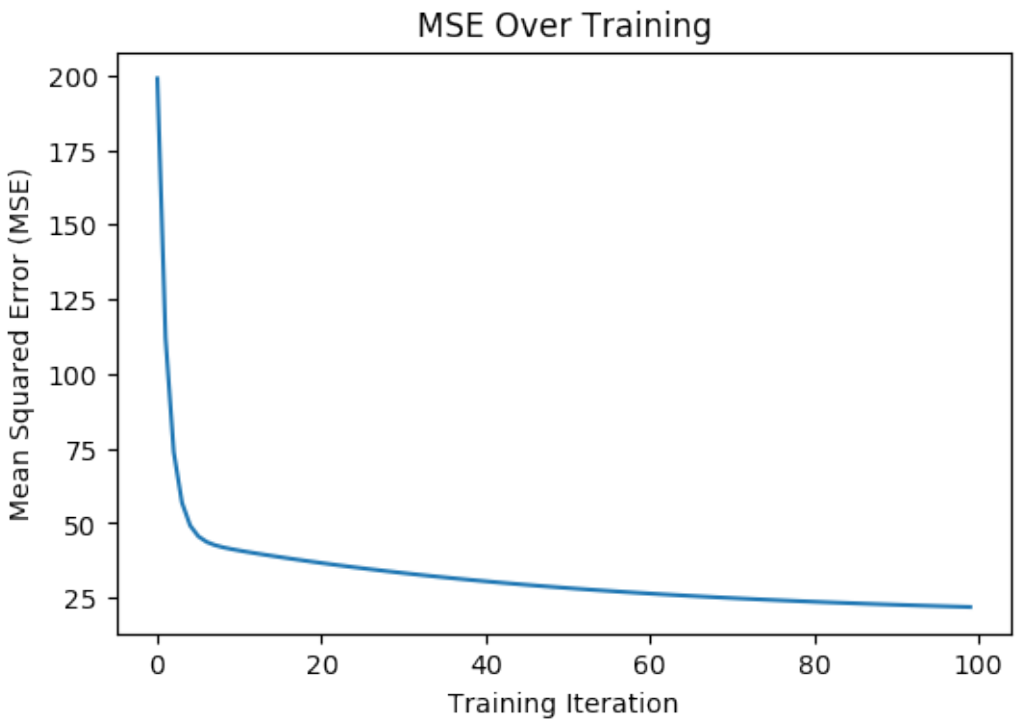
$$\theta_j = \theta_j + \alpha \frac{1}{N} \sum_{i=1}^{N} (t^i - (\theta_0 x_0^i + \theta_1 x_1^i)) x_j^i$$

The following code will now apply gradient descent to our synthetic dataset by repeatedly applying these update rules. Instead of a surface plot we will use a contour plot so that we can easily see how gradient descent moves through the parameter space to minimise our objective function. Much of the implementation below is vectorised so try to work through the matrix multiplications if you can to see how they correspond to the appropriate formula.

```
In [45]: # Add the column of 1s for the new bias term (x_0)
         X_plus_bias = np.concatenate((np.ones((num_points, 1)),
                                       np.expand_dims(X, axis=-1)),
                                      axis=-1)

         # Setup contour plot
         J = J / num_points # MSE
         fig = plt.figure()
         ax = plt.gca()
         CS = plt.contour(theta_0, theta_1, J, cmap=cm.coolwarm)

         # Perform Gradient Descent
         alpha = .01
         num_iters = 100
         MSEs = []

         for i in range(num_iters):
             plt.plot([theta[0]], [theta[1]], 'r*')
             pred = np.matmul(X_plus_bias, theta)
             theta = theta + (alpha *
                              (np.matmul(X_plus_bias.transpose(),(t-pred))
                               / num_points))
             mse = np.squeeze((1 / num_points) *
                              np.matmul((t - pred).transpose(),t-pred))
             MSEs.append(mse)
```

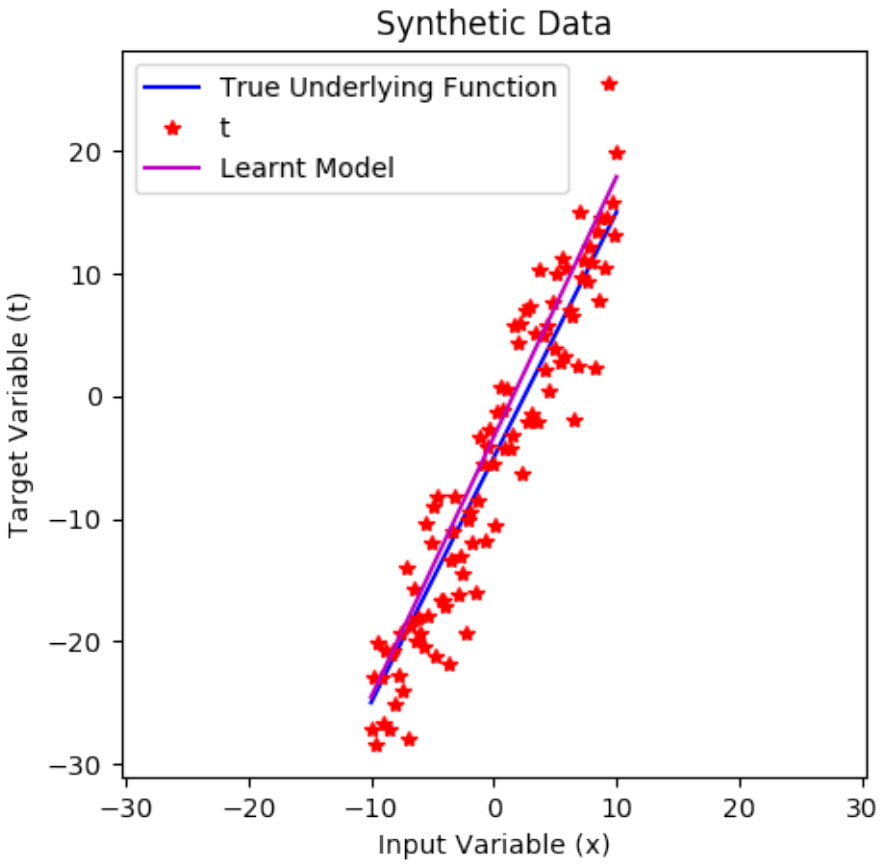
```
ax.clabel(CS, inline=1, fontsize=10)
plt.title('Gradient Descent')
ax.set_xlabel('$Theta_0$')
ax.set_ylabel('$Theta_1$')
plt.show()

plt.figure()
plt.plot(np.array(MSEs))
plt.title('MSE Over Training')
plt.xlabel('Training Iteration')
plt.ylabel('Mean Squared Error (MSE)')
plt.show()
```





We have just run gradient descent on our synthetic dataset! As you can see the MSE gradually decreases as learning progresses and our parameter values approach the true parameter values used to generate our dataset. The manitudes of the parameter updates start off large because of the steeper gradient but become smaller as the gradient becomes more gentle. Lets use the parameter values learnt by gradient descent to plot our learnt linear regression model:

```
In [46]: pred = np.matmul(X_plus_bias, theta)

         fig = plt.figure(figsize=(5,5))
         plt.plot(X, y, 'b', label='True Underlying Function')
         plt.plot(X, t, 'r*', label='t')
         plt.plot(X, pred, 'm', label='Learnt Model')
         plt.axis('equal')
         plt.legend(loc='upper left')
         plt.xlabel('Input Variable (x)')
         plt.ylabel('Target Variable (t)')
         plt.title('Synthetic Data')
         plt.show()
```



As you can see we have achieved a pretty good fit to the data just from 100 iterations of gradient descent. The great thing about gradient descent is that it is applicable across a wide range of machine learning algorithms. Gradient descent can work for any parametric model were you are able to calculate the partial derivates of your objective function with respect to the model parameters. Gradient descent also works well even when you have many many parameters! Imagine a parameter space that has a 1000 dimensions rather than just 2, we can still calculate the partial derivates for each parameter and then use these derivatives in the update rule to gradually approach a minimum of our objective function.

One last concept worth mentioning is stochastic gradient descent. In this version of gradient descent we do not calculate the objective function for all datapoints i.e. $J(\theta) = \sum_{i=1}^{N}(t_i - y(x_i, \theta))^2$, but instead do it for a single datapoint i.e. $J(\theta) = (t_i - y(x_i, \theta))^2$. We therefore lose the sum in the partial derivative of our objective function and the update rule in the case of linear regression becomes:

$$\theta_j = \theta_j + \alpha(t^i - (\theta_0 x_0^i + \theta_1 x_1^i))x_j^i$$

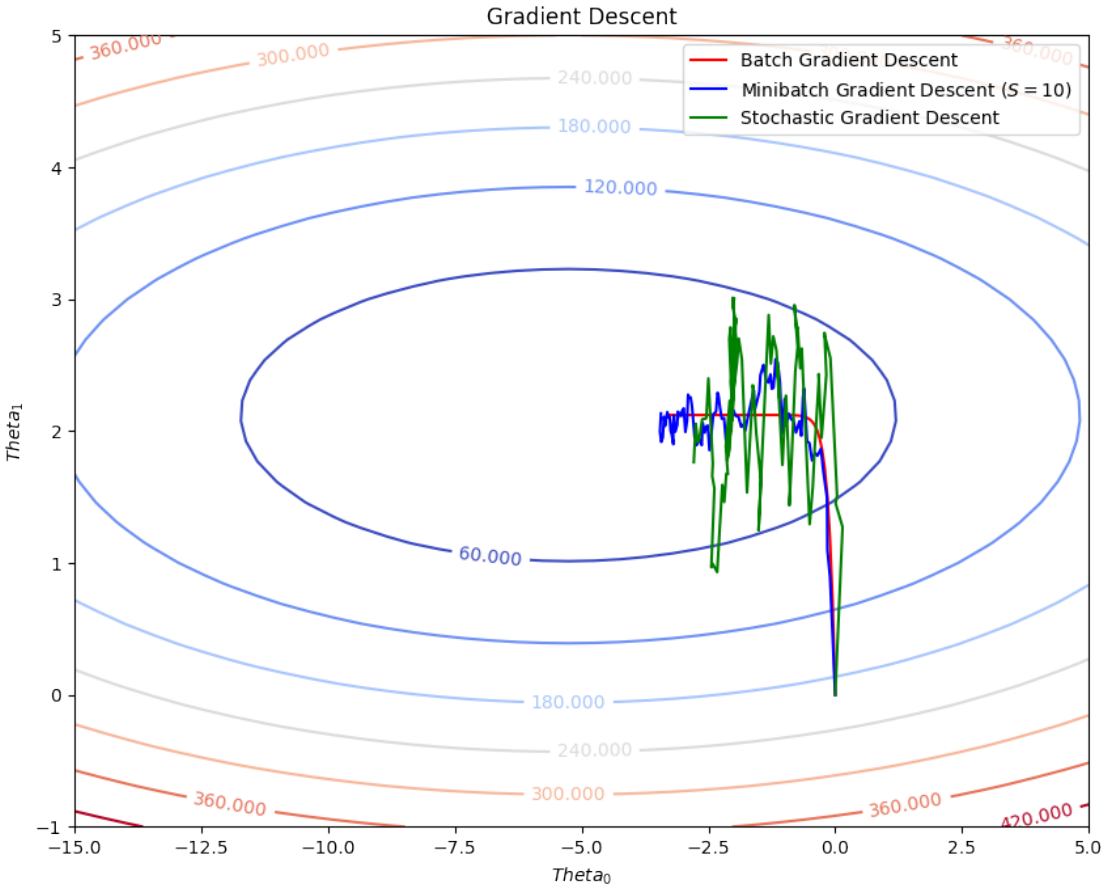
We apply this update rule iteratively, randomly selecting a single training example each time. This essentially calculates the 'cost' for one training example and updates the parameters based on that one training example. In normal gradient descent we take into account the 'cost' for all training examples and update the parameters correspondingly. Stochastic gradient descent may seem counterintuitive given that we want to find parameters that fit all our datapoints, but it does have a few significant advantages. Firstly, by removing the sum over all datapoints we make the calculation of the update rule much quicker and so convergence to the miminum may also be quicker. This is especially true for datasets where the number of training examples N is large. Secondly, due to its 'stochastic' nature, stochastic gradient descent can help us to avoid local mimima. Updating the parameter values based on a single training example does not guarantee that the cost will reduce for the other training examples. As a result, the cost over all training examples may actually increase after one update of stochastic gradient descent. This allows us to potentially climb out of local minima and is the reason stochastic gradient descent is considered 'stochastic'. While computing the update rule may be faster in stochastic gradient descent, its stochastic nature means that more updates may be needed to reach the minimum of our objective function.

Normal (batch) gradient descent and stochastic gradient descent represent two extremes of a continuum. In the middle of this continuum we have minibatch gradient descent, which uses an objective function that sums over a subset S of the training examples i.e. $J(\theta) = \sum_{i=1}^{S}(t_i - y(x_i, \theta))^2$. S can be tuned so that there is a nice trade-off between the advantages of stochastic gradient descent (fast updating, stochastic) and the advantages of batch gradient descent (more direct updates).

The code below runs batch, minibatch and stochastic gradient descent on our synthetic dataset so that you can see the differences in how they navigate the parameter space.

```
In [47]: fig = plt.figure(figsize=(10,8))
         ax = plt.gca()
         CS = plt.contour(theta_0, theta_1, J, cmap=cm.coolwarm)
         ax.clabel(CS, inline=1, fontsize=10)

         alpha = .01
         num_iters = 100

         # Perform Batch Gradient Descent
         theta = np.array([[0], [0]])
         thetas = []
         for i in range(num_iters):
             thetas.append([theta[0,0], theta[1,0]])
             pred = np.matmul(X_plus_bias, theta)
             theta = theta + (alpha *
                              (np.matmul(X_plus_bias.transpose(),
                                         (t - pred)) / num_points))
         thetas = np.array(thetas)
         plt.plot(thetas[:, 0], thetas[:, 1], 'r',
                  label='Batch Gradient Descent')
```

```
# Perform Minibatch Gradient Descent
S = 10
theta = np.array([[0], [0]])
thetas = []
for i in range(num_iters):
    thetas.append([theta[0,0], theta[1,0]])
    indexes = np.random.choice(num_points, S)
    pred = np.matmul(X_plus_bias[indexes,:], theta)
    theta = theta + (alpha *
                     (np.matmul(X_plus_bias[indexes,:].transpose(),
                                (t[indexes] - pred)) / S))
thetas = np.array(thetas)
plt.plot(thetas[:, 0], thetas[:, 1], 'b',
         label='Minibatch Gradient Descent ($S = 10$)')

# Perform Stochastic Gradient Descent
theta = np.array([0, 0])
thetas = []
for i in range(num_iters):
    thetas.append(theta)
    plt.plot([theta[0]], [theta[1]])
    index = np.random.randint(num_points)
    pred = np.dot(X_plus_bias[index,:], theta)
    theta = theta + (alpha * X_plus_bias[index,:] *
                     (t[index] - pred))
thetas = np.array(thetas)
plt.plot(thetas[:, 0], thetas[:, 1], 'g',
         label='Stochastic Gradient Descent')

plt.legend()
plt.title('Gradient Descent')
ax.set_xlabel('$Theta_0$')
ax.set_ylabel('$Theta_1$')
plt.show()
```



I hope this notebook has helped to develop your understanding of gradient descent and why it is such a useful method. If you have a good grasp of gradient descent then further topics such as training neural networks via backpropagation will become seemingly easy. There are loads of other great resources out there on gradient descent and I would urge you to read around a bit to really develop your understanding. If you have any questions then please feel free to contact me!