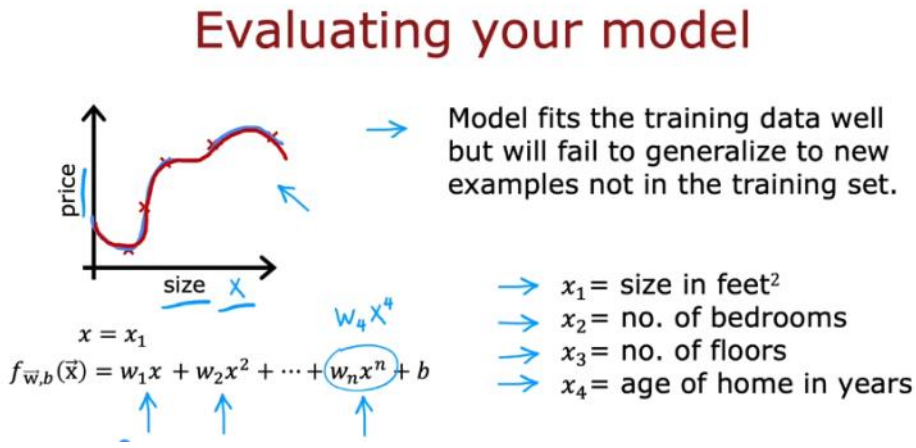
Module 3

05 December 2025 10:12

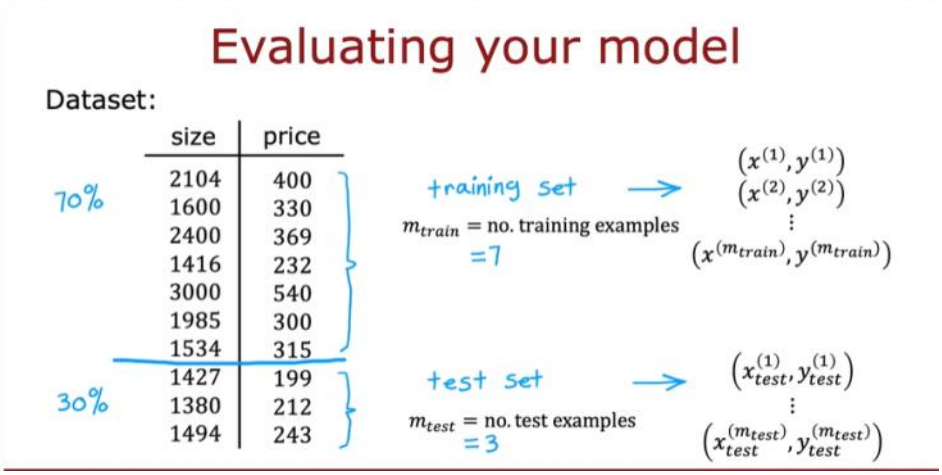
Let's see, you've trained a machine learning model. How do you evaluate that model's performance? You find that having a systematic way to evaluate performance will also hope paint a clearer path for how to improve its performance. So let's take a look at how to evaluate the model. Let's take the example of learning to predict housing prices as a function of the size. Let's say you've trained the model to predict housing prices as a function of the size x. And for the model that is a fourth order polynomial.
So features x, x squared, execute and x to the 4. Because we fit 1/4 order polynomial to a training set with five data points, this fits the training data really well. But, we don't like this model very much because even though the model fits the training data well, we think it will fail to generalize to new examples that aren't in the training set. So, when you are predicting prices, just a single feature at the size of the house, you could plot the model like this and we could see that the curve is very wiggly so we know this is probably isn't a good model. But if you were fitting this model with even more features, say we had x1 the size of house, number of bedrooms, the number of floors of the house, also the age of the home in years, then it becomes much harder to plot f because f is now a function of x1 through x4. And how do you plot a four dimensional function? So in order to tell if your model is doing well, especially for applications where you have more than one or two features, which makes it difficult to plot f of x.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/26yGi/evaluating-a-model>



From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/26yGi/evaluating-a-model>
We need some more systematic way to evaluate how well your model is doing. Here's a technique that you can use. If you have a training set and this is a small training set with just 10 examples listed here, rather than taking all your data to train the parameters w and p of the model, you can instead split the training set into two subsets. I'm going to draw a line here, and let's put 70% of the data into the first part and I'm going to call that the training set. And the second part of the data, let's say 30% of the data, I'm going to put into it a test set. And what we're going to do is train the models, parameters on the training set on this first 70% or so of the data, and then we'll test its performance on this test set
And it's not uncommon to split your dataset according to maybe a 70, 30 split or 80, 20 split with most of your data going into the training set, and then a smaller fraction going into the test set.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/26yGi/evaluating-a-model>

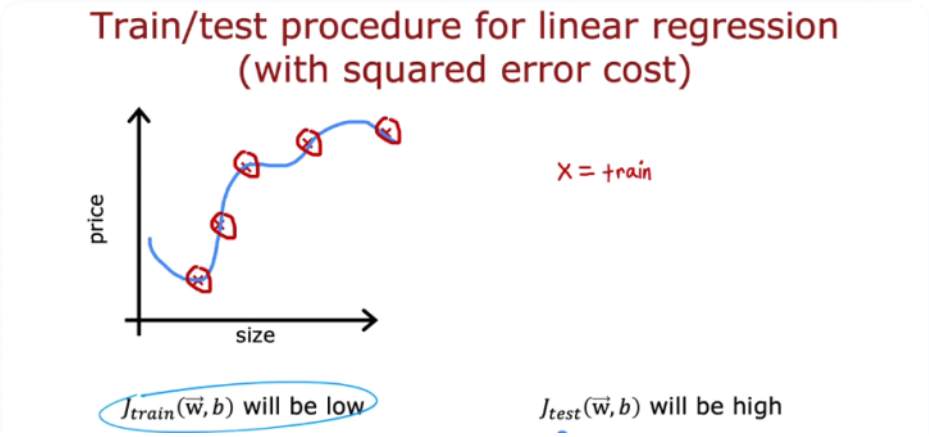


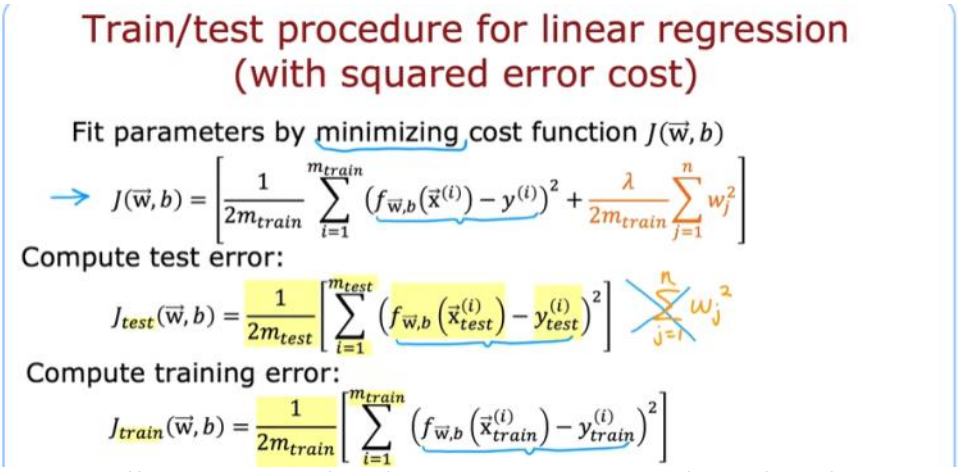
So, in order to train a model and evaluated it, this is what it would look like if you're using linear regression with a squared error cost. Start off by fitting the parameters by minimizing the cost function j of w,b. So this is the usual cost function minimize over w,b of this square error cost, plus regularization term lambda over 2m times some of the w,j squared. And then to tell how well this model is doing, you would compute J test of w,b, which is equal to the average error on the test set, and that's just equal to 1/2 times m test. That's the number of test examples. And then of some overall the examples from r equals 1, to the number of test examples of the squared era on each of the test examples like so. So it's a prediction on the i'th test example input minus the actual price of the house on the test example squared

And notice that the test error formula J test, it does not include that regularization term. And this will give you a sense of how well your learning algorithm is doing. One of the quantity that's often useful to computer as well as the training error, which is a measure of how well you're learning algorithm is doing on the training set. So let me define J train of w,b to be equal to the average over the training set. 1 over to 2m, or 1/2 m subscript train of some over your training set of this squared error term. And once again, this does not include the regularization term unlike the cost function that you are minimizing to fit the parameters. So, in the model like what we saw earlier in this video, J train of w,b will be low because the average era on your training examples will be zero or very close to zero. So J train will be very close to zero.

But if you have a few additional examples in your test set that the album had not trained on, then those test examples, might look like these. And there's a large gap between what the album is predicting as the estimated housing price, and the actual value of those housing prices. And so, J tests will be high. So seeing that J test is high on this model, gives you a way to realize that even though it does great on the training set, is actually not so good at generalizing to new examples to new data points that were not in the training set.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/26yGi/evaluating-a-model>





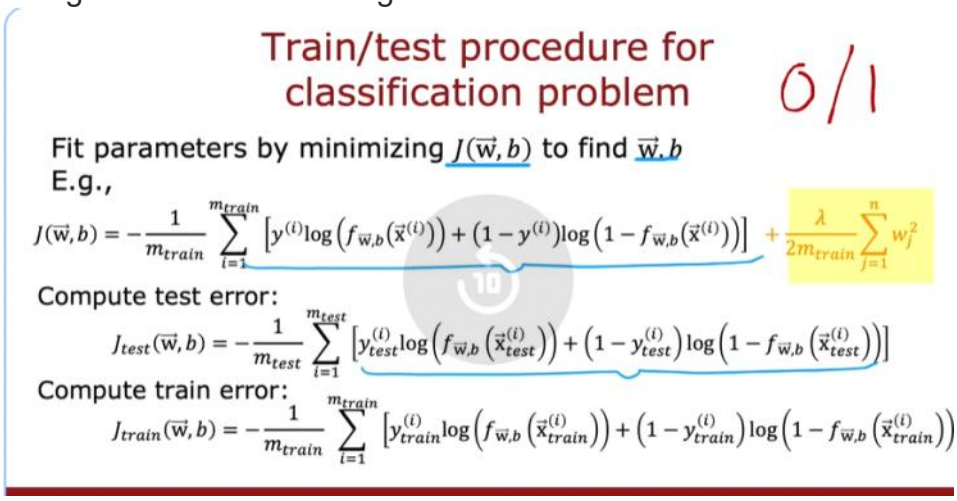
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/26yGi/evaluating-a-model>

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/26yGi/evaluating-a-model>

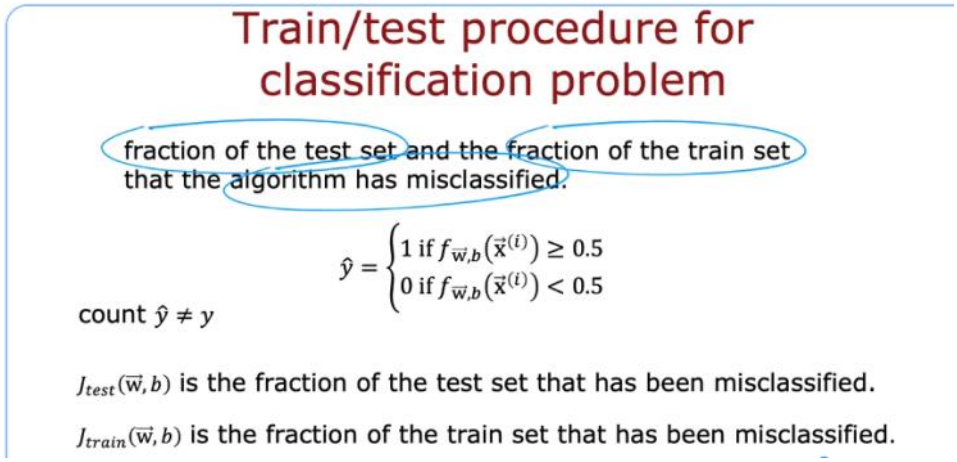
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/26yGi/evaluating-a-model>

So, that was regression with squared error cost. Now, let's take a look at how you apply this procedure to a classification problem. For example, if you are classifying between handwritten digits that are either 0 or 1,, so same as before, you fit the parameters by minimizing the cost function to find the parameters w,b.
For example, if you were training logistic regression, then this would be the cost function J of w,b, where this is the usual logistic loss function, and then plus also the regularization term. And to compute the test error, J test is then the average over your test examples, that's that 30% of your data that wasn't in the training set of the logistic loss on your test set. And the training error you can also compute using this formula, is the average logistic loss on your training data that the album was using to minimize the cost function J of w, b. Well, when I described here will work, okay, for figuring out if your learning algorithm is doing well, by seeing how I was doing in terms of test error. When applying machine learning to classification problems, there's actually

one other definition of J tests and J train that is maybe even more commonly used. Which is instead of using the logistic loss to compute the test error and the training error to instead measure what the fraction of the test set, and the fraction of the training set that the algorithm has misclassified
So, recall y_hat we would predict us 1 if f of x is greater or equal 0.5, and zero if it's less than 0.5. And you can then count up in the test set the fraction of examples where y_hat is not equal to the actual ground truth label while in the test set. So concretely, if you are classifying handwritten digits 0, 1 binary classification loss, then J tests would be the fraction of that test set, where 0 was classified as 1 of 1, classified as 0. And similarly, J train is a fraction of the training set that has been misclassified. Taking a dataset and splitting it into a training set and a separate test set gives you a way to systematically evaluate how well your learning algorithm is doing. By computing both J tests and J train, you can now measure how was doing on the test set and on the training set. This procedure is one step to what you'll be able to automatically choose what model to use for a given machine learning a



pplication.



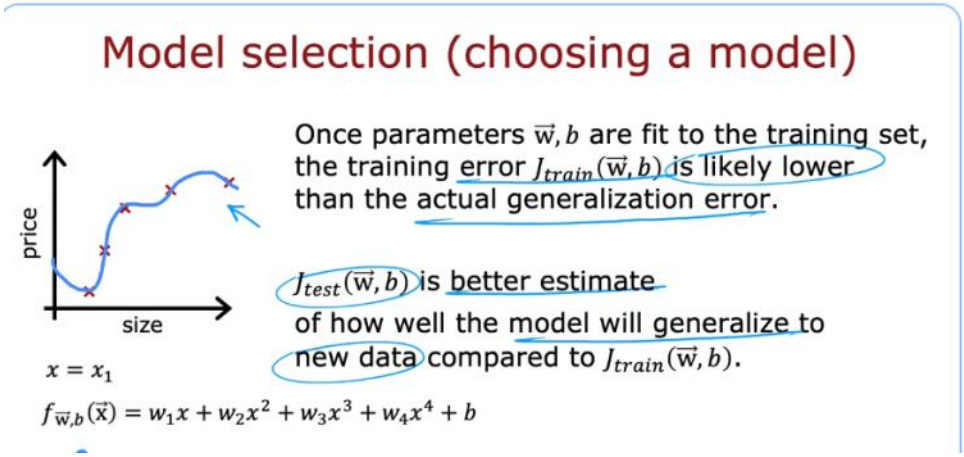
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/26yGi/evaluating-a-model>

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/26yGi/evaluating-a-model>

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/26yGi/evaluating-a-model>

One thing we've seen is that once the model's parameters w and b have been fit to the training set. The training error may not be a good indicator of how well the algorithm

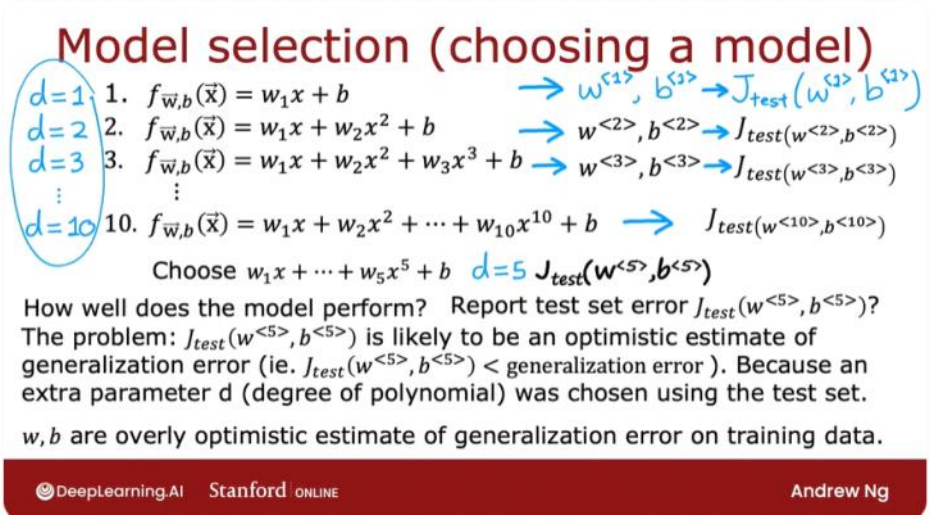
will do or how well it will generalize to new examples that were not in the training set, and in particular, for this example, the training error will be pretty much zero. That's likely much lower than the actual generalization error, and by that I mean the average error on new examples that were not in the training set. What you saw on the last video is that J test the performance of the algorithm on examples, is not trained on, that will be a better indicator of how well the model will likely do on new data. By that I mean other data that's not in the training set.



You can go on to try d equals 3, that's a third order or a degree three polynomial that looks like this, and fit parameters and similarly get J test. You might keep doing this until, say you try up to a 10th order polynomial and you end up with J test of w^10, b^10. That gives you a sense of how well the 10th order polynomial is doing. One procedure you could try, this turns out not to be the best procedure, but one thing you could try is, look at all of these J tests, and see which one gives you the lowest value. Say, you find that, J test for the fifth order polynomial for w^5, b^5 turns out to be the lowest. If that's the case, then you might decide that the fifth order polynomial d equals 5 does best, and choose that model for your application. If you want to estimate how well this model performs, one thing you could do, but this turns out to be a slightly flawed procedure, is to report the test set error, J test w^5, b^5.

The reason this procedure is flawed is J test of w^5, b^5 is likely to be an optimistic estimate of the generalization error. In other words, it is likely to be lower than the actual generalization error, and the reason is, in the procedure we talked about on this slide with basic fits, one extra parameter, which is d, the degree of polynomial, and we chose this parameter using the test set. On the previous slide, we saw that if you were to fit w, b to the training data, then the training data would be an overly optimistic estimate of generalization error. It turns out too, that if you want to choose the parameter d using the test set, then the test set J test is now an overly optimistic, that is lower than actual estimate of the generalization error. The procedure on this particular slide is flawed and I don't recommend using this. Instead, if you want to automatically choose a model, such as decide what degree polynomial to use. Here's how you modify the training and testing procedure in order to carry out model selection.

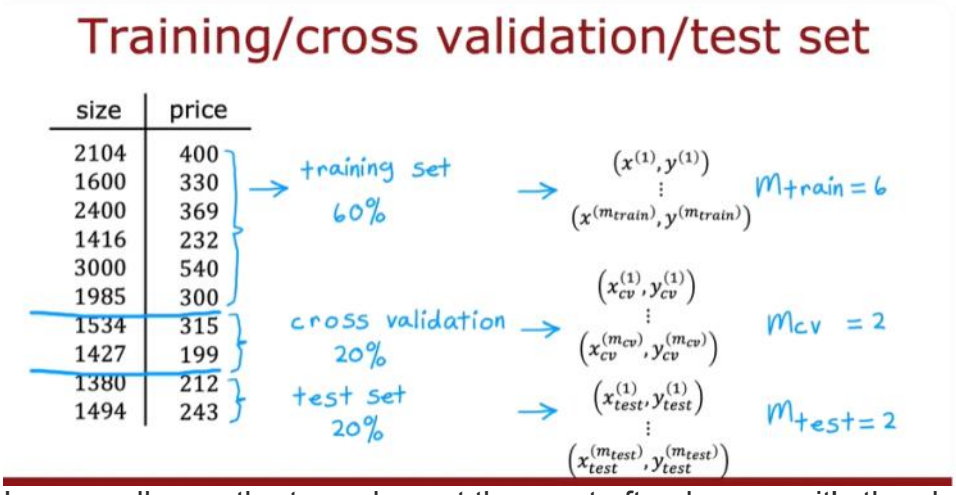
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/zqXm6/model-selection-and-training-cross-validation-test-sets>



From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/zqXm6/model-selection-and-training-cross-validation-test-sets>

The way we'll modify the procedure is instead of splitting your data into just two subsets, the training set and the test set, we're going to split your data into three different subsets, which we're going to call the training set, the cross-validation set, and then also the test set.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/zqXm6/model-selection-and-training-cross-validation-test-sets>



I personally use the term dev set the most often because it's the shortest, fastest way to say it but cross-validation is pretty used a little bit more often by machine learning practitioners. Onto these three subsets of the data training set, cross-validation set, and test set, you can then compute the training error, the cross-validation error, and the test error using these three formulas. Whereas usual, none of these terms include the regularization term that is included in the training objective, and this new term in the middle, the cross-validation error is just the average over your m_cv cross-validation examples of the average say, squared error.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/zqXm6/model-selection-and-training-cross-validation-test-sets>

Training/cross validation/test set

Training error: $J_{train}(\vec{w}, b) = \frac{1}{2m_{train}}\left[\sum_{i=1}^{m_{train}} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2\right]$

Cross validation error: $J_{cv}(\vec{w}, b) = \frac{1}{2m_{cv}}\left[\sum_{i=1}^{m_{cv}} \left(f_{\vec{w},b}(\vec{x}_{cv}^{(i)}) - y_{cv}^{(i)}\right)^2\right]$ (validation error, dev error)

Test error: $J_{test}(\vec{w}, b) = \frac{1}{2m_{test}}\left[\sum_{i=1}^{m_{test}} \left(f_{\vec{w},b}(\vec{x}_{test}^{(i)}) - y_{test}^{(i)}\right)^2\right]$

Then, in order to choose a model, you will look at which model has the lowest cross-validation error, and concretely, let's say that J_cv of w4, b4 as low as, then what that means is you pick this fourth-order polynomial as the model you will use for this application. Finally, if you want to report out an estimate of the generalization error of how well this model will do on new data. You will do so using that third subset of your data, the test set and you report out Jtest of w4,b4. You notice that throughout this entire procedure, you had fit these parameters using the training set. You then chose the parameter d or chose the degree of polynomial using the cross-validation set and so up until this point, you've not fit any parameters, either w or b or d to the test set and that's why Jtest in this example will be fair estimate of the generalization error of this model thus parameters w4,b4. This gives a better procedure for model selection and it lets you automatically make a decision like what order polynomial to choose for your linear regression model. This model selection procedure also works for choosing among other types of models.

Model selection

$d=1$ 1. $f_{\vec{w},b}(\vec{x}) = w_1x + b$ $w^{<1>}, b^{<1>} \rightarrow J_{cv}(w^{<1>}, b^{<1>})$

$d=2$ 2. $f_{\vec{w},b}(\vec{x}) = w_1x + w_2x^2 + b$ $\rightarrow J_{cv}(w^{<2>}, b^{<2>})$

$d=3$ 3. $f_{\vec{w},b}(\vec{x}) = w_1x + w_2x^2 + w_3x^3 + b$

⋮ ⋮ ⋮

$d=10$ 10. $f_{\vec{w},b}(\vec{x}) = w_1x + w_2x^2 + \cdots + w_{10}x^{10} + b$ $J_{cv}(w^{<10>}, b^{<10>})$

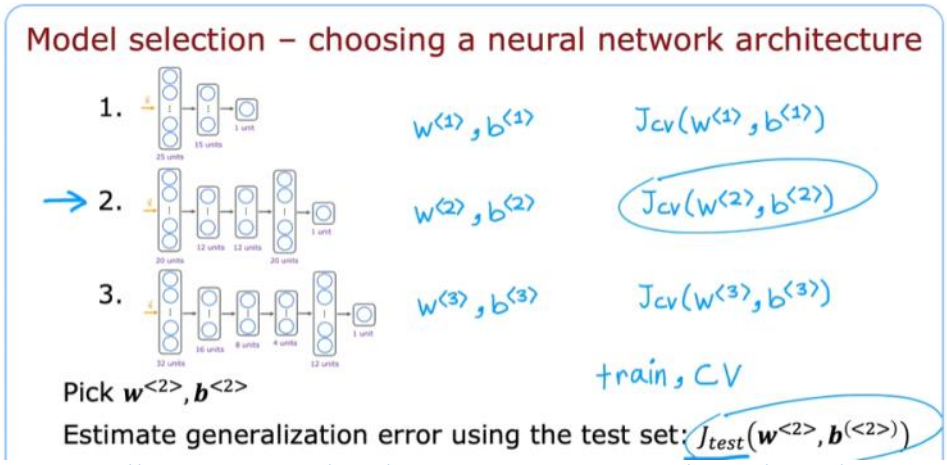
→ Pick $w_1x + \cdots + w_4x^4 + b$ $(J_{cv}(w^{<4>}, b^{<4>}))$

Estimate generalization error using test the set: $J_{test}(w^{<4>}, b^{<4>})$

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/zqXm6/model-selection-and-training-cross-validation-test-sets>

you might consider three models like this, maybe even a larger set of models than just me but here are a few different neural networks of small, somewhat larger, and then even larger. To help you decide how many layers do the neural network have and how many hidden units per layer should you have, you can then train all three of these models and end up with parameters w1, b1 for the first model, w2, b2 for the second model, and w3,b3 for the third model. You can then evaluate the neural networks performance using Jcv, using your cross-validation set Since this is a classification problem, Jcv the most common choice would be to compute this as the fraction of cross-validation examples that the algorithm has misclassified. You would compute this using all three models and then pick the model with the lowest cross validation error.

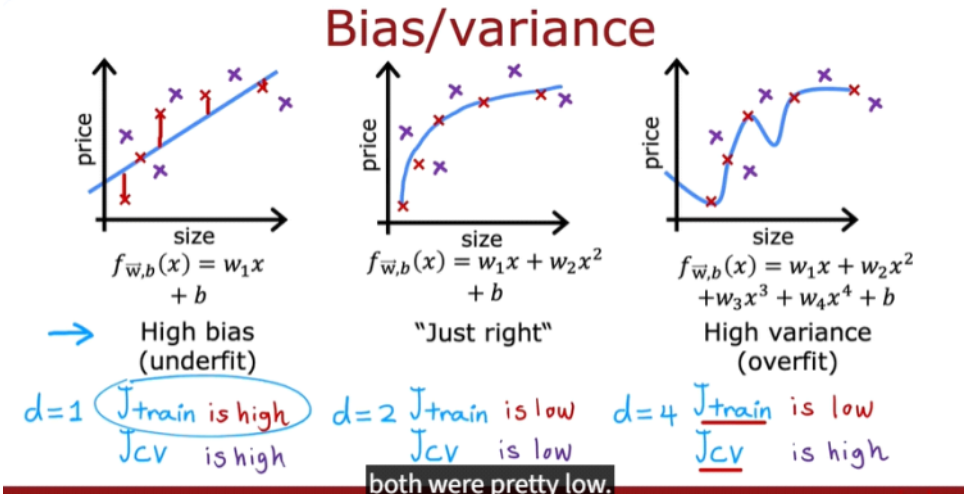
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/zqXm6/model-selection-and-training-cross-validation-test-sets>



From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/zqXm6/model-selection-and-training-cross-validation-test-sets>

We said that this algorithm has high bias or that it underfits this dataset. If you were to fit a fourth-order polynomial, then it has high-variance or it overfits. In the middle if you fit a quadratic polynomial, then it looks pretty good. Then I said that was just right. Because this is a problem with just a single feature x, we could plot the function f and look at it like this. But if you had more features, you can't plot f and visualize whether it's doing well as easily. Instead of trying to look at plots like this, a more systematic way to diagnose or to find out if your algorithm has high bias or high variance will be to look at the performance of your algorithm on the training set and on the cross validation set.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/L6SHx/diagnosing-bias-and-variance>



In particular, let's look at the example on the left. If you were to compute J_train, how well does the algorithm do on the training set? Not that well. I'd say J train here would be high because there are actually pretty large errors between the examples and the actual predictions of the model. How about J_cv? J_cv would be if we had a few new examples, maybe examples like that, that the algorithm had not previously seen. Here the algorithm also doesn't do that well on examples that it had not previously seen, so J_cv will also be high.
One characteristic of an algorithm with high bias, something that is under fitting, is that it's not even doing that well on the training set. When J_train is high, that is

your strong indicator that this algorithm has high bias. Let's now look at the example on the right. If you were to compute J_train, how well is this doing on the training set? Well, it's actually doing great on the training set. Fits the training data really well. J_train here will be low.

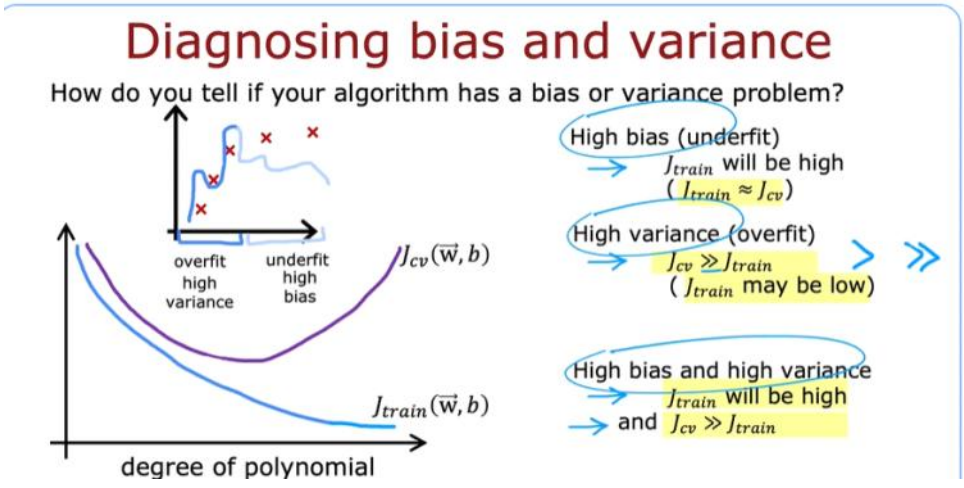
Over on the left we'll correspond to a small value of d, like d equals 1, which corresponds to fitting straight line. Over to the right we'll correspond to, say, d equals 4 or even higher values of d. We're fitting this high order polynomial. So if you were to plot J train or W, B as a function of the degree of polynomial, what you find is that as you fit a higher and higher degree polynomial, here I'm assuming we're not using regularization, but as you fit a higher and higher order polynomial, the training error will tend to go down because when you have a very simple linear function, it doesn't fit the training data that well, when you fit a quadratic function or third order polynomial or fourth-order polynomial, it fits the training data better and better.

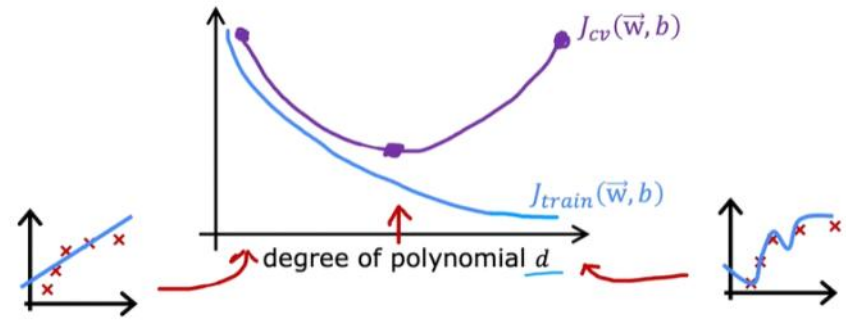
As the degree of polynomial increases, J train will typically go down. Next, let's look at J_cv, which is how well does it do on data that it did not get to fit to? What we saw was when d equals one, when the degree of polynomial was very low, J_cv was pretty high because it underfits, so it didn't do well on the cross validation set. Here on the right as well, when the degree of polynomial is very large, say four, it doesn't do well on the cross-validation set either, and so it's also high. But if d was in-between say, a second-order polynomial, then it actually did much better. If you were to vary the degree of polynomial, you'd actually get a curve that looks like this, which comes down and then goes back up. Where if the degree of polynomial is too low, it underfits and so doesn't do the cross validation set, if it is too high, it overfits and also doesn't do well on the cross validation set.

Is only if it's somewhere in the middle, that is just right, which is why the second-order polynomial in our example ends up with a lower cross-validation error and neither high bias nor high-variance.

Even though we've just seen bias in the areas, it turns out, in some cases, is possible to simultaneously have high bias and have high-variance. You won't see this happen that much for linear regression, but it turns out that if you're training a neural network, there are some applications where unfortunately you have high bias and high variance. One way to recognize that situation will be if J train is high, so you're not doing that well on the training set, but even worse, the cross-validation error is again, even much larger than the training set. The notion of high bias and high variance, it doesn't really happen for linear models applied to 1D.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/L6SHx/diagnosing-bias-and-variance>





From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/L6SHx/diagnosing-bias-and-variance>

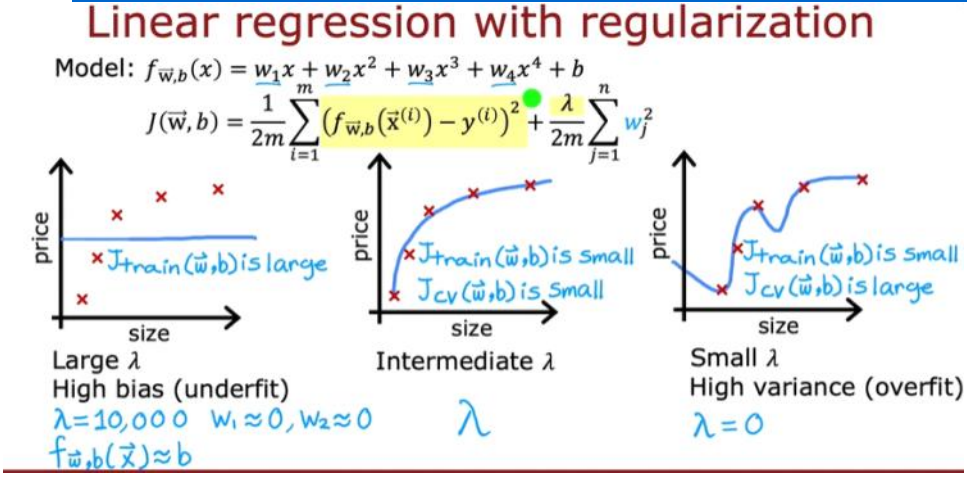
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/L6SHx/diagnosing-bias-and-variance>

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/L6SHx/diagnosing-bias-and-variance>

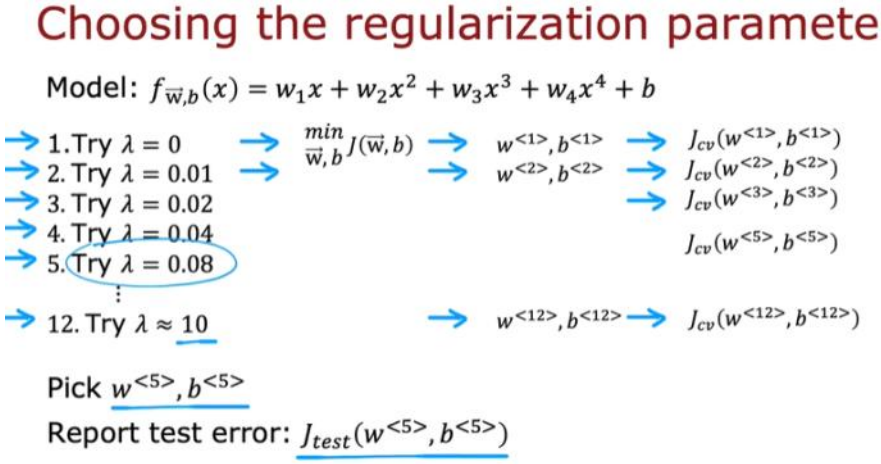
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/L6SHx/diagnosing-bias-and-variance>

This, it turns out, will be helpful for when you want to choose a good value of Lambda of the regularization parameter for your algorithm. Let's take a look. In this example, I'm going to use a fourth-order polynomial, but we're going to fit this model using regularization. Where here the value of Lambda is the regularization parameter that controls how much you trade-off keeping the parameters w small versus fitting the training data well. Let's start with the example of setting Lambda to be a very large value.
It would be if you have some intermediate value of Lambda, not really largely 10,000, but not so small as zero that hopefully you get a model that looks like this, that is just right and fits the data well with small J_train and small J_cv. If you are trying to decide what is a good value of Lambda to use for the regularization parameter, cross-validation gives you a way to do so as well. Let's take a look at how we could do so.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/JQZRO/regularization-and-bias-variance>



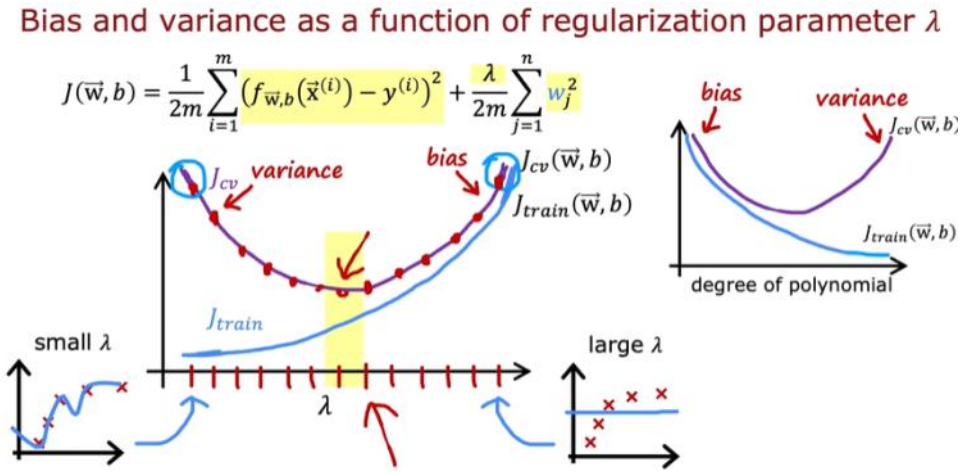
This would be procedures similar to what you had seen for choosing the degree of polynomial D using cross-validation. Specifically, let's say we try to fit a model using Lambda equals 0. We would minimize the cost function using Lambda equals 0 and end up with some parameters w1, b1 and you can then compute the cross-validation error, J_cv of w1, b1. Now let's try a different value of Lambda.



To further hone intuition about what this algorithm is doing, let's take a look at how training error and cross validation error vary as a function of the parameter Lambda. In this figure, I've changed the x-axis again. Notice that the x-axis here is annotated with the value of the regularization parameter Lambda, and if we look at the extreme of Lambda equals zero here on the left, that corresponds to not using any regularization, and so that's where we wound up with this very wiggly curve. If Lambda was small or it was even zero, and in that case, we have a high variance model, and so J train is going to be small and J_cv is going to be large because it does great on the training data but does much worse on the cross validation data. This extreme on the right were very large values of Lambda. Say Lambda equals 10,000 ends up with fitting a model that looks like that. This has high bias, it underfits the data, and it turns out J train will be high and J_cv will be high as well.

n fact, if you were to look at how J train varies as a function of Lambda, you find that J train will go up like this because in the optimization cost function, the larger Lambda is, the more the algorithm is trying to keep W squared small. That is, the more weight is given to this regularization term, and thus the less attention is paid to actually do well on the training set. This term on the left is J train, so the most trying to keep the parameters small, the less good a job it does on minimizing the training error. That's why as Lambda increases, the training error J train will tend to increase like so. Now, how about the cross-validation error? Turns out the cross-validation error will look like this. Because we've seen that if Lambda is too small or too large, then it doesn't do well on the cross-validation set.

It either overfits here on the left or underfits here on the right. There'll be some intermediate value of Lambda that causes the algorithm to perform best. What cross-validation is doing is, it's trying out a lot of different values of Lambda. This is what we saw on the last slide; trial Lambda equals zero, Lambda equals 0.01, logic is 0,02. Try a lot of different values of Lambda and evaluate the cross-validation error in a lot of these different points, and then hopefully pick a value that has low cross validation error, and this will hopefully correspond to a good model for your application.

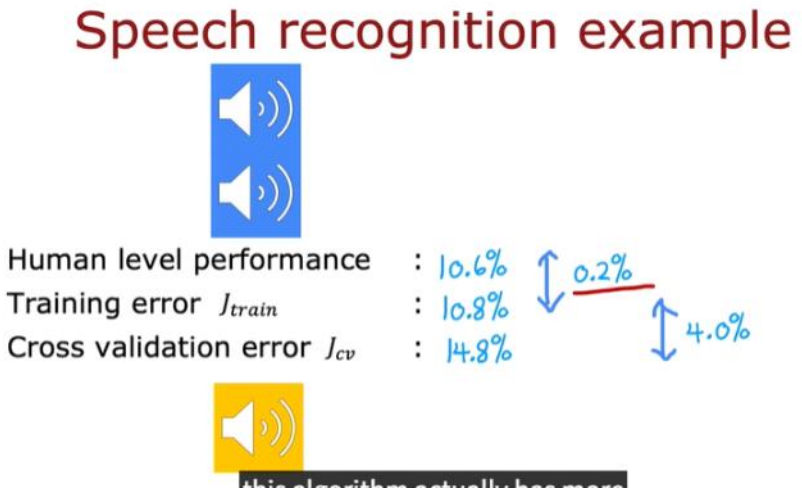


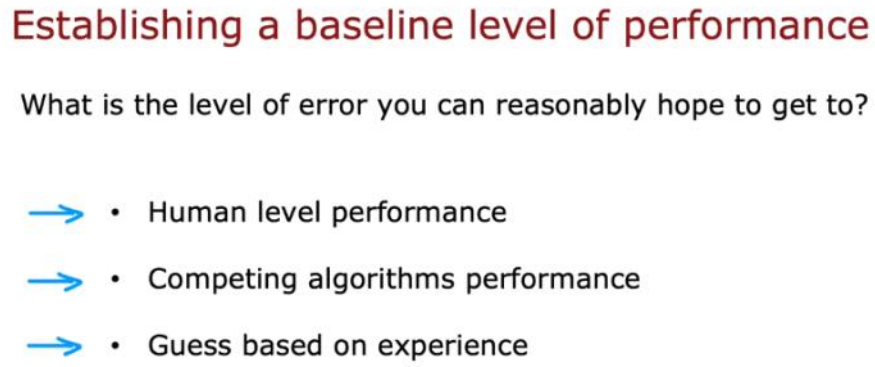
If you compare this diagram to the one that we had in the previous video, where the horizontal axis was the degree of polynomial, these two diagrams look a little bit not mathematically and not in any formal way, but they look a little bit like mirror images of each other, and that's because when you're fitting a degree of polynomial, the left part of this curve corresponded to underfitting and high bias, the right part corresponded to overfitting and high variance. Whereas in this one, high-variance was on the left and high bias was on the right.
But that's why these two images are a little bit like mirror images of each other. But in both cases, cross-validation, evaluating different values can help you choose a good value of t or a good value of Lambda

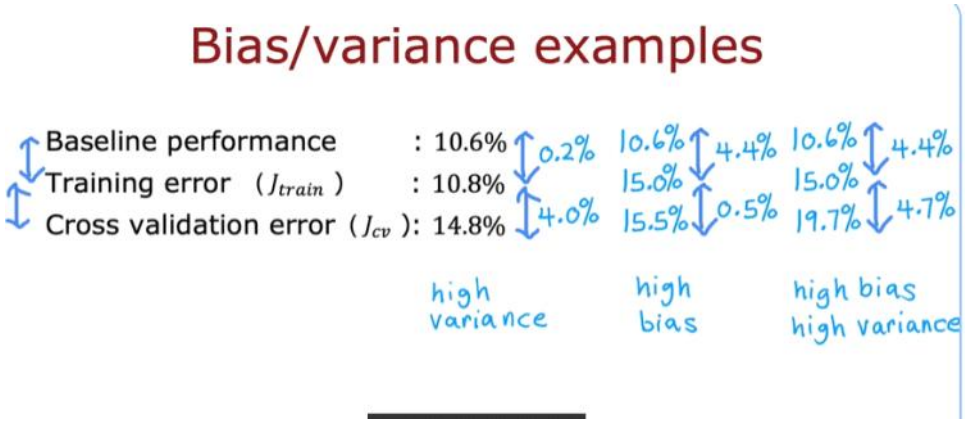
Let's say the training error for this data-set is 10.8 percent meaning that it transcribes it perfectly for 89.2 percent of your training set, but makes some mistake in 10.8 percent of your training set. If you were to also measure your speech recognition algorithm's performance on a separate cross-validation set, let's say it gets 14.8 percent error. If you were to look at these numbers it looks like the training error is really high, it got 10 percent wrong, and then the cross-validation error is higher but getting 10 percent of even your training set wrong that seems pretty high. It seems like that 10 percent error would lead you to conclude it has high bias because it's not doing well on your training set, but it turns out that when analyzing speech recognition it's useful to also measure one other thing which is what is the human level of performance? In other words, how well can even humans transcribe speech accurately from these audio clips? Concretely, let's say that you measure how well fluent speakers can transcribe audio clips and you find that human level performance achieves 10.6 percent error.
hy is human level error so high? It turns out that for web search, there are a lot of audio clips that sound like this, "I'm going to navigate to [inaudible]." There's a lot of noisy audio where really no one can accurately transcribe what was said because of the noise in the audio. If even a human makes 10.6 percent error, then it seems difficult to expect a learning algorithm to do much better. In order to judge if the training error is high, it turns out to be more useful to see if the training error is much higher than a human level of performance, and in this example it does just 0.2 percent worse than humans. Given that humans are actually really good at recognizing speech I think if I can build a speech recognition system that achieves 10.6 percent error matching human performance I'd be pretty happy, so it's just doing a little bit worse than humans. But in contrast, the gap or the difference between JCV and J-train is much larger.
There's actually a four percent gap there, whereas previously we had said

maybe 10.8 percent error means this is high bias. When we benchmark it to human level performance, we see that the algorithm is actually doing quite well on the training set, but the bigger problem is the cross-validation error is much higher than the training error which is why I would conclude that this algorithm actually has more of a variance problem than a bias problem. It turns out when judging if the training error is high is often useful to establish a baseline level of performance, and by baseline level of performance I mean what is the level of error you can reasonably hope your learning algorithm to eventually get to. One common way to establish a baseline level of performance is to measure how well humans can do on this task because humans are really good at understanding speech data, or processing images or understanding texts. Human level performance is often a good benchmark when you are using unstructured data, such as: audio, images, or texts. Another way to estimate a baseline level of performance is if there's some competing algorithm, maybe a previous implementation that someone else has implemented or even a competitor's algorithm to establish a baseline level of performance if you can measure that, or sometimes you might guess based on prior experience.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/acyFT/establishing-a-baseline-level-of-performance>





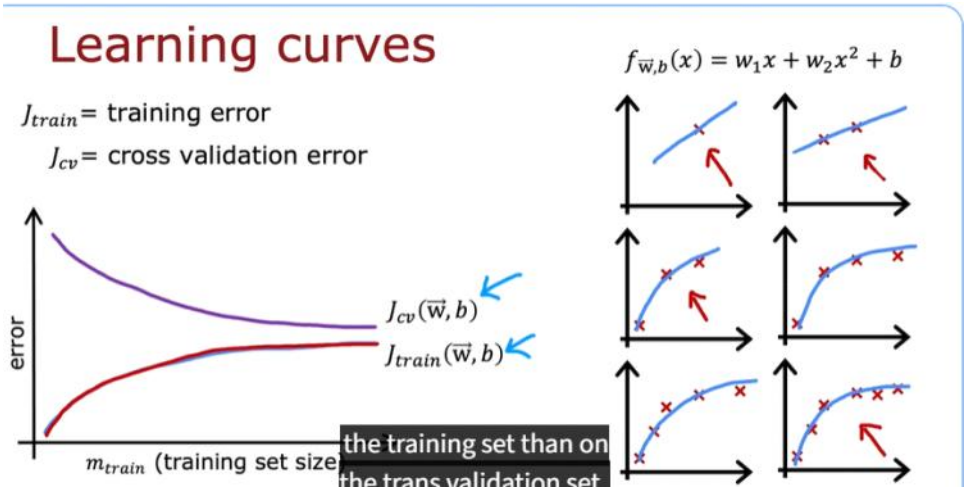


From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/acyFT/establishing-a-baseline-level-of-performance>

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/acyFT/establishing-a-baseline-level-of-performance>

Let's take a look. Let me plot the learning curves for a model that fits a second-order polynomial quadratic function like so. I'm going to plot both J_cv, the cross-validation error, as well as J_train the training error. On this figure, the horizontal axis is going to be m_train. That is the training set size or the number of examples so the algorithm can learn from. On the vertical axis, I'm going to plot the error
y error, I mean either J_cv or J_train. Let's start by plotting the cross-validation error. It will look something like this. That's what J_cv of (w, b) will look like. Is maybe no surprise that as m_train, the training set size gets bigger, then you learn a better model and so the cross-validation error goes down. Now, let's plot J_train of (w, b) of what the training error looks like as the training set size gets bigger. It turns out that the training error will actually look like this.
That as the training set size gets bigger, the training set error actually increases. Let's take a look at why this is the case. We'll start with an example of when you have just a single training example. Well, if you were to fit a quadratic model to this, you can fit easiest straight line or a curve and your training error will be zero. How about if you have two training examples like this? Well, you can again fit a straight line and achieve zero training error. In fact, if you have three training examples, the quadratic function can still fit this very well and get pretty much zero training error, but now, if your training set gets a little bit bigger, say you have four training examples, then it gets a little bit harder to fit all four examples perfectly.
e haven't even larger training sets it just gets harder and harder to fit every single one of your training examples perfectly.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/X8i9Z/learning-curves>



To recap, when you have a very small number of training examples like one or two or even three, is relatively easy to get zero or very small training error, but when you have a larger training set is harder for quadratic function to fit all the training examples perfectly. Which is why as the training set gets bigger, the training error increases because it's harder to fit all of the training examples perfectly.

Notice one other thing about these curves, which is the cross-validation error, will be typically higher than the training error because you fit the parameters to the training set. You expect to do at least a little bit better or when m is small, maybe even a lot better on the training set than on the trans validation set.

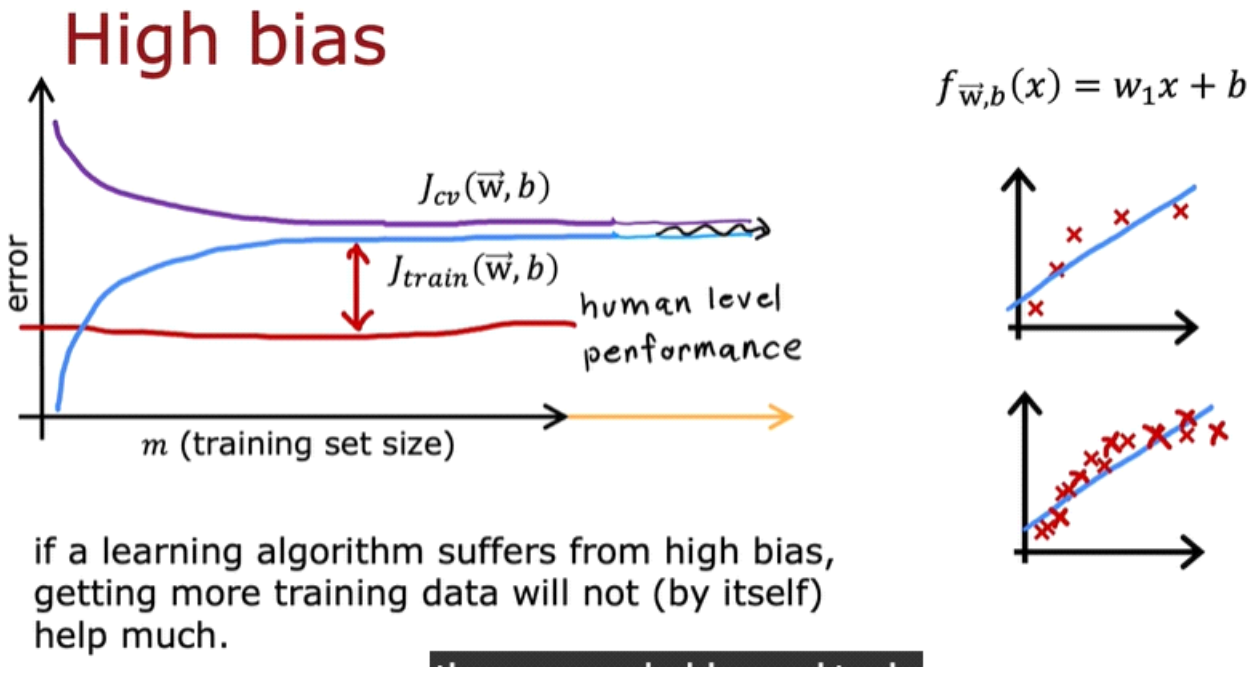
et's now take a look at what the learning curves will look like for an algorithm with high bias versus one with high variance. Let's start at the high bias or the underfitting case. Recall that an example of high bias would be if you're fitting a linear function, so curve that looks like this. If you were to plot the training error, then the training error will go up like so as you'd expect. In fact, this curve of training error may start to flatten out.

We call it plateau, meaning flatten out after a while. That's because as you get more and more training examples when you're fitting the simple linear function, your model doesn't actually change that much more.

Similarly, your cross-validation error will come down and also fattened out after a while, which is why J_cv again is higher than J_train, but J_cv will tend to look like that.

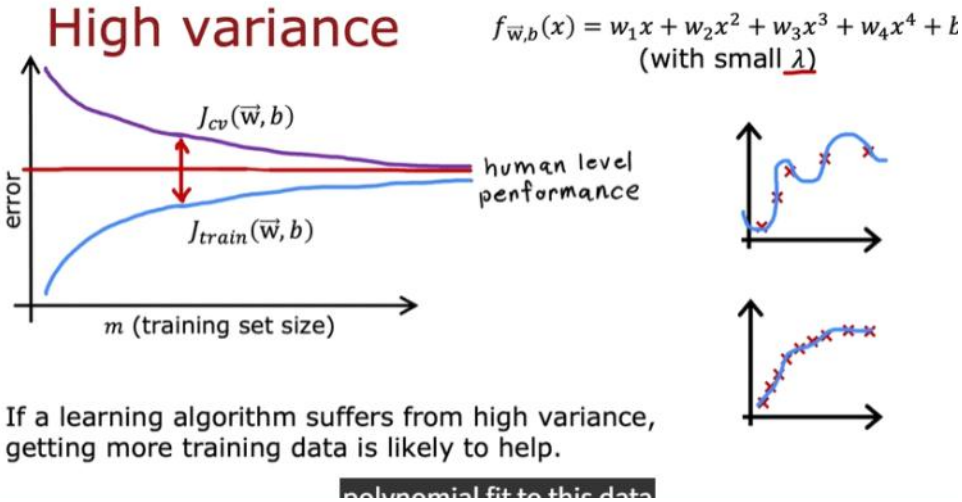
If you had a measure of that baseline level of performance, such as human-level performance, then they'll tend to be a value that is lower than your J_train and your J_cv. Human-level performance may look like this. There's a big gap between the baseline level of performance and J_train, which was our indicator for this algorithm having high bias. That is, one could hope to be doing much better if only we could fit a more complex function than just a straight line.

That gives this conclusion, maybe a little bit surprising, that if a learning algorithm has high bias, getting more training data will not by itself hope that much. I know that we're used to thinking that having more data is good, but if your algorithm has high bias, then if the only thing you do is throw more training data at it, that by itself will not ever let you bring down the error rate that much.



>
Let's now look at what a learning curve might look like in this high variance scenario. J train will be going up as the training set size increases, so you get a curve that looks like this, and J cv will be much higher, so your cross-validation error is much higher than your training error. The fact there's a huge gap here is what I can tell you that this high-variance is doing much better on the training set than it's doing on your cross-validation set. If you were to plot a baseline level of performance, such as human level performance, you may find that it turns out to be here, that J train can sometimes be even lower than the human level performance or maybe human level performance is a little bit lower than this. But when you're over fitting the training set, you may be able to fit the training set so well to have an unrealistically low error, such as zero error in this example over here, which is actually better than how well humans will actually be able to predict housing prices or whatever the application you're working on. But again, to signal for high variance is whether J cv is much higher than J train. When you have high variance, then increasing the training set size could help a lot, and in particular, if we could extrapolate these curves to the right, increase M train, then the training error will continue to go up, but then the cross-validation error hopefully will come down and approach J train
So in this scenario, it might be possible just by increasing the training set size to lower the cross-validation error and to get your algorithm to perform better and better, and this is unlike the high bias case, where if the only thing you do is get more training data, that won't actually help you learn your algorithm performance much. To summarize, if a learning algorithm suffers from high variance, then getting more training data is indeed likely to help. Because extrapolating to the right of this curve, you see that you can expect J cv to keep on coming down.

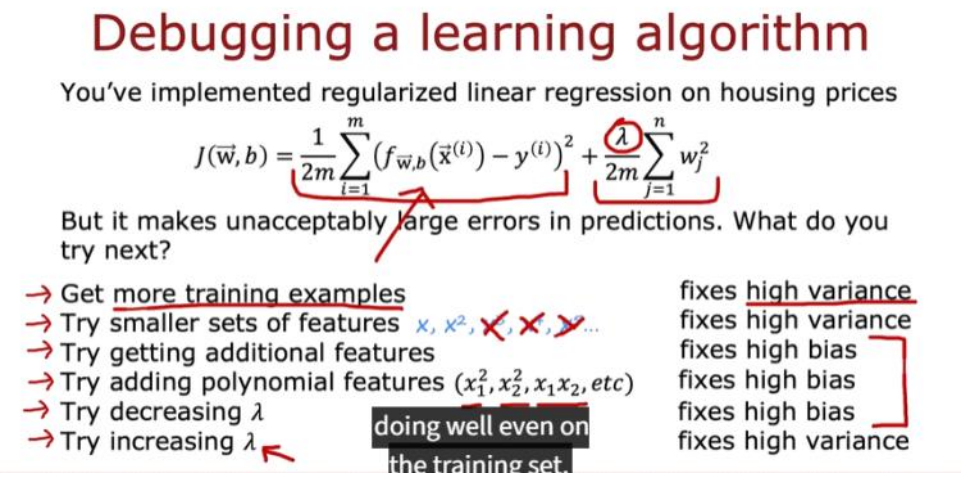
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/X8i9Z/learning-curves>



From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/X8i9Z/learning-curves>

But the takeaways I hope you have are, if you find that your algorithm has high variance, then the two main ways to fix that are; neither get more training data or simplify your model. By simplifying model I mean, either get a smaller set of features or increase the regularization parameter Lambda. Your algorithm has less flexibility to fit very complex, very wiggly curves. Conversely, if your algorithm has high bias, then that means is not doing well even on the training set. If that's the case, the main fixes are to make your model more powerful or to give them more flexibility to fit more complex or more wiggly functions. Some ways to do that are to give it additional features or add these polynomial features, or to decrease the regularization parameter Lambda.
Anyway, in case you're wondering if you should fix high bias by reducing the training set size, that doesn't actually help. If you reduce the training set size, you will fit the training set better, but that tends to worsen your cross-validation error and the performance of your learning algorithm, so don't randomly throw away training examples just to try to fix a high bias problem.

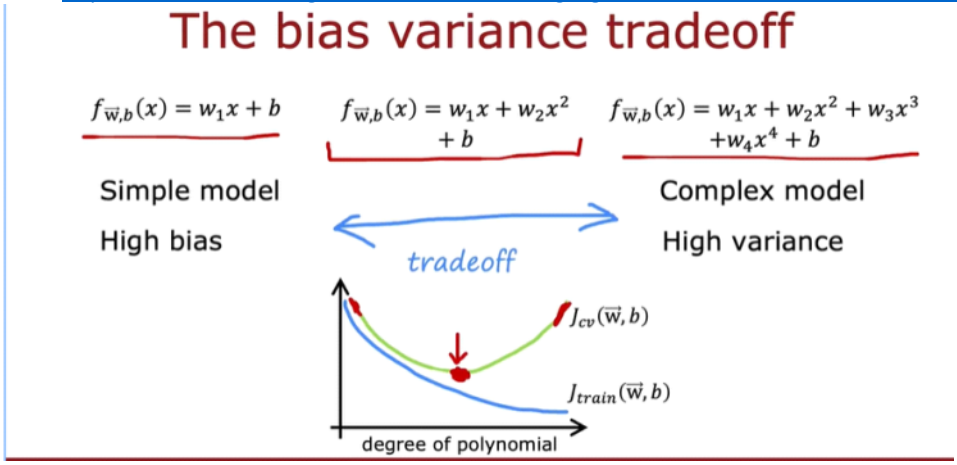
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/WbRtr/deciding-what-to-try-next-revisited>



From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/WbRtr/deciding-what-to-try-next-revisited>

You saw that if you're fitting different order polynomial is to a data set, then if you were to fit a linear model like this on the left. You have a pretty simple model that can have high bias whereas you were to fit a complex model, then you might suffer from high variance. And there's this tradeoff between bias and variance, and in our example it was choosing a second order polynomial that helps you make a tradeoff and pick a model with lowest possible cross validation error.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/d1EGK/bias-variance-and-neural-networks>

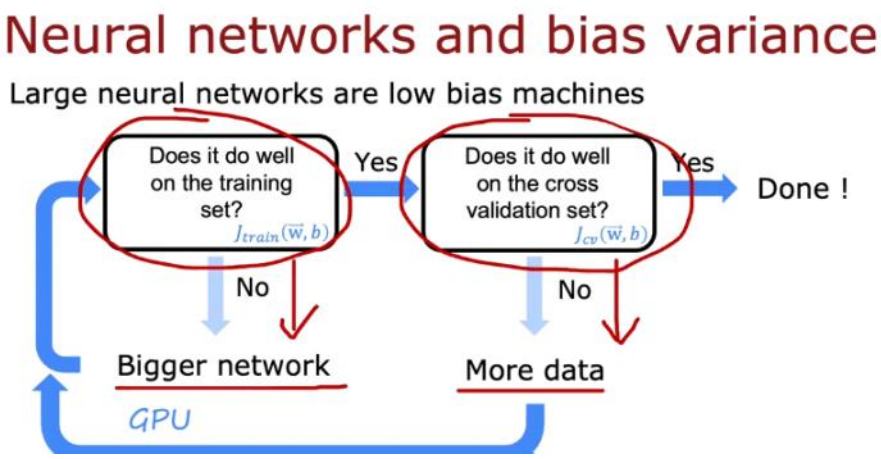


And if you hear machine learning engineers talk about the bias variance tradeoff. This is what they're referring to where if you have too simple a model, you have high bias, too complex a model high variance. And you have to find a tradeoff between these two bad things to find probably the best possible outcome. But it turns out that neural networks offer us a way out of this dilemma of having to tradeoff bias and variance with some caveats. And it turns out that large neural networks when trained on small term moderate sized datasets are low bias machines.

And what I mean by that is, if you make your neural network large enough, you can almost always fit your training set well. So long as your training set is not enormous. And what this means is this gives us a new recipe to try to reduce bias or reduce variance as needed without needing to really trade off between the two of them. But if it applies can be very powerful for getting an accurate model using a neural network which is first train your algorithm on your training set and then asked does it do well on the training set. So measure Jtrain and see if it is high and by high, I mean for example, relative to human level performance or some baseline level of performance and if it is not doing well then you have a high bias problem, high trains error. And one way to reduce bias is to just use a bigger neural network and by bigger neural network, I mean either more hidden layers or more hidden units per layer.

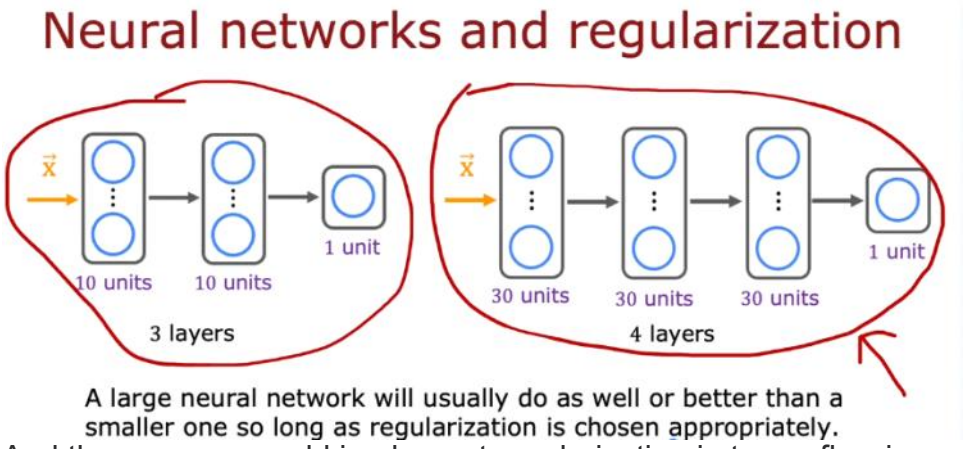
And you can then keep on going through this loop and make your neural network bigger and bigger until it does well on the training set. Meaning that achieves the level of error in your training set that is roughly comparable to the target level of error you hope to get to, which could be human level performance. After it does well on the training set, so the answer to that question is yes. You then ask does it do well on the cross validation set? In other words, does it have high variance and if the answer is no, then you can conclude that the algorithm has high variance because it doesn't want to train set does not do on the cross validation set. So that big gap in Jcv and Jtrain indicates you probably have a high variance problem, and if you have a high variance problem, then one way to try to fix it is to get more data. To get more data and go back and retrain the model and just double-check, do you just want the training set?

If not, have a bigger network, or it does see if it does when the cross validation set and if not get more data. And if you can keep on going around and around and around this loop until eventually it does well in the cross validation set. Then you're probably done because now you have a model that does well on the cross validation set and hopefully will also generalize to new examples as well.



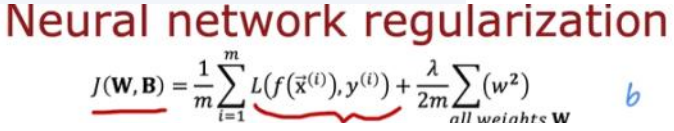
When you train your neural network, one thing that people have asked me before is, hey Andrew, what if my neural network is too big? Will that create a high variance problem? It turns out that a large neural network with well-chosen regularization, well usually do as well or better than a smaller one. And so for example, if you have a small neural network like this, and you were to switch to a much larger neural network like this, you would think that the risk of overfitting goes up significantly. But it turns out that if you were to regularize this larger neural network appropriately, then this larger neural network usually will do at least as well or better than the smaller one.

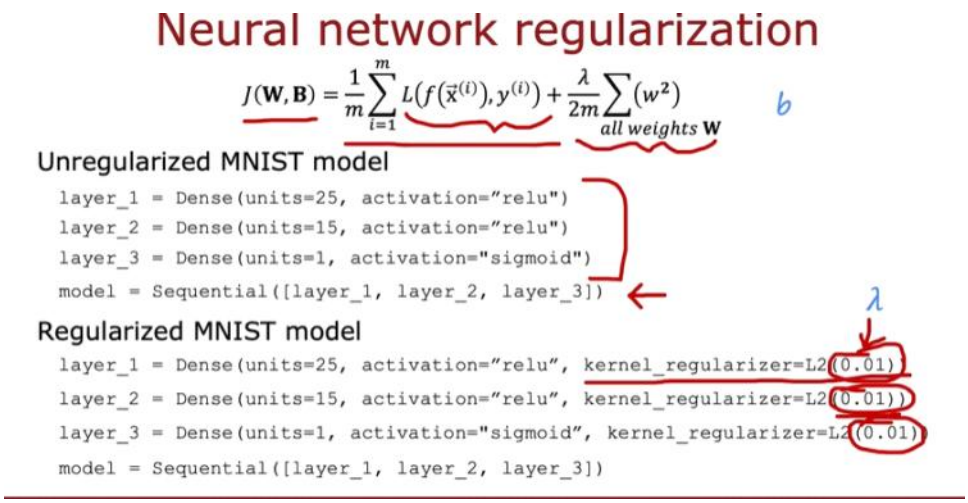
So another way of saying this is that it almost never hurts to go to a larger neural network so long as you regularized appropriately with one caveat, which is that when you train the larger neural network, it does become more computational e expensive. So the main way it hurts, it will slow down your training and your inference process and very briefly to regularize a neural network.



And the way you would implement regularization in tensorflow is recall that this was the code for implementing an unregulated Rised handwritten digit classification model. We create three layers like so with a number of fitting units activation And then create a sequential model with the three layers.

If you want to add regularization then you would just add this extra term colonel regularize A equals l. two and then 0.01 where that's the value of longer in terms of though actually lets you choose different values of lambda for different layers although for simplicity you can choose the same value of lambda for all the weights and all of the different layers as follows. And then this will allow you to implement regularization in your neural network. So to summarize two Takeaways, I hope you have from this video are one. It hardly ever hurts to have a larger neural network so long as you regularize appropriately. one caveat being that having a larger neural network can slow down your algorithm. So maybe that's the one way it hurts, but it shouldn't hurt your algorithm's performance for the most part and in fact it could even help it significantly. And second so long as your training set isn't too large. Then a neural network, especially large neural network is often a low bias machine.

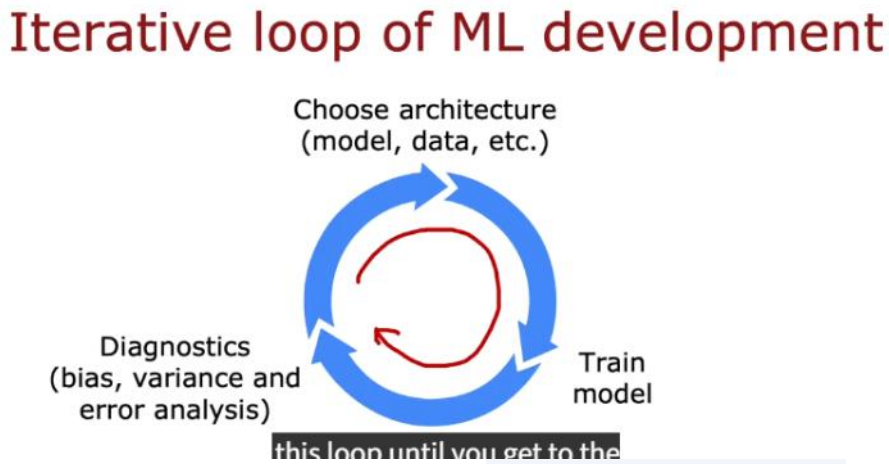




>

This is what developing a machine learning model will often feel like. First, you decide on what is the overall architecture of your system. That means choosing your machine learning model as well as deciding what data to use, maybe picking the hyperparameters, and so on. Then, given those decisions, you would implement and train a model. As I've mentioned before, when you train a model for the first time, it will almost never work as well as you want it to.
The next step that I recommend then is to implement or to look at a few diagnostics, such as looking at the bias and variance of your algorithm as well as something we'll see in the next video called error analysis. Based on the insights from the diagnostics, you can then make decisions like do want to make your neural network bigger or change the Lambda regularization parameter, or maybe add more data or add more features or subtract features. Then you go around this loop again with your new choice of architecture, and it will often take multiple iterations through this loop until you get to the performance that you want.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/uOXJM/iterative-loop-of-ml-development>



Let's look at an example of building an email spam classifier.
Spammers will sometimes deliberately misspell words like these, watches, medicine, and mortgages in order to try to trip up a spam recognizer.

Spam classification example

From: cheapsales@buystufffromme.com
To: Andrew Ng
Subject: Buy now!

Deal of the week! Buy now!
Rolex w4tchs - $100
Med1cine (any kind) - £50
Also low cost M0rgages
available.

From: Alfred Ng
To: Andrew Ng
Subject: Christmas dates?

Hey Andrew,
Was talking to Mom about plans
for Xmas. When do you get off
work. Meet Dec 22?
Alf

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/uOXJM/iterative-loop-of-ml-development>

One way to do so would be to train a supervised learning algorithm where the input features x will be the features of an email and the output label y will be one or zero depending on whether it's spam or non-spam. This application is an example of text classification because you're taking a text document that is an email and trying to classify it as either spam or non-spam. One way to construct the features of the email would be to say, take the top 10,000 words in the English language or in some other dictionary and use them to define features x_1, x_2 through x_10,000. For example, given this email on the right, if the list of words we have is a, Andrew buy deal discount and so on
Then given the email on the right, we would set these features to be, say, 0 or 1, depending on whether or not that word appears. The word a does not appear. The word Andrew does appear. The word buy does appear, deal does, discount does not, and so on, and so you can construct 10,000 features of this email. There are many ways to construct a feature vector. Another way would be to let these numbers not just be 1 or 0, but actually, count the number of times a given word appears in the email. If buy appears twice, maybe you want to set this to 2, but setting into just 1 or 0.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/uOXJM/iterative-loop-of-ml-development>

Building a spam classifier

Supervised learning: $\vec{x}$ = features of email
y = spam (1) or not spam (0)

Features: list the top 10,000 words to compute $x_1, x_2, \cdots, x_{10,000}$

$$\vec{x} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ \vdots \end{bmatrix} \begin{matrix} a \\ andrew \\ buy \\ deal \\ discount \\ \vdots \end{matrix}$$

From: cheapsales@buystufffromme.com
To: Andrew Ng
Subject: Buy now!

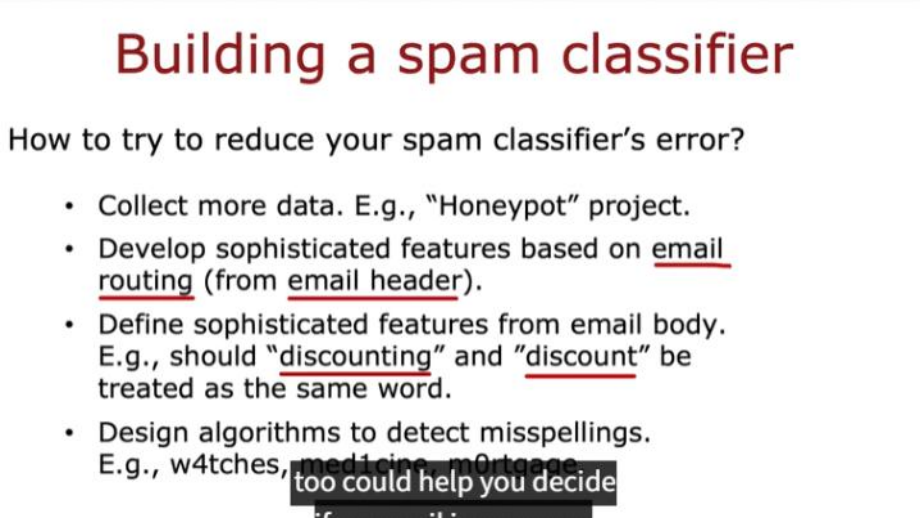
Deal of the week! Buy now!
Rolex w4tchs - $100
Med1cine (any kind) - £50
Also low cost M0rgages
available.

There are many ways to

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/uOXJM/iterative-loop-of-ml-development>

It actually works decently well. Given these features, you can then train a classification algorithm such as a logistic regression model or a neural network to predict y given these features x. After you've trained your initial model, if it doesn't work as well as you wish, you will quite likely have multiple ideas for improving the learning algorithm's performance
For example, is always tempting to collect more data. In fact, I have friends that have worked on very large-scale honeypot projects. These are projects that create a large number of fake email addresses and tries to deliberately to get these fake email addresses into the hands of spammers so that when they send spam email to these fake emails well we know these

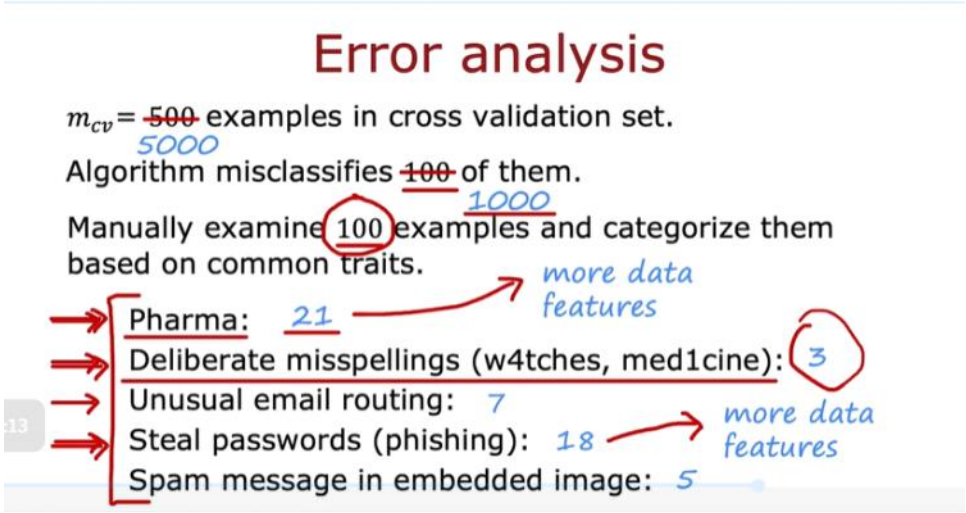
are spam email messages and so this is a way to get a lot of spam data. Or you might decide to work on developing more sophisticated features based on the email routing.
Email routing refers to the sequence of compute service. Sometimes around the world that the email has gone through all this way to reach you and emails actually have what's called email header information. That is information that keeps track of how the email has traveled across different servers, across different networks to find its way to you. Sometimes the path that an email has traveled can help tell you if it was sent by a spammer or not. Or you might work on coming up with more sophisticated features from the email body that is the text of the email. In the features I talked about last time, discounting and discount may be treated as different words, and maybe they should be treated as the same words. Or you might decide to come up with algorithms to detect misspellings or deliberate misspellings like watches, medicine, and mortgage and this too could help you decide if an email is spammy. choosing the more promising path forward can speed up your project easily 10 times compared to if you were to somehow choose some of the less promising directions. For example, we've already seen that if your algorithm has high bias rather than high variance, then spending months and months on a honeypot project may not be the most fruitful direction. But if your algorithm has high variance, then collecting more data could help a lot.

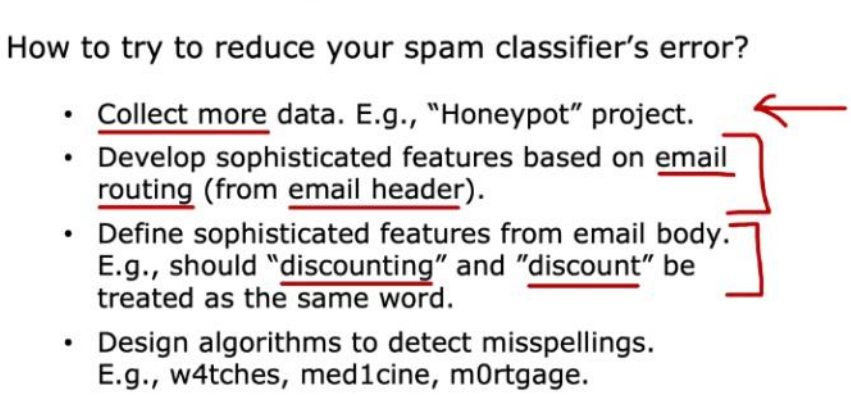


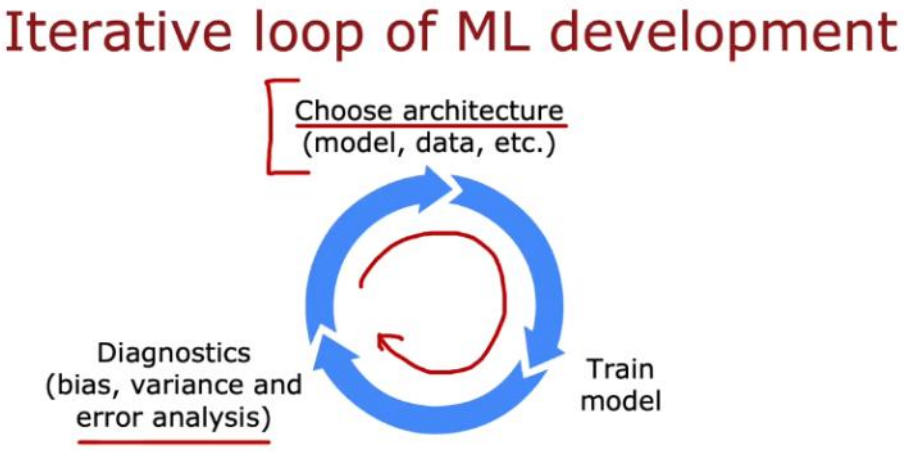
In terms of the most important ways to help you run diagnostics to choose what to try next to improve your learning algorithm performance, I would say bias and variance is probably the most important idea and error analysis would probably be second on my list. Let's take a look at what this means. Concretely, let's say you have m_cv equals 500 cross validation examples and your algorithm misclassifies 100 of these 500 cross validation examples. The error analysis process just refers to manually looking through these 100 examples and trying to gain insights into where the algorithm is going wrong. Specifically, what I will often do is find a set of examples that the algorithm has misclassified examples from the cross validation set and try to group them into common teams or common properties or common traits.
In terms of the most important ways to help you run diagnostics to choose what to try next to improve your learning algorithm performance, I would say bias and variance is probably the most important idea and error analysis would probably be second on my list. Let's take a look at what this means. Concretely, let's say you have m_cv equals 500 cross validation examples and your algorithm misclassifies 100 of these 500 cross validation examples. The error analysis process just refers to manually looking through these 100 examples and trying to gain insights into where the algorithm is going

wrong. Specifically, what I will often do is find a set of examples that the algorithm has misclassified examples from the cross validation set and try to group them into common teams or common properties or common traits.
The net impact seems like it may not be that large. Doesn't mean it's not worth doing? But when you're prioritizing what to do, you might therefore decide not to prioritizes this as highly.
Just a couple of notes on this process. These categories can be overlapping or in other words they're not mutually exclusive.
For example, there can be a pharmaceutical spam email that also has unusual routing or a password that has deliberate misspellings and is also trying to carry out the phishing attack. One email can be counted in multiple categories.
If you have a larger cross validation set, say we had 5,000 cross validation examples and if the algorithm misclassified say 1,000 of them then you may not have the time depending on the team size and how much time you have to work on this project. You may not have the time to manually look at all 1,000 examples that the algorithm misclassifies. In that case, I will often sample randomly a subset of usually around a 100, maybe a couple 100 examples because that's the amount that you can look through in a reasonable amount of time. Hopefully looking through maybe around a 100 examples will give you enough statistics about whether the most common types of errors and therefore where maybe most fruitful to focus your attention
After this analysis, if you find that a lot of errors are pharmaceutical spam emails then this might give you some ideas or inspiration for things to do next. For example, you may decide to collect more data but not more data of everything, but just try to find more data of pharmaceutical spam emails so that the learning algorithm can do a better job recognizing these pharmaceutical spam. Or you may decide to come up with some new features that are related to say specific names of drugs or specific names of pharmaceutical products of the spammers are trying to sell in order to help your learning algorithm become better at recognizing this type of pharma spam. Then again this might inspire you to make specific changes to the algorithm relating to detecting phishing emails.
The point of this error analysis is by manually examining a set of examples that your algorithm is misclassifying or mislabeling.
Often this will create inspiration for what might be useful to try next and sometimes it can also tell you that certain types of errors are sufficiently rare that they aren't worth as much of your time to try to fix. Returning to this list, a bias variance analysis should tell you if collecting more data is helpful or not. Based on our error analysis in the example we just went through, it looks like more sophisticated email features could help but only a bit whereas more sophisticated features to detect pharma spam or phishing emails could help a lot. These detecting misspellings would not help nearly as much. In general I found both the bias variance diagnostic as well as carrying out this form of error analysis to be really helpful to screening or to deciding which changes to the model are more promising to try on next. Now one limitation of error analysis is that it's much easier to do for problems that humans are good at. You can look at the email and say you think is a spam email, why did the algorithm get it wrong?
Error analysis can be a bit harder for tasks that even humans aren't good at. For example, if you're trying to predict what ads someone will click on on the website. Well, I can't predict what someone will click on. Error analysis there actually tends to be more difficult. But when you apply error analysis to problems that you can it can be extremely helpful for focusing attention on the more promising things to try. That in turn can easily save you months of otherwise fruitless work.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/FaPgS/error-analysis>

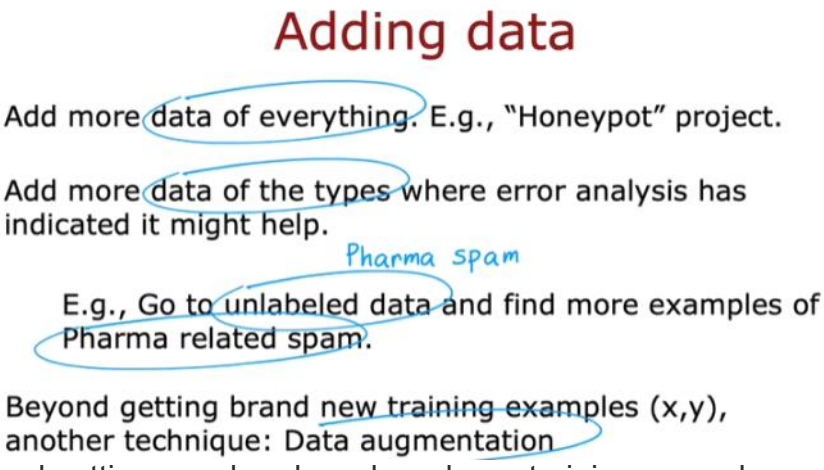






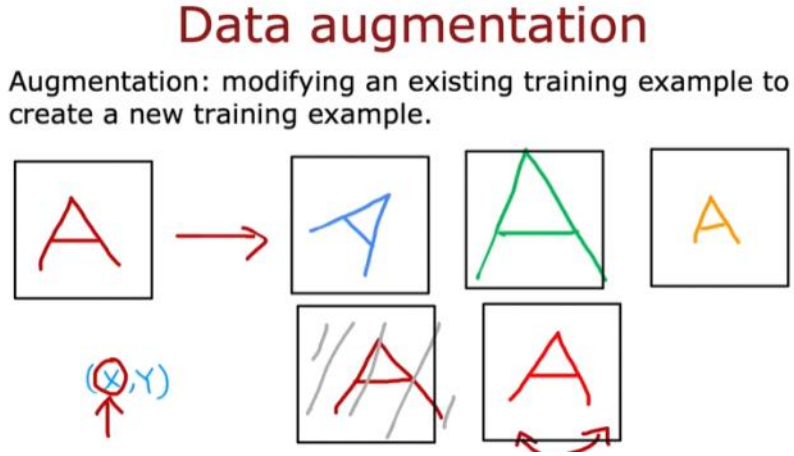
But let's take a look at some tips for how to add data for your application. When training machine learning algorithms, it feels like always we wish we had even more data almost all the time. And so sometimes it's tempting to let's just get more data of everything. But, trying to get more data of all types can be slow and expensive. Instead, an alternative way of adding data might be to focus on adding more data of the types where analysis has indicated it might help. In the previous slide we saw if error analysis reviewed

that pharma spam was a large problem, then you may decide to have a more targeted effort not to get more data everything under the sun but to stay focused on getting more examples of pharma spam And with a more modest cost this could let you add just the emails you need to hope you're learning and get smarter on recognizing pharma spam. if error analysis has indicated that there are certain subsets of the data that the algorithm is doing particularly poorly on. And that you want to improve performance on, then getting more data of just the types where you wanted to do better. Be it more examples of pharmaceutical spam or more examples of phishing spam or something else.

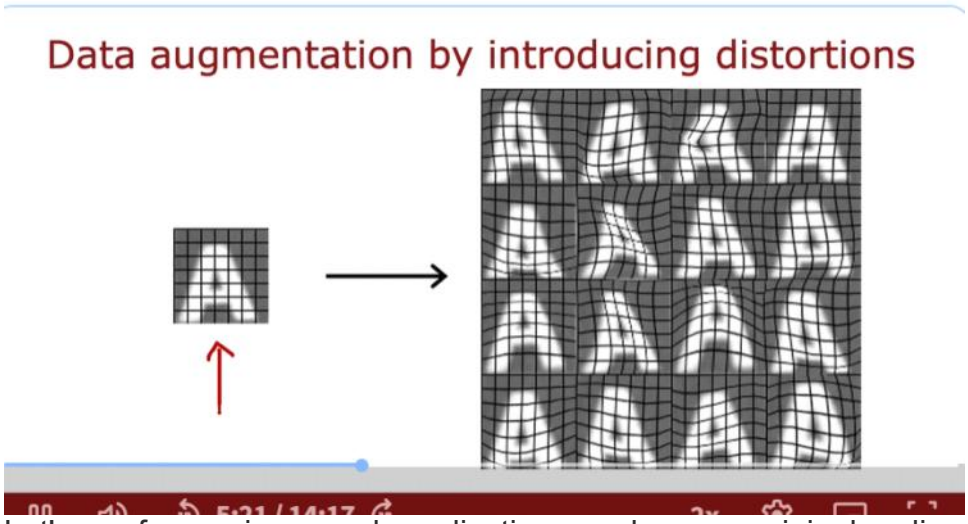


Beyond getting your hands on brand new training examples xy. There's another technique that's widely used especially for images and audio data that can increase your training set size significantly. This technique is called data augmentation. And what we're going to do is take an existing training example to create a new training example. For example if you're trying to recognize the letters from A to Z for an OCR optical character recognition problem. So not just the digits 0-9 but also the letters from A to Z.

Given an image like this, you might decide to create a new training example by rotating the image a bit. Or by enlarging the image a bit or by shrinking a little bit or by changing the contrast of the image. And these are examples of distortions to the image that don't change the fact that this is still the letter A And for some letters but not others you can also take the mirror image of the letter and it still looks like the letter A. But this only applies to some letters but these would be ways of taking a training example X, Y. And applying a distortion or transformation to the input X ,in order to come up with another example that has the same label. And by doing this you're telling the algorithm that the letter A rotated a bit or enlarged a bit or shrunk a little bit it is still the letter A. And creating additional examples like this holds the learning algorithm, do a better job learning how to recognize the letter A.

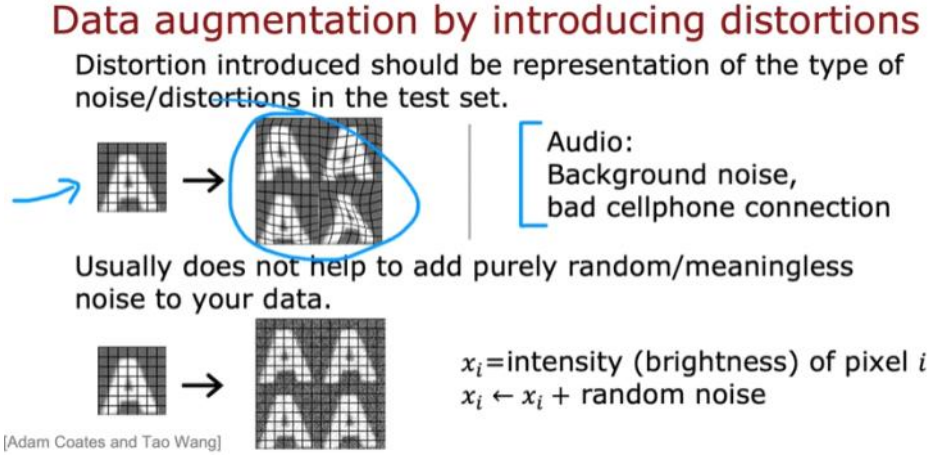


For a more advanced example of data augmentation. You can also take the letter A and place a grid on top of it. And by introducing random warping of this grid, you can take the letter A



Let's say for a voice search application, you have an original audio clip that sounds like this. >> What is today's weather. >> One way you can apply data augmentation to speech data would be to take noisy background audio like this. For example, this is what the sound of a crowd sounds like. And it turns out that if you take these two audio clips, the first one and the crowd noise and you add them together, then you end up with an audio clip that sounds like this. >> What is today's weather. >> And you just created an audio clip that sounds like someone saying what's the weather today.

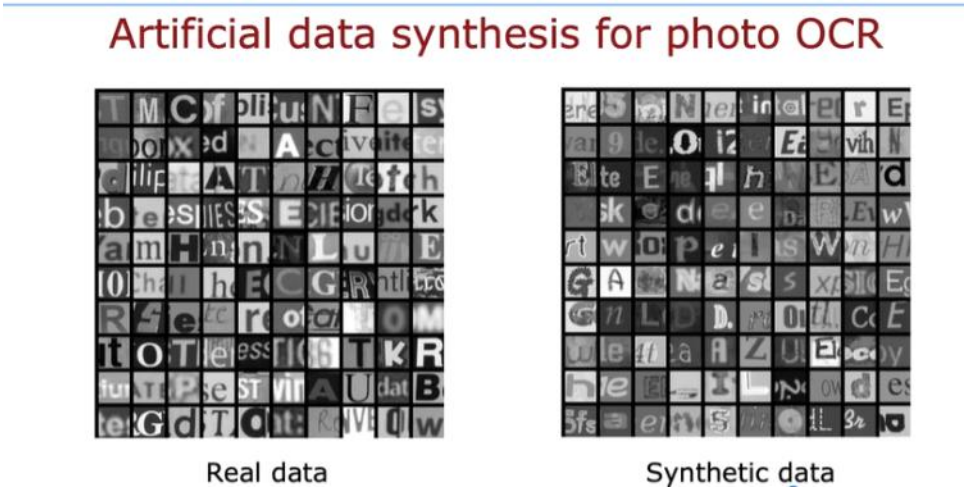
In contrast is usually not that helpful at purely random meaningless noise to data. For example,, you have taken the letter A and I've added per pixel noise where if Xi is the intensity or the brightness of pixel i, if I were to just add noise to each pixel, they end up with images that look like this. But if to the extent that this isn't that representative of what you see in the test set because you don't often get images like this in the test set is actually going to be less helpful.



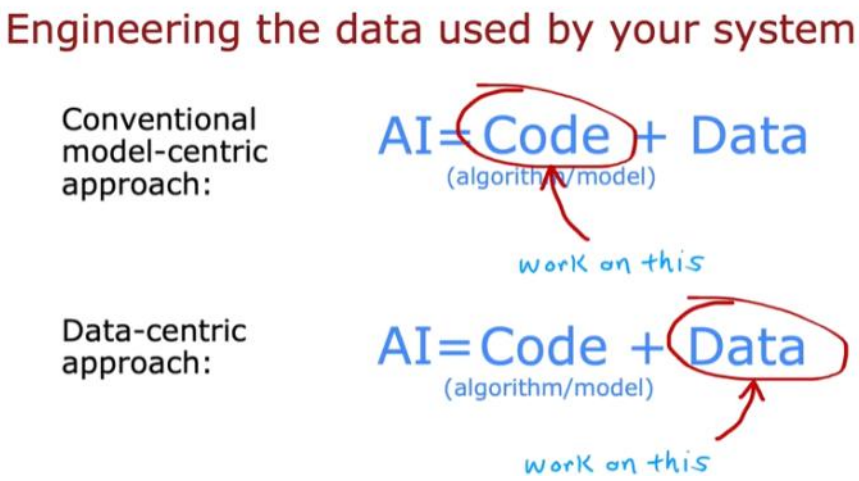
There's one of the techniques which is data synthesis in which you make up brand new examples from scratch. Not by modifying an existing example but by creating brand new examples. So take the example of photo OCR. Photo OCR or photo optical character

recognition refers to the problem of looking at an image like this and automatically having a computer read the text that appears in this image.

And one key step with the photo OCR task is to be able to look at the little image like this, and recognize the letter at the middle. So this has T in the middle, this has the letter L in the middle, this has the letter C in the middle and so on. So one way to create artificial data for this task is if you go to your computer's text editor, you find that it has a lot of different fonts and what you can do is take these fonts and basically type of random text in your text editor.
And screenshotted using different colors and different contrasts and very different fonts and you get synthetic data like that on the right.
Synthetic data generation has been used most probably for computer vision tasks and less for other applications.



A machine learning system or an AI system includes both code to implement your album or your model, as well as the data that you train the algorithm model. and over the last few decades, most researchers doing machine learning research would download the data set and hold the data fixed while they focus on improving the code or the algorithm or the model. Thanks to that paradigm of machine learning research. I find that today the algorithm we have access to such as linear regression, logistic regression, neural networks, also decision trees we should see next week.
There are algorithms that already very good and will work well for many applications. And so sometimes it can be more fruitful to spend more of your time taking a data centric approach in which you focus on engineering the data used by your algorithm. And this can be anything from collecting more data just on pharmaceutical spam. If that's what error analysis told you to do. To using data augmentation to generate more images or more audio or using data synthesis to just create more training examples. And sometimes that focus on the data can be an efficient way to help your learning algorithm improve its performance. So I hope that this video gives you a set of tools to be efficient and effective in how you add more data to get your learning algorithm to work better.



For an application where you don't have that much data, transfer learning is a wonderful technique that lets you use data from a different task to help on your application. This is one of those techniques that I use very frequently. Let's take a look at how transfer learning works.

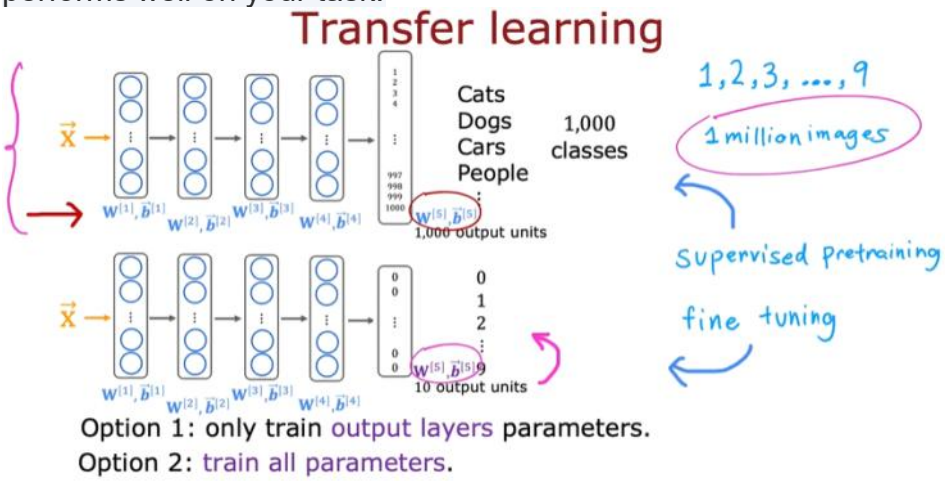
Say you find a very large datasets of one million images of pictures of cats, dogs, cars, people, and so on, a thousand classes.

You can then start by training a neural network on this large dataset of a million images with a thousand different classes and train the algorithm to take as input an image X, and learn to recognize any of these 1,000 different classes. In this process, you end up learning parameters for the first layer of the neural network W^1, b^1, for the second layer W^2, b^2, and so on, W^3, b^3, W^4, b^4, and W^5, b^5 for the output layer. To apply transfer learning, what you do is then make a copy of this neural network where you would keep the parameters W^1, b^1, W^2, b^2, W^3, b^3, and W^4, b^4. But for the last layer, you would eliminate the output layer and replace it with a much smaller output layer with just 10 rather than 1,000 output units. These 10 output units will correspond to the classes zero, one, through nine that you want your neural network to recognize. Notice that the parameters W^5, b^5 they can't be copied over because the dimension of this layer has changed, so you need to come up with new parameters W^5, b^5 that you need to train from scratch rather than just copy it from the previous neural network. In transfer learning, what you can do is use the parameters from the first four layers, really all the layers except the final output layer as a starting point for the parameters and then run an optimization algorithm such as gradient descent or the Adam optimization algorithm with the parameters initialized using the values from this neural network up on top.

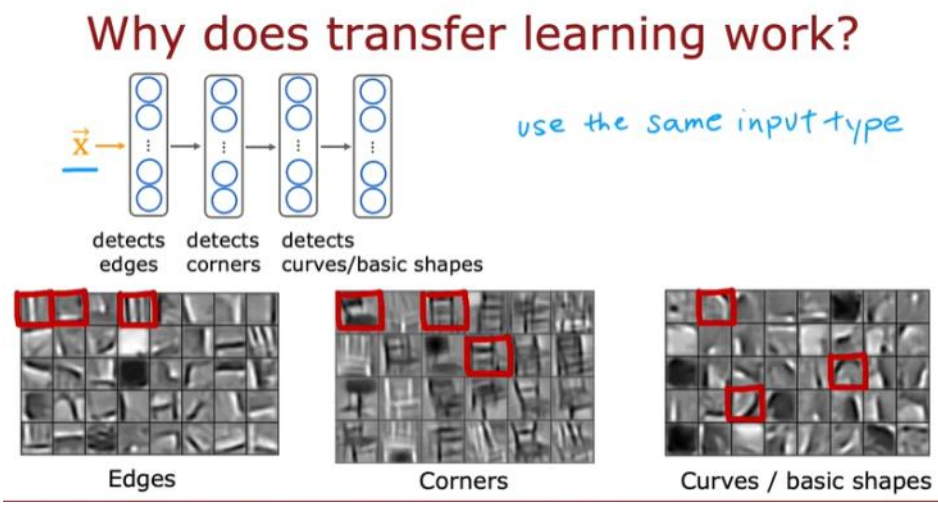
Option 1 is you only train the output layers parameters. You would take the parameters W^1, b^1, W^2, b^2 through W^4, b^4 as the values from on top and just hold them fix and don't even bother to change them, and use an algorithm like Stochastic gradient descent or the Adam optimization algorithm to only update W^5, b^5 to lower the usual cost function that you use for learning to recognize these digits zero to nine from a small training set of these digits zero to nine, so this is Option 1. Option 2 would be to train all the parameters in the network including W^1, b^1, W^2, b^2 all the way through W^5, b^5 but the first four layers parameters would be initialized using the values that you had trained on top. If you have a very small training set then Option 1 might work a little bit better, but if you have a training set that's a little bit larger then Option 2 might work a little bit better.

Then by transferring these parameters to the new neural network, the new neural network starts off with the parameters in a much better place so that we have just a little

bit of further learning.
These two steps of first training on a large dataset and then tuning the parameters further on a smaller dataset go by the name of supervised pre-training for this step on top. That's when you train the neural network on a very large dataset of say a million images of not quite the related task. Then the second step is called fine tuning where you take the parameters that you had initialized or gotten from supervised pre-training and then run gradient descent further to fine tune the weights to suit the specific application of handwritten digit recognition that you may have.
that means is rather than carrying out the first step yourself, you can just download the neural network that someone else may have spent weeks training and then replace the output layer with your own output layer and carry out either Option 1 or Option 2 to fine tune a neural network that someone else has already carried out supervised pre-training on, and just do a little bit of fine tuning to quickly be able to get a neural network that performs well on your task.



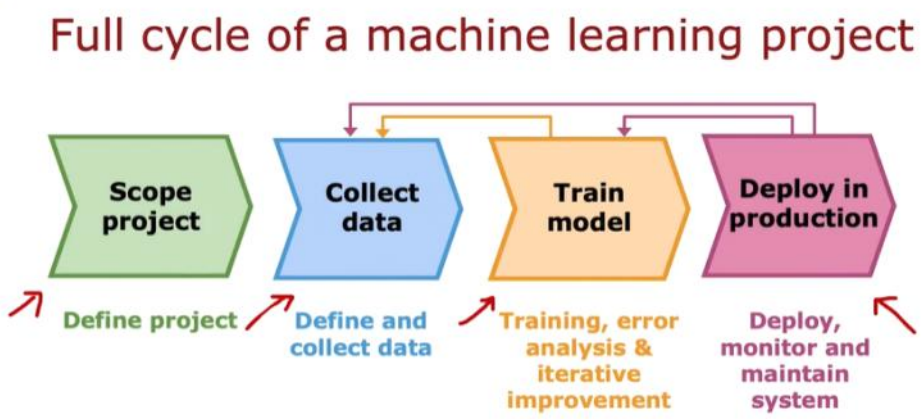
If you are training a neural network to detect, say, different objects from images, then the first layer of a neural network may learn to detect edges in the image. We think of these as somewhat low-level features in the image which is to detect edges. Each of these squares is a visualization of what a single neuron has learned to detect as learn to group together pixels to find edges in an image. The next layer of the neural network then learns to group together edges to detect corners. Each of these is a visualization of what one neuron may have learned to detect, must learn to technical, simple shapes like corner like shapes like this. The next layer of the neural network may have learned to detect some are more complex, but still generic shapes like basic curves or smaller shapes like these. That's why by learning on detecting lots of different images, you're teaching the neural network to detect edges, corners, and basic shapes.
That's why by training a neural network to detect things as diverse as cats, dogs, cars and people, you're helping it to learn to detect these pretty generic features of images and finding edges, corners, curves, basic shapes. This is useful for many other computer vision tasks, such as recognizing handwritten digits. One restriction of pre-training though, is that the image type x has to be the same for the pre-training and fine-tuning steps. If the final task you want to solve is a computer vision tasks, then the pre-training step also has been a neural network trained on the same type of input, namely an image of the desired dimensions.



To summarize, these are the two steps for transfer learning. Step 1 is download neural network with parameters that have been pre-trained on a large dataset with the same input type as your application. That input type could be images, audio, texts, or something else, or if you don't want to download the neural network, maybe you can train your own. But in practice, if you're using images, say, is much more common to download someone else's pre-trained neural network. Then further train or fine tune the network on your own data. I found that if you can get a neural network pre-trained on large dataset, say a million images, then sometimes you can use a much smaller dataset, maybe a thousand images, maybe even smaller, to fine tune the neural network on your own data and get pretty good results. I'd sometimes train neural networks on as few as 50 images that were quite well using this technique, when it has already been pre-trained on a much larger dataset.

The first step of machine learning project is to scope the project. In other words, decide what is the project and what you want to work on.
After deciding what to work on you have to collect data. Decide what data you need to train your machine learning system and go and do the work to get the audio and get the transcripts of the labels for your dataset. That's data collection. After you have your initial data collection you can then start to train the model.
Here you would train a speech recognition system and carry out error analysis and iteratively improve your model. Is not at all uncommon. After you've started training the model for error analysis or for a bias-variance analysis to tell you that you might want to go back to collect more data. Maybe collect more data of everything or just collect more data of a specific type where your error analysis tells you you want to improve the performance of your learning algorithm.
You go around this loop a few times, train the model, error analysis, go back to collect more data, maybe do this for a while until eventually you say the model is good enough to then deploy in a production environment. What that means is you make it available for users to use. When you deploy a system you have to also make sure that you continue to monitor the performance of the system and to maintain the system in case the performance gets worse to bring us performance back up instead of just hosting your machine learning model on a server.
But after this deployment, sometimes you realize that is not working as well as you hoped and you go back to train the model to improve it again or even go back and get more data. In fact, if users and if you have permission to use data from your production

deployment, sometimes that data from your working speech system can give you access to even more data with which to keep on improving the performance of your system. Now, I think you have a sense of what scoping a project means and we've talked a bunch about collecting data and training models in this course.

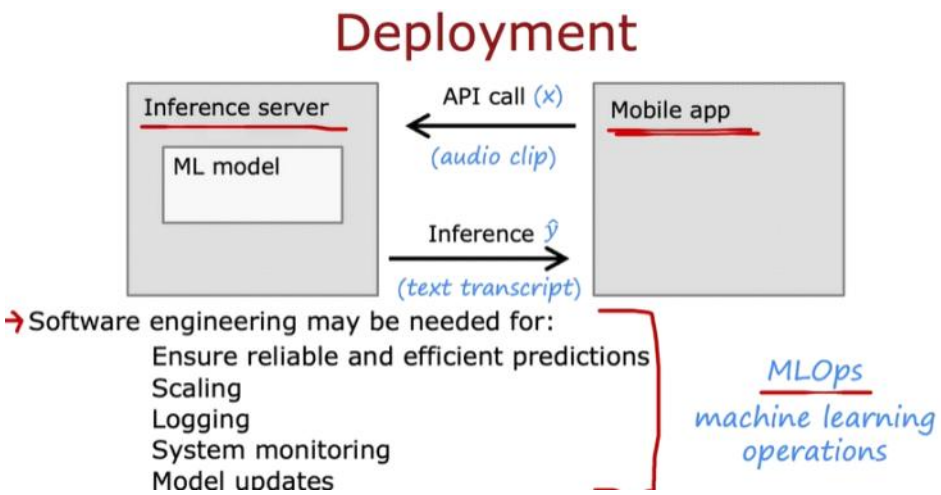


But let me share with you a little bit more detail about what deploying in production might look like. After you've trained a high performing machine learning model, say a speech recognition model, a common way to deploy the model would be to take your machine learning model and implement it in a server, which I'm going to call an inference server, whose job it is to call your machine learning model, your trained model, in order to make predictions. Then if your team has implemented a mobile app, say a search application, then when a user talks to the mobile app, the mobile app can then make an API call to pass to your inference server the audio clip that was recorded and the inference server's job is supply the machine learning model to it and then return to it the prediction of your model, which in this case would be the text transcripts of what was said. This would be a common way of implementing an application that calls via the API and inference server that has your model repeatedly make predictions based on the input, x. This were common pattern where depend on the application does implemented. You have an API call to give your learning algorithm the input, x, and your machine learning model within output to prediction, say y hat. To implement this some software engineering may be needed to write all the code that does all of these things.

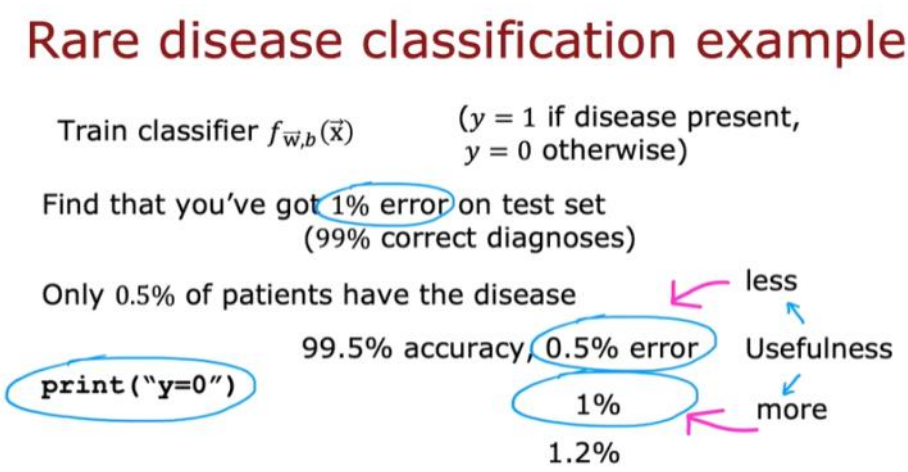
Depending on whether your application needs to serve just a few handful of users or millions of users the amounts of software engineer needed can be quite different. Depending on scale application needed, software engineering may be needed to make sure that your inference server is able to make reliable and efficient predictions hopefully not too high of computational cost. Software engineering may be needed to manage scaling to a large number of users. You often want to log the data you're getting both the inputs, x, as well as the predictions, y hat, assuming that user privacy and consent allows you to store this data. This data, if you can access to it, is also very useful for system monitoring.

But there is a growing field in machine learning called MLOps. This stands for Machine Learning Operations.

This refers to the practice of how to systematically build and deploy and maintain machine learning systems. To do all of these things to make sure that your machine learning model is reliable, scales well, has good laws, is monitored, and then you have the opportunity to make updates to the model as appropriate to keep it running well.



If you're working on a machine learning application where the ratio of positive to negative examples is very skewed, very far from 50-50, then it turns out that the usual error metrics like accuracy don't work that well
Let's start with an example. Let's say you're training a binary classifier to detect a rare disease in patients based on lab tests or based on other data from the patients. Y is equal to 1 if the disease is present and y is equal to 0 otherwise. Suppose you find that you've achieved one percent error on the test set, so you have a 99 percent correct diagnosis. This seems like a great outcome. But it turns out that if this is a rare disease, so y is equal to 1, very rarely, then this may not be as impressive as it sounds. Specifically, if it is a rare disease and if only 0.5 percent of the patients in your population have the disease, then if instead you wrote the program, that just said, print y equals 0. It predicts y equals 0 all the time. This very simple even non-learning algorithm, because it just says y equals 0 all the time, this will actually have 99.5 percent accuracy or 0.5 percent error. This really dumb algorithm outperforms your learning algorithm which had one percent error, much worse than 0.5 percent error. But I think a piece of software that just prints y equals 0, is not a very useful diagnostic tool. It's difficult to know which of these is actually the best algorithm. Because if you have an algorithm that achieve 0.5 percent error and a different one that achieves one percent error and a different one that achieves 1.2 percent error, it's difficult to know which of these is the best algorithm. Because the one with the lowest error may be is not particularly useful prediction like this that always predicts y equals 0 and never ever diagnose any patient as having this disease. Quite possibly an algorithm that has one percent error, but that at least diagnosis some patients as having the disease could be more useful than just printing y equals 0 all the time.



we usually use a different error metric rather than just classification error to figure out how well your learning algorithm is doing. In particular, a common pair of error metrics are precision and recall, which we'll define on the slide. In this example, y equals one will be the rare class, such as the rare disease that we may want to detect.

In particular, to evaluate a learning algorithm's performance with one rare class it's useful to construct what's called a confusion matrix, which is a two-by-two matrix or a two-by-two table that looks like this. On the axis on top, I'm going to write the actual class, which could be one or zero. On the vertical axis, I'm going to write the predicted class, which is what did your learning algorithm predicts on a given example, one or zero? To evaluate your algorithm's performance on the cross-validation set or the test set say, we will then count up how many examples? Was the actual class, 1, and the predicted class 1?

I'm actually going to give names to these four cells. When the actual class is one and the predicted class is one, we're going to call that a true positive because you predicted positive and it was true there's a positive example.

In this cell on the lower right, where the actual class is zero and the predicted class is zero, we will call that a true negative because you predicted negative and it was true. It really was a negative example. This cell on the upper right is called a false positive because the algorithm predicted positive, but it was false. It's not actually positive, so this is called a false positive. This cell is called the number of false negatives because the algorithm predicted zero, but it was false. It wasn't actually negative. The actual class was one.

Having divided the classifications into these four cells, two common metrics you might compute are then the precision and recall.

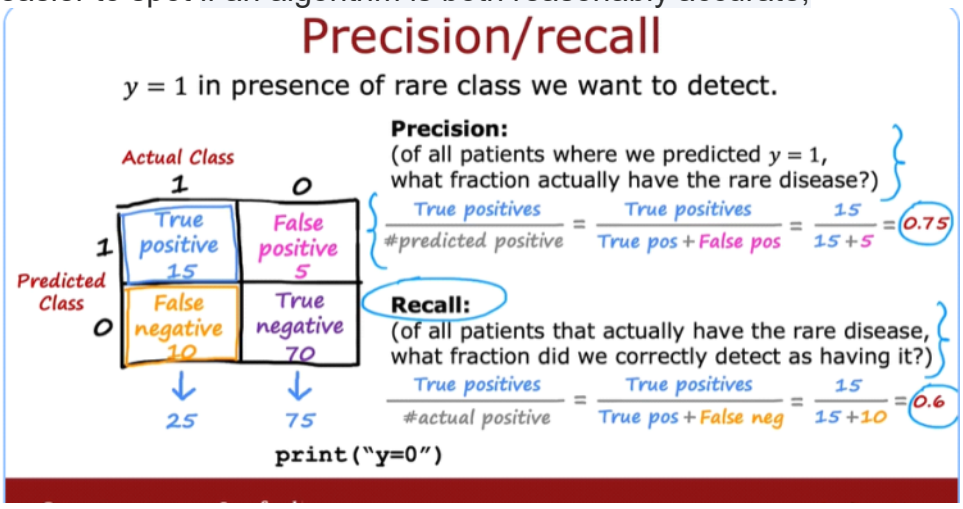
precision is defined as the number of true positives divided by the number classified as positive. In other words, of all the examples you predicted as positive, what fraction did we actually get right. Another way to write this formula would be true positives divided by true positives plus false positives because it is by summing this cell and this cell that you end up with the total number that was predicted as positive. I

The second metric that is useful to compute is recall. And recall asks: Of all the patients that actually have the rare disease, what fraction did we correctly detect as having it? Recall is defined as the number of true positives divided by the number of actual positives. Alternatively, we can write that as number of true positives divided by the number of actual positives. Well, it's this cell plus this cell. So it's actually the number of true positives plus the number of false negatives because it's by summing up this upper-left cell and this lower-left cell that you get the number of actual positive examples.

The recall metric in particular helps you detect if the learning algorithm is predicting zero all the time. Because if your learning algorithm just prints y equals 0, then the number of true positives will be zero because it never predicts positive, and so the recall will be equal to zero divided by the number of actual positives, which is equal to zero.

In general, a learning algorithm with either zero precision or zero recall is not a useful algorithm. But just as a side note, if an algorithm actually predicts zero all the time, precision actually becomes undefined because it's actually zero over. zero. But in practice, if an algorithm doesn't predict even a single positive, we just say that precision

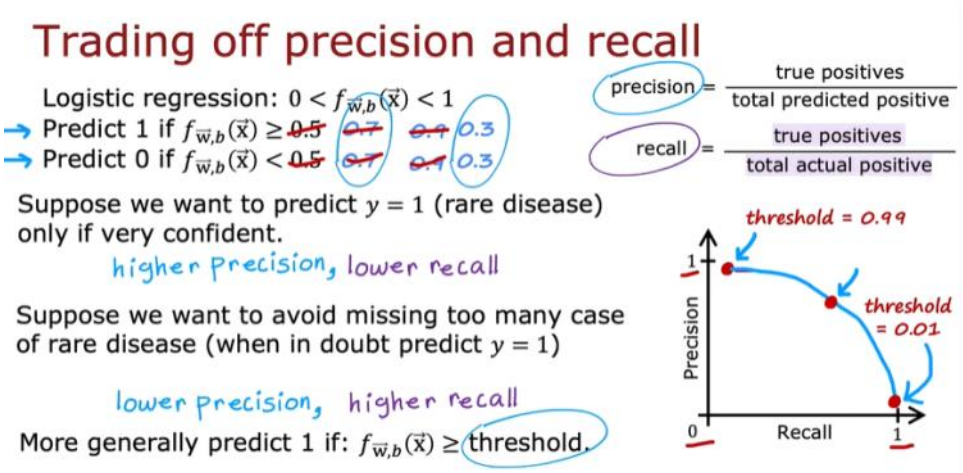
is also equal to zero. But we'll find that computing both precision and recall makes it easier to spot if an algorithm is both reasonably accurate,



If you're using logistic regression to make predictions, then the logistic regression model will output numbers between 0 and 1. We would typically threshold the output of logistic regression at 0.5 and predict 1 if f of x is greater than equal to 0.5 and predict 0 if it's less than 0.5. But suppose we want to predict that y is equal to 1. That is, the rare disease is present only if we're very confident. If our philosophy is, whenever we predict that the patient has a disease, we may have to send them for possibly invasive and expensive treatment. If the consequences of the disease aren't that bad, even if left not treated aggressively, then we may want to predict y equals 1 only if we're very confident. In that case, we may choose to set a higher threshold where we will predict y is 1 only if f of x is greater than or equal to 0.7, so this is saying we'll predict y equals 1 only we're at least 70 percent sure, rather than just 50 percent sure and so this number also becomes 0.7.
By raising this threshold, you predict y equals 1 only if you're pretty confident and what that means is that precision will increase because whenever you predict one, you're more likely to be right so raising the thresholds will result in higher precision, but it also results in lower recall because we're now predicting one less often and so of the total number of patients with the disease, we're going to correctly diagnose fewer of them.
On the flip side, suppose we want to avoid missing too many cases of the rare disease, so if what we want is when in doubt, predict y equals 1, this might be the case where if treatment is not too invasive or painful or expensive but leaving a disease untreated has much worse consequences for the patient. In that case, you might say, when in doubt in the interests of safety let's just predict that they have it and consider them for treatment because untreated cases could be quite bad.

If for your application, that is the better way to make decisions, then you would take this threshold instead lower it, say, set it to 0.3. In that case, you predict one so long as you think there's maybe a 30 percent chance or better of the disease being present and you predict zero only if you're pretty sure that the disease is absent. As you can imagine, the impact on precision and recall will be opposite to what you saw up here, and lowering this threshold will result in lower precision because we're now looser, we're more willing to predict one even if we aren't sure but to result in higher recall, because of all the patients that do have that disease, we're probably going to

correctly identify more of them.
It turns out that for most learning algorithms, there is a trade-off between precision and recall. Precision and recall both go between zero and one and if you were to set a very high threshold, say a threshold of 0.99, then you enter with very high precision, but lower recall and as you reduce the value of this threshold, you then end up with a curve that trades off precision and recall until eventually, if you have a very low threshold, so the threshold equals 0.01, then you end up with very low precision but relatively high recall. Sometimes by plotting this curve, you can then try to pick a threshold which corresponds to picking a point on this curve.



It turns out that if you want to automatically trade-off precision and recall rather than have to do so yourself, there is another metric called the F1 score that is sometimes used to automatically combine precision recall to help you pick the best value or the best trade-off between the two.
In order to help you decide which algorithm to pick, it may be useful to find a way to combine precision and recall into a single score, so you can just look at which algorithm has the highest score and maybe go with that one. One way you could combine precision and recall is to take the average, this turns out not to be a good way, so I don't really recommend this. But if we were to take the average, you get 0.45, 0.4, and 0.5. But it turns out that computing the average and picking the algorithm with the highest average between precision and recall doesn't work that well because this algorithm has very low precision, and in fact, this corresponds maybe to an algorithm that actually does print y equals 1 and diagnosis all patients as having the disease, that's why recall is perfect but the precision is really low. Algorithm 3 is actually not a particularly useful algorithm, even though the average between precision and recall is quite high. Let's not use the average between precision and recall.
Instead, the most common way of combining precision recall is a compute something called the F1 score, and the F1 score is a way of combining P and R precision and recall but that gives more emphasis to whichever of these values is lower. Because it turns out if an algorithm has very low precision or very low recall is pretty not that useful. The F1 score is a way of computing an average of sorts that pays more attention to whichever is lower. The formula for computing F1 score is this, you're going to compute one over P and one over R, and average them, and then take the inverse of that. Rather than averaging P and R precision recall we're going to average one over P and one over R, and then take one over that. If you simplify this equation it can also be computed as follows. But by averaging one over P and one over R this gives a much greater emphasis to if either P or R turns out to be very small.
If you were to compute the F1 score for these three algorithms, you'll find that the F1 score for Algorithm 1 is 0.444, and for the second algorithm is 0.175. You notice that 0.175 is much closer to the lower value than the higher value and for the third algorithm is 0.0392. F1 score gives away to trade-off precision and recall, and in this case, it will tell us that maybe the first algorithm is better than the second or the third algorithms.

F1 score

How to compare precision/recall numbers?

F1 score

How to compare precision/recall numbers?

	Precision (P)	Recall (R)	Average	F_1 score
Algorithm 1	0.5	0.4	0.45	0.444
Algorithm 2	0.7	0.1	0.4	0.175
Algorithm 3	0.02	1.0	0.501	0.0392

```
print("y=1")
```

Harmonic mean

Average $= \frac{P+R}{2}$

$F_1 \text{ score} = \frac{1}{\frac{1}{2}\left(\frac{1}{P} + \frac{1}{R}\right)} = 2\frac{PR}{P+R}$