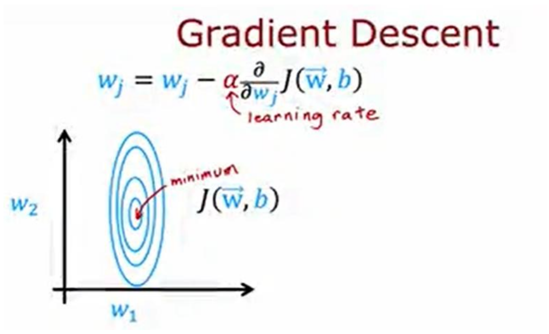
Advanced optimization

Gradient descent is an optimization algorithm that is widely used in machine learning, and was the foundation of many algorithms like linear regression and logistic regression and early implementations of neural networks. But it turns out that there are now some other optimization algorithms for minimizing the cost function, that are even better than gradient descent.

In this video, we'll take a look at an algorithm that can help you train your neural network much faster than gradient descent. Recall that this is the expression for one step of gradient descent. A parameter w_j is updated as w_j minus the learning rate Alpha times this partial derivative term. How can we make this work even better? In this example, I've plotted the cost function J using a contour plot including these ellipsis, and the minimum of this cost function is at the center of this ellipsis down here.

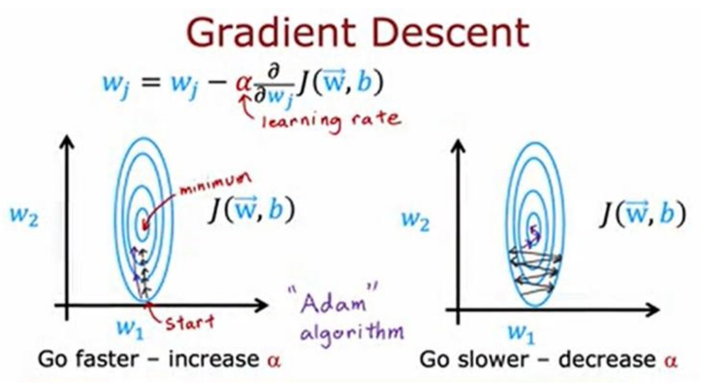


Gradient Descent
$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$
learning rate
minimum
$J(\vec{w}, b)$
w_2
w_1

Now, if you were to start gradient descent down here, one step of gradient descent, if Alpha is small, may take you a little bit in that direction. Then another step, then another step, then another step, then another step, and you notice that every single step of gradient descent is pretty much going in the same direction, and if you see this to be the case, you might wonder, well, why don't we make Alpha bigger, can we have an algorithm to automatically increase Alpha?

They just make it take bigger steps and get to the minimum faster. There's an algorithm called the Adam algorithm that can do that. If it sees that the learning rate is too small, and we are just taking tiny little steps in a similar direction over and over, we should just make the learning rate Alpha bigger.

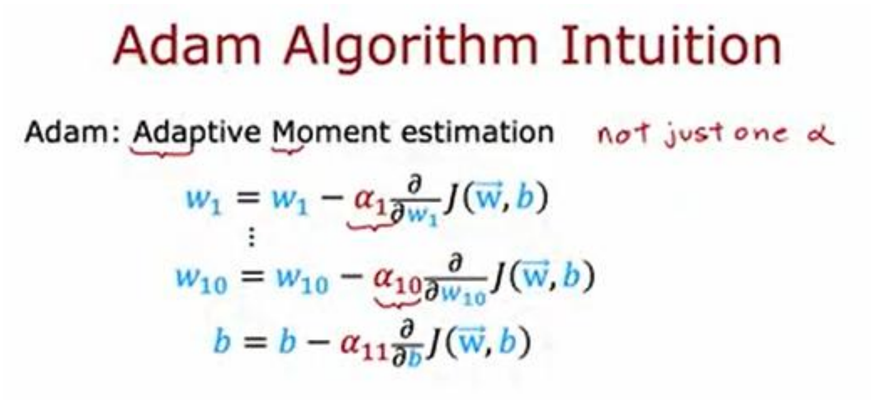
In contrast, here again, is the same cost function if we were starting here and have a relatively big learning rate Alpha, then maybe one step of gradient descent takes us here, in the second step takes us here, third step, and the fourth step, and the fifth step, and the sixth step, and if you see gradient descent doing this, is oscillating back and forth. You'd be tempted to say, well, why don't we make the learning rates smaller?



Gradient Descent
$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$
learning rate
minimum
$J(\vec{w}, b)$
w_2
w_1
start
"Adam" algorithm
Go faster – increase α
$J(\vec{w}, b)$
w_2
w_1
Go slower – decrease α

The Adam algorithm can also do that automatically, and with a smaller learning rate, you can then take a more smooth path toward the minimum of the cost function. Depending on how gradient descent is proceeding, sometimes you wish you had a bigger learning rate Alpha, and sometimes you wish you had a smaller learning rate Alpha.

The Adam algorithm can adjust the learning rate automatically. Adam stands for Adaptive Moment Estimation, or A-D-A-M, and don't worry too much about what this name means, it's just what the authors had called this algorithm. But interestingly, the Adam algorithm doesn't use a single global learning rate Alpha. It uses a different learning rates for every single parameter of your model. If you have parameters w_1 through w_10, as was b, then it actually has 11 learning rate parameters, Alpha_1, Alpha_2, all the way through Alpha_10 for w_1 to w_10, as well as I'll call it Alpha_11 for the parameter b.



Adam Algorithm Intuition

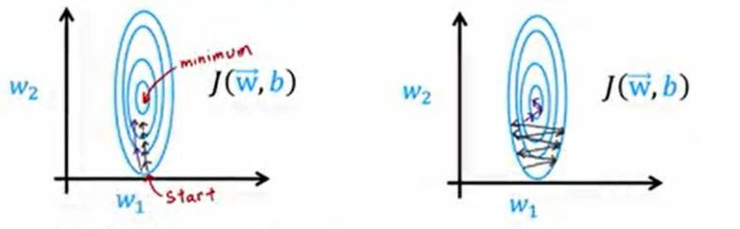
Adam: Adaptive Moment estimation   not just one α

$$w_1 = w_1 - \alpha_1 \frac{\partial}{\partial w_1} J(\vec{w}, b)$$
$$\vdots$$
$$w_{10} = w_{10} - \alpha_{10} \frac{\partial}{\partial w_{10}} J(\vec{w}, b)$$
$$b = b - \alpha_{11} \frac{\partial}{\partial b} J(\vec{w}, b)$$

The intuition behind the Adam algorithm is, if a parameter w_j, or b seems to keep on moving in roughly the same direction. This is what we saw on the first example on the previous slide. But if it seems to keep on moving in roughly the same direction, let's increase the learning rate for that parameter.

Let's go faster in that direction. Conversely, if a parameter keeps oscillating back and forth, this is what you saw in the second example on the previous slide. Then let's not have it keep on oscillating or bouncing back and forth. Let's reduce Alpha_j for that parameter a little bit.

Adam Algorithm Intuition



If w_j (or b) keeps moving in same direction, increase α_j.

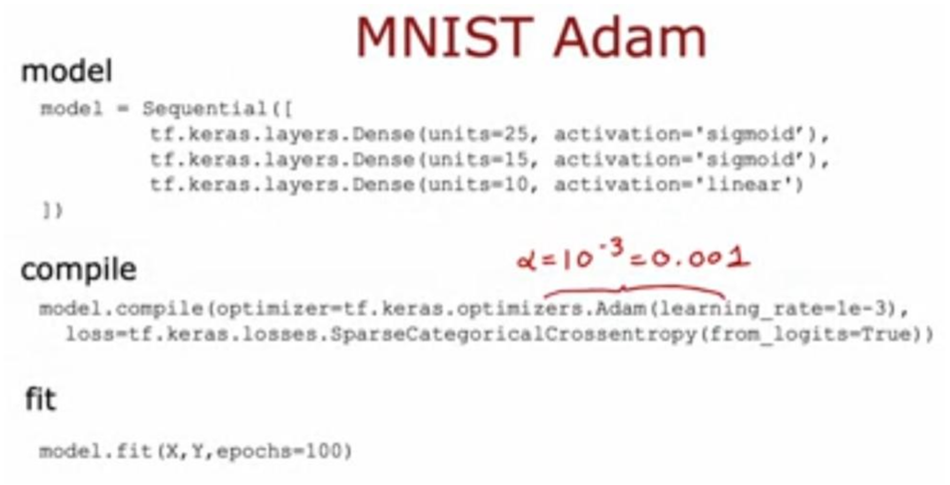
If w_j (or b) keeps oscillating, reduce α_j.

The details of how Adam does this are a bit complicated and beyond the scope of this course, but if you take some more advanced deep learning courses later, you may learn more about the details of this Adam algorithm, but in codes this is how you implement it.

The model is exactly the same as before, and the way you compile the model is very similar to what we had before, except that we now add one extra argument to the compile function, which is that we specify that the optimizer you want to use is tf.keras.optimizers.Adam optimizer.

The Adam optimization algorithm does need some default initial learning rate Alpha, and in this example, I've set that initial learning rate to be 10^ negative 3. But when you're using the Adam algorithm in practice, it's worth trying a few values for this default global learning rate. Try some large and some smaller values to see what gives you the fastest learning performance.

Compared to the original gradient descent algorithm that you had learned in the previous course though, the Adam algorithm, because it can adapt the learning rate a bit automatically, it is more robust to the exact choice of learning rate that you pick. Though there's still way tuning this parameter little bit to see if you can get somewhat faster learning.



MNIST Adam

model

```
model = Sequential([
        tf.keras.layers.Dense(units=25, activation='sigmoid'),
        tf.keras.layers.Dense(units=15, activation='sigmoid'),
        tf.keras.layers.Dense(units=10, activation='linear')
])
```

compile

$\alpha = 10^{-3} = 0.001$

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True))
```

fit

```
model.fit(X,Y,epochs=100)
```

That's it for the Adam optimization algorithm. It typically works much faster than gradient descent, and it's become a de facto standard in how practitioners train their neural networks. If you're trying to decide what learning algorithm to use, what optimization algorithm to use to train your neural network.

A safe choice would be to just use the Adam optimization algorithm, and most practitioners today will use Adam rather than the optional gradient descent algorithm, and with this, I hope that your learning algorithms will be able to learn much more quickly. Now, in the next couple of videos, I'd like to touch on some more advanced concepts for neural networks, and in particular, in the next video, let's take a look at some alternative layer types.