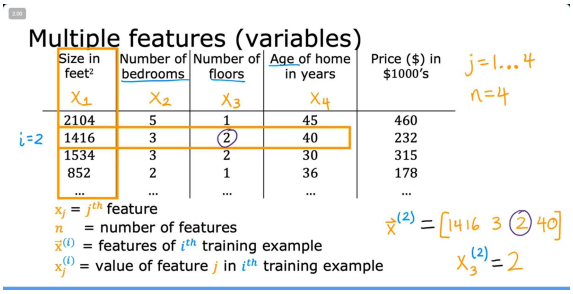
Module 2

22 July 2025 23:04

To introduce a little bit of new notation,
we're going to use the variables X_1,
X_2, X_3 and X_4,
to denote the four features.
For simplicity, let's introduce
a little bit more notation.
We'll write X subscript j
or sometimes I'll just say for short,
X sub j, to represent the list of features.
Here, j will go from one to four,
because we have four features.
I'm going to use lowercase
n to denote the total number of features,
so in this example,
n is equal to 4.
As before, we'll use X superscript
i to denote the ith training example.
Here X superscript i is actually
going to be a list of four numbers,
or sometimes we'll call this a vector that
includes all the features of the ith training example.
As a concrete example,
X superscript in parentheses 2,
will be a vector of
the features for the second training example,
so it will equal to this 1416, 3,
2 and 40

From <https://www.coursera.org/learn/machine-learning/lecture/gFuSx/multiple-features>
To refer to a specific feature
in the ith training example,
I will write X superscript i, subscript j,
so for example, X superscript 2 subscript
3 will be the value of the third feature,
that is the number of floors in
the second training example and
so that's going to be equal to 2.
Sometimes in order to emphasize that this X^2
is not a number but is
actually a list of numbers that is a vector,
we'll draw an arrow on top of that

From <https://www.coursera.org/learn/machine-learning/lecture/gFuSx/multiple-features>

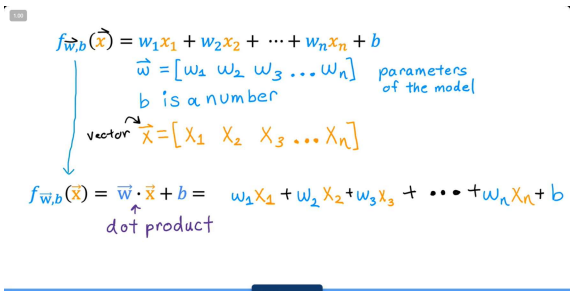


Here n is equal to 3.
Notice that in linear algebra,
the index or the counting starts from 1 and so
the first value is subscripted w1 and x1.
In Python code,
you can define these variables w,
b, and x using arrays like this.
Here, I'm actually using
a numerical linear algebra library
in Python called NumPy,
which is by far
the most widely used numerical linear algebra library
in Python and in machine learning.
Because in Python,
the indexing of arrays while
counting in arrays starts from 0,
you would access the first value of
w using w square brackets 0.
The second value using w square bracket 1,
and the third and using w square bracket 2.
The indexing here, it goes from 0,1 to
2 rather than 1, 2 to 3.

What we're going to do next is
introduce a little bit of notation
to rewrite this expression in
a simpler but equivalent way.
Let's define W as a list of
numbers that list the parameters W_1,
W_2, W_3,
all the way through W_n.
In mathematics, this is called a vector
and sometimes to designate that this is a vector,
which just means a list of numbers,
I'm going to draw a little arrow on top.

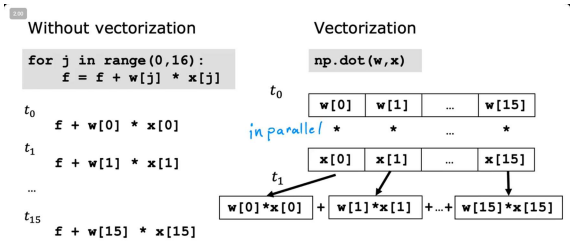
From <https://www.coursera.org/learn/machine-learning/lecture/gFuSx/multiple-features>
Next, same as before,
b is a single number and not a vector and so this vector
W together with this number b
are the parameters of the model.
Let me also write X as a list or a vector,
again a row vector that
lists all of the features X_1, X_2,
X_3 up to X_n,
this is again a vector,
so I'm going to add a little arrow up on top to signify.
In the notation up on top,
we can also add little arrows here and here
to signify that that W and that X
are actually these lists of numbers,
that they're actually these vectors.
With this notation, the model can now be rewritten more
succinctly as f of x equals,
the vector w dot and this dot refers to
a dot product from linear algebra of X the vector,
plus the number b.
What is this dot product thing?
Well, the dot products of
two vectors of two lists of numbers W and X,
is computed by checking
the corresponding pairs of numbers,
W_1 and X_1 multiplying that,
W_2 X_2 multiplying that,
W_3 X_3 multiplying that,
all the way up to W_n and X_n multiplying
that and then summing up all of these products.
Writing that out,
this means that the dot products is equal to W_1X_1 plus
W_2X_2 plus W_3X_3 plus all the way up to W_nX_n.
Then finally we add back in the b on top.
You notice that this gives us
exactly the same expression as we had on top.
The dot traffic notation lets you write the model in
a more compact form with fewer characters.
The name for this type of linear regression model with
multiple input features is multiple linear regression.
This is in contrast to univariate regression,
which has just one feature.
By the way, you might think
this algorithm is called multivariate regression,
but that term actually refers to something
else that we won't be using here.
I'm going to refer to this model
as multiple linear regression.
That's it for linear regression with multiple features,
which is also called multiple linear regression.
In order to implement this,
there's a really neat trick called vectorization,
which will make it much simpler to implement
this and many other learning algorithms.

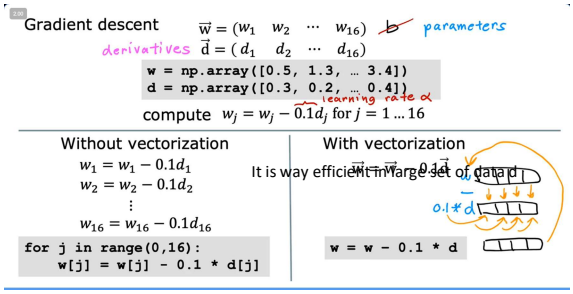
From <https://www.coursera.org/learn/machine-learning/lecture/gFuSx/multiple-features>



Similarly, to access individual features of x,
you will use x0,
x1, and x2.
Many programming languages including Python
start counting from 0 rather than 1.
Now, let's look at an implementation without
vectorization for computing the model's prediction.
In codes, it will look like this.
You take each parameter
w and multiply it by his associated feature.
Now, you could write your code like this,
but what if n isn't three but instead n is a 100 or a
100,000 is both inefficient for you
the code and inefficient for your computer to compute.
Here's another way. Without using
vectorization but using a for loop.
In math, you can use
a summation operator to add all the products of w_j
and x_j for j equals 1 through
n. Then I'll cite the summation you add b at the end.
To summation goes from j equals 1 up
to and including n. For n equals 3,
j therefore goes from 1, 2 to 3.
In code, you can initialize after 0.
Then for j in range from 0 to n,
this actually makes j go from 0 to n minus 1.
From 0, 1 to 2,
you can then add to f the product of w_j times x_j.
Finally, outside the for loop, you add b.
Notice that in Python,
the range 0 to n means that j goes from 0
all the way to n minus 1 and does not include n itself.
This is written range n in Python.
But in this video, I added a 0
here just to emphasize that it starts from 0.
While this implementation is
a bit better than the first one,
this still doesn't use factorization,
and isn't that efficient?
Now, let's look at how you can
do this using vectorization.
This is the math expression of the function f,
which is the dot product of w and x plus b,
and now you can implement this
with a single line of code by
computing fp equals np dot dot,
I said dot dot because the first dot is the period and
the second dot is the function or the method called DOT.
But is fp equals np dot dot w comma x and
this implements the mathematical dot products
between the vectors w and x.
Then finally, you can add b to it at the end.
This NumPy dot function is a vectorized implementation of
the dot product operation between
two vectors and especially when n is large,
this will run much faster
than the two previous code examples.
I want to emphasize that vectorization
actually has two distinct benefits.
First, it makes code shorter,
is now just one line of code. Isn't that cool?
Second, it also results in your code running much faster
than either of the two previous implementations
that did not use vectorization.
The reason that the vectorized implementation
is much faster is behind the scenes.
The NumPy dot function is
able to use parallel hardware in
your computer and this is true whether
you're running this on a normal computer,
that is on a normal computer CPU
or if you are using a GPU,
a graphics processor unit,
that's often used to accelerate machine learning jobs.
The ability of the NumPy dot function
to use parallel hardware makes it much more
efficient than the for loop or
the sequential calculation that we saw previously.
Now, this version is much more
practical when n is large because you are
not typing w0 times x0 plus w1 times x1
plus lots of additional terms
like you would have had for the previous version.
But while this saves a lot on the typing,
is still not that computationally
efficient because it still doesn't use vectorization.

From <https://www.coursera.org/learn/machine-learning/lecture/ismjc/vectorization-part-1>

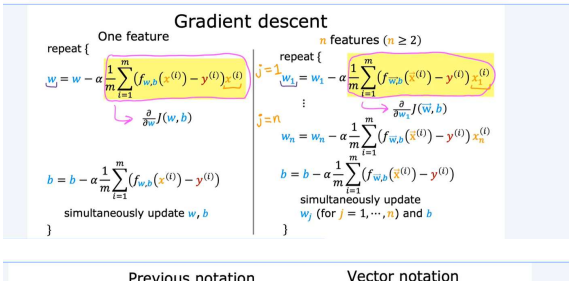




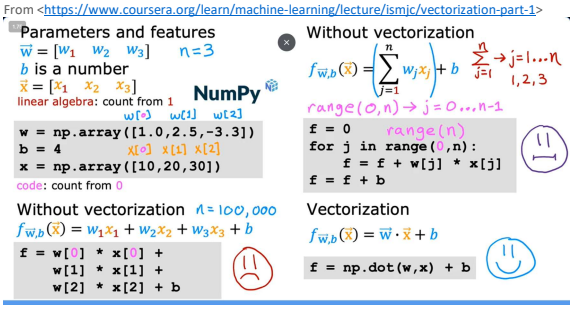
ue to parallel processing hardware

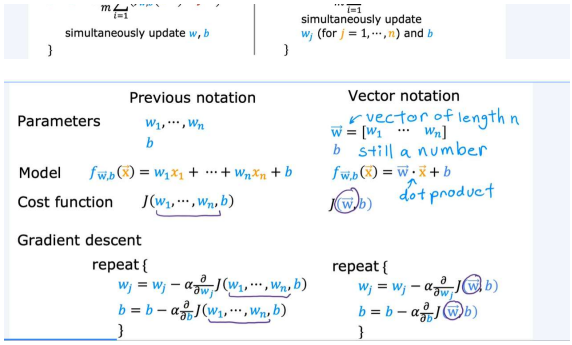
his term here is the derivative of the cost function J
with respect to the parameter w. Similarly,
we have an update rule for parameter b,
with univariate regression, we had only one feature.
We call that feature xi without any subscript.
Now, here's a new notation for where we have n features,
where n is two or more.
We get this update rule for gradient descent.
Update w_1 to be w_1 minus
Alpha times this expression here
and this formula is actually
the derivative of the cost J with respect to w_1.
The formula for the derivative of
J with respect to w_1 on
the right looks very similar to
the case of one feature on the left.
The error term still takes
a prediction f of x minus the target y.
One difference is that w and x are now vectors
and just as w on the left
has now become w_1 here on the right,
xi here on the left is now instead xi _1 here on
the right and this is just for J equals 1.
For multiple linear regression,
we have J ranging from 1 through n and
so we'll update the parameters w_1,
w_2, all the way up to w_n,
and then as before, we'll update b.
If you implement this,
you get gradient descent for multiple regression.
That's it for gradient descent for multiple regression.

From <https://www.coursera.org/learn/machine-learning/lecture/ltMMp/gradient-descent-for-multiple-linear-regression>



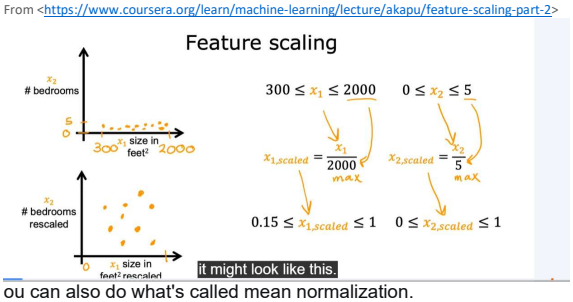
From <https://www.coursera.org/learn/machine-learning/lecture/ismjc/vectorization-part-1>





How do you actually scale features?
Well, if x_1 ranges from 3-2,000,
one way to get a scale version of x_1 is to take
each original x1_ value and divide by 2,000,
the maximum of the range.
The scale x_1 will range from 0.15 up to one.
Similarly, since x_2 ranges from 0-5,
you can calculate a scale version of x_2 by
taking each original x_2 and dividing by five,
which is again the maximum.
So the scale is x_2 will now range from 0-1.
If you plot the scale to x_1 and x_2 on a graph,
it might look like this.

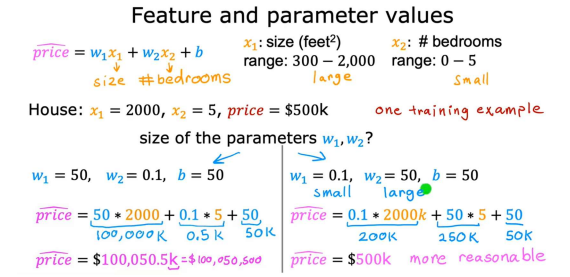
From <https://www.coursera.org/learn/machine-learning/lecture/akapu/feature-scaling-part-2>



ou can also do what's called mean normalization.
What this looks like is,
you start with the original features and then you
re-scale them so that both
of them are centered around zero.
Whereas before they only had values greater than zero,
now they have both negative and positive values
that may be usually between negative one and plus one.
To calculate the mean normalization of x_1,
first find the average,
also called the mean of x_1 on your training set,
and let's call this mean Mu_1,
with this being the Greek alphabets Mu.
For example, you may find that the average of feature 1,
Mu_1 is 600 square feet.
Let's take each x_1,
subtract the mean Mu_1,
and then let's divide by the difference 2,000 minus 300,
where 2,000 is the maximum and 300 the minimum,
and if you do this,
you get the normalized x_1 to
range from negative 0.18-0.82.
Similarly, to mean normalized x_2,
you can calculate the average of feature 2.
For instance, Mu_2 may be 2.3.
Then you can take each x_2,
subtract Mu_2 and divide by 5 minus 0.
Again, the max 5 minus the mean, which is 0.
The mean normalized x_2 now ranges
from negative 0.46-0 54.
If you plot the training data
using the mean normalized x_1 and x_2,
it might look like this.
There's one last common re-scaling
method call Z-score normalization.
To implement Z-score normalization,
you need to calculate something called

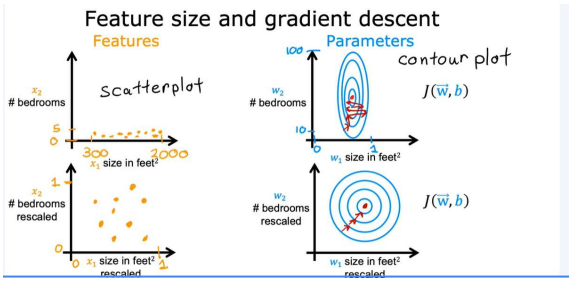
So hopefully you might notice that when a possible range of values of a feature
is large, like the size and square feet which goes all the way up to 2000.
It's more likely that a good model will learn to choose a relatively small
parameter value, like 0.1.
Likewise, when the possible values of the feature are small,
like the number of bedrooms, then a reasonable value for
its parameters will be relatively large like 50.

From <https://www.coursera.org/learn/machine-learning/lecture/KMDV3/feature-scaling-part-1>



This is what might end up happening if you were to run great in dissent,
if you were to use your training data as is.
Because the contours are so tall and
skinny gradient descent may end up bouncing back and forth for
a long time before it can finally find its way to the global minimum.
In situations like this, a useful thing to do is to scale the features.
This means performing some transformation of your training data so
that x1 say might now range from 0 to 1 and x2 might also range from 0 to 1.
So the data points now look more like this and you might notice that the scale of
the plot on the bottom is now quite different than the one on top.
Play video starting at :5:45 and follow transcript5:45
The key point is that the re scale x1 and
x2 are both now taking comparable ranges of values to each other.
And if you run gradient descent on a cost function to find on this,
re scaled x1 and x2 using this transformed data, then the contours
will look more like this more like circles and less tall and skinny.
And gradient descent can find a much more direct path to the global minimum.

From <https://www.coursera.org/learn/machine-learning/lecture/KMDV3/feature-scaling-part-1>



et's take a look. As a reminder,
here's the gradient descent rule.
One of the key choices is
the choice of the learning rate Alpha.
Here's something that I often do to make
sure that gradient descent is working well.
Recall that the job of
gradient descent is to find parameters w
and b that hopefully minimize the cost function J.

it might look like this.
There's one last common re-scaling
method call Z-score normalization.
To implement Z-score normalization,
you need to calculate something called
the standard deviation of each feature.

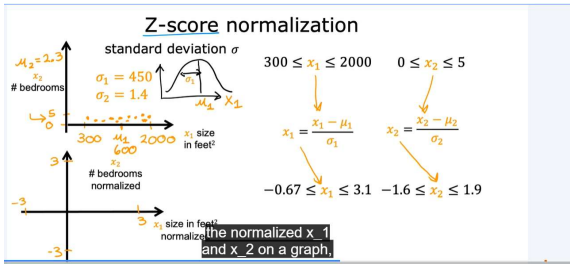
From <https://www.coursera.org/learn/machine-learning/lecture/akapu/feature-scaling-part-2>

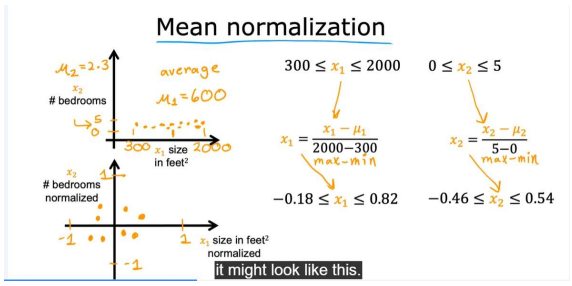
Or if you've heard of
the normal distribution or the bell-shaped curve,
sometimes also called the Gaussian distribution,
this is what the standard deviation
for the normal distribution looks like.
But if you haven't heard of this,
you don't need to worry about that either.
But if you do know what is the standard deviation,
then to implement a Z-score normalization,
you first calculate the mean Mu,
as well as the standard deviation,
which is often denoted by
the lowercase Greek alphabet Sigma of each feature.
For instance, maybe feature 1 has
a standard deviation of 450 and mean 600,
then to Z-score normalize x_1,
take each x_1,
subtract Mu_1, and
then divide by the standard deviation,
which I'm going to denote as Sigma 1.
What you may find is that the Z-score normalized
x_1 now ranges from negative 0.67-3.1.
Play video starting at :4:9 and follow transcript

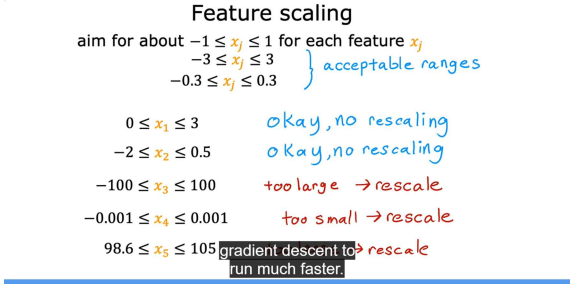
From <https://www.coursera.org/learn/machine-learning/lecture/akapu/feature-scaling-part-2>

Similarly, if you calculate the
second features standard deviation
to be 1.4 and mean to be 2.3,
then you can compute x_2 minus Mu_2 divided by Sigma_2,
and in this case,
the Z-score normalized by x_2 might now
range from negative 1.6-1.9.
If you plot the training data on
the normalized x_1 and x_2 on a graph,
it might look like this.

From <https://www.coursera.org/learn/machine-learning/lecture/akapu/feature-scaling-part-2>







sure that gradient descent is working well.
Recall that the job of
gradient descent is to find parameters w
and b that hopefully minimize the cost function J.
What I'll often do is plot the cost function J,
which is calculated on the training set,
and I plot the value of J at
each iteration of gradient descent.
Remember that each iteration means after
each simultaneous update of the parameters w and b.

From <https://www.coursera.org/learn/machine-learning/lecture/rOTkB/checking-gradient-descent-for-convergence>
This differs from previous graphs you've
seen where the vertical axis was cost
J and the horizontal axis was
a single parameter like w or b.
This curve is also called a learning curve.
Note that there are
a few different types of learning
curves used in machine learning,

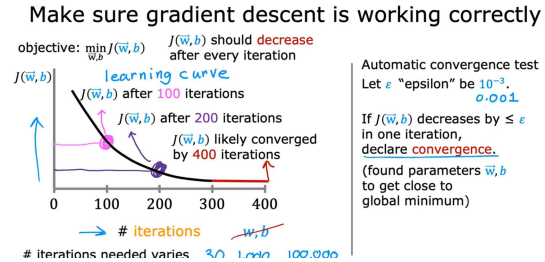
From <https://www.coursera.org/learn/machine-learning/lecture/rOTkB/checking-gradient-descent-for-convergence>

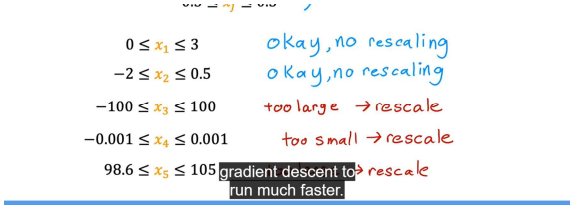
If J ever increases after one iteration,
that means either Alpha is chosen poorly,
and it usually means Alpha is too large,
or there could be a bug in the code.
Another useful thing that this part can tell
you is that if you look at this curve,
by the time you reach maybe 300 iterations also,
the cost J is leveling
off and is no longer decreasing much.
By 400 iterations,
it looks like the curve has flattened out.
This means that gradient descent has more or less
converged because the curve is no longer decreasing.
Looking at this learning curve,
you can try to spot whether or not
gradient descent is converging.
By the way, the number
of iterations that gradient descent
takes a conversion can vary
a lot between different applications.
In one application, it may
converge after just 30 iterations.
For a different application,
it could take 1,000 or 100,000 iterations.
It turns out to be very difficult to tell in
advance how many iterations
gradient descent needs to converge,
which is why you can create
a graph like this, a learning curve.

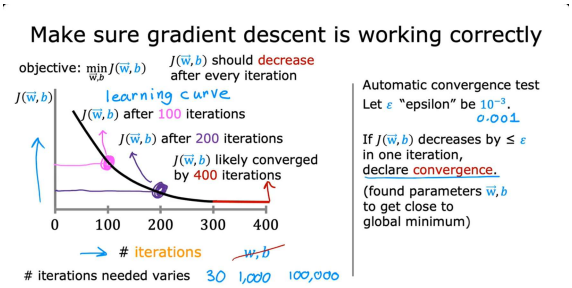
From <https://www.coursera.org/learn/machine-learning/lecture/rOTkB/checking-gradient-descent-for-convergence>

Another way to decide when your model is done training
is with an automatic convergence test.
Play video starting at :4:29 and follow transcript4:29
Here is the Greek alphabet epsilon.
Let's let epsilon be a
variable representing a small number,
such as 0.001 or 10^-3.
If the cost J decreases by less
than this number epsilon on one iteration,
then you're likely on this flattened part of
the curve that you see on
the left and you can declare convergence.
Remember, convergence,
hopefully in the case that you found parameters
w and b that are close to the minimum possible value of
J. I usually find
that choosing the right threshold
epsilon is pretty difficult.
I actually tend to look at graphs
like this one on the left,
rather than rely on automatic convergence tests.

From <https://www.coursera.org/learn/machine-learning/lecture/rOTkB/checking-gradient-descent-for-convergence>







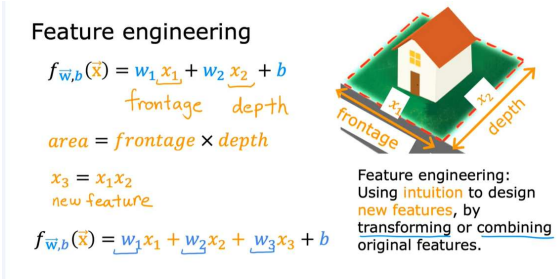
if the learning rate is too big,
then if you start off here,
your update step may overshoot
the minimum and end up here,
and in the next update step here,
your gain overshooting so you end up here and so on.
That's why the cost can sometimes go
up instead of decreasing.
To fix this, you can use a smaller learning rate.
Then your updates may start
here and go down a little bit and down a bit,
and we'll hopefully consistently
decrease until it reaches the global minimum.
Sometimes you may see that the cost
consistently increases after each iteration,
like this curve here.
This is also likely due to
a learning rate that is too large,
and it could be addressed by
choosing a smaller learning rate.
But learning rates like this could
also be a sign of a possible broken code.
For example, if I wrote my code so that w_1 gets
updated as w_1 plus Alpha times this derivative term,
this could result in the cost consistently
increasing at each iteration.
This is because having the derivative term moves
your cost J further from
the global minimum instead of closer.
So remember, you want to use in minus sign,
so the code should be updated w_1 updated
by w_1 minus Alpha times the derivative term.
One debugging tip for a correct implementation of
gradient descent is that
with a small enough learning rate,
the cost function should
decrease on every single iteration.
So if gradient descent isn't working,
one thing I often do
and I hope you find this tip useful too,
one thing I'll often do is just set Alpha to be
a very small number and see if that
causes the cost to decrease on every iteration.
If even with Alpha set to a very small number,
J doesn't decrease on every single iteration,
but instead sometimes increases,
then that usually means
there's a bug somewhere in the code.
Note that setting Alpha
to be really small is meant here as
a debugging step and a very small value of Alpha
is not going to be the most efficient choice
for actually training your learning algorithm.
One important trade-off is
that if your learning rate is too small,
then gradient descents can take
a lot of iterations to converge.
So when I am running gradient descent,
I will usually try a range of
values for the learning rate Alpha

From <https://www.coursera.org/learn/machine-learning/lecture/10ZVv/choosing-the-learning-rate>
What I'll do is try a range of
values until I found the value of that's too
small and then also make
sure I've found a value that's too large.
I'll slowly try to
pick the largest possible learning rate,
or just something slightly smaller than
the largest reasonable value that I found.
When I do that, it usually gives

In fact, for many practical applications,
choosing or entering the right features is
a critical step to making the algorithm work well.
In this video, let's take a look at how you can choose or
engineer the most appropriate features
for your learning algorithm.
Let's take a look at feature engineering by revisiting
the example of predicting the price of a house.

From <https://www.coursera.org/learn/machine-learning/lecture/dgZYR/feature-engineering#>
You might notice that the area of
the land can be calculated
as the frontage or width times the depth.
You may have an intuition that
the area of the land is more predictive of the price,
than the frontage and depth as separate features.
You might define a new feature,
x_3, as x_1 times x_2.
This new feature x_3 is
equal to the area of the plot of land.
With this feature, you can then have a model f_w,
b of x equals w_1x_1 plus w_2x_2
plus w_3x_3 plus b
so that the model can now choose parameters w_1,
w_2, and w_3,
depending on whether the data shows that
the frontage or the depth or the area
x_3 of the lot turns out to be
the most important thing for
predicting the price of the house.
What we just did, creating a new feature
is an example of what's called feature engineering,
in which you might use your knowledge or
intuition about the problem to design new features
usually by transforming or
combining the original features of
the problem in order to make it
easier for the learning algorithm
to make accurate predictions.
Depending on what insights you
may have into the application,
rather than just taking
the features that you happen to have
started off with sometimes by defining new features,
you might be able to get a much better model.
That's feature engineering

From <https://www.coursera.org/learn/machine-learning/lecture/dgZYR/feature-engineering#>



me a good learning rate for my model.
I hope this technique too
will be useful for you to choose
a good learning rate for
your implementation of gradient descent.

From <https://www.coursera.org/learn/machine-learning/lecture/10ZVv/choosing-the-learning-rate>