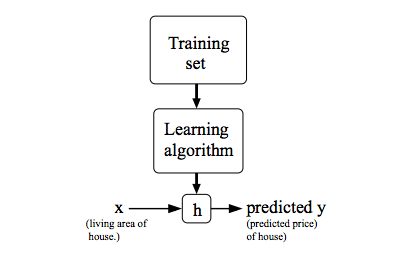
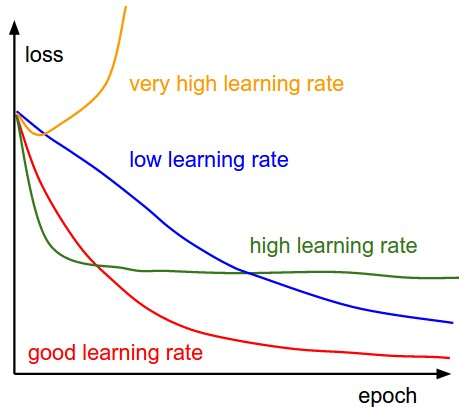
To establish notation for future use, we’ll use *x(i)*and *y(i)* to denote the **input features** and **class label** or output value that we are trying to predict. A pair of *(x(i), y(i)* is called a **training example**, and the dataset that we’ll be using to learn is a list of *m* training examples {*(x(i), y(i)*; i = 1, …, m) is called a **training set**. is the value of feature j in the ith training example.

If we define the , h is the hypothesis function that take a function and tries to output the estimated value, it is the function we want to fit. It does some task like this showing in image. We can measure the accuracy of our hypothesis function by using a cost function, . To break it apart, it is where is the mean of the square of , or the difference between the predicted value and the actual value. The constant is convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the term. This takes an average difference of all the results of the hypothesis with inputs from x’s and the actual output y’s. This cost function is also called the half mean squared error(MSE) or simplify squared error cost function. It turns out that these squared error cost function is a reasonable choice and works well for problems for most regression programs. There are other cost functions that will work well. But the square cost function is probably the most common used on for regression problems. The is the model’s prediction for the ith training example using the Ѳ. Cost function is also called, loss function, objective function, scoring function or error function, which estimates how well your training model’s prediction vary from the expected result. Here the Ѳ is the chosen vector (Ѳ0, Ѳ1, Ѳ2 …) that represent the weights of each component of input data. The objective of the learning algorithm, then, is to find the parameters Ѳ of which give , the minimum possible average amount of error measured by J(Ѳ), that also provide the best fit to our training set.

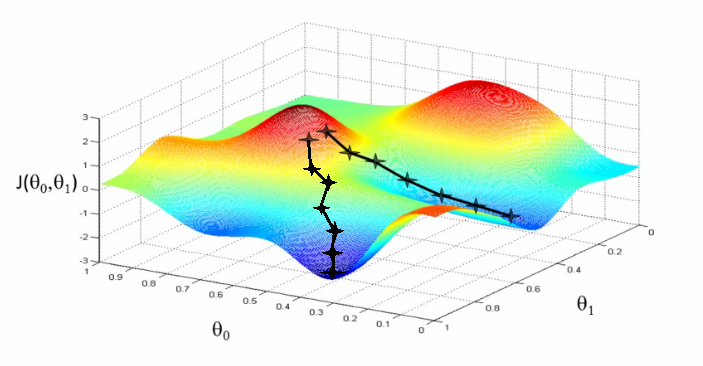
We’re going to use gradient descent to find Ѳ. It is an iterative algorithm, it starts with a random value of Ѳ, typically Ѳ=0. On each iteration, we take a small step α follow the opposite of the gradient of J(Ѳ) which tells us the direction to move Ѳ in and how big of a step to take, and keep repeating until convergence, and our h function will more and more accurate. Alpha is a hyper-parameter called the learning rate, and should be treated with care when choosing its value. It is basically a scalar value suggesting how fast we want to go to the minima., it may vary during the process. There are many types of gradient descent algorithms.

Full Batch Gradient Descent (BGD): it updates each component of Ѳ by taking the partial derivative of J(Ѳ), . On each iteration, you use whole dataset to compute the gradient. If the data is large, it will be slow.

Stochastic Gradient Descent (SGD): SGD also known as incremental gradient descent, typically reaches convergence much faster than BSD since it updates weight more frequently. Unlike the BGD, SGD tries to find minimums by iteration from the gradient of a single randomly picked training example, and the error is typically noisier. However, this can also have the advantage that SGD can escapes shallow local minima more easily. We can write, while . That is ith loss function associated with ith example, we use it when updating. , and i is random choice.

Mini-batch gradient descent: A compromise between computing the true gradient and the gradient at a single example, is to compute the gradient against more than one training example (called “mini-batch”) at each step. This can perform significantly better than true SGD because the code can make use of vectorization libraries rather than computing each step separately. It may also result in smoother convergence as the gradient computed at each step uses more training examples.

Let’s we see a example of MSE.

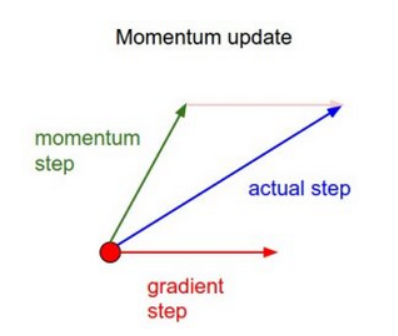
2. Similarly with
3. Cost Function:
4. Objective:
5. Update rules

1. Derivatices:

When implementing the update rule in software, Ѳ1 and Ѳ2 should not be updated until after you have computed the new values for both. Note that each update of the theta variables is averaged over the training set. This means that the statistics of your training are being considered during the learning process. An outlier training example (or even a mislabeled/corrupted example) is going to have less influence over the final weights.

The image is the “cost space” that represents how our algorithm would perform when we choose different Ѳ1, Ѳ2. So, on the y-axis, we have the cost J(Ѳ) against our parameters Ѳ1 and Ѳ2 on x-axis and z-axis respectively. Here, red region represented which have high cost, and blue region, which have low cost. Our goal is move in bluest region.

Limitations and challenges

1. It is very difficult to perform optimization using gradient descent when data poses a non-convex optimization problem.
2. Even when optimizing a convex optimization problem, there may be numerous local minimal points. Our aim is to go to global minima while avoiding local minima.
3. There is also a saddle point that the gradient is zero but is not an optimal point.
4. When the gradient is too small or too large, it may lead to problems like vanishing gradient or exploding gradient problems. And because of the algorithms do not converge.

Variants of Gradient Descent Algorithms

1. Vanilla Gradient Descent
2. Gradient Descent with Momentum. This introduce a new term called velocity which considers the previous update and a constant which is called momentum.
3. ADAGRAD uses adaptive technique for learning rate updating. Based on how the gradient has been changing for all the previous iterations we try to change the learning rate. The epsilon is a constant which is used to keep rate of change of learning rate in check.
4. ADAM is one more adaptive technique, you can consider this as momentum + ADAGRAD.

For rapid prototyping, use adaptive techniques like Adam/ADAGRAD. These help in getting quicker results with much less efforts. As here, you don’t require hyper-parameter tuning. To get the best results, we should use vanilla or momentum. These is slower, but they are mostly better than adaptive techniques. If your data is small and can be fit in a single iteration, you can use 2nd order techniques like I-BFGS. This is because 2nd order techniques are extremely fast and accurate, but are only feasible when data is small enough. There also an emerging method to use learned features to predict learning rates of gradient descent.