Lecture 13 [Debugging ML Models and Error Analysis]

https://www.youtube.com/watch?v=ORrStCArmP4&list=PLoROMvodv4rMiGQp3WXShtMGgzqpfVfbU&index=13

[
Art vs Science:
How to get started on a machine learning problem:
Premature (statistical) optimization:
]

Advice on getting learning algorithms to work:

A lot of the workflow of developing ***learning algorithms*** will actually feel like a debugging workflow

What happens all the time is you have an idea for a machine learning application, you implement something, and then it won't work as well as you hoped. The key question is *"what do you do next?"*

Let's say we're building an ***anti-spam classifier*** and let's say you've carefully chosen a small set of **100** words to use as features. Instead of using **10,000** or **50,000** words, you've chosen **100** words that you think could be the most relevant to anti-spam

Let's say you start off implementing ***logistic regularization*** (you can think of this as ***bayesian logistic regression*** where you have the ***maximum likelihood*** term on the left and the ***regularization*** term on the right). And let's say that ***logistic regression*** with ***regularization*** or ***bayesian logistic regression***, gets **20% *test error***, which is unacceptably high; making **1** in **5** mistakes on your spam filter

So, what do you do next?

Debugging learning algorithms

Motivating example:

- Anti-spam. You carefully choose a small set of 100 words to use as features. (Instead of using all 50000+ words in English.)
- Logistic regression with regularization (Bayesian Logistic regression), implemented with gradient ascent, gets 20% test error, which is unacceptably high.

$$\max_{\theta} \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}, \theta) - \lambda||\theta||^2$$

- What to do next?

When you implement an algorithm like this, what many teams will do is try improving the algorithm in different ways. So what many teams will do is say *"we like big data, more data always. So let's get some more data and hope that solves the problem"*

Some teams will say *"Let's get more training examples"*. It's true that more data pretty much never hurts, it almost always helps, but the key question is *"how much?"*

You could try using a smaller set of features. With **100** features, probably some weren't that relevant, so we'll get rid of some features

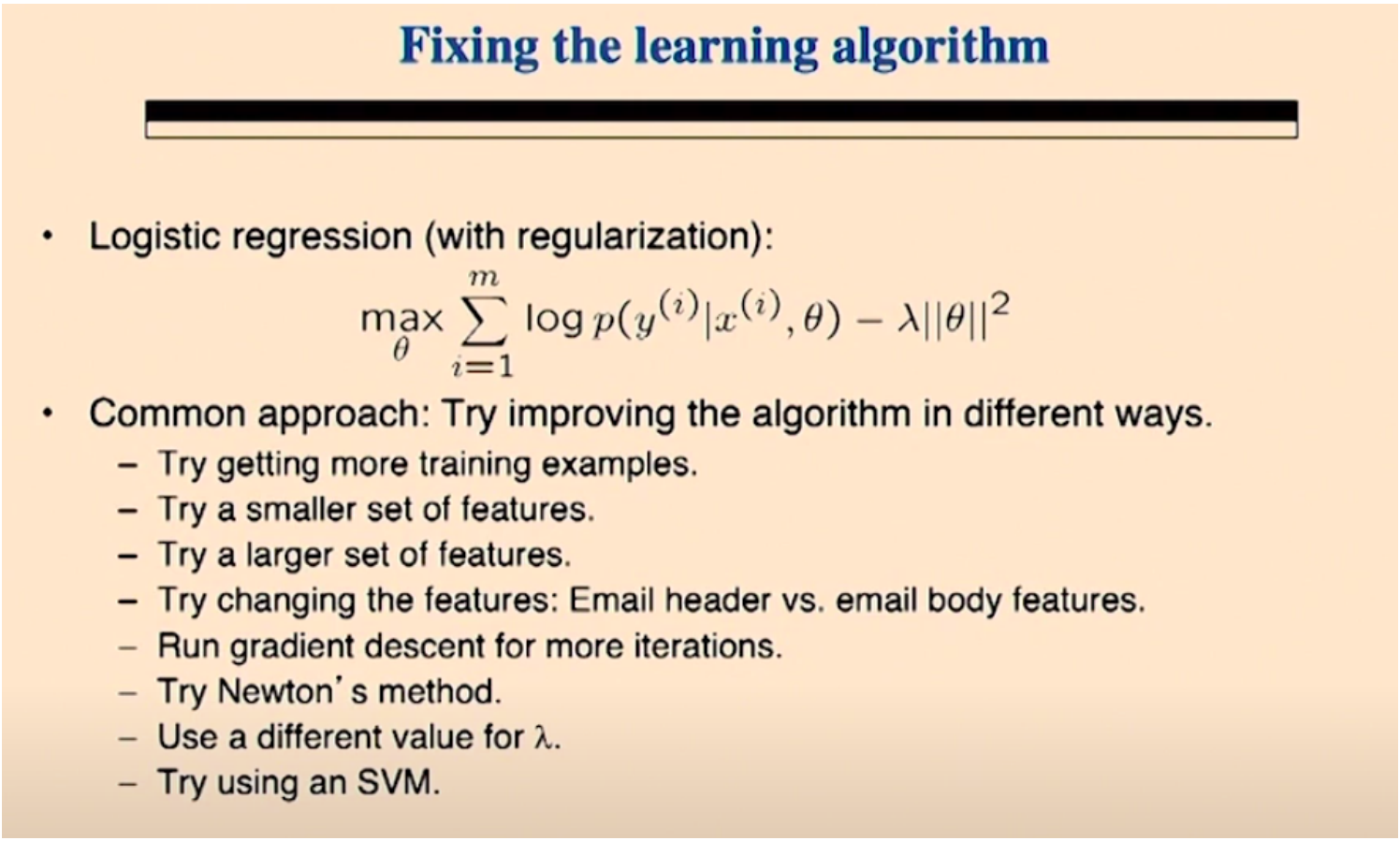
You could try having a larger set of features

You might want other designs of the features. Instead of just using features in an email body, you can use features from the email header

You could try running ***gradient descent*** for more iterations. That usually never hurts

Instead of ***gradient descent***, we can switch to ***Newton's methods***. We can also try a different value for **λ**

Or we could forget about ***bayesian logistic regression*** or logistic regression with regularization, and use a totally different algorithm like a ***SVM*** or ***Neural networks***



What happens in a lot of teams is, someone will pick one of these ideas, kind of at random. Unless you analyze these different options, it's hard to know which of these is actually the best option

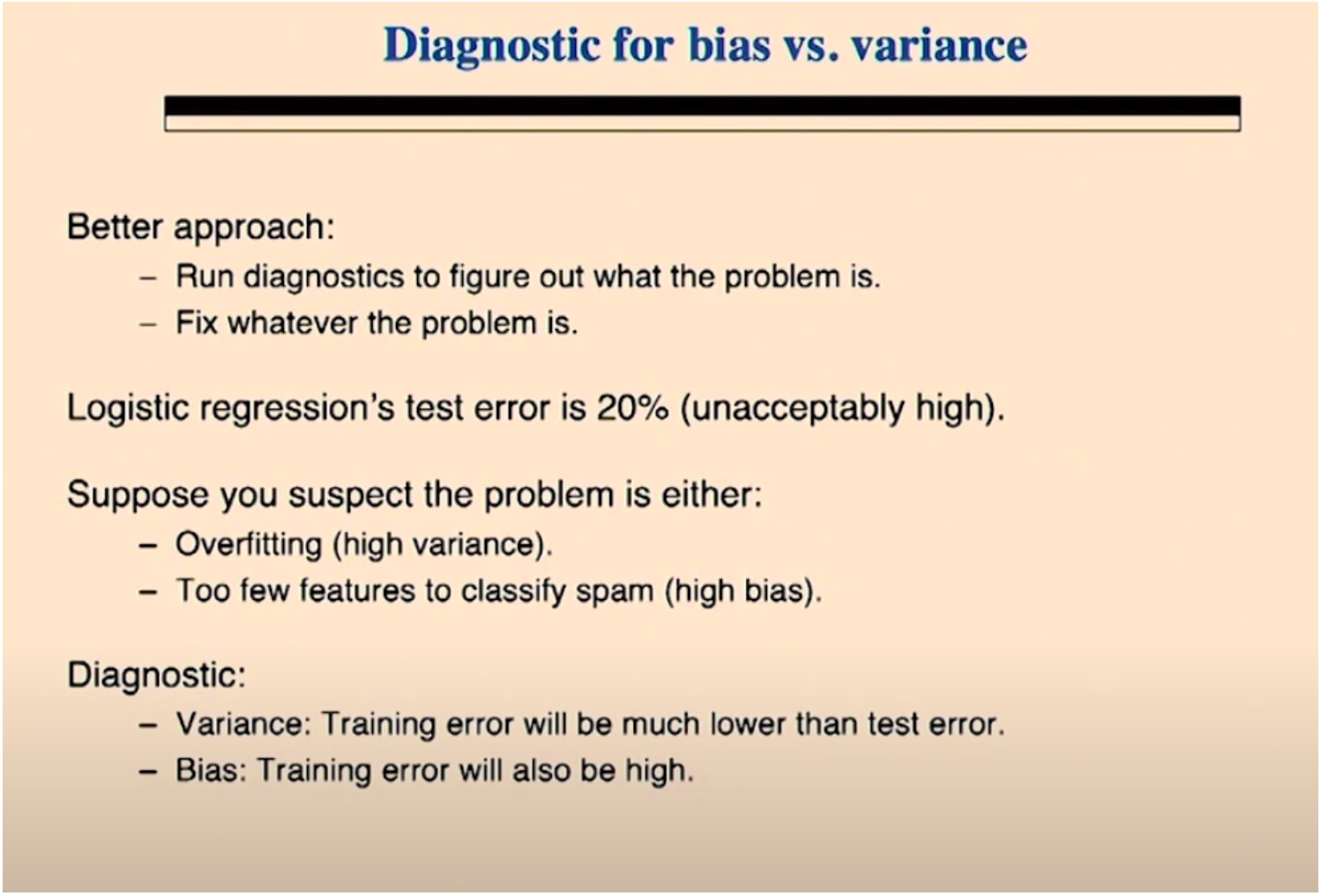
Diagnostics for debugging learning algorithms:

One of the most common diagnostic you may end up using in developing learning algorithms is a ***bias vs variance diagnostic*** to understand how much of your learning algorithm's problem comes from ***bias*** and how much of it comes from ***variance***

We're going to describe a workflow where you would run some diagnostics to figure out what the problem is, and then try to fix what

that problem is

To summarize this example, this logistic error is unacceptable high and you suspect the problem is due to high ***variance*** or high ***bias***



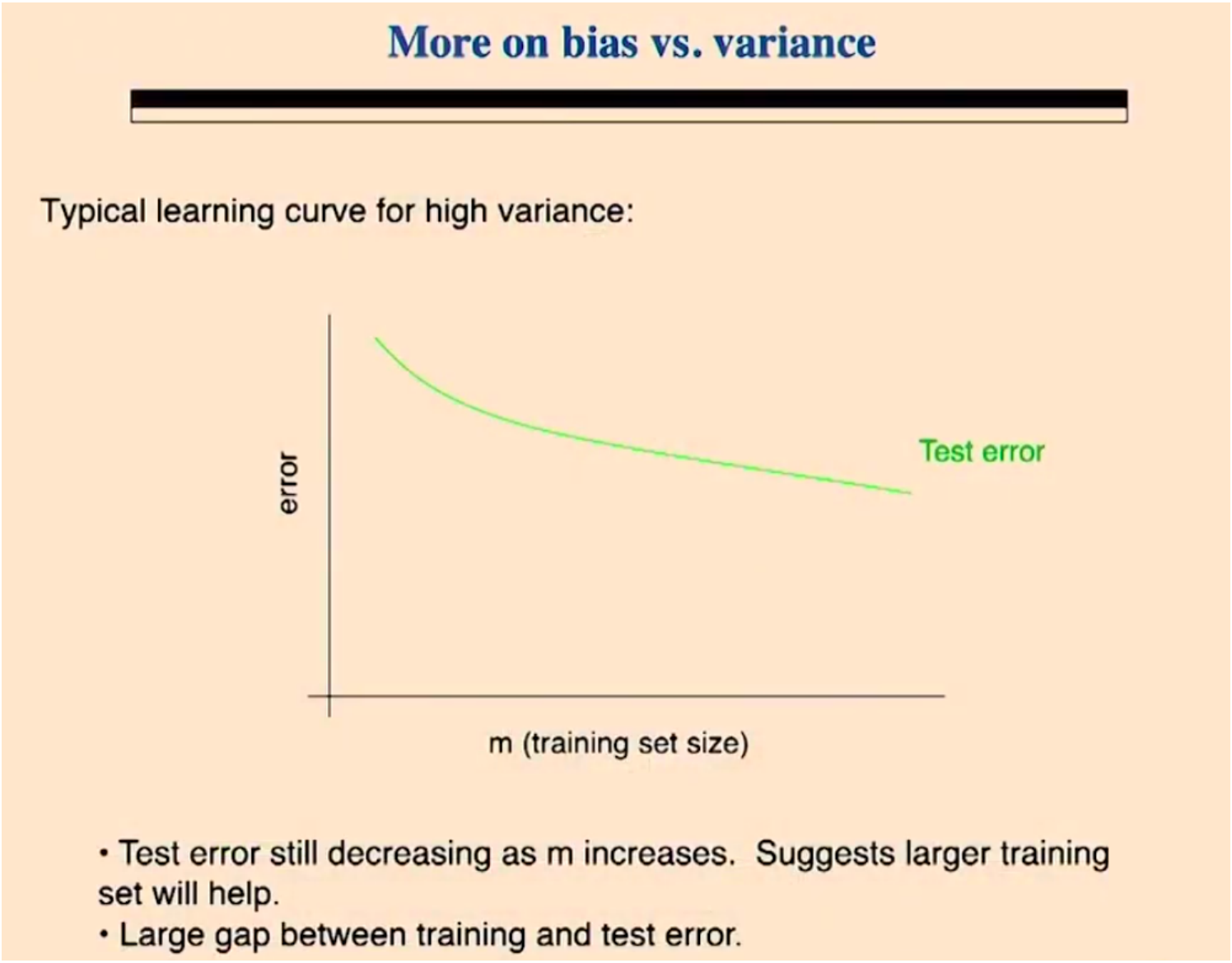
There's a diagnostic that lets you look at your algorithm's performance and try to figure out how much of the problem is ***variance*** and how much of the problem is ***bias***

We said ***test error*** but if you are developing, you should be doing this with a ***dev set*** rather than a ***test set***

To explain this diagnostic in greater detail, it turns out that if you have a ***classifier*** with very high ***variance***, the performance on the **development set** (it would be better practice to use the ***hold-out cross validation*** for the ***test set***), you'll see that the error that you classify has much lower error on the ***training set*** than on the ***development set***

In contrast, if you have ***high bias***, then the ***training error*** and the ***test set error*** and the ***dev set error*** will both be high

We'll illustrate this with a picture:

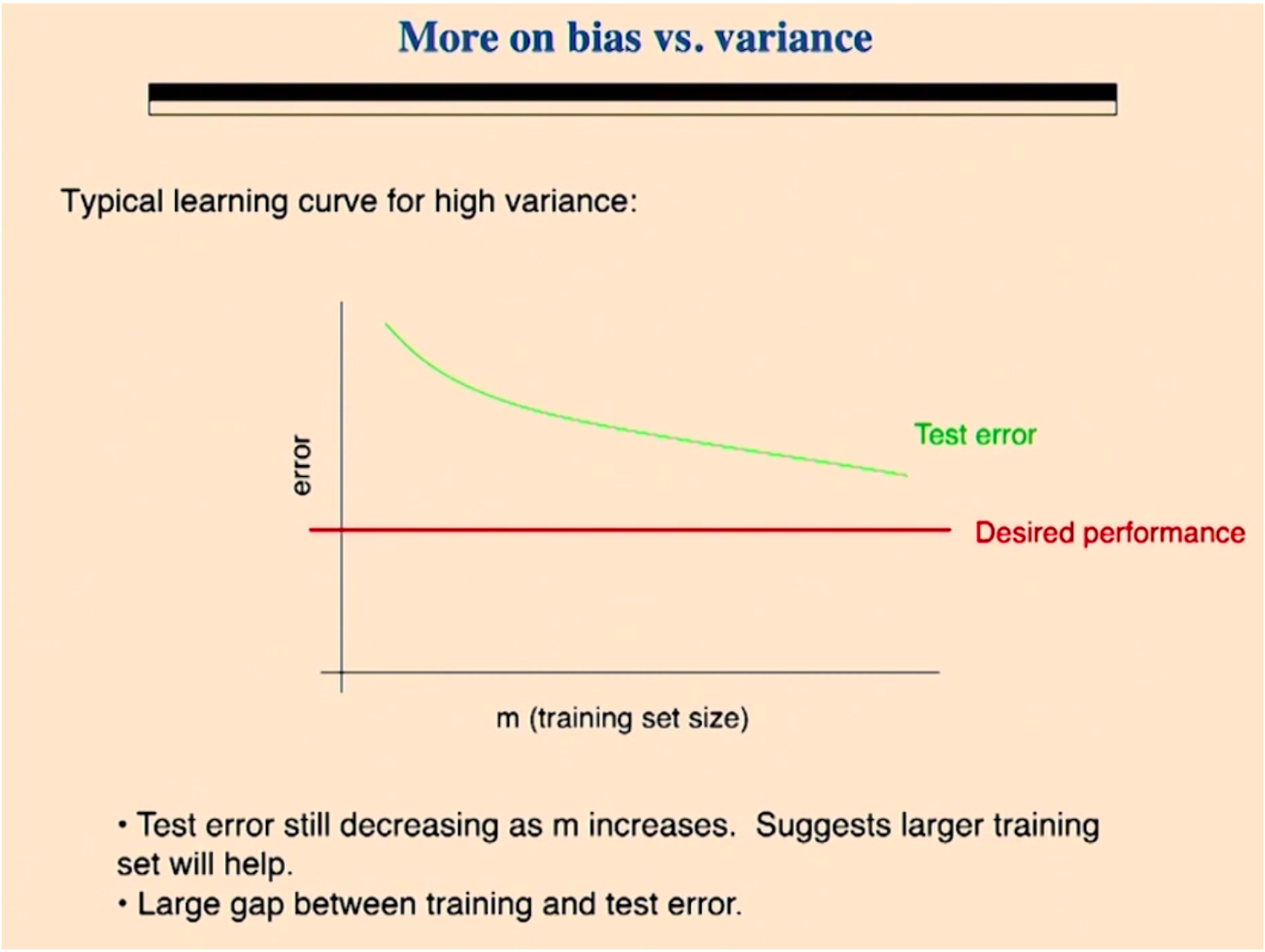


This is a learning curve and what that means is, on the horizontal axis, you're going to vary the number of ***training examples***

When we talked about ***bias*** and ***variance*** before, we had a plot where the horizontal axis was the ***degree of polynomial***. In this plot, the horizontal axis is different. It's the number of ***training examples***

Whenever you train a learning algorithm, the more data you have, usually the better your ***development set error*** and the better your ***test set error***. This error usually goes down when you increase the number of training examples

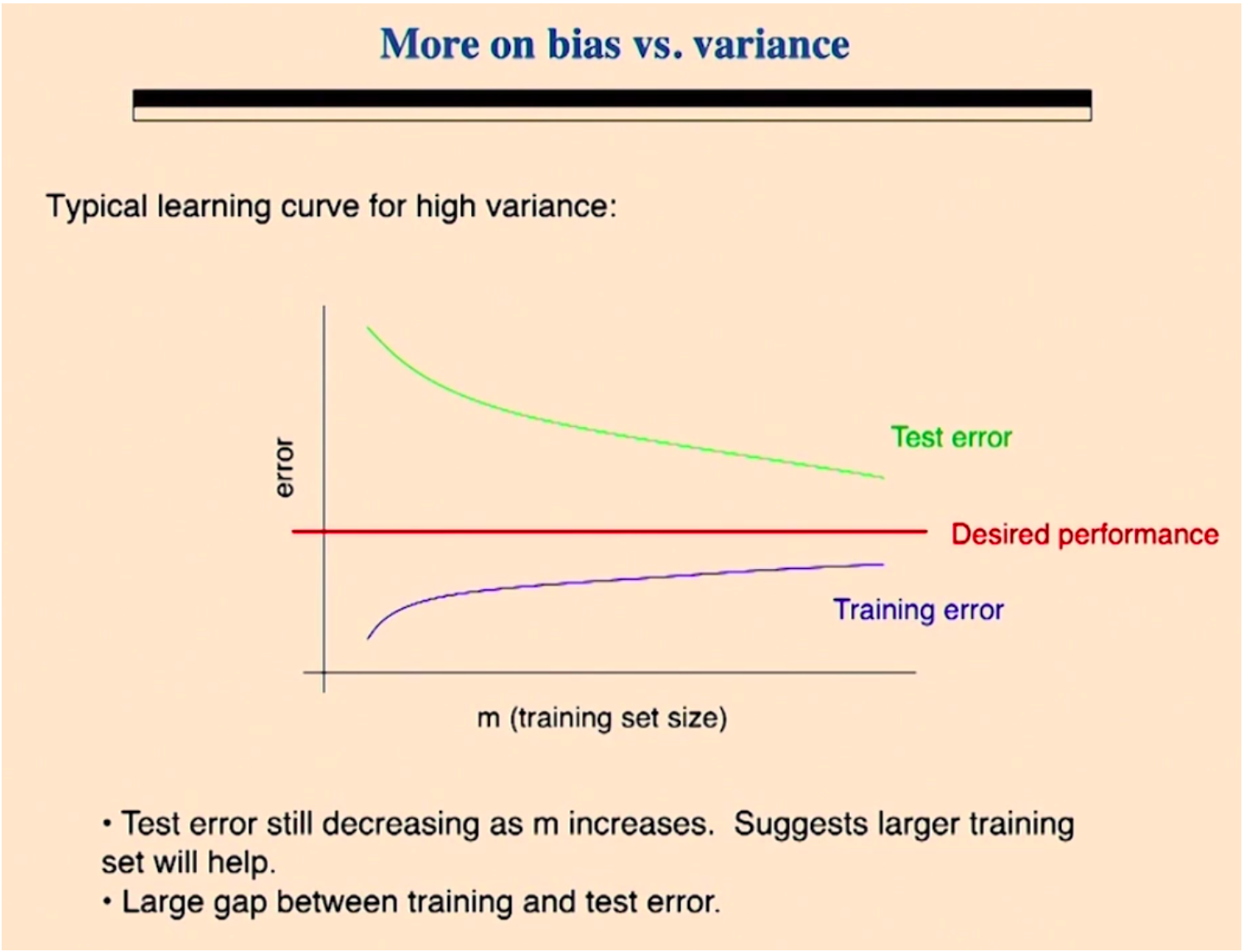
Let's say that you're hoping to achieve a certain level of desired performance and often, sometimes desired level of performance is to do about as well as a human can. That's a common business objective depending on your application, but sometimes it can be different



The other plot to add to this, which will help you analyze ***bias vs variance*** is to plot the ***training error***

One seen property of ***training error*** is that it increases as the ***training set*** size increases. Let's say you're building a spam classifier and you have only one training example, then any algorithm can fit one training example perfectly. And so, if your ***training set*** size is very small, the ***training set error*** is usually **0**. It's only if you have a bigger ***training set*** that it becomes harder for the learning algorithm to fit your training data that well. It's only if you have a very large ***training set*** that a classifier like ***logistic regression*** or ***linear regression*** may have a harder time fitting all of your training examples

That's why ***training error*** or ***average training error*** averaged over your ***training set***, generally increases as you increase the ***training set*** size



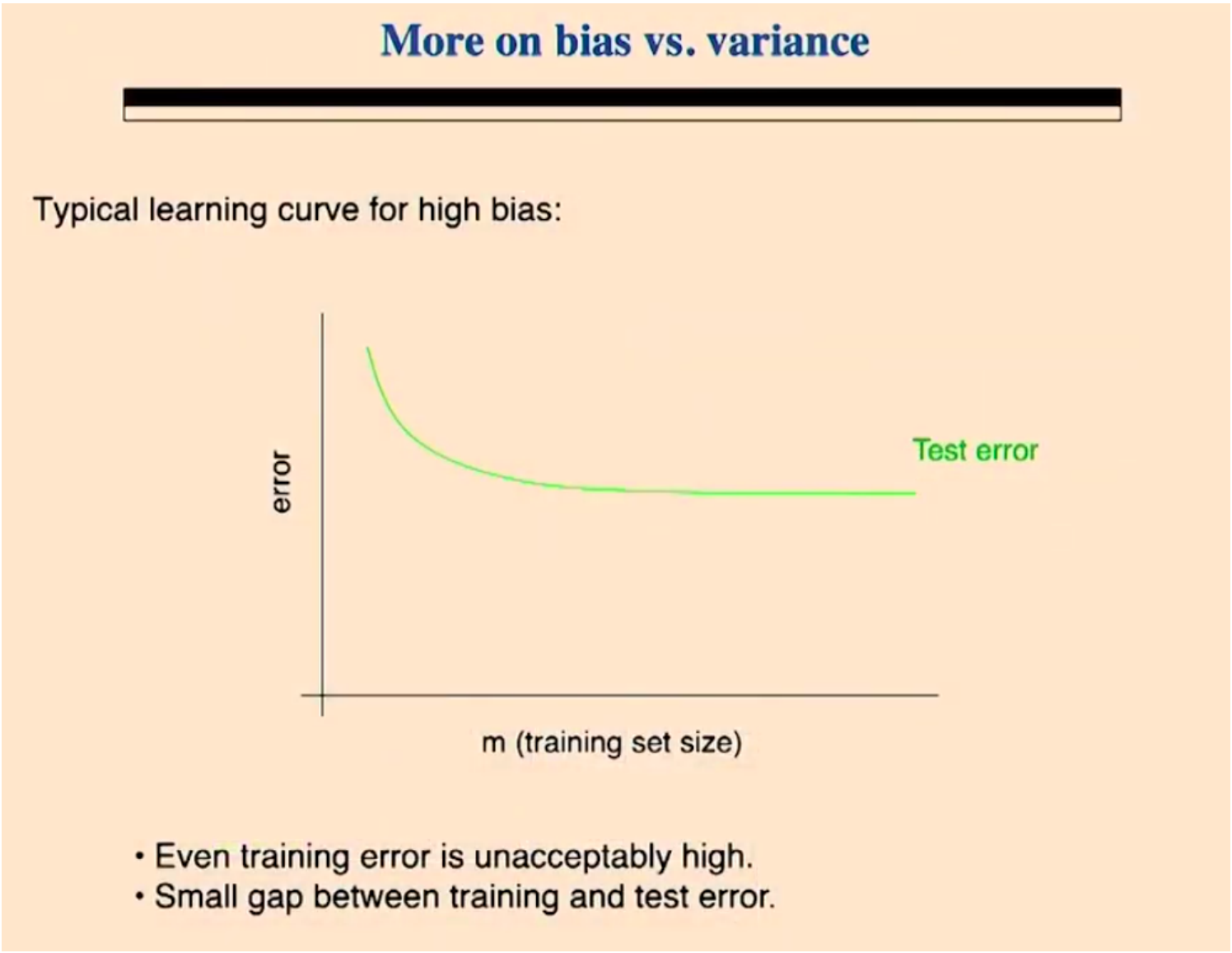
There are two characteristics of this plot that, if you plot the learning curves, suggests that the algorithm has a large ***bias*** problem

The two properties written at the bottom:

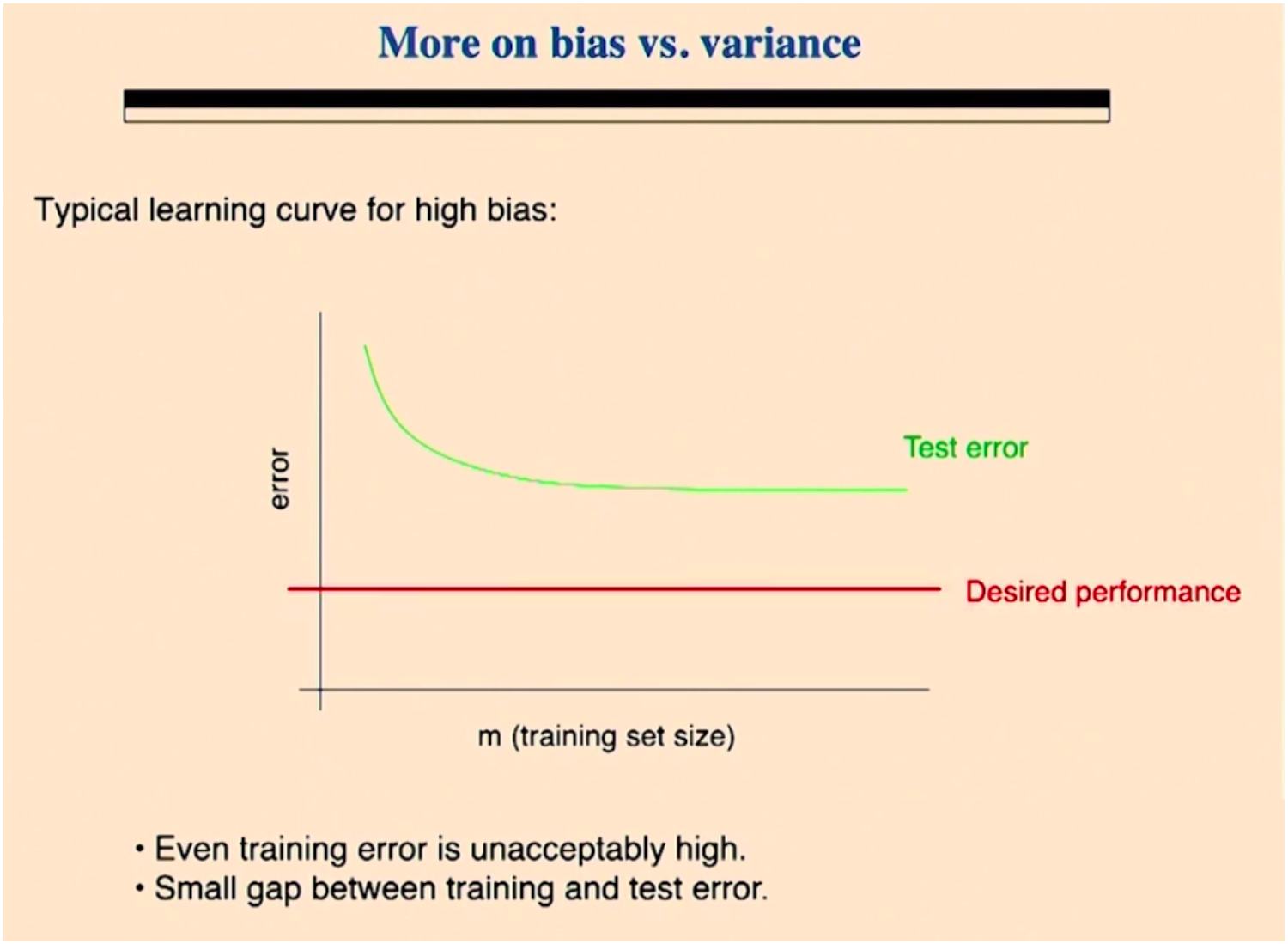
(1) The weaker signal, the one that's harder to rely on, is that the ***development set error*** or the ***test set error*** is still decreasing as you increase the ***training set size***. The **green** curve still looks like it's going down, so this suggests that if you increase the ***training set size*** and extrapolate further to the right, the curve would keep on going down. This turns out to be a weaker signal because sometimes we look at a curve like that, it's actually quite hard to tell (to extrapolate to the right). If you double the ***training set size*** ,how much further would the **green** curve go down? It's hard to tell. This is a useful signal, but sometimes it's a bit hard to judge exactly where the curve will go if you extrapolate to the right

(2) The stronger signal is actually the second one. The fact that there's a huge gap between your ***training error*** and your ***test set error***, or your ***training error*** and your ***dev set error*** which would be the better thing to look at, it's actually a stronger signal that this particular learning algorithm has high **variance**. As you increase the ***training set*** size, you find that the gap between the ***train*** and ***test error*** usually reduces. There's still a lot of room for making your ***test set error*** become closer to your ***training error***, so if you see a learning curve like this, this is a strong sign that you have a ***variance*** problem

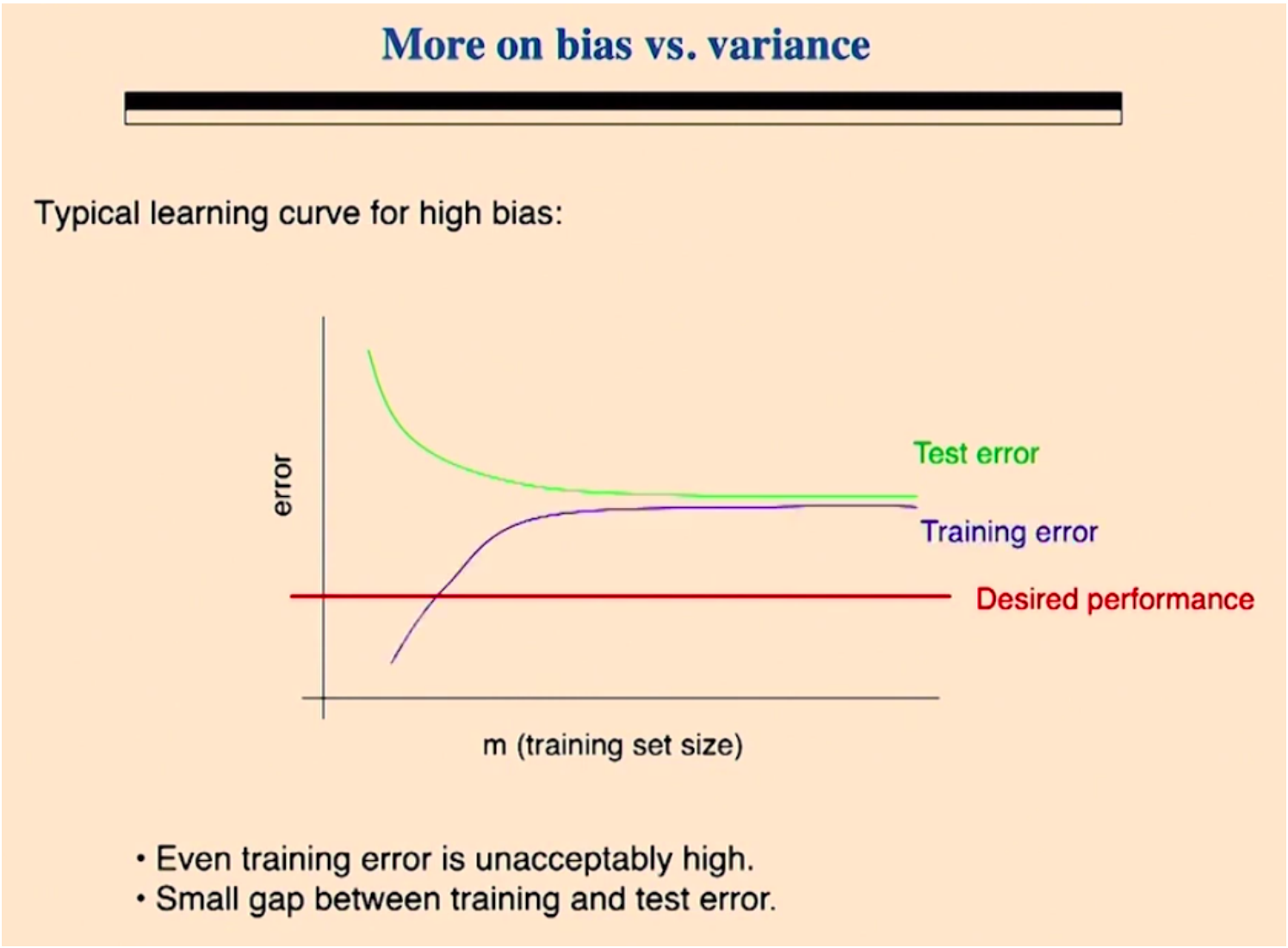
Let's look at what the learning curve will look like if you have a ***bias*** problem:



This is a typical learning curve for high ***bias***. That's your ***dev set error*** or your ***development set holdout cross-validation test error*** and you're hoping to hit a level of performance like this:



Your training error looks like this:

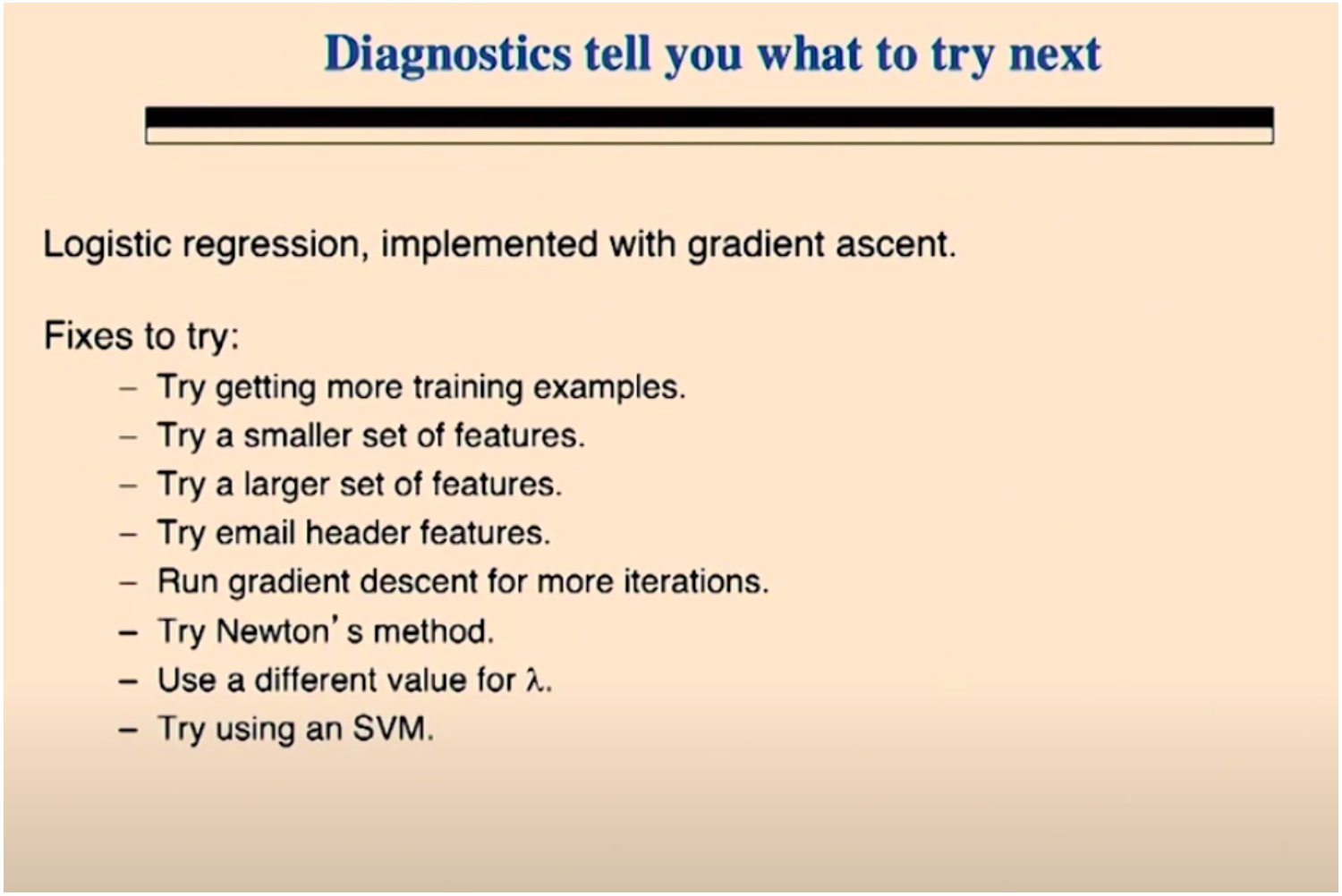


One sign that you have a high ***bias*** problem is that this algorithm is not even doing that well on the ***training set***. Even on the ***training set***, you're not achieving your desired level of performance. Imagine, you're looking at learning algorithms and this algorithm has seen these examples and even for examples its seen, it's not doing as well as you were hoping. So clearly the algorithm's not fitting the data well enough. This is a sign that you have a high ***bias*** problem (not enough features, your learning algorithm is too simple), and the other signal is that there is a very small gap between the ***training*** and the ***test error***

When you see a plot like this, no matter how much more data you get and no matter how far you extrapolate to the right of this plot, the **blue** curve (the ***training error***) is never going to come back down to hit the desired level of performance. Because the ***test set error*** is generally higher than your ***training set error***, no matter how much more data you have, no matter how far you extrapolate to the right, the error is never going to come back down to your desired level of performance

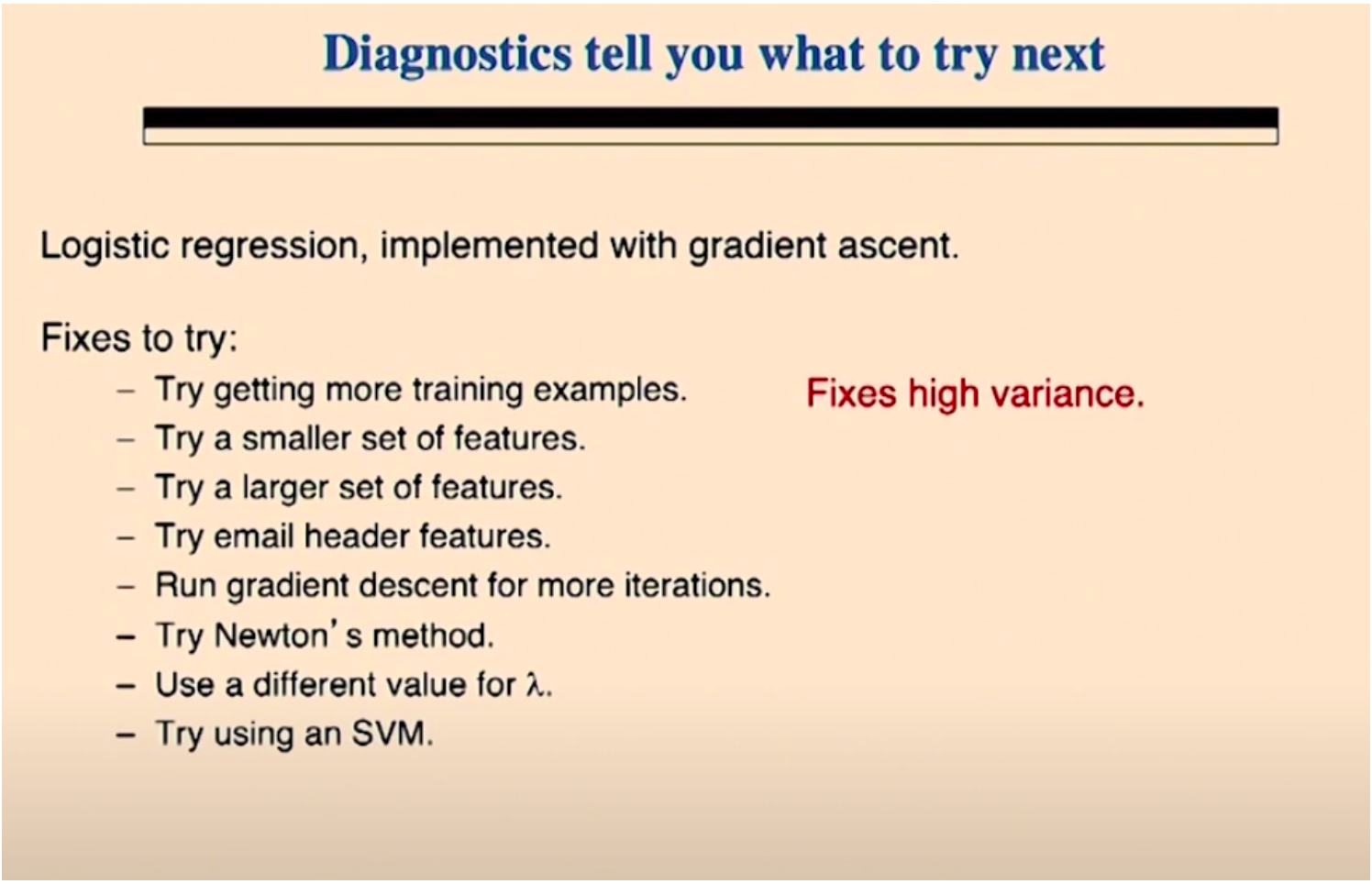
If you get a ***training error*** and ***test error*** curve that looks like this, you kind of know that while getting more training data may help (the **green** curve could come down a little bit), the act of getting more training data by itself will never get you to where you want to go

For each of the first four ideas fixes either a ***high variance*** or a ***high bias*** problem

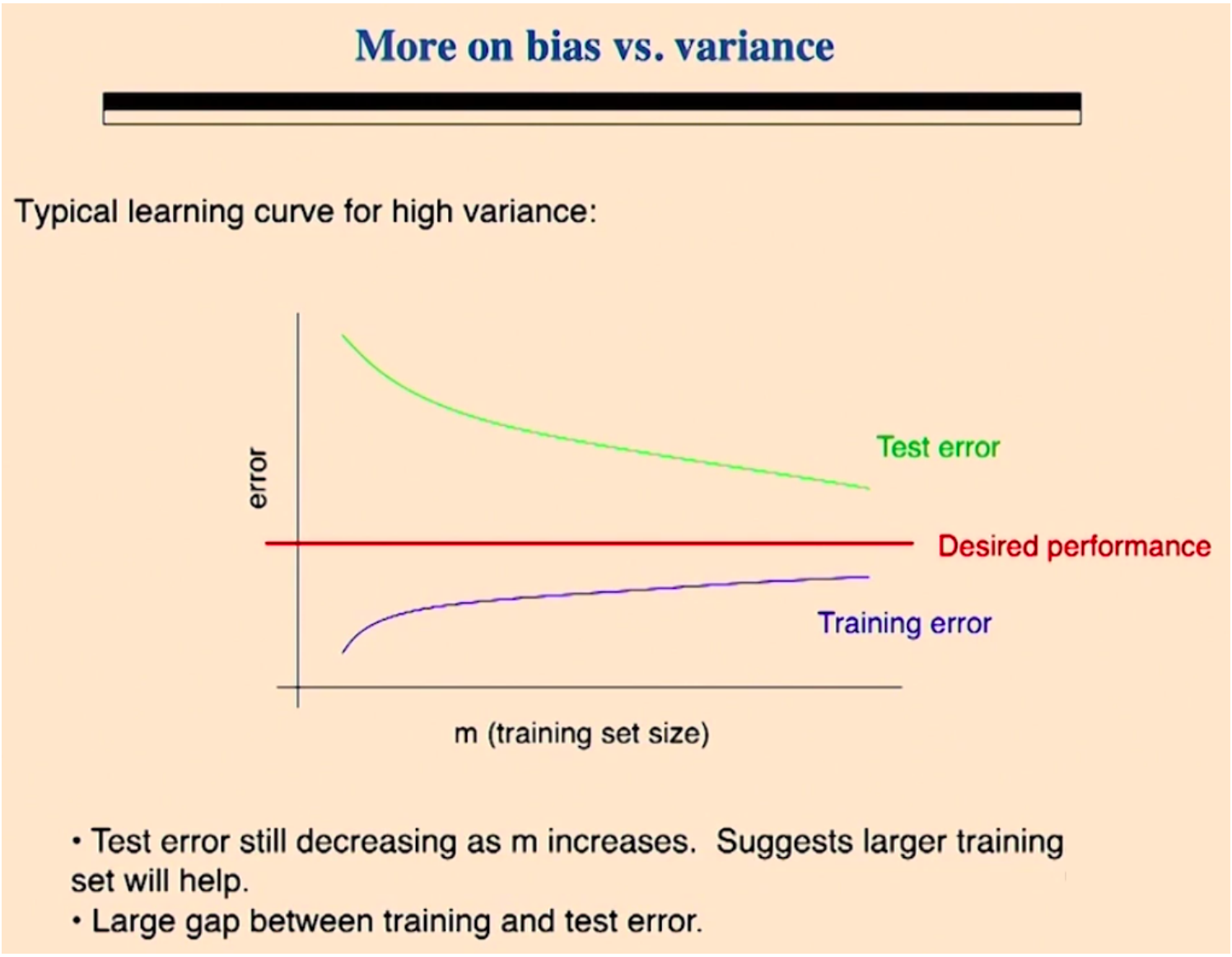


"Try getting more training examples" : This one fixes high ***variance***

If you're fitting a very high order polynomial that overfits, if you have more data then it won't oscillate like that

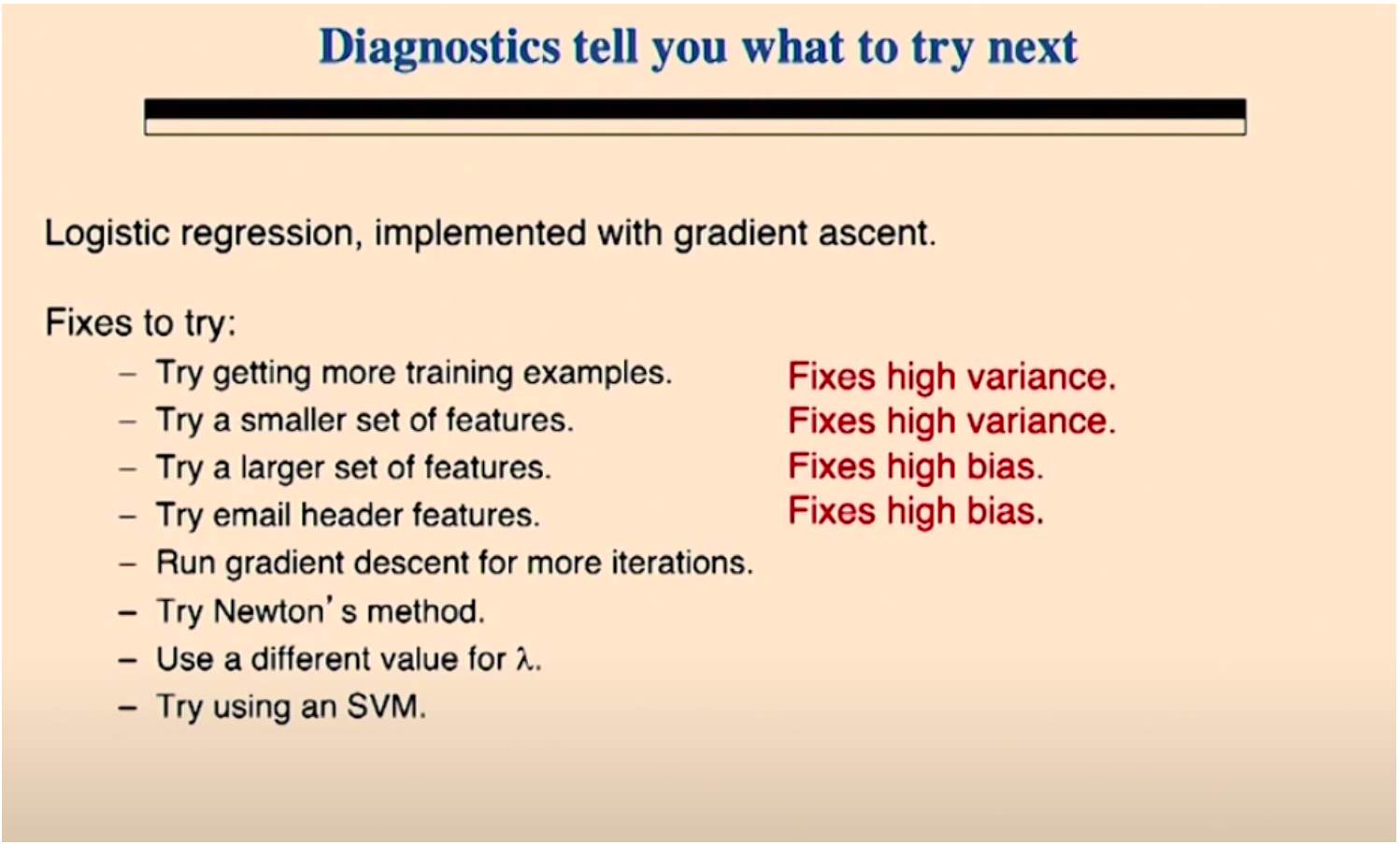


If you look at a high ***variance*** curve:



This is a high ***variance*** plot and if you have a learning algorithm of high ***variance***, if you extrapolate to the right, there is some hope that the **green** curve will keep on coming down. So getting more training data if you have high ***variance***, which is if you're in this situation, looks like it's worth trying. You can't guarantee that it will work, but it's worth trying

#*# The curves will look like this assuming that your training data is ***IID*** (***Independently Identicaly Distributed***); the ***training and dev and test sets*** are all drawn from the same distribution. There is ***learning theory*** that suggests that in most cases the **green** curve should decay as **1/√m**. That's the rate at which it should decay until it reaches some ***Bayes error***. Learning algorithms' error don't always decay to **0**, but the theory says that as **m** increases, it will decay at roughly a rate of **1/√m** toward that baseline error which is called ***Bayes error*** which is the best that you could possibly hope anything could do given how noisy the data is



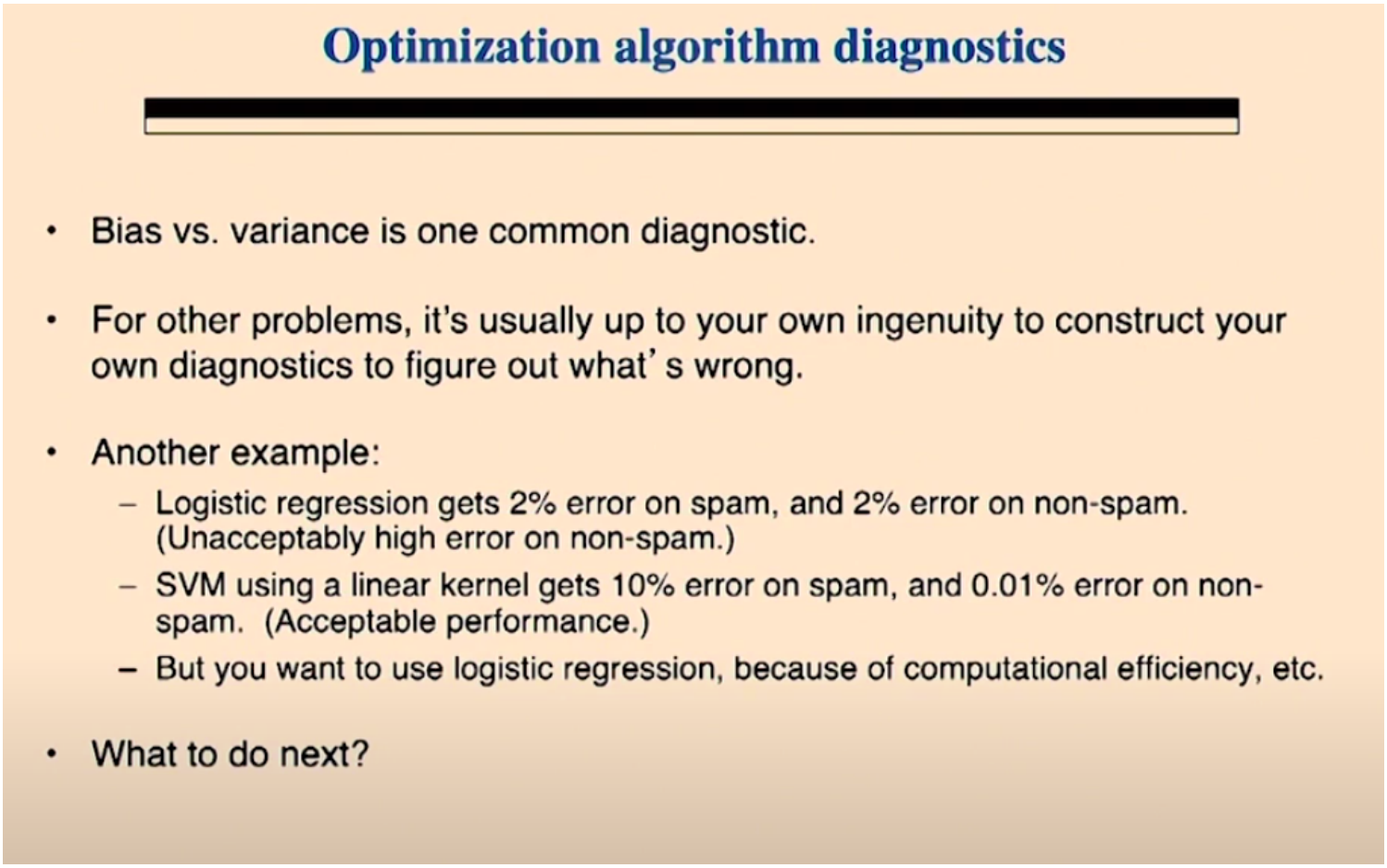
One of the things about building learning algorithms is that, for a new application problem, it's difficult to know in advance if you're going to run into a ***high bias*** or ***high variance*** problem. It's very difficult to know in advance what's going to go wrong with your learning algorithm

In the workflow of how you develop a learning algorithm for a new application, it's recommended that you implement a quick and dirty learning algorithm (such as starting with ***logistic regression***, something simple) so you can run this ***bias-variance*** type of analysis to see what went wrong and then use that to decide what to do next. You could go to a more complex algorithm, you could try adding more data, etc. The one exception to this is if you're working on a domain in which you have a lot of experience

Bias and variance is the single most powerful tool for analyzing the performance of learning algorithms

There's one other pattern that is seen quite often, which is addresses the second set, which is optimization algorithm working. When you implement a learning algorithm, you often have a few guesses for what's wrong, and if you can systematically test if that hypothesis is right before you spend a lot of work to try to fix it, then you could be much more efficient

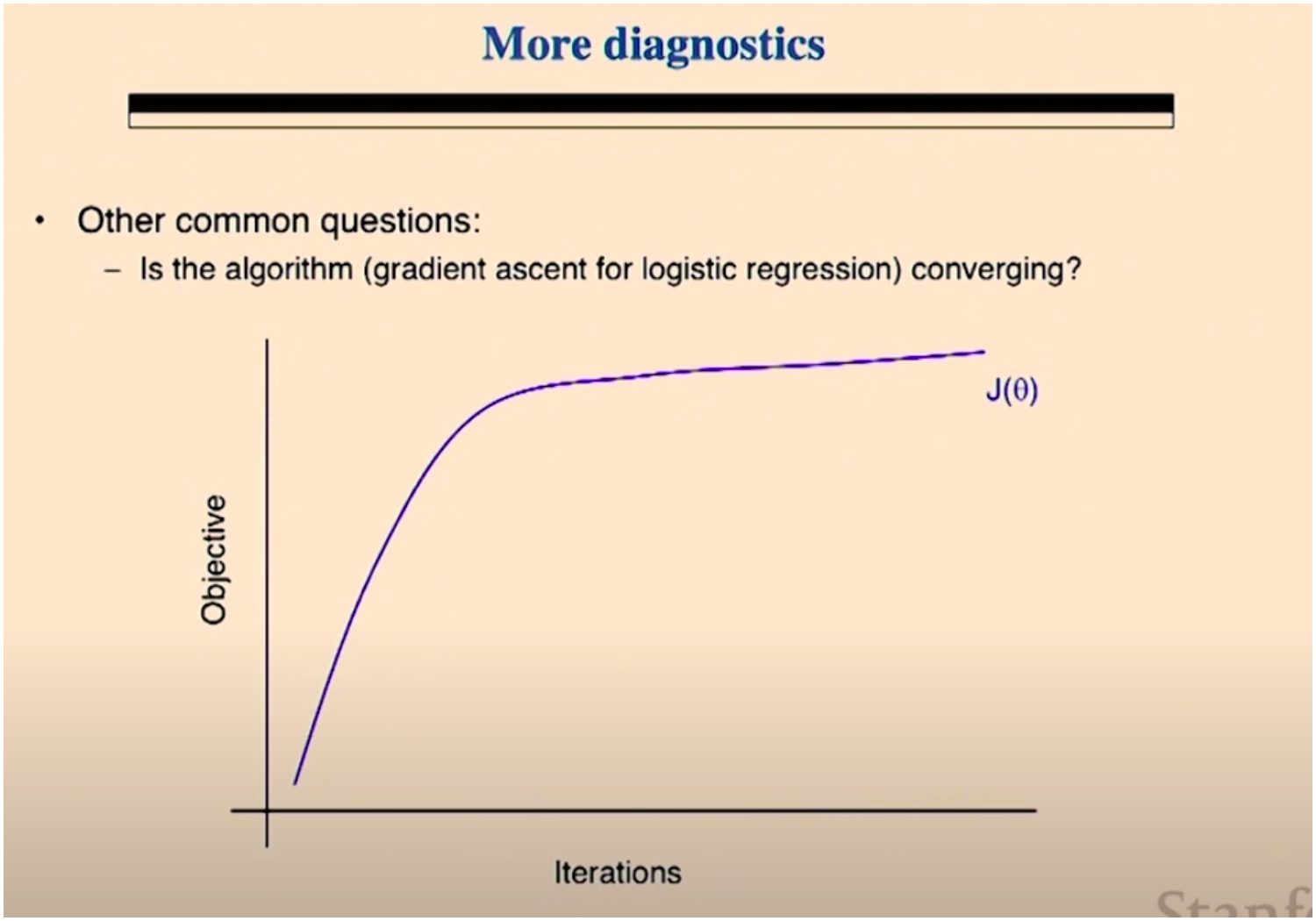
Let's say that you tuned your ***logistic regression*** algorithm for a while and let's say:



Logistic regression is more computationally efficient and it may be easier to update, you get more examples, run a few more iterations of ***gradient descent***, and let's say you want to ship a ***logistic regression*** implementation rather than a ***SVM*** implementation. What do you do next?

It turns out that one common question you have when training your learning algorithm is you often wonder *"is your optimization algorithm converging?"*

One thing you might do is draw a plot of the ***training optimization objective***:



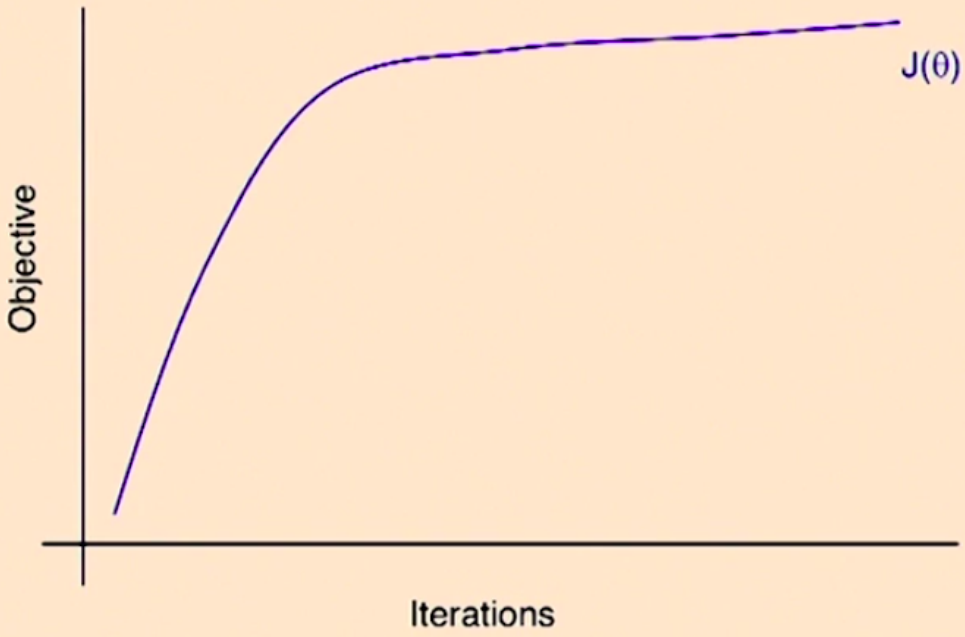
Often, the plot will look like this

The curve is kind of going up but not that fast, and if you train it twice as long or even **10** times as long, will that help? Again, training the algorithm for more iterations pretty much never hurts. If you regularize the algorithm properly, training the algorithm longer almost always helps, but is it the right thing to do to go and burn another 48 hours of CPU/GPU cycles to just train this thing longer and hoping it works better? Maybe, maybe not

Is there a systematic way or a better way to tell if you should invest a lot more time in running the ***optimization algorithm***? Sometimes it's just hard to tell

More diagnostics

- Other common questions:
  - Is the algorithm (gradient ascent for logistic regression) converging?



It's often very hard to tell if an algorithm has converged yet by looking at the objective.

A lot of where this iteration of debugging learning algorithms is looking at what your learning algorithm is doing and just asking yourself *"what are my guesses for what could be wrong?"*, and maybe one of your guesses is *"maybe optimizing the wrong **cost function**"*

More diagnostics

- Other common questions:
  - Is the algorithm (gradient ascent for logistic regression) converging?
  - Are you optimizing the right function?

What you care about is this ***weighted accuracy criteria*** where the sum over your ***dev set*** or ***test set*** of weights o different examples of whether it gets it right, where the weights are higher for non-spam than spam because you really want to make sure you label non-spam email correctly

More diagnostics

- Other common questions:
  - Is the algorithm (gradient ascent for logistic regression) converging?
  - Are you optimizing the right function?
  - I.e., what you care about:

$$a(\theta) = \sum_i w^{(i)} 1\{h_\theta(x^{(i)}) = y^{(i)}\}$$

(weights $w^{(i)}$ higher for non-spam than for spam).

So maybe that's the ***weighted accuracy criteria*** you care about

But for ***logistic regression***, you're maximizing this ***cost function***

More diagnostics

- Other common questions:
  - Is the algorithm (gradient ascent for logistic regression) converging?
  - Are you optimizing the right function?
  - I.e., what you care about:

$$a(\theta) = \sum_i w^{(i)} 1\{h_\theta(x^{(i)}) = y^{(i)}\}$$

(weights $w^{(i)}$ higher for non-spam than for spam).

  - Logistic regression? Correct value for λ?

$$\max_\theta J(\theta) = \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}, \theta) - \lambda||\theta||^2$$

So you're optimizing this **J(Θ)** when what you actually care about is **A(Θ)**. So maybe you're optimizing the wrong ***cost function***. Then one way to change the ***cost function*** would be to fiddle with the parameter **λ**. That's one way to change the definition of **J(Θ)**. Another way to change **J(Θ)** is to just totally change the ***cost function*** you are maximizing, like change it to the ***SVM objective*** and part of that also means choosing the appropriate value for **C**

More diagnostics

- Other common questions:
  - Is the algorithm (gradient ascent for logistic regression) converging?
  - Are you optimizing the right function?
  - I.e., what you care about:

$$a(\theta) = \sum_i w^{(i)} 1\{h_\theta(x^{(i)}) = y^{(i)}\}$$

(weights $w^{(i)}$ higher for non-spam than for spam).

  - Logistic regression? Correct value for λ?

$$\max_\theta J(\theta) = \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}, \theta) - \lambda||\theta||^2$$

  - SVM? Correct value for C?

$$\begin{aligned} \min_{w,b} \quad & ||w||^2 + C\sum_{i=1}^m \xi_i \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} - b) \geq 1 - \xi_i \end{aligned}$$

There is a second diagnostic which is the problem of your optimization algorithm. In other words, *“is gradient ascent not converging?”* or "*is the problem that you're just optimizing the wrong function?*"

There's a diagnostic that can help you figure that out. To summarize this scenario (this running example we're using):

Diagnostic

An SVM outperforms logistic regression, but you really want to deploy logistic regression for your application.

Let θ_{SVM} be the parameters learned by an SVM.

Let θ_{BLR} be the parameters learned by logistic regression. (BLR = Bayesian logistic regression.)

You care about weighted accuracy:

$$a(\theta) = \max_{\theta} \sum_{i} w^{(i)} 1\{h_\theta(x^{(i)}) = y^{(i)}\}$$

θ_{SVM} outperforms θ_{BLR}. So:

$$a(\theta_{SVM}) > a(\theta_{BLR})$$

How can you tell if the problem is your ***optimization algorithm***, meaning that you need to run gradient ascent longer to actually maximize **J(Θ)**?

Diagnostic

An SVM outperforms logistic regression, but you really want to deploy logistic regression for your application.

Let θ_{SVM} be the parameters learned by an SVM.

Let θ_{BLR} be the parameters learned by logistic regression. (BLR = Bayesian logistic regression.)

You care about weighted accuracy:

$$a(\theta) = \max_{\theta} \sum_{i} w^{(i)} 1\{h_\theta(x^{(i)}) = y^{(i)}\}$$

θ_{SVM} outperforms θ_{BLR}. So:

$$a(\theta_{SVM}) > a(\theta_{BLR})$$

BLR tries to maximize:

$$J(\theta) = \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}, \theta) - \lambda||\theta||^2$$

So how do you tell? We have two possible hypotheses you want to distinguish between. One is that the learning algorithm is not actually

finding the value of **Θ** that maximizes **J(Θ)**

For some reason, ***gradient ascent*** is not converging. So that would be a problem with the ***optimization algorithm***

For the problem to be with the ***optimization algorithm***, it means that if only we could have an algorithm that maximizes **J(Θ)**, we would do great, but for some reason gradient ascent isn't doing well. That's one hypothesis. The second hypothesis is that **J(Θ)** is just the wrong function to be optimizing. It's just a bad choice of ***cost function*** that **J(Θ)** is too different from **a(Θ)**, that maximizing **J(Θ)** doesn't give you a classifier that does well on **a(Θ)**, which is what you actually care about. This is a problem setup

#!# Why not maximize a(Θ) directly?

Because **a(Θ)** is non-differentiable. Maximizing **a(Θ)** explicitly is ***NP-hard***, we just don't have great algorithms to try and do that

It turns out there's a diagnostic you could use to distinguish between these two different problems:

Diagnostic

An SVM outperforms logistic regression, but you really want to deploy logistic regression for your application.

Let θ_{SVM} be the parameters learned by an SVM.

Let θ_{BLR} be the parameters learned by logistic regression. (BLR = Bayesian logistic regression.)

You care about weighted accuracy:

$$a(\theta) = \max_{\theta} \sum_{i} w^{(i)} 1\{h_\theta(x^{(i)}) = y^{(i)}\}$$

θ_{SVM} outperforms θ_{BLR}. So:

$$a(\theta_{SVM}) > a(\theta_{BLR})$$

BLR tries to maximize:

$$J(\theta) = \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}, \theta) - \lambda||\theta||^2$$

Diagnostic:

$$J(\theta_{SVM}) > J(\theta_{BLR})?$$

The diagnostic is, check the ***cost function*** that ***logistic regression*** is trying to maximize, so **J**, and compute that ***cost function*** on the parameters found by the ***SVM*** and compute that ***cost function*** on the parameters found by ***Bayesian logistic regression***, and just see which value is higher

There are two cases. Either $\mathbf{J(\Theta_{SVM})}$ is greater or it is less than or equal to. They're just two possible cases

Diagnostic

An SVM outperforms logistic regression, but you really want to deploy logistic regression for your application.

Let θ_{SVM} be the parameters learned by an SVM.

Let θ_{BLR} be the parameters learned by logistic regression. (BLR = Bayesian logistic regression.)

You care about weighted accuracy:

$$a(\theta) = \max_{\theta} \sum_{i} w^{(i)} 1\{h_\theta(x^{(i)}) = y^{(i)}\}$$

θ_{SVM} outperforms θ_{BLR}. So:

$$a(\theta_{SVM}) > a(\theta_{BLR})$$

BLR tries to maximize:

$$J(\theta) = \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}, \theta) - \lambda||\theta||^2$$

Diagnostic:

$$J(\theta_{SVM}) > J(\theta_{BLR})?$$

We're going to copy over this equation. That's just a fact that the ***SVM*** does better than ***Bayesian logistic regression*** on a problem:

Diagnostic

An SVM outperforms logistic regression, but you really want to deploy logistic regression for your application.

Let θ_{SVM} be the parameters learned by an SVM.

Let θ_{BLR} be the parameters learned by logistic regression. (BLR = Bayesian logistic regression.)

You care about weighted accuracy:

$$a(\theta) = \max_{\theta} \sum_{i} w^{(i)} 1\{h_\theta(x^{(i)}) = y^{(i)}\}$$

θ_{SVM} outperforms θ_{BLR}. So:

$$a(\theta_{SVM}) > a(\theta_{BLR})$$

BLR tries to maximize:

$$J(\theta) = \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}, \theta) - \lambda||\theta||^2$$

Diagnostic:

$$J(\theta_{SVM}) > J(\theta_{BLR})?$$

$\lesseqgtr$

We're gonna copy over this first equation, and then we're going to consider these two cases separately:

Diagnostic

An SVM outperforms logistic regression, but you really want to deploy logistic regression for your application.

Let θ_{SVM} be the parameters learned by an SVM.

Let θ_{BLR} be the parameters learned by logistic regression. (BLR = Bayesian logistic regression.)

You care about weighted accuracy:

$$a(\theta) = \max_{\theta} \sum_{i} w^{(i)} 1\{h_\theta(x^{(i)}) = y^{(i)}\}$$

θ_{SVM} outperforms θ_{BLR}. So:

$$a(\theta_{SVM}) > a(\theta_{BLR})$$

BLR tries to maximize:

$$J(\theta) = \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}, \theta) - \lambda||\theta||^2$$

Diagnostic:

$$J(\theta_{SVM}) > J(\theta_{BLR})?$$

≤

So **>** will be ***case (1)*** and **≤** will be ***case (2)***

Two cases

Case 1:

$$a(\theta_{SVM}) > a(\theta_{BLR})$$
$$J(\theta_{SVM}) > J(\theta_{BLR})$$

But BLR was trying to maximize J(θ). This means that θ_{BLR} fails to maximize J, and the problem is with the convergence of the algorithm. Problem is with optimization algorithm.

In ***case (1)***, $\mathbf{J(\Theta_{SVM})} > \mathbf{J(\Theta_{BLR})}$, meaning that, whatever the **SVM** was doing, it found a value for **Θ**, which we have written as:

Two cases

Case 1: $a(\theta_{SVM}) > a(\theta_{BLR})$

$J(\theta_{SVM}) > J(\theta_{BLR})$

But BLR was trying to maximize J(θ). This means that θ_{BLR} fails to maximize J, and the problem is with the convergence of the algorithm. Problem is with optimization algorithm.

Θ_{SVM} has a higher value on the ***cost function*** **J** than Θ_{BLR}

Bayesian logistic regression was trying to maximize **J(Θ)**. ***Bayesian logistic regression*** is just using gradient ascent to try to maximize **J(Θ)**

So, under ***case (1)***, this shows that whatever the ***SVM*** was doing, it managed to find a value for **Θ** that actually achieves a higher value of **J(Θ)** than your implementation of ***Bayesian logistic regression***. This means that Θ_{BLR} fails to maximize the ***cost function*** **J**, and that the problem is with the ***optimization algorithm***

Two cases

Case 1: $a(\theta_{SVM}) > a(\theta_{BLR})$

$J(\theta_{SVM}) > J(\theta_{BLR})$

But BLR was trying to maximize J(θ). This means that θ_{BLR} fails to maximize J, and the problem is with the convergence of the algorithm. Problem is with optimization algorithm.

Case 2: $a(\theta_{SVM}) > a(\theta_{BLR})$

$J(\theta_{SVM}) \leq J(\theta_{BLR})$

For ***case (2)***, we're just copying over the first equation because it's just part of our analysis (it is part of the problem setup), and the second line is now a **≤** sign

Two cases

Case 1: $a(\theta_{SVM}) > a(\theta_{BLR})$

$J(\theta_{SVM}) > J(\theta_{BLR})$

But BLR was trying to maximize J(θ). This means that θ_{BLR} fails to maximize J, and the problem is with the convergence of the algorithm. Problem is with optimization algorithm.

Case 2: $a(\theta_{SVM}) > a(\theta_{BLR})$

$J(\theta_{SVM}) \leq J(\theta_{BLR})$

This means that BLR succeeded at maximizing J(θ). But the SVM, which does worse on J(θ), actually does better on weighted accuracy a(θ).

This means that J(θ) is the wrong function to be maximizing, if you care about a(θ). Problem is with objective function of the maximization problem.

If you look at the second equation, it looks like Bayesian logistic regression did a better job than the SVM of maximizing **J(Θ)**

You tell ***Bayesian logistic regression*** to maximize **J(Θ)** and it found a value of **Θ** that achieves a higher value of **J(Θ)** than the ***SVM***. It did a good job of trying to find a value of **Θ** that drives up **J(Θ)** as much as possible

Two cases

Case 1: $a(\theta_{SVM}) > a(\theta_{BLR})$

$J(\theta_{SVM}) > J(\theta_{BLR})$

But BLR was trying to maximize J(θ). This means that θ_{BLR} fails to maximize J, and the problem is with the convergence of the algorithm. Problem is with optimization algorithm.

Case 2: $a(\theta_{SVM}) > a(\theta_{BLR})$

$J(\theta_{SVM}) \leq J(\theta_{BLR})$

This means that BLR succeeded at maximizing J(θ). But the SVM, which does worse on J(θ), actually does better on weighted accuracy a(θ).

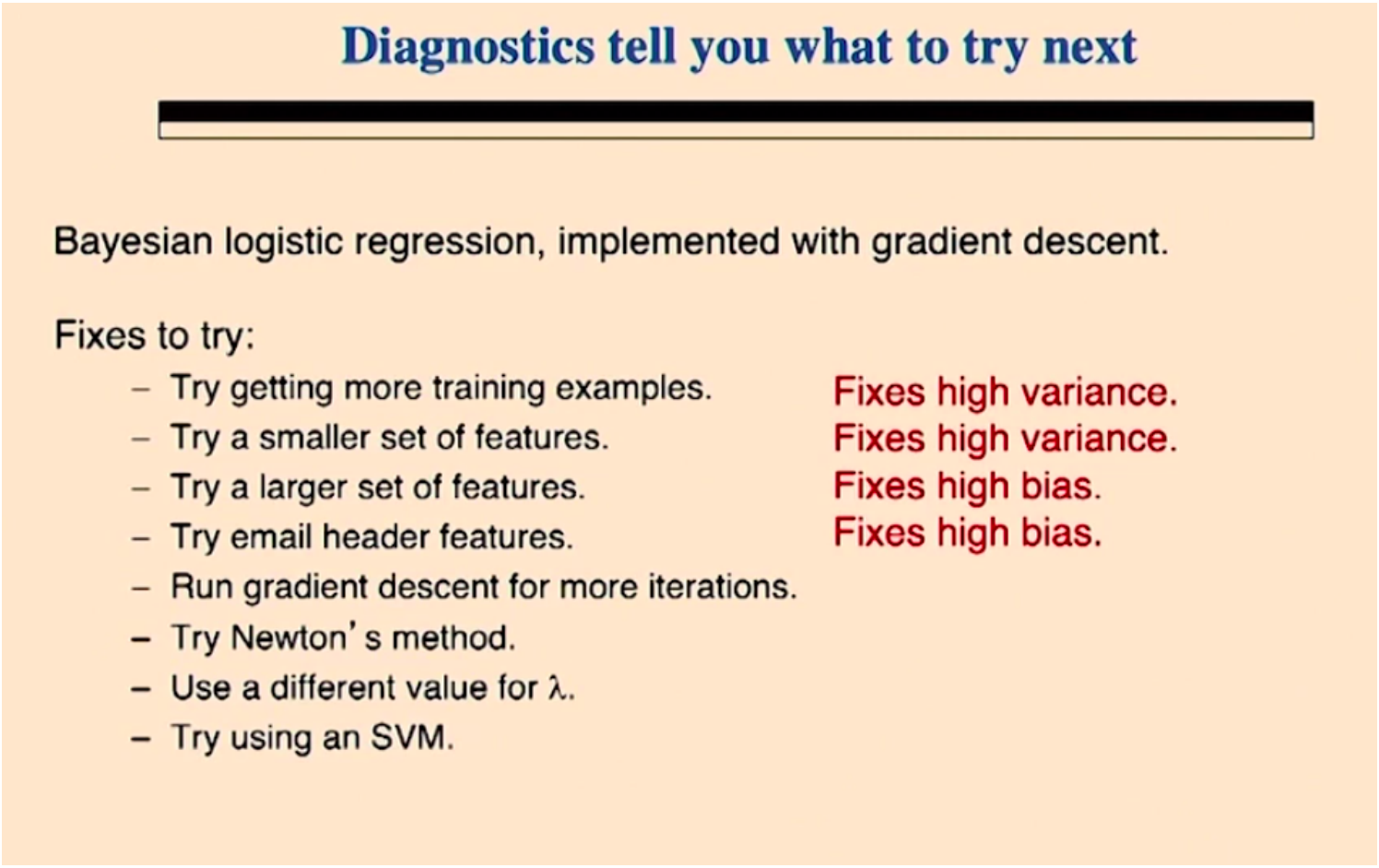
This means that J(θ) is the wrong function to be maximizing, if you care about a(θ). Problem is with objective function of the maximization problem.

But if you look at these two equations in combination, what we have is that the ***SVM*** does worse on the ***cost function*** **J**, but it does better on the thing you actually care about, which is **a(Θ)**

What these two equations in combination tells you, is that having the best value (thei highest value for **J(Θ)**, does not correspond to having the best possible value for **a(Θ)**. It tells you that maximizing **J(Θ)**, doesn't mean you're doing a good job on **a(Θ)**, and therefore maybe **J(Θ)** is not such a good thing to be maximizing. Maximizing it doesn't actually give you the result you ultimate;y care about

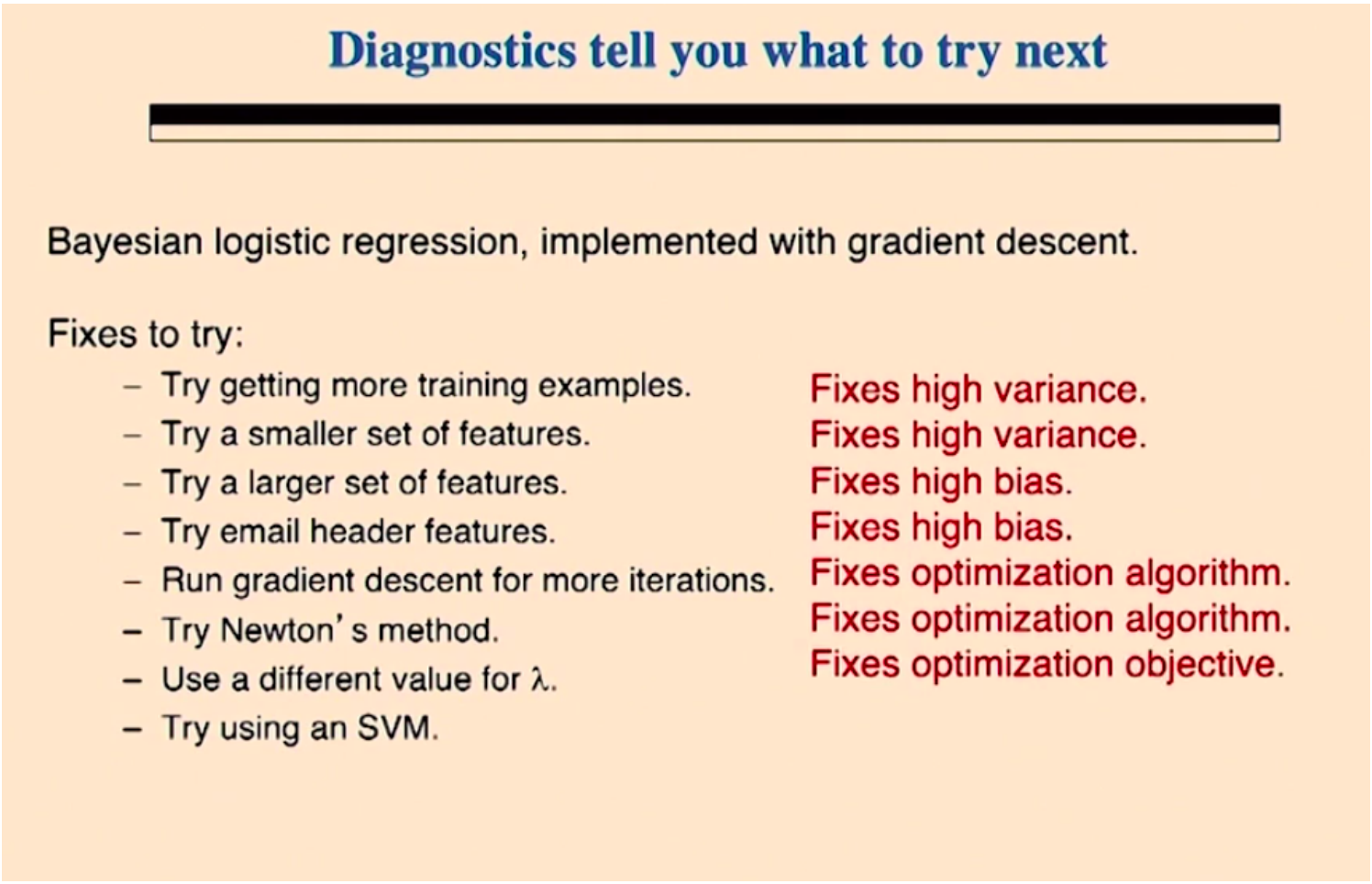
Under ***case (2)***, you can be convinced that **J(Θ)** is not the best function to be maximizing because getting a high value of **J(Θ)** doesn't get you a high value for what you actually care about. So the problem is with the ***objective function*** of the ***maximization problem***, and maybe we should just find a different function to maximize

#!# For these second four bullet points, does it fix the ***optimization algorithm*** or does it fix the ***optimization objective***?



Newton's method still looks at the same ***cost function*** **J(Θ)**, but in some cases it just optimizes it much more efficiently

Usually you fiddle with **λ** to trade off ***bias*** and ***variance*** things, but this is one way to change the ***optimization objective***. Usually you change **λ** to adjust ***bias*** and ***variance*** rather than this



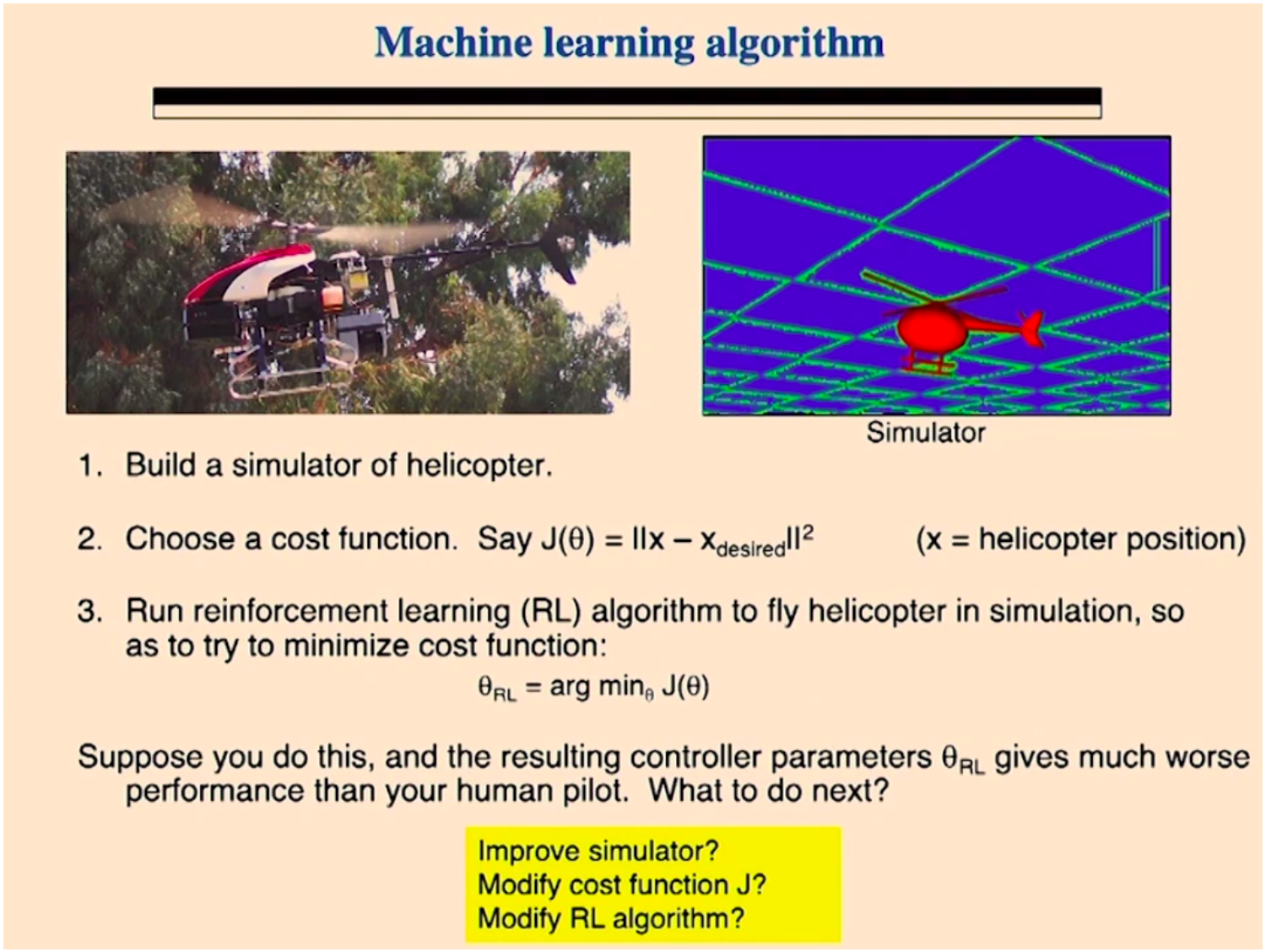
Try to use an ***SVM*** would be one way to totally change the ***optimization objective***

Sometimes when you find you have the wrong ***optimization objective***, there isn't always an obvious thing to do. Sometimes you have to brainstorm a few ideas. There isn't always one obvious thing to try, but at least it tells you that "*category of things*" of trying out different ***optimization objectives***, is what you want

Now we'll go through a more complex example that will illustrate some of these concepts that we've been going through

If you go to ***Andrew Ng's*** office, grab this helicopter, and are asked to write a piece of code (use the learning algorithm) to make this fly by itself, how do you go about doing so?

A good way to a helicopter fly by itself is to do the following:



Step (1) is to build a ***computer simulator*** (***video game simulator***) for a helicopter. The advantage of using a ***video game simulator*** of a helicopter, is you could try a lot of things, crash a lot in the simulation because it's cheap whereas crashing a helicopter in real life is slightly dangerous and more expensive

Step (2) is to choose a ***cost function***. Today we're just using a relatively simple cost function which is ***squared error***. You want the helicopter to fly to the position $\mathbf{x_{desired}}$ and your helicopter instead wandered off to some other place **x**, so we'll use a ***squared error*** to penalize it

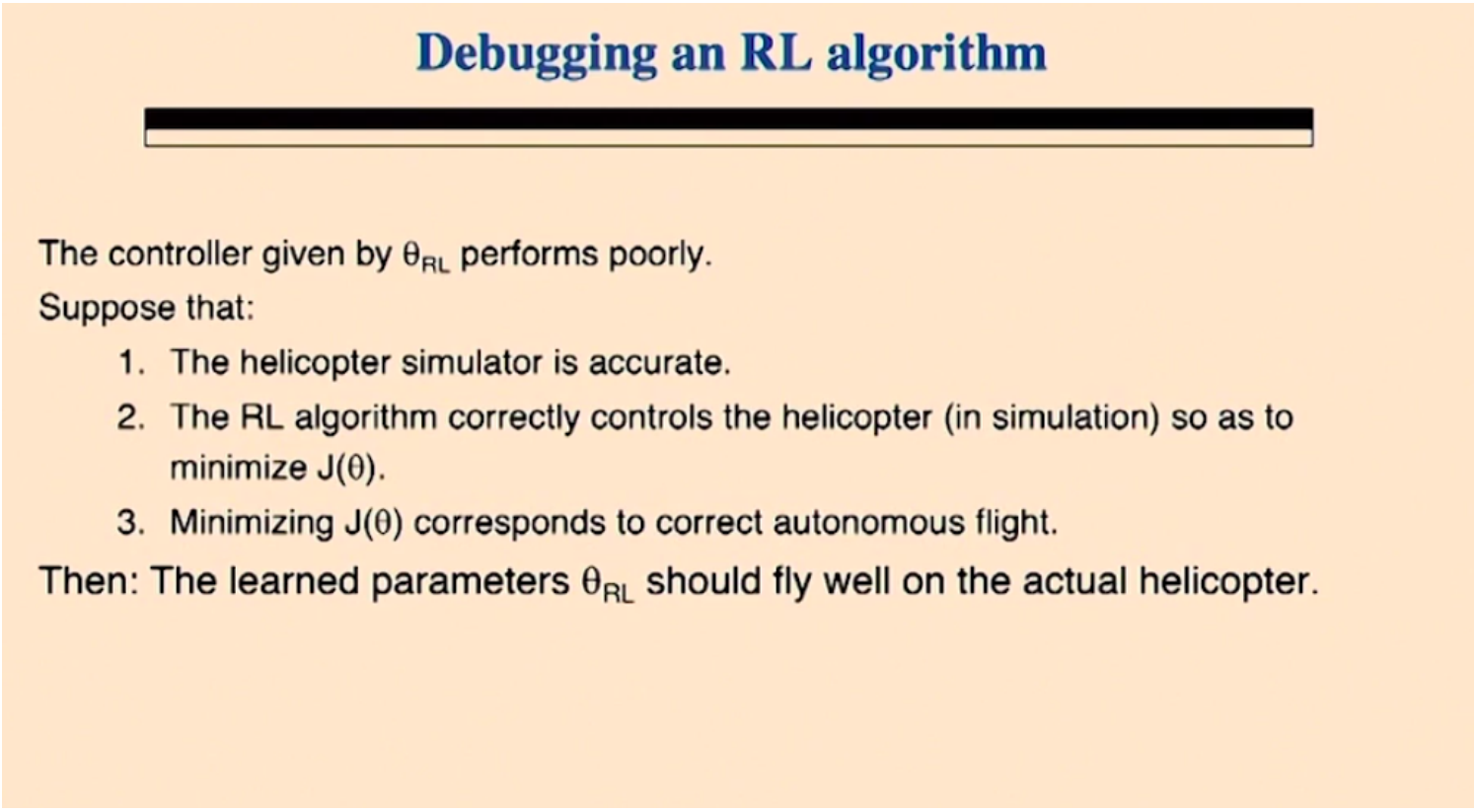
Step (3) Run a ***reinforcement algorithm*** and what the ***reinforcement algorithm*** does is it tries to minimize that cost function **J(Θ)**. You learn some set of parameters $\mathbf{\Theta_{RL}}$ for controlling the helicopter

Let's say you do this and the resulting ***controller*** (the way you fly the helicopter) gets much worse performance than your human pilot. So the helicopter wobbles all over the place and doesn't quite stay where you are hoping it wil. What do you do next? We have some options corresponding to the above three steps. You could work on improving your helicopter flight simulator, or maybe you think that the ***cost function*** is messed up (maybe ***squared error*** isn't the best metric because your optimal orientation is not **0**), or maybe you want to modify ***reinforcement learning algorithm*** because you secretly suspect that your algorithm in not doing a great job of minimizing that ***cost function*** (that it's not actually finding the value of **Θ** that absolutely minimizes **J(Θ)**

Each one of these topics can easily be a ***PhD thesis***. You could definitely work for six years on any one of these topics. The problem is "maybe your helicopter simulator is good enough and you can spend six years improving it but will that actually get you the result? It's not totally clear if that's the key thing for you to spend time on

We'll describe a set of diagnostics that allows you to use this sort of logical step-by-step reasoning to debug which of these three things is what you should actually be spending time on

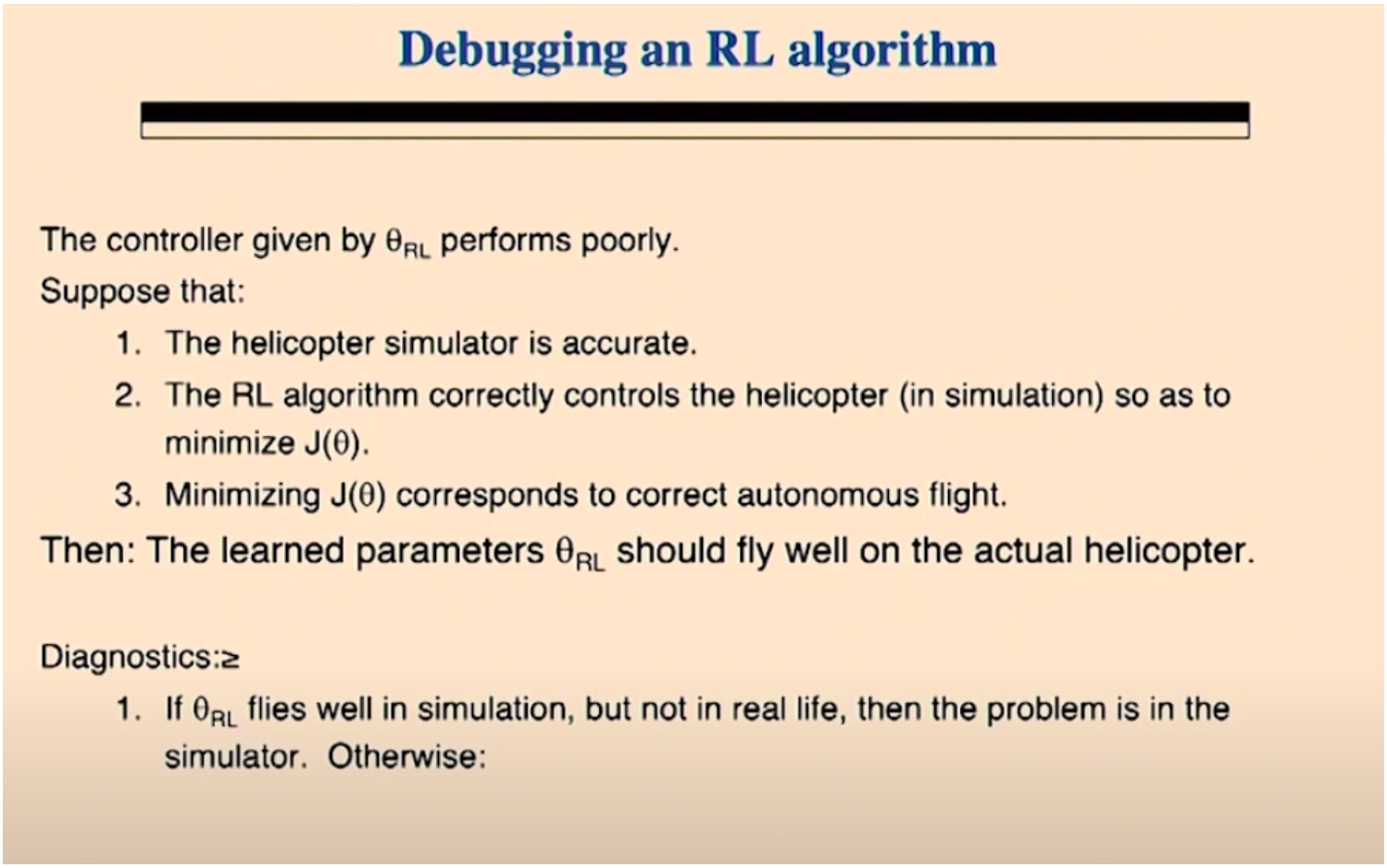
Just to summarize a scenario, The ***controller*** given by $\mathbf{\Theta_{RL}}$ performs poorly. This is how we would reason through a learning algorithm:



Suppose all of these things were true. Suppose that, corresponding to the previous three steps, suppoe the helicopter simulator was accurate and suppose the learning algorithm correctly minimizes the ***cost function*** and suppose **J(Θ)** is a good ***cost function***. If all of these things were true, then the learned parameters should fly well on the actual helicopter, but it doesn't fly well on the helicopter. So one of these three things is false and our job is to identify at least one of these three statements, **(1)**, **(2)**, or **(3)**, that's false because that lets you sink your teeth into something to work on

To make an analogy to more conventional software debugging: If a big complicated program, and for some reason your program crashes (like a ***core dump*** for example), if you can isolate this big complicated program into one component that crashes, then you can focus your attention on that component that you know crashes for some reason and try to find the bug there. So instead of trying to look over a huge code base, if you could do ***binary search*** or try to isolate the problem in a smaller part of your code base, then you can focus your debugging efforts on that part of your code base, try to figure out why it crashes, and then fix that first. After you fix that, it might still crash, then there may be a second problem to work on but at least you know that trying to fix the first bug seems like a worthwhile thing to do

What we're going to do is come up with a set of diagnostics to isolate the problem to one of these three components



The first step is to look at how well the algorithm flies in the simulation. You ran the algorithm and it resulted in a set of parameters that does not do well on your actual helicopter. So the first thing you will do is just check how well this thing even does in the simulation. There are two possible cases, **(1a)** if it flies well in simulation but doesn't do well in real life, then it means something's wrong with the simulator. This means it's actually worth working on the simulator because, if it's already working well in the simulator, what else could you expect the ***reinforcement learning algorithms*** to do? You told the ***reinforcement learning algorithm*** to go and fly well in the simulator because this is just trained in simulation and it's already doing well in the simulator, so there's not much to improve on there (or at least, it's hard to improve on that)

If you're learning algorithm does well in the simulator but not in real life, then this means that the simulator isn't matching real life well, and so that's strong evidence (strong grounds) for you to spend some time to improve your simulator

#!# Is it ever the case that it flies bad in the simulator but well in real life?

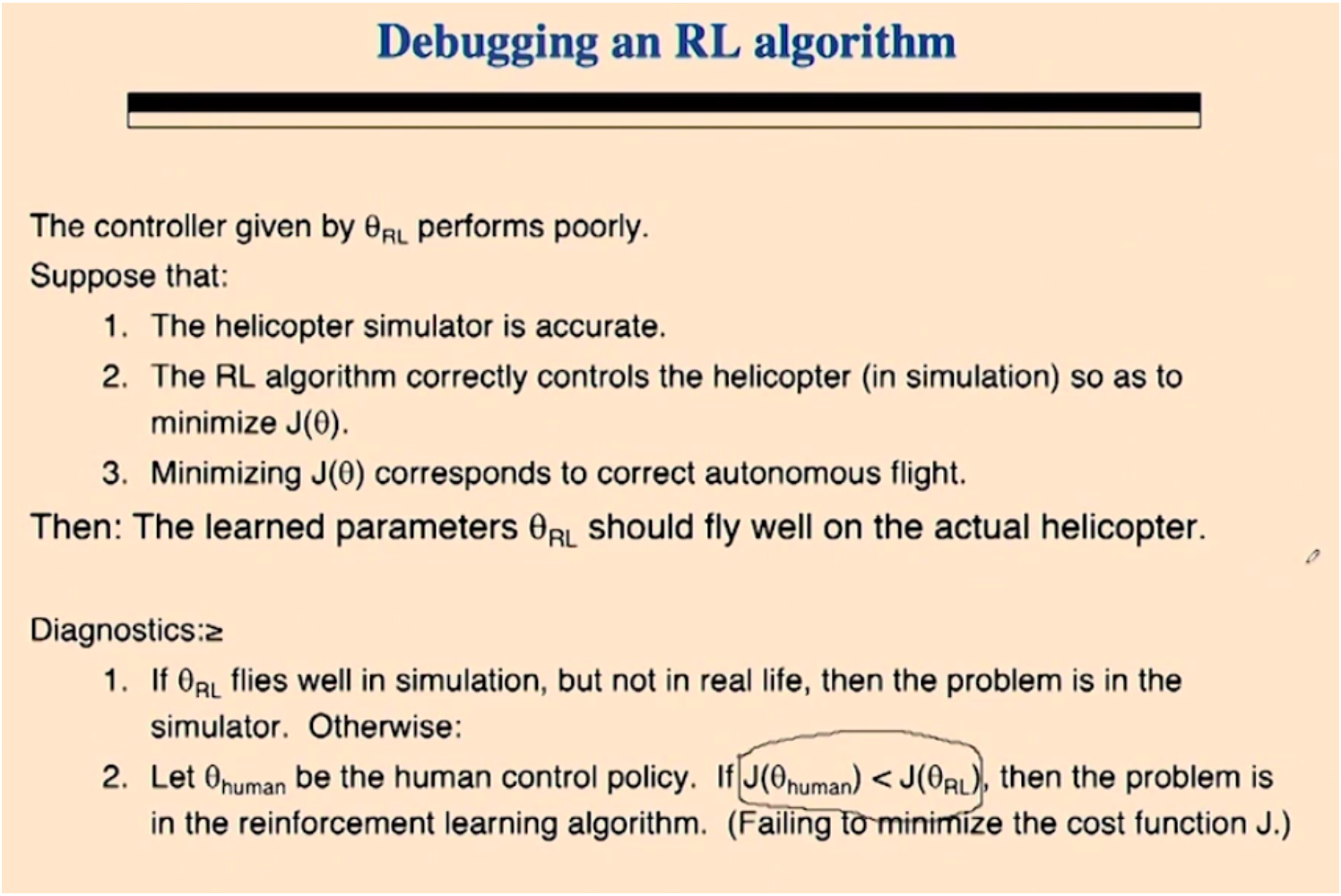
Very rarely There is actually one scenario where that happens. It turns out that when we train this helicopter (or really any robot) in the simulator, we often add a lot of ***noise*** to the simulator because one of the lessons we've learned is that if your simulator is noisy (because simulations are always wrong, any digital simulation is only an approximation to the real world), because we think that if a learning algorithm is robust to all this ***noise*** you've thrown at it in simulation, then whatever the ***noise*** the real world throws at it, it has a bigger chance of being robust too

So we tend to throw a lot of ***noise*** into simulators and one case where that does happen is when we find we threw too much ***noise*** at it in simulation and that might be a sign that we should dial back the ***noise*** a bit

This first diagnostic tells you that you should work on improving the simulation. If there's a big mismatch between ***simulation performance*** and ***real world performance***, that's a good sign that you should improve the simulation

Second, this is actually very similar to the diagnostic we use on the Spam ***Bayesian logistic regression*** and ***SVM*** example, what

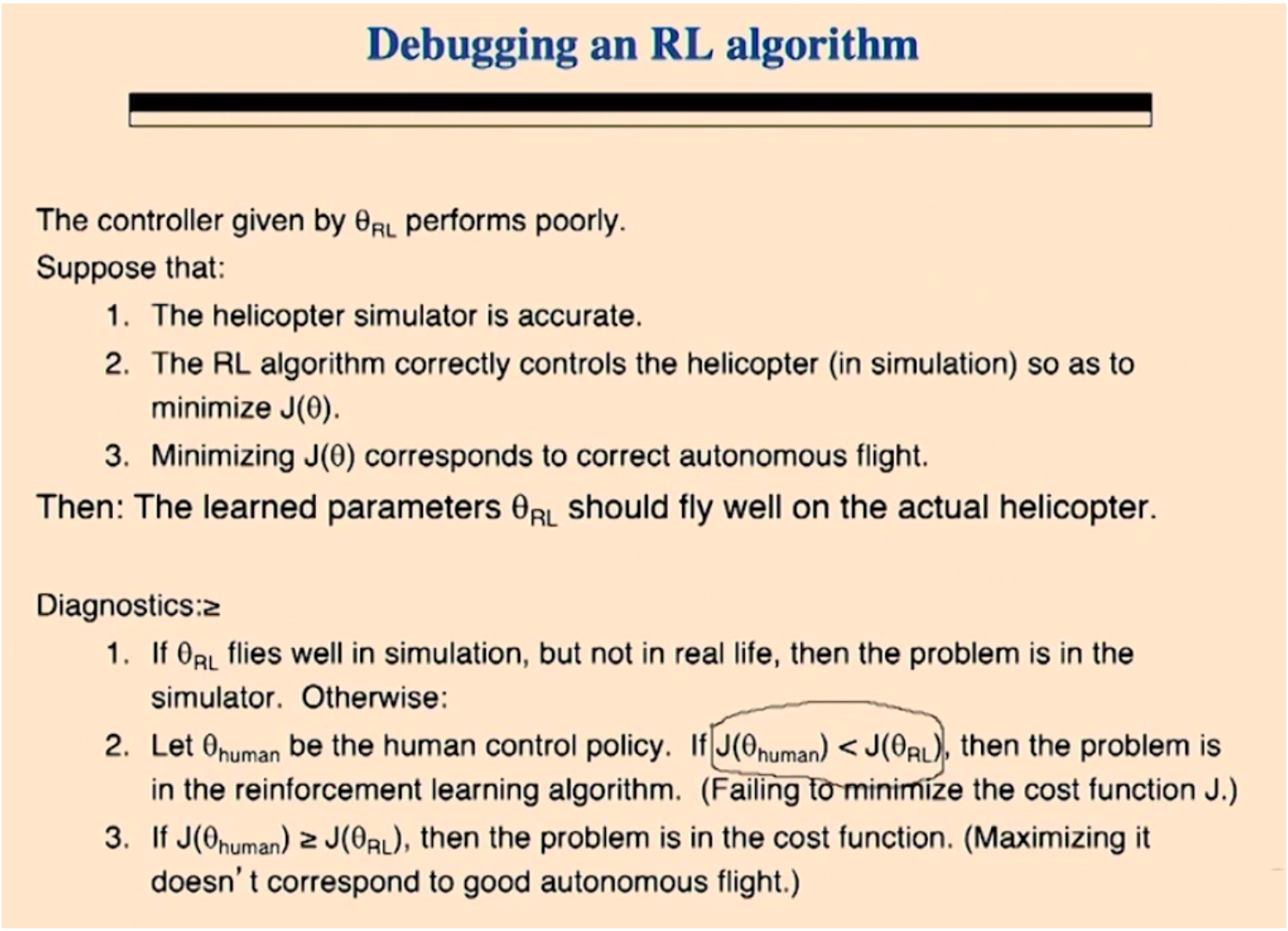
we're going to do is we're going to measure this equation which is, similar to the previous example, take the ***cost function J*** that ***reinforcement learning*** was told to minimize and see if the human achieves better ***squared error*** than the ***reinforcement learning algorithm*** on this ***squared error cost function***



There are two cases, that equation will be either be less than or it will be greater or equal to

Case (1) is say that the $J(\theta_{human}) < J(\theta_{RL})$, which would be this above case. Then that tells you that the problem is with the ***reinforcement learning algorithm***, that somehow the human achieves a lower ***squared error***. So the learning algorithm is not finding the best possible ***squared error***, that is, some other ***controller***, as evidenced by whatever the human is doing, actually achieves a lower ***cost function***. So in this case, we think the ***reinforcement learning algorithm*** is not doing a good job minimizing that and we'll work on the reinforcement learning algorithm

The other case, ***case (2)***, would be if the sign of the inequality is the other way around:



In this case, you can infer that the problem is in the ***cost function*** because what happens here is the human is flying better than your ***reinforcement learning algorithm***, but the human is achieving what looks like a worst ***cost*** than your ***reinforcement learning algorithm***. What this tells you is that minimizing **J(Θ)** does not correspond to flying well. Your learning algorithm achieves a better value for **J(Θ)**

The ***reinforcement learning algorithm***, as far as it knows, is doing a great ob because it's finding a value of **Θ** where **J(Θ)** is really really small, but in this last case, you know that finding such a small value of **J(Θ)** doesn't correspond to flying well because a human doesn't achieve such a good value in the ***cost function*** but the helicopter actually just looks better (was flying in a more satisfactory way). That tells you that this ***squared error cost function*** is not the right ***cost function*** for what flying accurately means

So, through this set of diagnostics, you could decide which one of these three things (improving the ***simulator***, improving the ***reinforcement learning algorithm***, or improving the ***cost function***) is the thing you should work on

What happens in this particular project and what often happens in ***machine learning applications*** is, you run this set of diagnostics, the most acute problems shifts right after you've cleared out one set of problems. It might be the case that now the bottleneck is the simulator. So we often use this workflow to constantly drive prioritization for what to work on next

#!# How do you find the new cost function?

Finding a new ***cost function*** is actually not that easy. It's actually really difficult to write down an equation to specify what is a good way of making a turn, for example

Error analyses and ablative analysis:

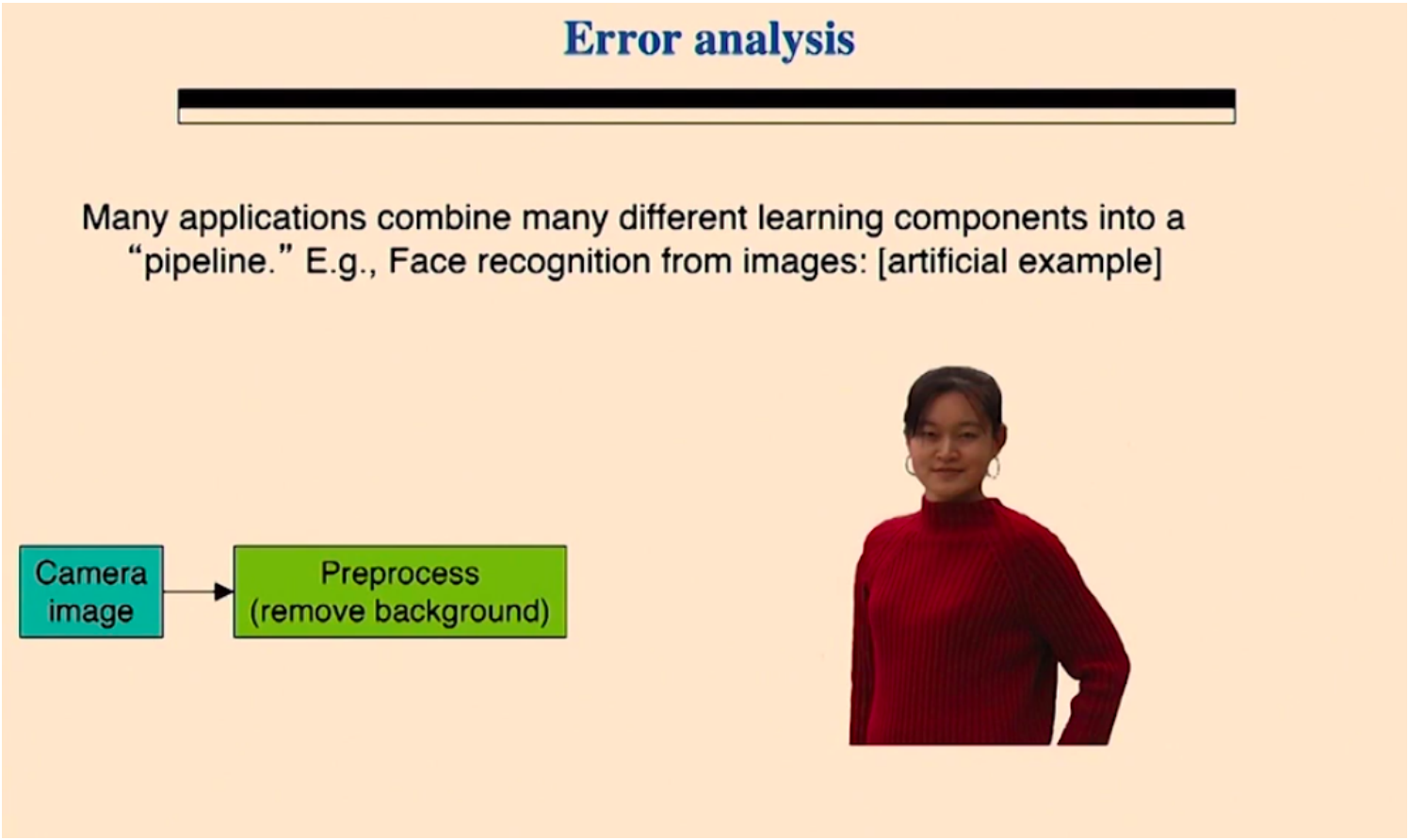
In addition to these specific diagnoses of ***bias*** vs ***variance*** and ***optimization algorithms*** vs ***optimization objective***

In addition to these type of diagnostics (how to debug learning algorithms), there's one other set of tools you'll find very useful, which is ***error analysis tools*** which is another way for you to figure out what's working, what's not working, or really what's not working in the learning algorithm

Let's say you're building a security system, so when someone walks in front of a door you unlock the door knob based on whether or not that person is authorized to enter that place. There are a lot of machine learning applications where it's not just one learning algorithm, but instead you have a ***pipeline*** (you string together many different steps)

So how do you actually build a ***face recognition algorithm*** to decide if someone approaching your front door is authorized to unlock the door?

Something you could do is you start with a camera image and then you could do ***preprocessing*** to remove the background. It turns out that when you have a camera against a static background, you could actually do this, with a little bit of noise, relatively easily, because if you have a fixed camera that's just mounted (such as on your door frame), it always sees the same background



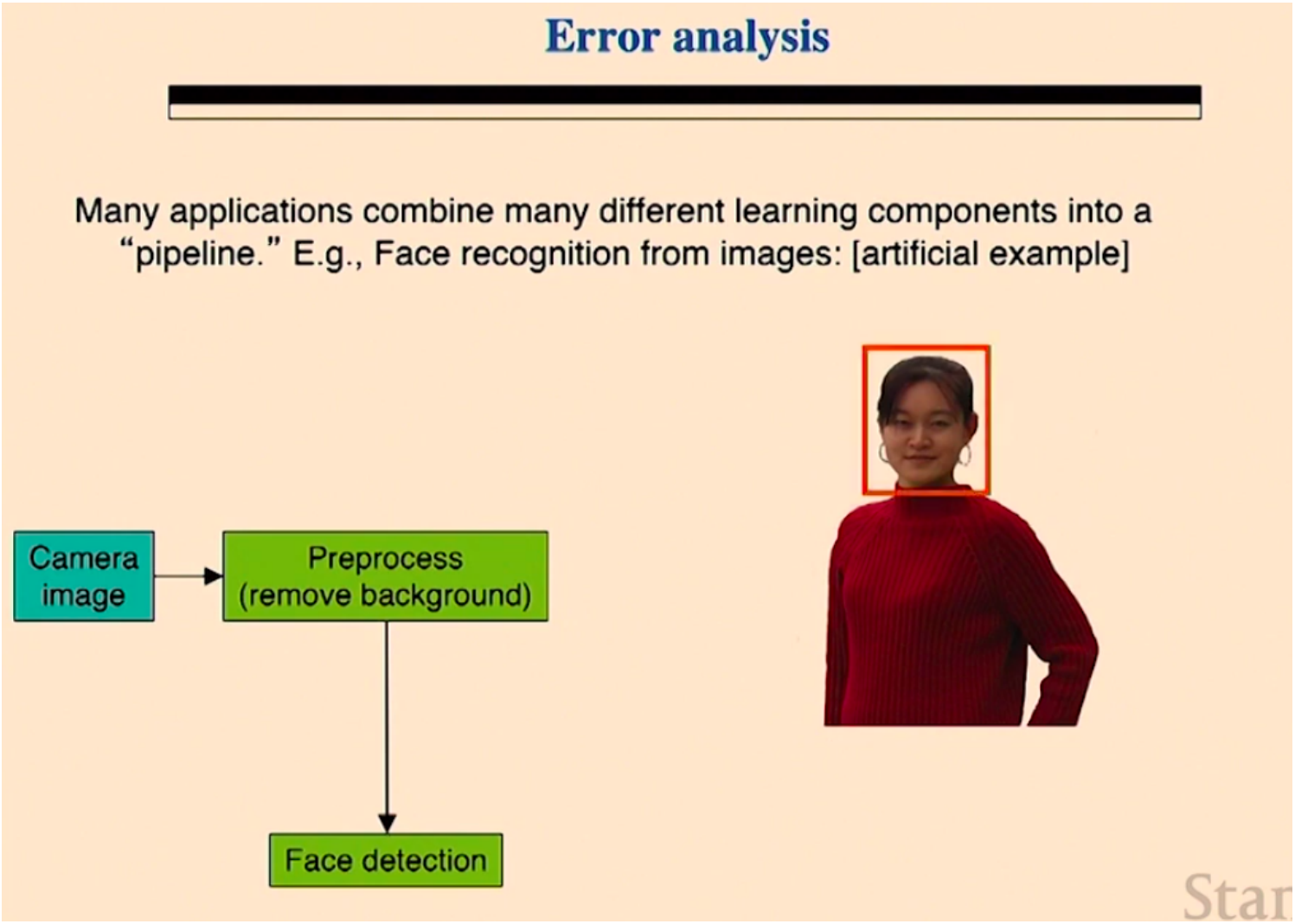
So you can just look at what pixels have changed and just keep the pixels that have changed

Error analysis

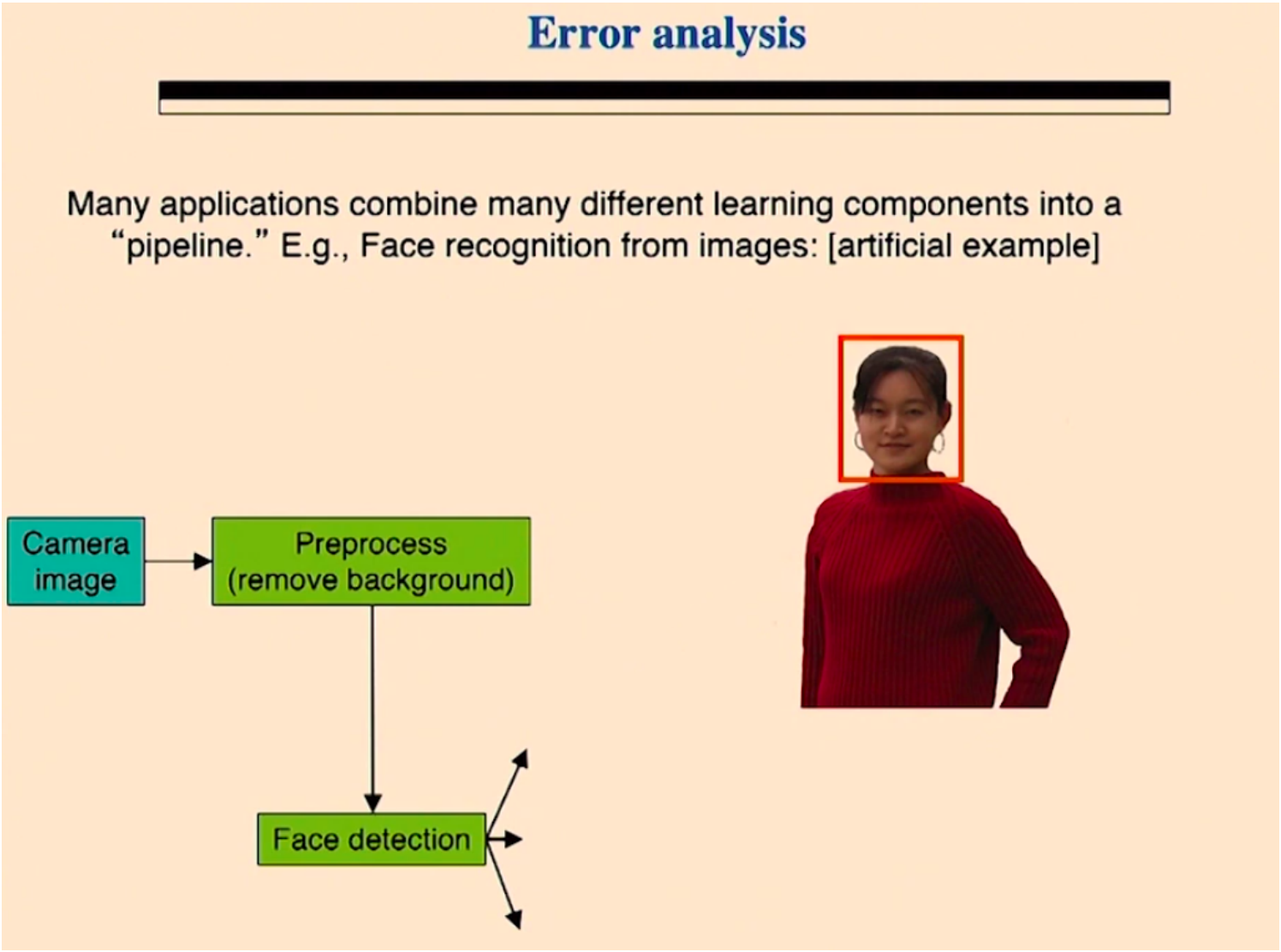
Many applications combine many different learning components into a "pipeline." E.g., Face recognition from images: [artificial example]

This camera always see that grey background and that brown bench in the back, and so you just look at what pixels have changed a lot and it does this background removal. This is actually feasible by just looking at what pixels have changed and keeping the pixels that have changed relative to that

After getting to the background, you could run the ***face detection algorithm***:



After detecting the face, it turns out that for some of the leading face recognition systems, depending on the details, but for some of them it turns out that the appearance of the eyes is a very important cue for recognizing people

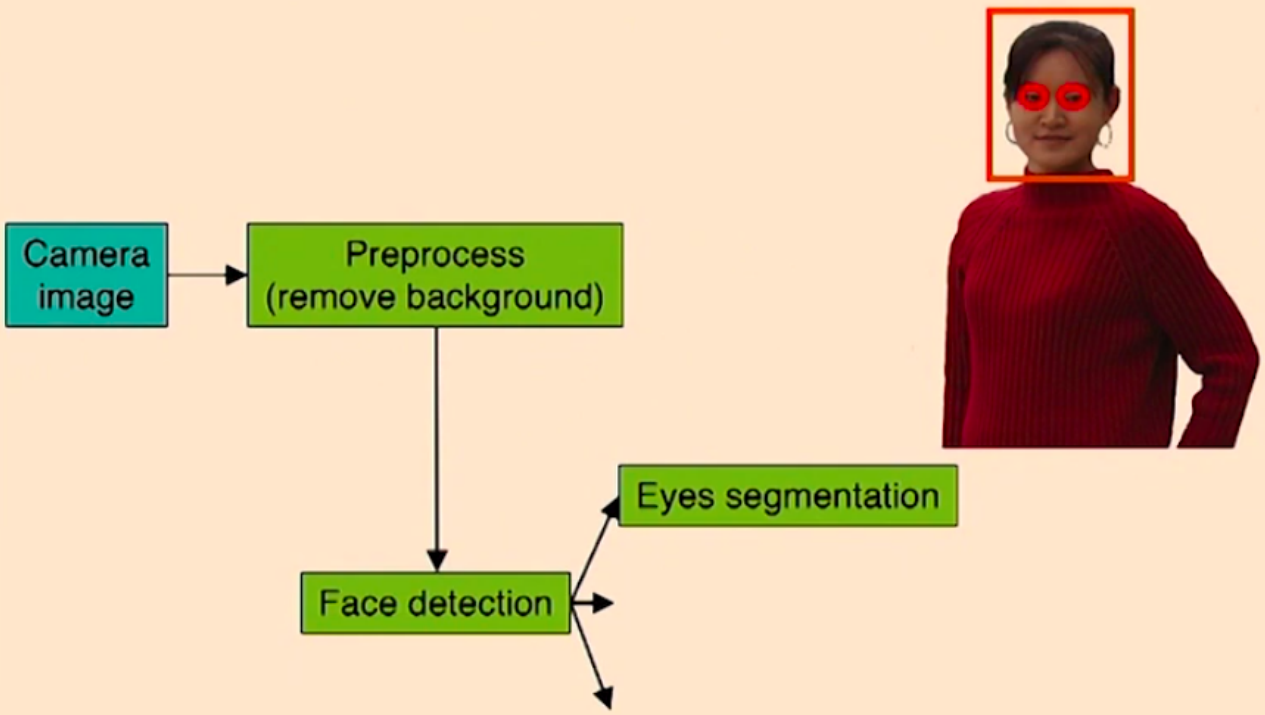


That's why if you cover your eyes, you actually have a much harder time recognizing people as eyes are very distinctive of people

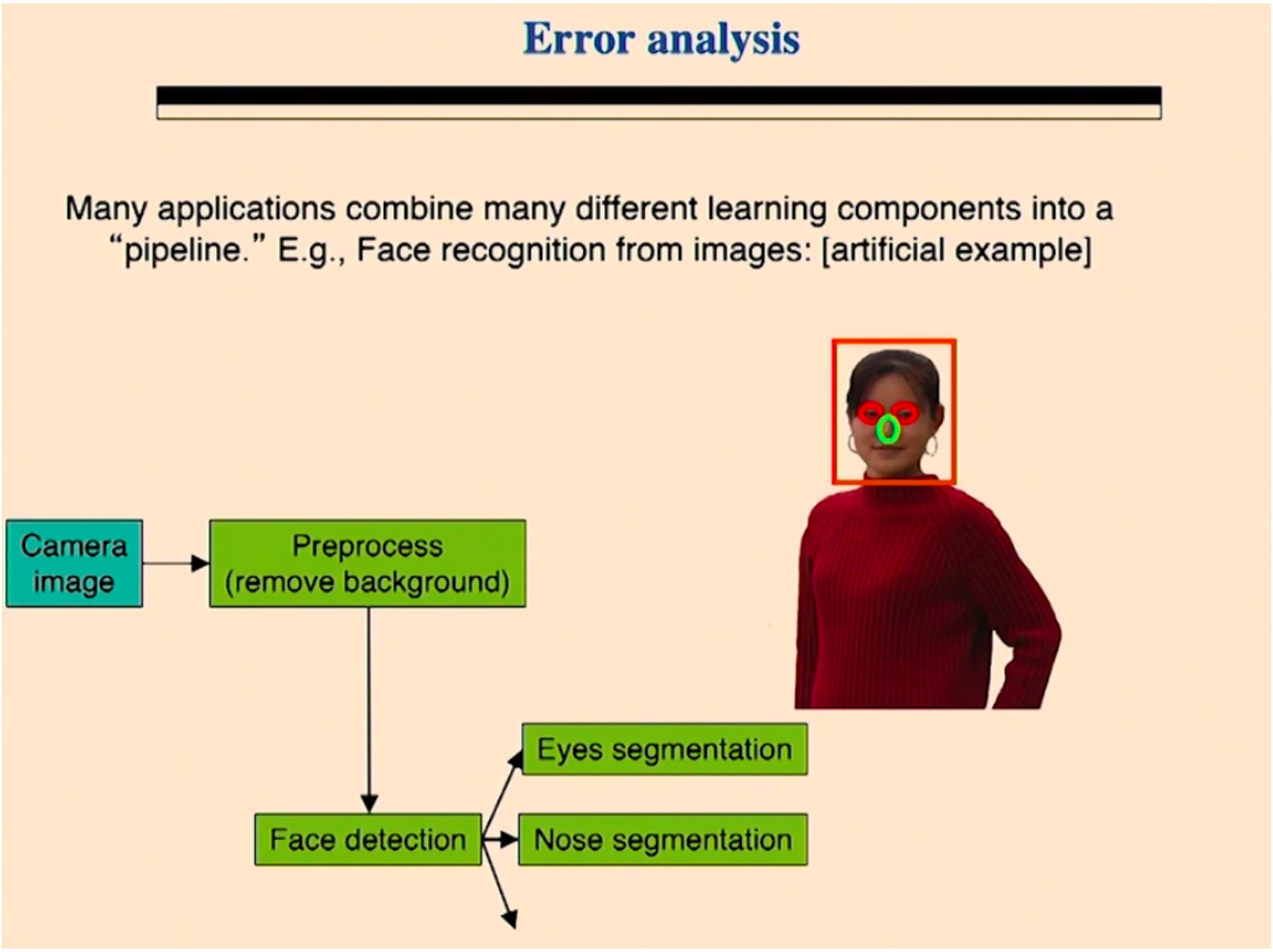
Just segment out the eyes:

Error analysis

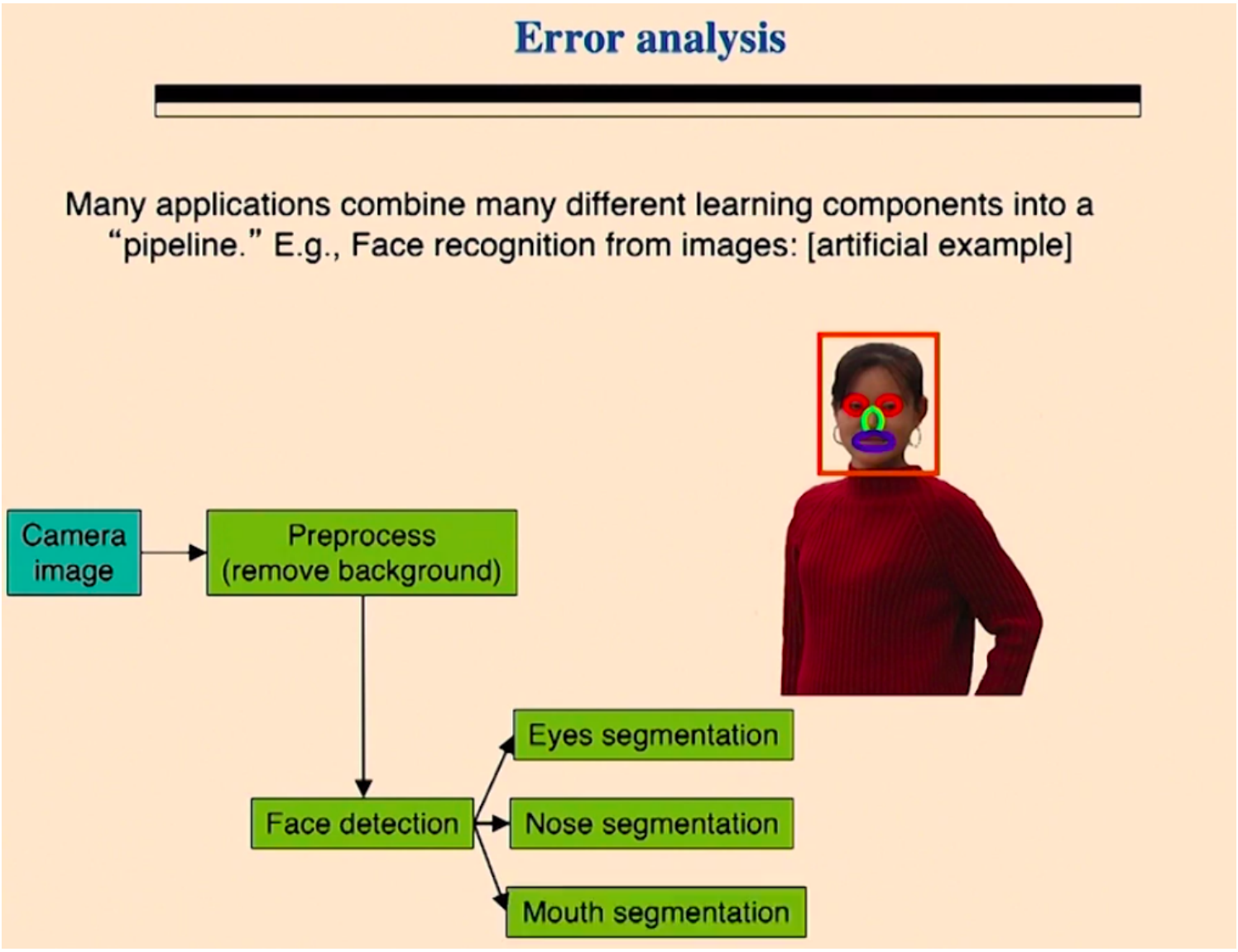
Many applications combine many different learning components into a "pipeline." E.g., Face recognition from images: [artificial example]



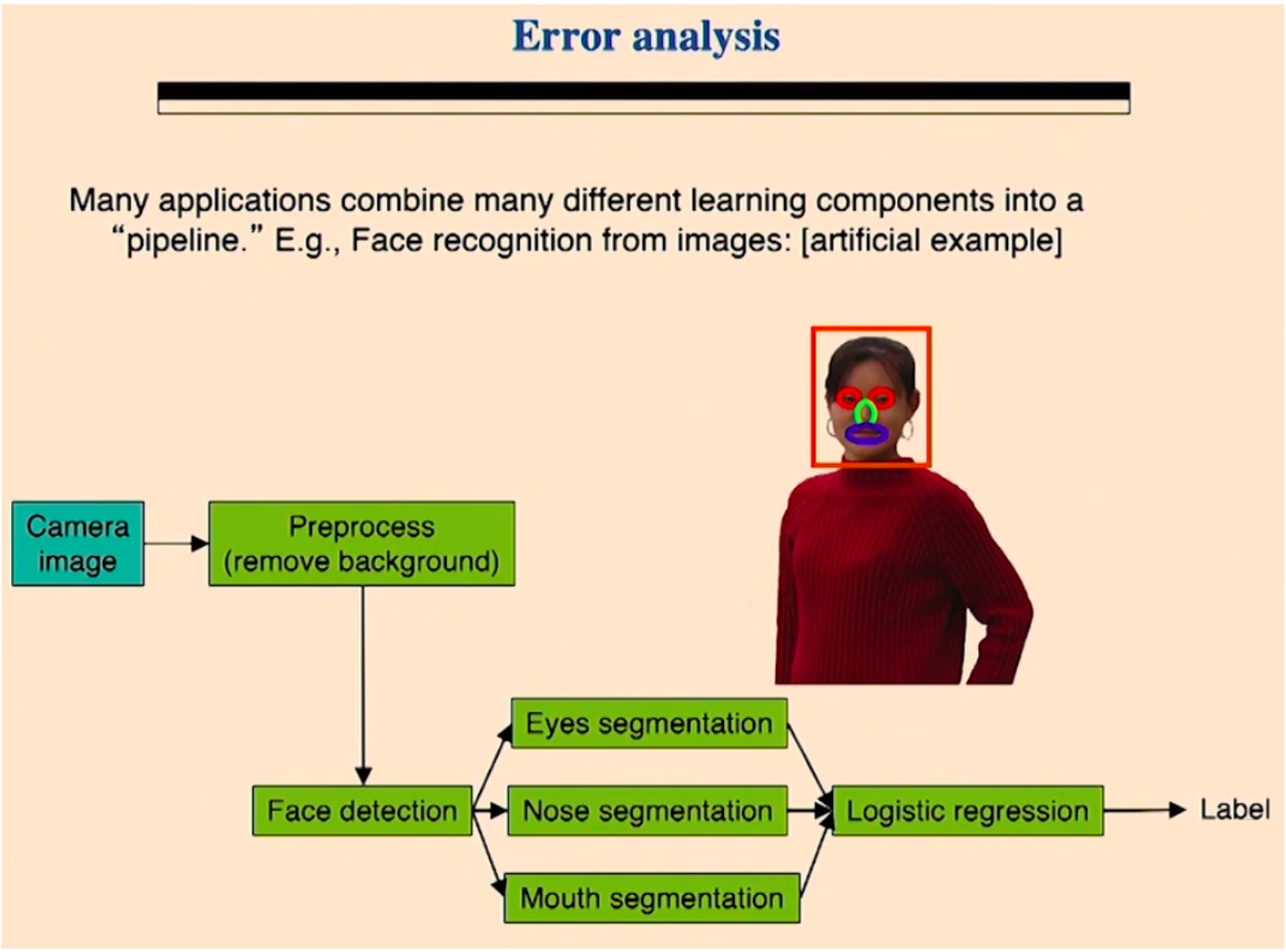
Segment out the nose:



And the other thing you segment out is the mouth:

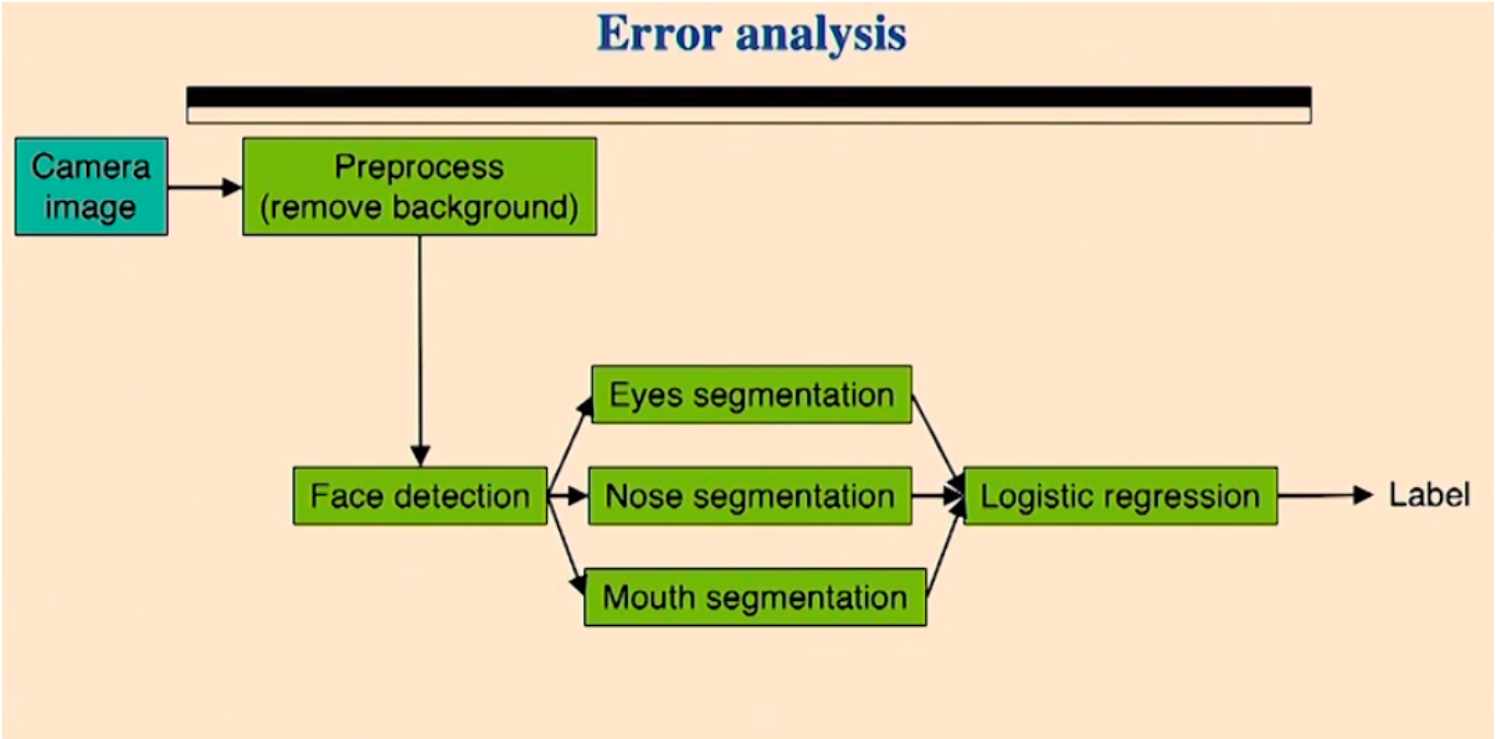


And then you feed these features into some other algorithms, say ***logistic regression*** for example:



That then finally outputs a label that says ***"is this the person that you're authorized to open the door for?"***

In many learning algorithms, you have a complicated ***pipeline*** like this of different components that have to be strung together

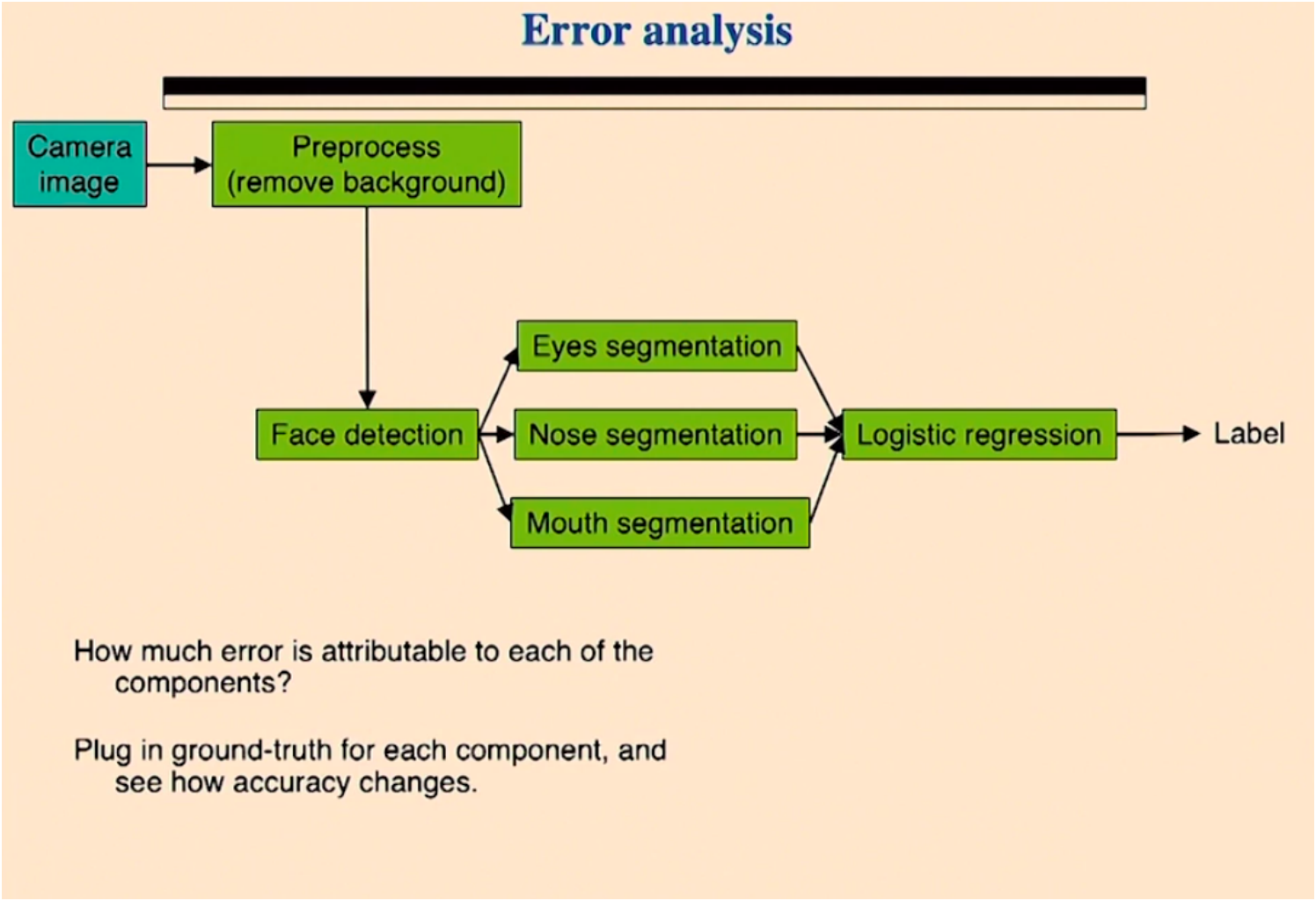


It turns out that for a lot of practical applications, if you don't have a gazillion examples, you end up designing much more complex ***machine learning pipelines*** like this where it's not just one monolithic learning algorithm but instead there are many different smaller components

Having a lot of data's great but big data has also been a little bit over-hyped and there are a lot of things you could do with small data sets as well, and you may find that if you have a relatively small dataset, often you can still get great results. You can often get great results with **100** images (**100** training examples or something)

But when you have small data, it often takes more insightful design of ***machine learning pipelines*** like this

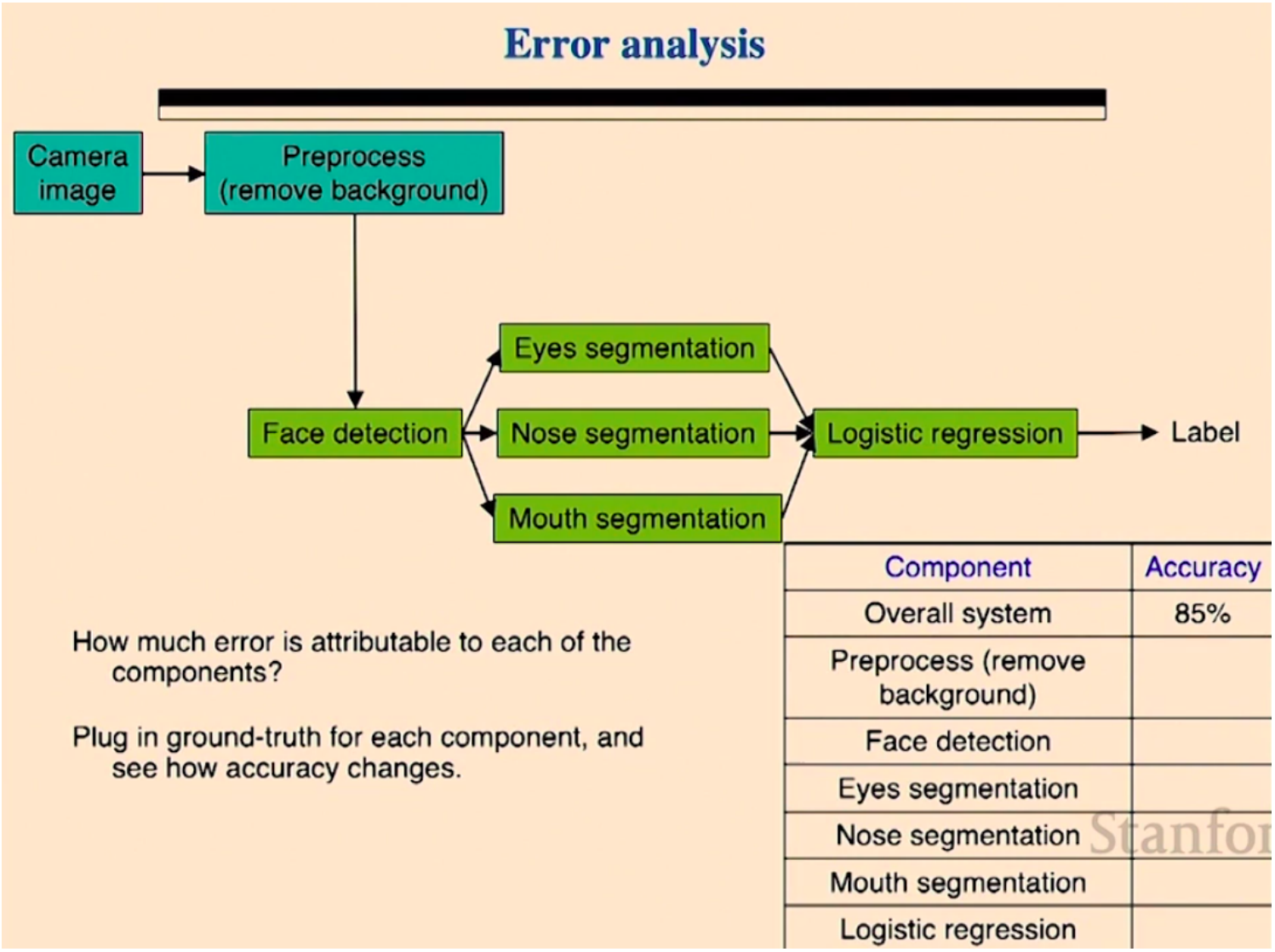
You build a pipeline like this and it doesn't work, this a common workflow. You build something, it doesn't work, so you want to debug it. In order to decide which part of the pipeline to work on, it's very useful if you can look at the error of your system and try to attribute the error to the different components so that you can decide which component to work on next



With the preprocess background removal step, since you're getting rid of the background, it turns out that there are a lot of details of how to do background removal. For example, the simple way to do it is to look at every pixel and just see which pixels have changed, but it turns out that if there's a tree in the background that waves a little bit because the wind moves the tree and blows the leaves and branches around a little bit, then sometimes the background pixels do change a little bit. So they're actually really complicated ***background removal algorithms*** that try to model, basically the trees and the bushes moving around a little bit in the background. So you know that, even though the pixels of the tree that moves around is part of the background, you just get rid of it

So background removal, there's simple versions where you just look at each pixel and see how much it's changed and there's incredibly complicated versions

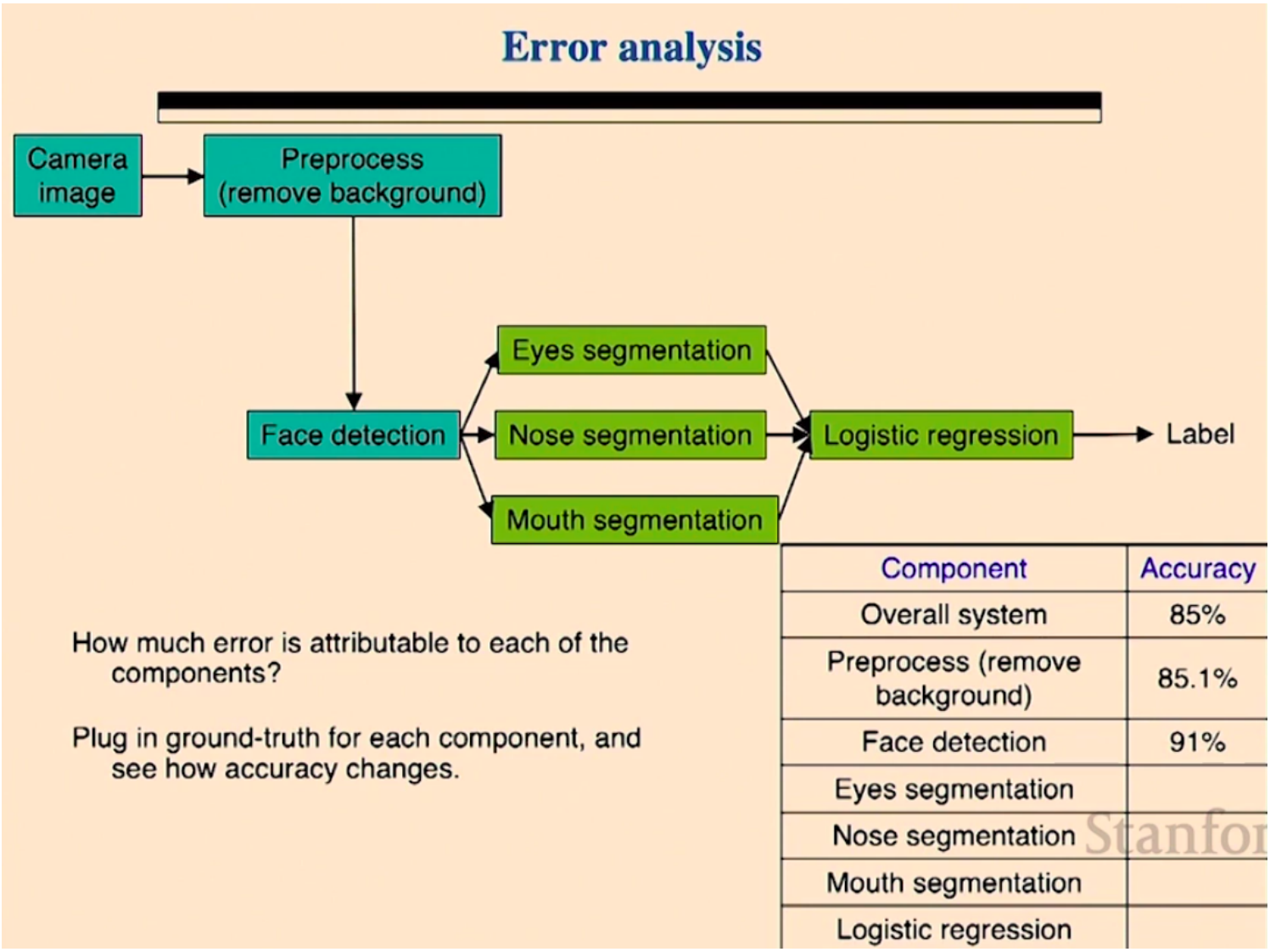
Here's what you can do with ***error analysis***, which is, say your overall system has **85%** accuracy:



Here's what you could do. You could go in and in your ***development set*** (the ***hold out cross-validation set***), go in and for every one of your examples in the ***dev set***, you would plug in the ground truth for the background, meaning that rather than using some ***approximate heuristic algorithm*** for roughly cleaning out the background, which may or may not work out well, you could just use ***Photoshop***, and for every example in the ***dev set***, you would give it the perfect background removal

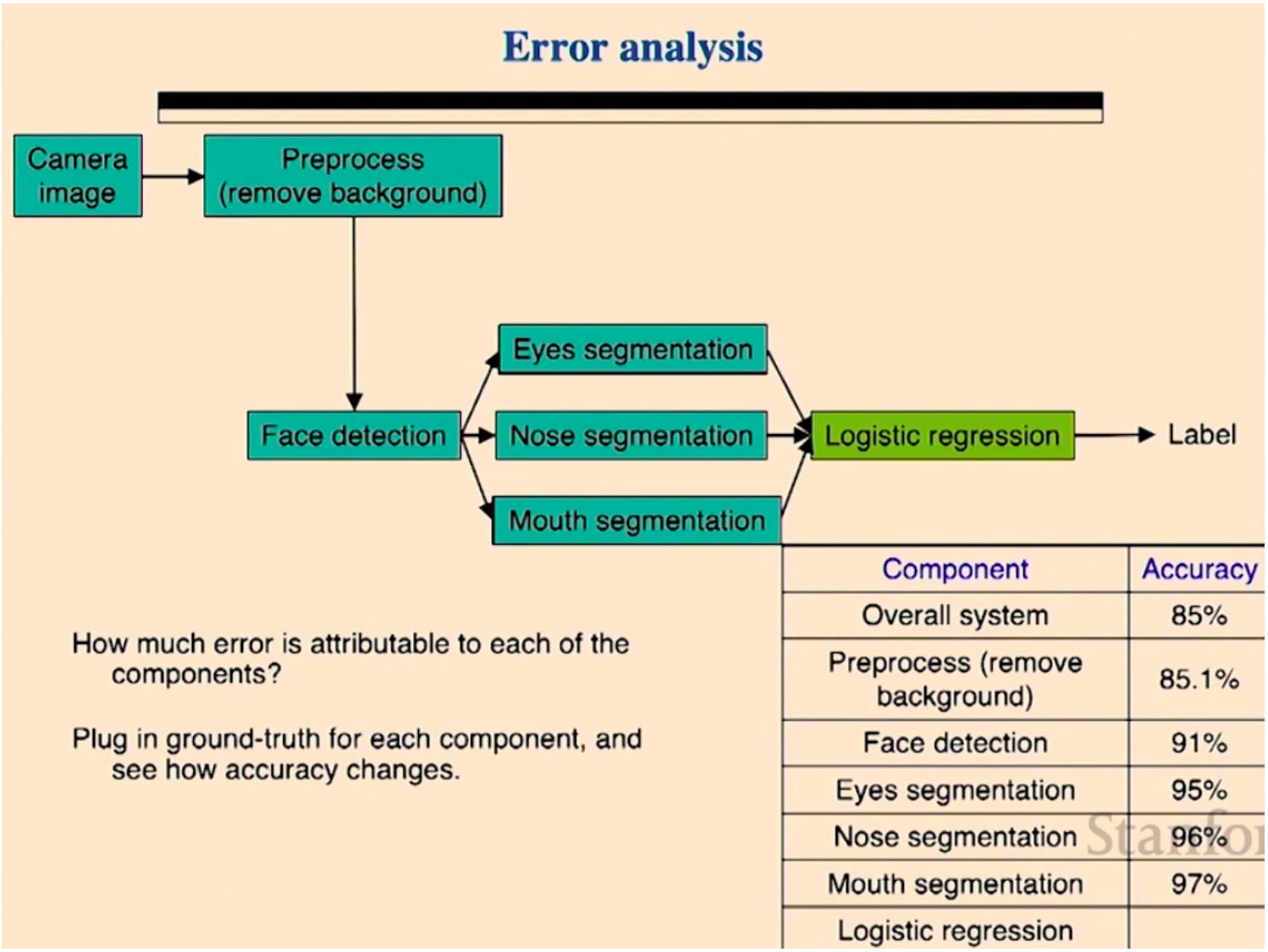
Imagine if instead of some noisy algorithm trying to remove the background, this step of the algorithm just had perfect performance, and then you can give it perfect performance on your ***dev set*** or your ***test set*** just by using ***Photoshop*** to just tell it ***"this is a background, this is a foreground"***

Let's say that when you plug in this perfect background removal, the accuracy improves to **85.1%**, and then you can keep on going from left to right in this ***pipeline*** which is now, instead of using some learning algorithm to do ***face detection***, let's just go in and for the ***test set***, modify (kind of have the ***face detection algorithm*** cheat) it. Have it just memorize the right location for the face in the ***test set*** and just give it a perfect result in the ***test set***

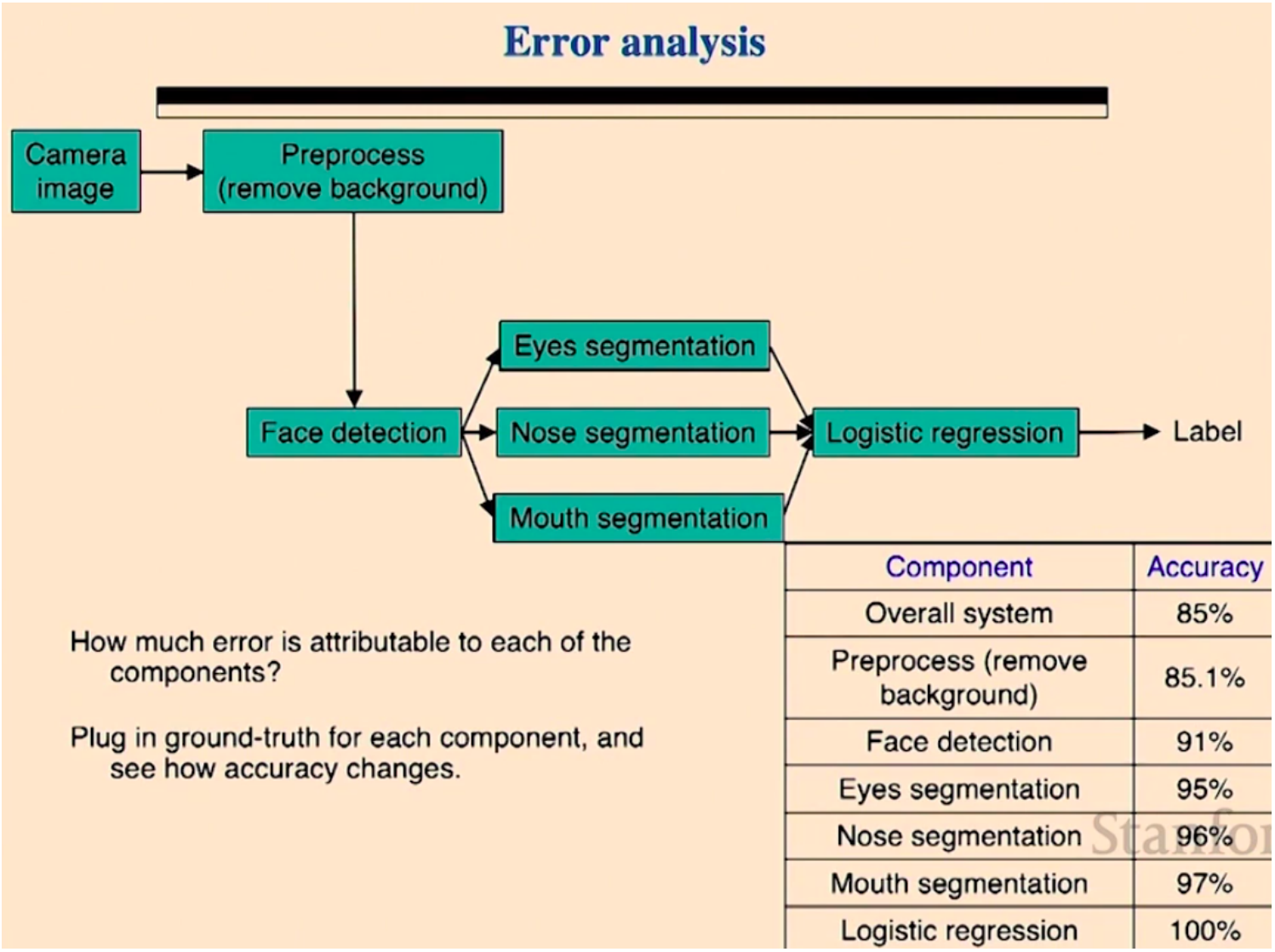


When we shaded these things, that means we're giving it the perfect result

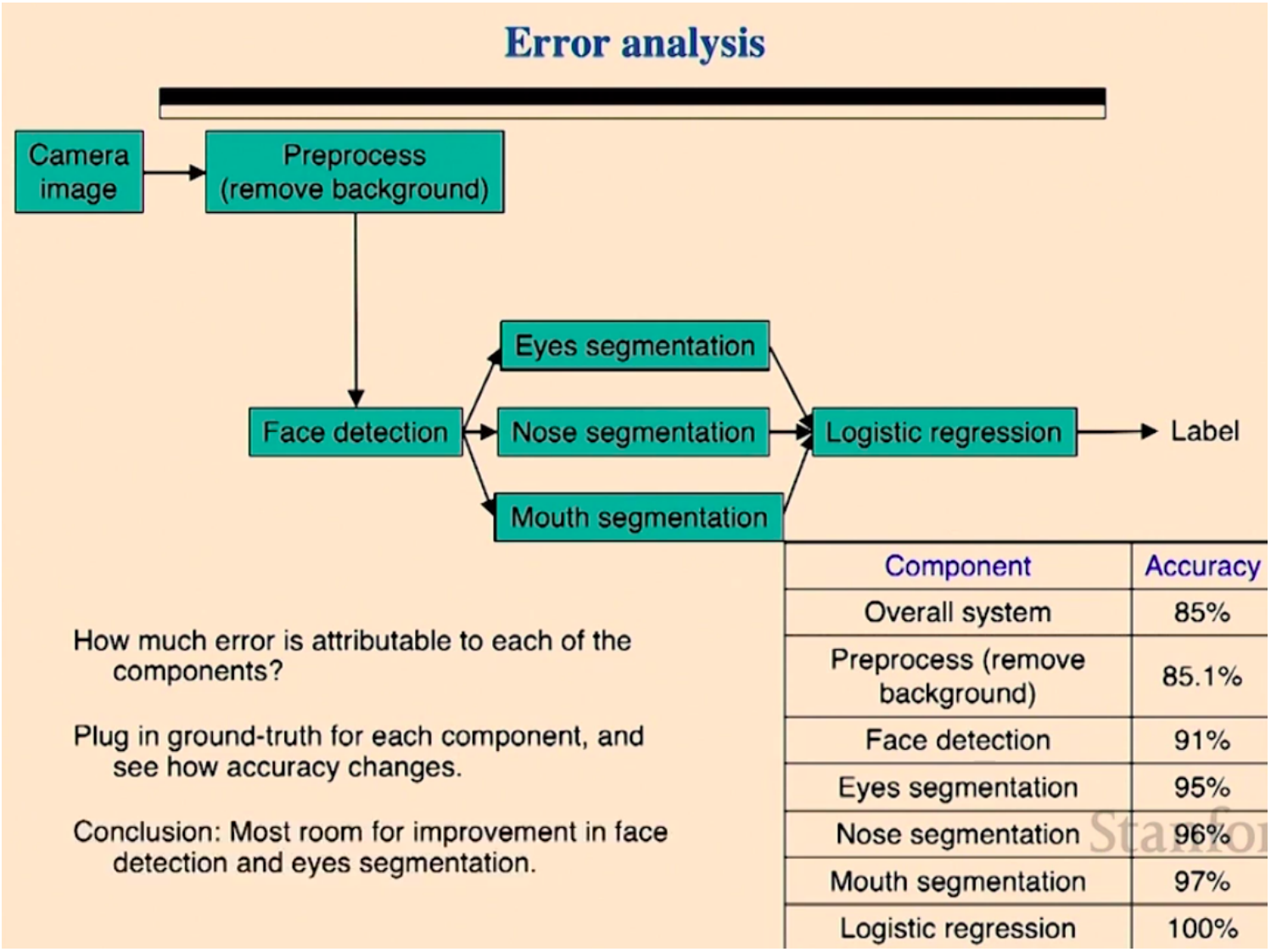
So let's just go in and on the ***test set***, give it the ***perfect face detection*** for every single example and then look at the final output and see how that changes the accuracy of the final output. And then, the same for these components (eyes segmentation, nose segmentation, mouth segmentation), and you do these one at a time



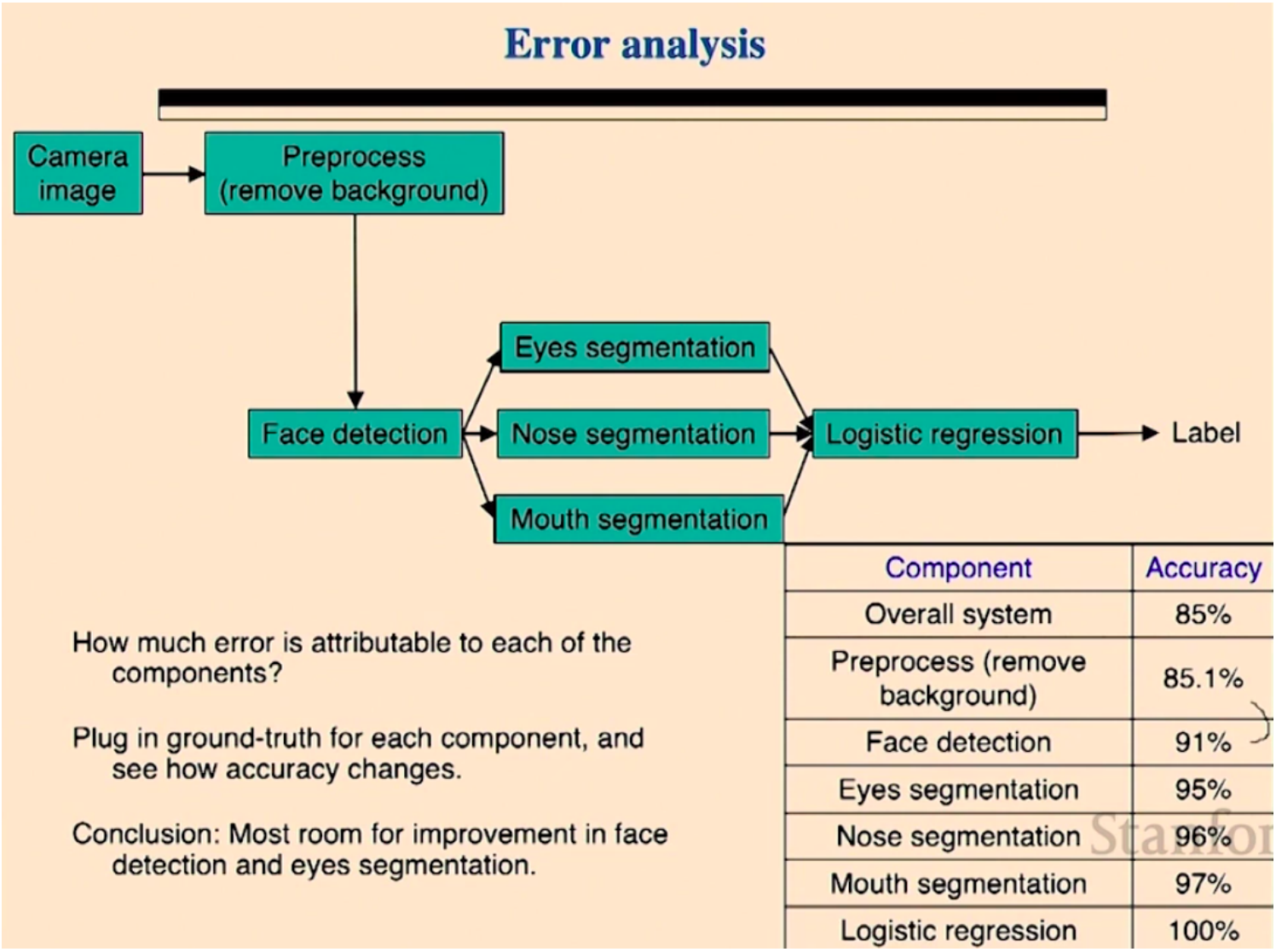
And then finally for ***logistic regression***, if you give it the perfect output, your accuracy should be **100%**



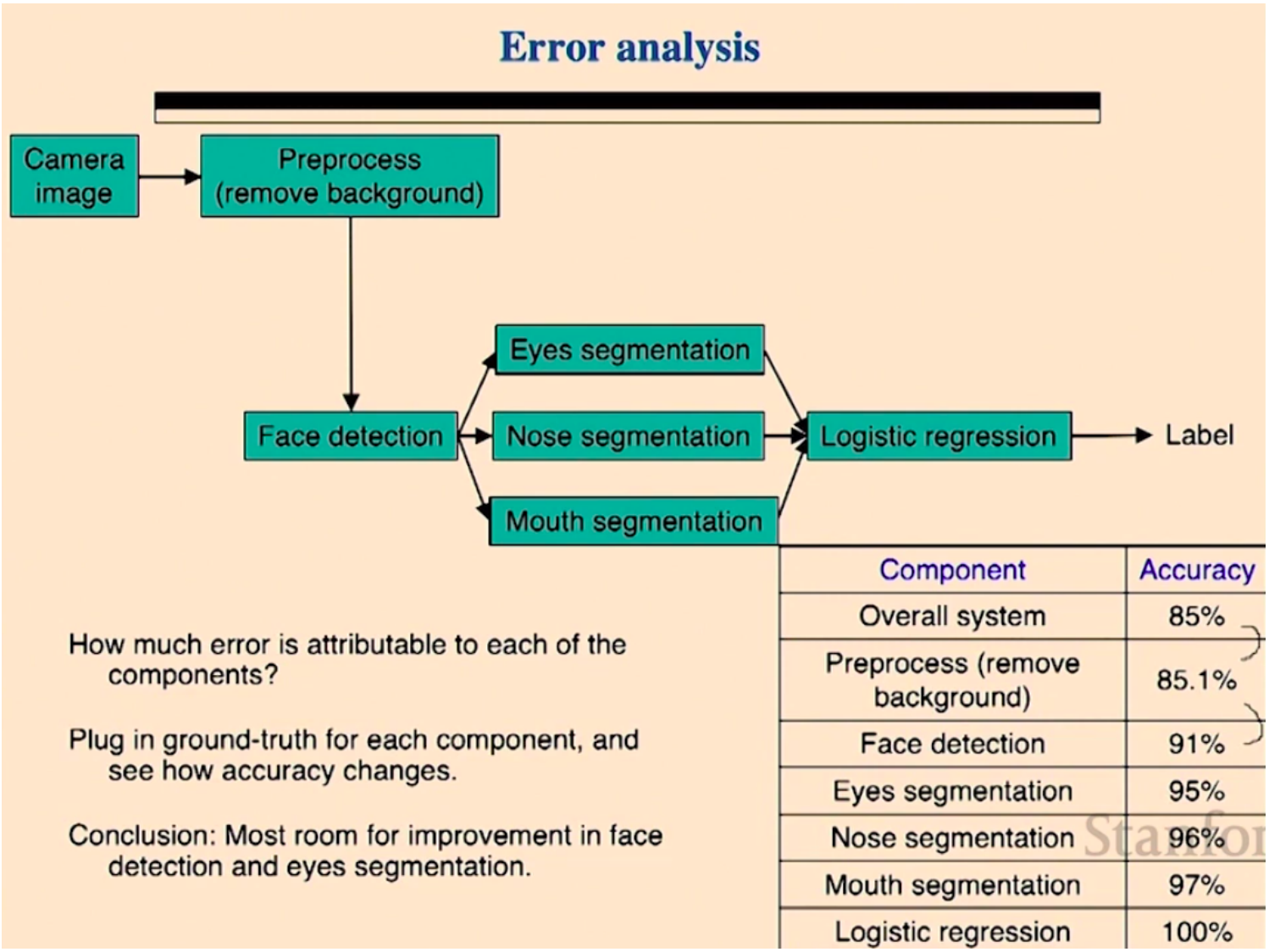
So now, what you can do is look at the sequence of steps and see which one gave you the biggest gain



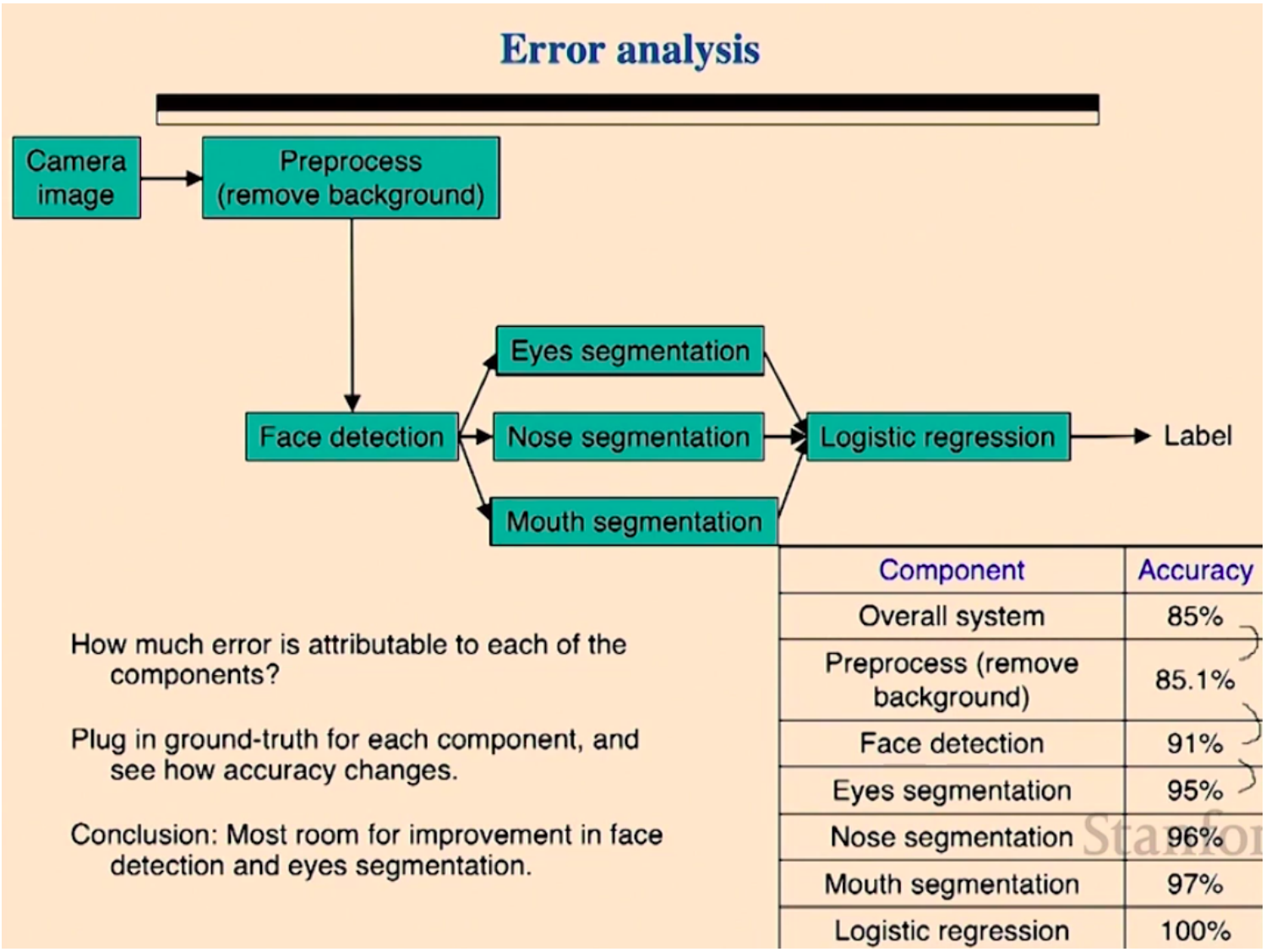
It looks like, in this example, when you gave it perfect face detection, the accuracy improved from **85.1%** to **91%** (so roughly a **6%** improvement) and that tells you that if only you can improve your face detection algorithm, maybe your overall system could get better by as much as **6%**. So this gives you faith that maybe it's worth improving on your face detection component



In contrast, this tells you that even if you had perfect background removal, it's only **0.1%** better so maybe don't spend too much time on that



And it looks like that when you gave it perfect eye segmentation, it went up another **4%**. So maybe that's another good project to prioritize



If you're in a team, one common structure would be to do this type of analysis and then have some people work on face detection, some people work on eyes segmentation. You could usually do a few things in parallel if you have a large engineering team, but at least this should give you a sense of the relative privitization of the different things

For the eyes, nose, mouth, you can do it cumulatively or one at a time and you'll probably get relatively similar results. No guarantee, you might get different results in terms of conclusions. But to the extent that you are wondering if doing it cumulatively vs non-cumulatively might give you different results, you could just do it both ways

Error analysis is not a hard mathematical rule. It is not that you do this and then there's a formula that tell you to work on face detection, for example

This should be married with judgements on how hard do you think it is to improve face detection vs eye segmentation. This at least gives you a sense of prioritization and it's worth doing this in multiple ways if you're concerned in the discrepancy in the culmulative and non-cumulative versions

When we have a complex machine learning pipeline, this type of ***error analysis*** helps you break down the error (so attribute the error to different components) which lets you focus your attention on what to work on

#!# If you do face detection accurately and then your error drops, what does that entail?

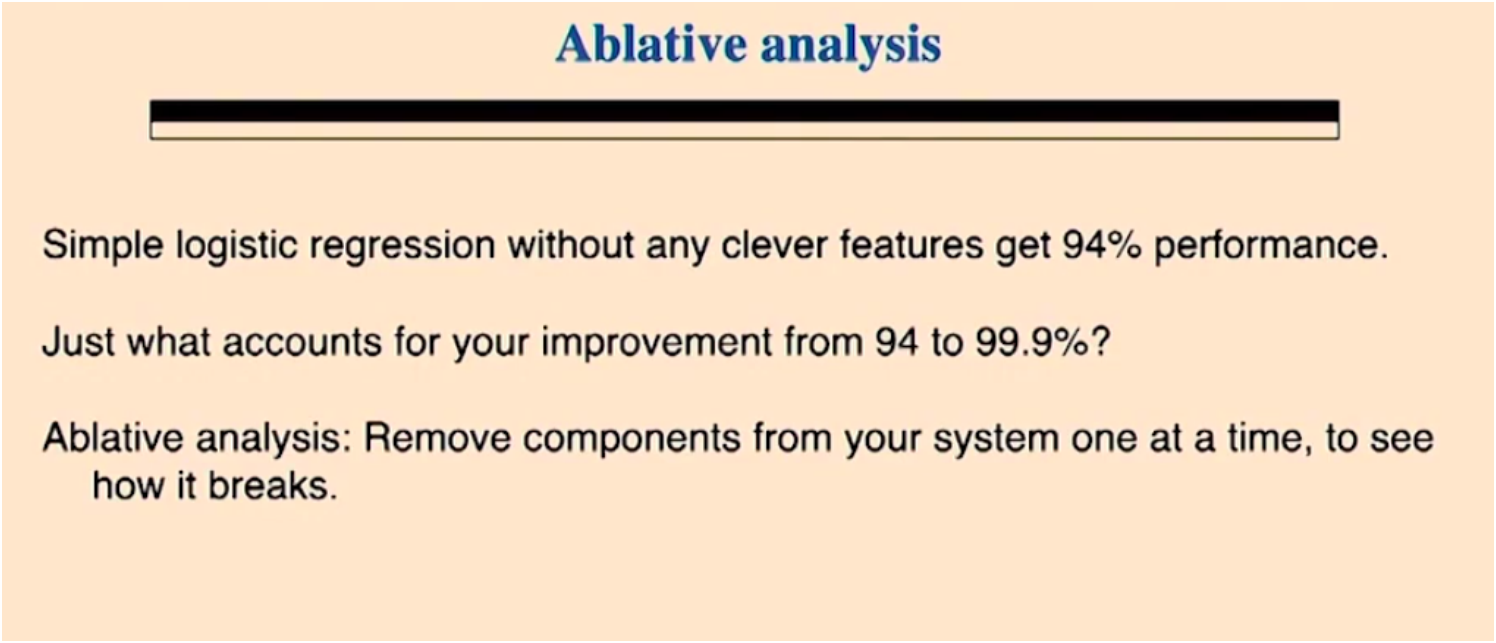
It's not impossible for that to happen, it would be quite rare. At a high level, what you could do is go in and try to figure out what's actually going on, you shouldn't ignore that. It's quite rare but not impossible

Error analysis helps figure out the difference between where you are now (**85%** overall system accuracy and **100%**). So it tries to

explain the difference between where you are and perfect performance

There's a different type of analysis called ***ablative analysis*** which figures out the difference between where you are and something much worse

Let's say that you built a good anti-spam classifier by adding lots of clever features in ***logistic regression***. So spelling correction because spam is trying to misspell words to mess up the tokenizer, to make spammy words not look like spammy words, sender host features (what machine did the email come from), header features, you could have a parser from ***NLP*** (***Natural Language Processing***), parse a text, use a Javascript parser to understand, or even fetch the webpages that the email refers to and parse that. And the question is ***"how much would these components really help?"***



Simple logistic regression without all these clever features got **94%** performance and with your addition of all these clever features, you got **99%** accuracy

Ablative analysis

Simple logistic regression without any clever features get 94% performance.

Just what accounts for your improvement from 94 to 99.9%?

Ablative analysis: Remove components from your system one at a time, to see how it breaks.

Component	Accuracy
Overall system	99.9%
Spelling correction	
Sender host features	
Email header features	
Email text parser features	
Javascript parser	
Features from images	

An ***ablative analysis***, what you would do is remove the components one at a time to see how it breaks. Just now we were adding to the system by making components perfect with error analysis, this is how it improves. Here, we're going to remove things one at a time to see how it breaks

Ablative analysis

Simple logistic regression without any clever features get 94% performance.

Just what accounts for your improvement from 94 to 99.9%?

Ablative analysis: Remove components from your system one at a time, to see how it breaks.

Component	Accuracy
Overall system	99.9%
Spelling correction	99.0
Sender host features	98.9%
Email header features	98.9%
Email text parser features	95%
Javascript parser	94.5%
Features from images	94.0%

When you remove all of these features, you end up there. You could do this cumulatively or remove one and put it back, remove one and put it back, or you could do it both ways and see if they give you slightly different insights

Ablative analysis

Simple logistic regression without any clever features get 94% performance.

Just what accounts for your improvement from 94 to 99.9%?

Ablative analysis: Remove components from your system one at a time, to see how it breaks.

Component	Accuracy	
Overall system	99.9%	
Spelling correction	99.0	
Sender host features	98.9%	
Email header features	98.9%	
Email text parser features	95%	
Javascript parser	94.5%	
Features from images	94.0%	[baseline]

Conclusion: The email text parser features account for most of the improvement.

The conclusion from this particular analysis is that the biggest gap is from the text parser features because when you remove that, the error (the accuracy) went down by **4%**

Ablative analysis

Simple logistic regression without any clever features get 94% performance.

Just what accounts for your improvement from 94 to 99.9%?

Ablative analysis: Remove components from your system one at a time, to see how it breaks.

Component	Accuracy	
Overall system	99.9%	
Spelling correction	99.0	
Sender host features	98.9%	
Email header features	98.9%	
Email text parser features	95%	
Javascript parser	94.5%	
Features from images	94.0%	[baseline]

Conclusion: The email text parser features account for most of the improvement.

So you know this is strong evidence. If you want to publish a paper, you can say ***"text parser features significantly improves spam filter accuracy in that level of insight"***

This type of ***error analysis*** gives you intuition about what's important and what's not, and helps you decide to maybe even double down on text parser features or maybe if the sender host features is too computationally expensive to compute, tells you maybe you can just get rid of that and without too much harm

Also, if you're publishing a paper or sending a report, this gives much more insight to your report