Gradient Descent

As we have seen before, Machine Learning is an optimization problem:

- minimize Cost/Loss
- maximize Utility

Most of the models we have seen thus far have relatively simple Cost functions that are amenable to closed form solutions.

But the art of Machine Learning is in crafting Cost functions that express the goals of the problem.

So we can no longer count on closed form solutions for the optimization.

Gradient Descent is a way of finding the parameters (e.g, Θ) that minimize a given cost Function, ie., solving optimization problems.

The advantage of Gradient Descent is its ability to handle complex Cost Functions.

Gradient Descent

- minimizes convex functions
- is a type of search algorithm
- is a critical tool for Deep Learning
    - most cost functions won't be amenable to closed form solutions

Cost/Loss, Utility/Optimization: review

- The prediction $\hat{\mathbf{y}}^{(i)}$ for example $\mathbf{x}^{(\mathbf{i})}$ is perfect if it matches the true label $\mathbf{y}^{(\mathbf{i})}$

$$\hat{\mathbf{y}}^{(\mathbf{i})} = \mathbf{y}^{(\mathbf{i})}$$

- The distance between $\hat{\mathbf{y}}^{(i)}$, $\mathbf{y}^{(\mathbf{i})}$ is called the *Loss* (or *Cost*) for example i:

$$\mathcal{L}_{\Theta}^{(\mathbf{i})} = L(\hat{\mathbf{y}}^{(\mathbf{i})}, \mathbf{y})$$

where $L(a, b)$ is a function that is 0 when $a = b$ and increasing as a increasingly differs from b.

Two versions L that we've seen are Mean Squared Error (for Regression) and Cross Entropy Loss (for classification).

The Loss for the entire training set is simply the average (across examples) of the Loss for the example

$$\mathcal{L}_{\Theta} = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}_{\Theta}^{(\mathbf{i})}$$

Whereas Loss describes how "bad" our prediction is, we sometimes refer to the converse -- how "good" the prediction is.

We call the "goodness" of the prediction the *Utility* U_Θ.

So we could state the optimization objective either as "minimize Cost" or "maximize Utility".

By convention, the DL optimization problem is usually framed as one of minimization (of cost or loss) rather than maximization of utility.

- The goal of fitting/training is to solve for the Θ that minimizes the training set loss L_Θ (or conversely, maximizes the Utility U_Θ.
- The method for finding Θ is called optimization.
- There is one optimization method that we will study in depth: Gradient Descent.
- We can use this in Classical ML but it will become a key tool once we move on to Deep Learning.
- One focus of this lecture will be variations on Gradient Descent.

Gradients

Gradient Descent is a method for optimizing the Optimization Objective.

It works for any model but we will illustrate it with Linear Regression.

For Regression with MSE as the loss, $L(a, b) = (a - b)^2$,so the loss for example i is:

$$\mathcal{L}_{\Theta}^{(\mathbf{i})} = (\text{error}^{(\mathbf{i})})^2$$

where

$$\text{error}^{(\mathbf{i})} = \hat{\mathbf{y}}^{(\mathbf{i})} - \mathbf{y}^{(\mathbf{i})}$$

The average cost

$$\mathcal{L}_{\Theta} = \text{MSE}(\mathbf{X}, \Theta) = \frac{1}{m} \sum_{i=1}^{m} (\text{error}^{(i)})^2$$

where $\mathbf{X}$ is the entire training set.

There may be added elements (e.g., regularization constraints) of the objective as well (discussed later).

A function of Θ, $\mathcal{L}_\Theta$, can be minimized by taking its derivative with respect to Θ and setting it equal to 0.

This is an equation that can be solved for Θ.

Because Θ is a vector, there is one derivative per feature. Hence the vector of derivatives (called the **gradient**) is

$$\nabla_\Theta \mathcal{L}_\Theta = \begin{pmatrix} \frac{\partial}{\partial \Theta_0} \mathcal{L}_\Theta \\ \frac{\partial}{\partial \Theta_1} \mathcal{L}_\Theta \\ \vdots \\ \frac{\partial}{\partial \Theta_n} \mathcal{L}_\Theta \end{pmatrix}$$

For $\mathcal{L}_\Theta = \text{MSE}(\mathbf{X}, \Theta)$

$$\nabla_\Theta \mathcal{L}_\Theta = \begin{pmatrix} \frac{\partial}{\partial \Theta_0} \text{MSE}(X, \boldsymbol{\theta}) \\ \frac{\partial}{\partial \Theta_1} \text{MSE}(X, \boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \Theta_n} \text{MSE}(X, \boldsymbol{\theta}) \end{pmatrix}$$

$$
\begin{aligned}
\frac{\partial}{\partial \theta_j} \mathrm{MSE}(X, \Theta) &= \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial \theta_j} (\mathrm{error}^{(i)})^2 \\
&= \frac{1}{m} \sum_{i=1}^{m} 2 \times \mathrm{error}^{(i)} \times \frac{\partial}{\partial \theta_j} \mathrm{error}^{(i)} \\
&= \frac{2}{m} \sum_{i=1}^{m} \mathrm{error}^{(i)} \times \frac{\partial}{\partial \theta_j} \hat{\mathbf{y}}^{(\mathbf{i})} && \text{since error}^{(\mathbf{i})} = \hat{\mathbf{y}}^{(\mathbf{i})} - \mathbf{y}^{(\mathbf{i})}, \\
&= \frac{2}{m} \sum_{i=1}^{m} \mathrm{error}^{(i)} \times \mathbf{x}_j^{(\mathbf{i})} && \text{since } \hat{y}^{(\mathbf{i})} = \Theta^T \cdot \mathbf{x} = \sum_{k=} \\
& && \text{and } \Theta_j \mathbf{x}_j^{(\mathbf{i})} \text{ is only term inv}
\end{aligned}
$$

Thus the gradient for Linear Regression can be written in matrix form as

$$\nabla_\theta \, \mathrm{MSE}(X, \theta) == \frac{2}{m} \mathbf{X}^T (\theta^T \mathbf{X} - \mathbf{y})$$

This will be particularly useful when working with NumPy as the gradient calculation is a vector operation that is implemented so as to be fast.

Batch Gradient Descent

The basic algorithm is:

1. Initialize Θ randomly
2. Repeat until done
    A. Compute the Gradient
    B. Update Θ by taking a step in the (negative) direction of the Gradient

Intuition

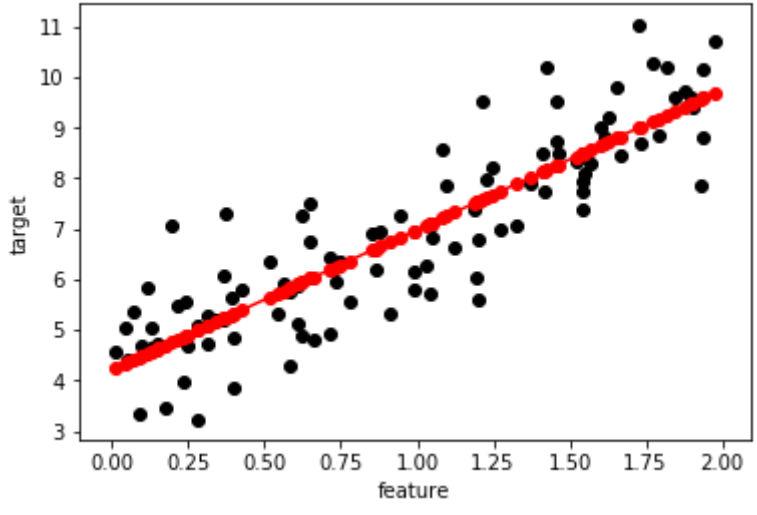
- Our objective is to minimize the Average Loss $\mathcal{L}_\Theta$
- The gradient, with respect to Θ_j, is the increase in $\mathcal{L}_\Theta$ for a unit change in Θ_j
    - we multiply by -1 to *decrease* the loss
- So by taking a step (size to be determined) in the negative direction of the gradient we decrease $\mathcal{L}_\Theta$

Let's illustrate Batch Gradient Descent on an example.

First, we use sklearn's `LinearRegression` as a baseline against which we will compare the Θ obtained from Gradient Descent.

In [3]:
```
X_lr, y_lr = gdh.gen_lr_data()
clf_lr = gdh.fit_lr(X_lr,y_lr)
fig, ax = gdh.plot_lr(X_lr, y_lr, clf_lr)

theta_lr = (clf_lr.intercept_, clf_lr.coef_)
```



Now let's perform Batch Gradient Descent and compare the Θ's

In [4]:
```
gd_theta = gdh.batchGradientDescent_lr(X_lr, y_lr)
theta_lr - gd_theta
```

Out[4]:
```
array([[ 7.99360578e-15],
       [-7.99360578e-15]])
```

The Θ's are equal up to 15 decimal points.

Let's look at the code for Batch Gradient Descent and examine the details

alpha = 0.1 n_iterations = 1000 m = 100 theta = np.random.randn(2,1) for iteration in range(n_iterations): gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y) theta = theta - alpha * gradients

- You can see the code that implements the steps described in English
    - `alpha` is the step size: how fast we adjust Θ in the direction of the gradient
    - `X_b` is the matrix
        - whose first column is 1
        - whose other columns are the non-intercept features
    - `X_b.dot(theta)` are the predicted values for all observations
    - `X_b.dot(theta) - y` are the errors for all observations

Intuition

- Why is the proportional to `gradients`?
    - When the gradient is large, we are far from the minimum $\mathcal{L}_\Theta$: take a big step
    - When the gradient is small, we are close to the minimum $\mathcal{L}_\Theta$: take a small step
- Why $\alpha < 1$?
    - If we take too big a step, we may overshoot the minimum, as we will see

Since the Θ's computed by Gradient Descent and Linear Regression are the same, it's no surprise that the predictions are too.

- as demonstrated in the following code

In [5]:
```
X_new = np.array([[0], [2]])
gd_y_pred = gdh.predict(X_new, theta_lr)
clf_y_pred = clf_lr.predict(X_new)

gd_y_pred == clf_y_pred
```

Out[5]:
```
array([[ True],
       [ True]])
```

Batch gradient descent: the movie

We can hopefully gain intuition by watching Gradient Descent at work.

- you will see the effect of each update of Θ
- you will see the effect of changing η, which scales the step size

In [6]:

```
%%capture
movie_file = os.path.join(MOVIE_DIR,'batch_gradient_descent_eta_10.mp4')

if CREATE_MOVIE:
    gd_anim = gdh.create_movie(X_lr, y_lr, n_iterations=10)
    gd_anim.save(movie_file, codec='h264')


if CREATE_MOVIE:
    gdh.show_movie(gd_anim)
else:
    print("To view movie:\n Use link in following cell, or use browser to visit
 file {f}".format(f=movie_file))
```

Movie (images/batch_gradient_descent_eta_10.mp4)

Initializing Θ

What would have happened if, instead of initializing Θ to random numbers

- we had initialized it to 0 ?
- we had initialized it to a very large number

Step size

What's a good choice for `alpha` ? We had used 0.1 and obtained convergence in around 10 steps.

Le'ts try a smaller step size: `alpha` = 0.2

In [7]:

```
%%capture
movie_file = os.path.join(MOVIE_DIR,'batch_gradient_descent_eta_02.mp4')

if CREATE_MOVIE:
    gd_anim_eta_02 = gdh.create_movie(X_lr, y_lr, alpha=0.02, n_iterations=30)
    gd_anim_eta_02.save(movie_file, codec='h264')


if CREATE_MOVIE:
    gdh.show_movie(gd_anim_eta_02)
else:
    print("To view movie:\n Use link in following cell, or use browser to visit
 file {f}".format(f=movie_file))
```

Movie (images/batch_gradient_descent_eta_02.mp4)

Like watching paint dry !

How about something bigger ?

In [8]:

```
%%capture
movie_file = os.path.join(MOVIE_DIR,'batch_gradient_descent_eta_45.mp4')
if CREATE_MOVIE:
    gd_anim_eta_45 = gdh.create_movie(X_lr, y_lr, alpha=0.45, n_iterations=20)
    gd_anim_eta_45.save(movie_file, codec='h264')


if CREATE_MOVIE:
    gdh.show_movie(gd_anim_eta_45)
else:
    print("To view movie:\n Use link in following cell, or use browser to visit
 file {f}".format(f=movie_file))
```

Movie (images/batch_gradient_descent_eta_45.mp4)

And even bigger

In [9]:

```
%%capture
movie_file = os.path.join(MOVIE_DIR,'batch_gradient_descent_eta_50.mp4')

if CREATE_MOVIE:
    gd_anim_eta_50 = gdh.create_movie(X_lr, y_lr, alpha=0.50, n_iterations=20)
    gd_anim_eta_50.save(movie_file, codec='h264')


if CREATE_MOVIE:
    gdh.show_movie(gd_anim_eta_50)
else:
    print("To view movie:\n Use link in following cell, or use browser to visit
 file {f}".format(f=movie_file))
```

Movie (images/batch_gradient_descent_eta_50.mp4)

Lost in space !

- the vertical intercept magnitude increases until it is off the screen

Learning rate schedule

We see from the above that if the step size is too small, it takes long to converge.

But if the step size is too big, we may overshoot.

An adaptive learning rate schedule may be the solution:

- take big steps at first
- take smaller steps toward end

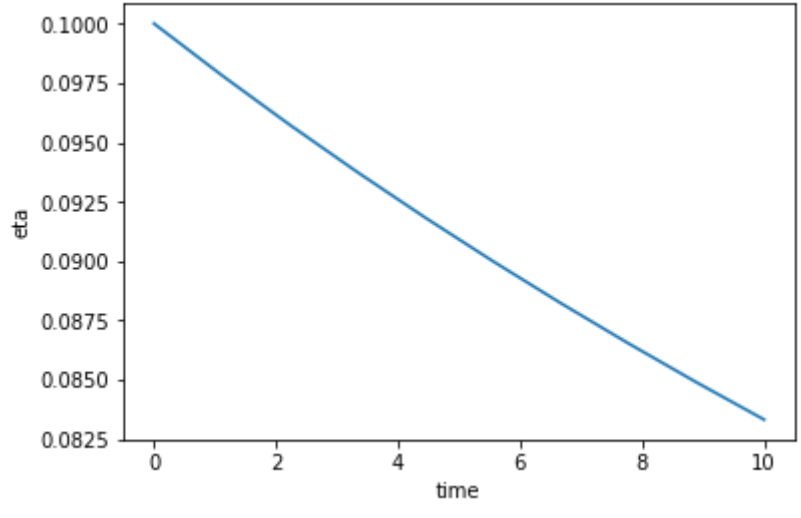
In [10]:

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

def learning_schedule(t):
    return t0 / (t + t1)

t = np.linspace(0, 10, 10)

fig = plt.figure()
ax  = fig.add_subplot(1,1,1)
_ =ax.plot(t, learning_schedule(t))
_ = ax.set_xlabel("time")
_ = ax.set_ylabel("eta")
```



When to stop

Can we do better than running for a fixed number of iterations ? Yes:

- Let Cost_t be the Cpst Function at step t
- Stop if
    - $\mathrm{Cost}_{t-1} - \mathrm{Cost}_t < \epsilon$
    - That is: stop if improvement of Cost Function is not big enough

A word on derivatives

Preview of part 2 of the course:

- the derivatives we used were *analytic* and not numerical approximations
- how can we automate calculation of analytic derivatives ?

Other cost functions

- Ridge Regression Cost Function
    - MSE, with a penalty large Θ
        - it's easy to compute the derivative of this cost function
        - try Minibatch Gradient Descent on this Cost Functions

Gradient Boosing

In the Decision Tree lecture we described Gradient Boosting (using Decision Tree Regression as an example).

Herei

- We built a *sequence* of Trees $T_{(0)}, T_{(1)}, \ldots$
- The relevance to us is that this resulted in a sequence of predictions $\hat{\mathbf{y}}_{(0)}, \mathbf{y}_{(1)}, \ldots$
- We updated $\hat{\mathbf{y}}$ via

$$\hat{\mathbf{y}}_{(\mathbf{t})} = \hat{\mathbf{y}}_{(t-1)} + \alpha * \hat{\mathbf{e}}_{(\mathbf{t})}$$

  - where $\mathbf{e}_{(\mathbf{t})} = \mathbf{y} - \hat{\mathbf{y}}_{(t-1)}$

For an MSE loss function

$$\mathcal{L}^{(\mathbf{i})} = (\hat{\mathbf{y}}_{(t-1)} - \mathbf{y})^2$$

the derivative with respect to $\hat{\mathbf{y}}_{(t-1)}$ is

$$\begin{aligned} \frac{\partial \mathcal{L}^{(\mathbf{i})}}{\partial \hat{\mathbf{y}}_{(t-1)}} &= 2 * (\hat{\mathbf{y}}_{(t-1)} - \mathbf{y}) \quad \text{by chain rule} \\ &= -2 * \mathbf{e_{(t)}} \end{aligned}$$

So the update rule for Gradient Boosting is the same as for Gradient Descent !

Thus Gradient Boosting, which was originally described informally

- is the minimization an MSE loss function by Gradient Descent.
    - n.b., we often describe a loss function as 0.5 time that described above
        - in to cancel out the 2 from the derivative of the sqaured error

Improvements to Gradient Descent

Simon Ruder survey (https://arxiv.org/abs/1609.04747)

Gradient Descent Cheatsheet (https://towardsdatascience.com/10-gradient-descent-optimisation-algorithms-86989510b5e9)

The update step

$$\Theta = \Theta - \alpha * \frac{\partial \mathcal{L}_\Theta}{\partial \Theta}$$

where α is the learning rate.

The improvements to Gradient Descent modify

- α, the learning rate
- $\frac{\partial \mathcal{L}_\Theta}{\partial \Theta}$ the gradient

In order to be able to flexibly change the definiton of both the gradient and the learning rate at each time step t, e will re-write the update step at time t as

$$\Theta_{(\mathbf{t})} = \Theta_{(t-1)} - \alpha' * V_{(\mathbf{t})}$$

$V_{(\mathbf{t})}$ will be our modified gradient and α' our modified learning rate.

Momentum: modify the gradient

In vanilla Gradient Descent, the gradients at time $t-1$ and time t are completely independent.

This has the potential for gradients to rapidly change direction (recall, they are a vector).

To smooth out jumps we could compute a modified gradient $V_{(\mathbf{t})}$ as:

$$V_{(\mathbf{t})} = \beta_V * V_{(t-1)} + (1-\beta_V) * \frac{\partial \mathcal{L}_\Theta}{\partial \Theta}$$

(Initialize $V_0 = 0$)

That is, the modified gradient is a weighted combination of the previous gradient and the new gradient.

Typically $\beta_V \approx 0.9$ so the old gradient dominates.

$V_{(\mathbf{t})}$ is the exponentially weighted moving average of the gradient.

Hence, there is "momentum" in the gradients in that they can't jump suddenly.

RMSprop: Modify the learning rate

Let

$$S_{(\mathbf{t})} = \beta_S * S_{(t-1)} + (1 - \beta_S) * \left(\frac{\partial \mathcal{L}_\Theta}{\partial \Theta} \right)^2$$

That is, $S_{(\mathbf{t})}$ is the exponentially weighted *variance* of the gradient.

(Initialize $S_0 = 0$)

Rather than using a learning rate of α, the RMSprop algorithm uses

$$\alpha' = \frac{1}{\sqrt{S_{(\mathbf{t})} + \epsilon}} * \alpha$$

The intuition is that if the gradient with respect to Θ_j is noisy (i.e., large variance) we want to damp updates in that component.

This also has the advantage that a rarely updated element Θ_i, having a low variance, will have a relatively larger update when it is encountered than a more frequently encountered feature.

Typically $\beta_S \approx 0.9$ so the old variance dominates.

Why the extra ϵ ? We've seen this before (e.g., $\log(x + \epsilon)$): it's to avoid mathematical issues of certain functions (inverse, log) when the argument is 0.

AdaM: Modify both the gradient and the learning rate

The AdaAM (Adaptive Moment) algorithm modifies both the gradient and learning rates via exponentially moving averages of the gradient as well as its variance.

$$V_{(\mathbf{t})} = \beta_V * V_{(t-1)} + (1 - \beta_V) \frac{\partial \mathcal{L}_\Theta}{\partial \Theta}$$

$$S_{(\mathbf{t})} = \beta_S * S_{(t-1)} + (1 - \beta_S) * \left(\frac{\partial \mathcal{L}}{\partial \Theta} \right)^2$$

$$\alpha' = \frac{1}{\sqrt{S_{(\mathbf{t})} + \epsilon}} * \alpha$$

Bias correction

You will have observed that we initialized to 0 the moving averages for gradients ($V_0 = 0$) and the variance of the gradients ($S_0 = 0$).

So the values are "biased" towards 0 with the bias having greatest effect for small t (i.e., when the number of "actual" values is small).

We can correct for the bias by dividing by $(1 - \beta^t)$:

$$\hat{V} = \frac{V_{(t)}}{1-\beta_V^t}$$

$$\hat{S} = \frac{S_{(t)}}{1-\beta_S^t}$$

In [11]:
```
print("Done")
```

```
Done
```