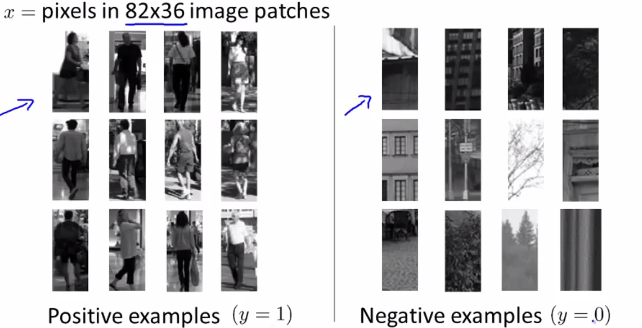
***Photo OCR***

**I. Problem Description and Pipeline**

* ML application case study centered around an application called **Photo OCR**
* 3 reasons for this case study
* To show an example of how a complex ML system can be put together.
* Concepts of a ML pipeline + how to allocate resources when trying to decide what to do next.
* Either in the context of working by yourself on a big application or in team of developers trying to build a complex application together.
* An excuse to talk about a couple more interesting ideas for ML.
* How to apply ML to computer vision problems
* The idea of **artificial data synthesis**
* Photo OCR = **Photo Optical Character Recognition.**
* W/ the growth of digital photography + more recently the growth of cameras in cell phones, we now have tons of visual pictures we take all over the place
* 1 of the things that has interested many developers is how to get CPUs to understand content of these pictures a little bit better.
* The photo OCR problem focuses on how to get CPUs to read the text that appears in images we take
* Given an image, a CPU would read the text in it so that if you're trying to look for this picture again, you’d type in the words seen in it + have the CPU automatically pull up the picture so that you're not spending lots of time digging through your photo collection
* Photo OCR does exactly this in several steps
* 1) Given the picture, it has to look through the image + detect where there is text in the picture.
* 2) After it has done that successfully, it looks at these text regions + actually reads the text in those regions (hopefully correctly) + comes up w/ transcriptions of the text in the image.
* Whereas **OCR** (**optical character recognition**) of scanned documents is relatively easier, doing OCR from photographs is still a very difficult ML problem
* Not only can Photo OCR help CPUs to understand the content of images better, there are also applications for it
* Ex: Provide a camera to a blind person that can look at what's in front of them + tell them the words that may be on the a street sign in front of them.
* W/ car navigation systems 🡺 car could read street signs + help navigate to a destination
* In order to perform photo OCR, here's what we can do
* 1) **Text Detection =** Go through the image + find regions where there's text
* 2) **Character Segmentation =** Given a rectangle around a text region, do character segmentation to take a text box + try to segment it out into the locations of the individual characters.
* **3) Character Classification** = Having segmented out individual characters, we can then run a classifier = looks at the images of the visual characters + tries to figure the characters + then hopefully figure out an entire phrase
* There’re some photo OCR systems that do more complex things, like spelling correction at the end
* Ex: Character segmentation + character classification system tells you it sees the word “c 1 e a n i n g” 🡪a spelling correction system might tell you this is probably the word 'cleaning', + your character classification algorithm had just mistaken the l for a 1.
* A system like what is described above is what we call a **ML pipeline.**
* In many complex ML systems, pipelines are common where you can have multiple modules w/in it, each of which may be ML component, or sometimes it may not be a ML component but a set of modules that act one after another on some piece of data in order to produce the output you want,
* If designing a ML system, 1 of the most important decisions will often be *what exactly is the pipeline that you want to put together?*
* In other words, given the photo OCR problem, how do you break this problem down into a sequence of different modules?
* The design of a pipeline + the performance of each module in a pipeline will often have a big impact on the final performance of your algorithm.
* If you have a team of engineers working on a problem, it’s very common to have different individuals work on different modules.
* 1-5 engineers doing character segmentation, another 1-5 doing character recognition, etc.
* Having a pipeline often offers a natural way to divide up a workload amongst different members of an engineering team
* All this work could also be done by just 1 person if that's how you want to do it.
* In complex ML systems the idea of a ML pipeline is pervasive.

**II. Sliding Windows**

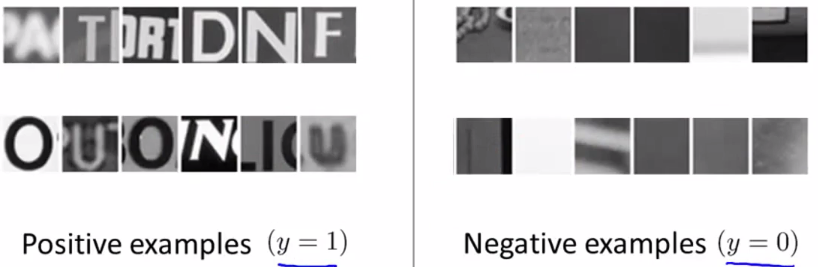
* The 1st stage of the photo OCR pipeline was Text detection, an unusual problem in CPU vision.
* Depending on the length of the text you're trying to find, rectangles that surround that text can have different aspects.
* Simpler example = pedestrian detection
* Want to take an image + find the individual pedestrians that appear in the image
* Slightly simpler than text detection b/c aspect ratios of most pedestrians are pretty similar
* Fixed ratio 🡺 ratio between height + width of rectangles are all the same for different pedestrians
* For text detection this ratio is different for different lines of text
* Pedestrians can be different distances away from the camera, + so the height of rectangles can be different, but the aspect ratio is the same.
* In order to build a pedestrian detection system: say we decide to standardize on an aspect ratio of 82x36
* Then go out + collect large training sets of positive + negative examples



* May have anywhere from a 1k up to maybe 10k training examples, or even more if you can get even larger training sets
* Then train an NN or some other learning algorithm to take this input (an image patch of dimension 82x36) + to classify that image patch Y as either containing a pedestrian or not.
* This gives you a way of applying supervised learning in order to take an image patch can determine whether or not a pedestrian appears in that image capture.
* Say we get a new/test set image 🡪 want to find pedestrian in the image
* Start by taking an 82x36 rectangular patch of the image + run that patch through our classifier to determine whether or not there is a pedestrian in that image patch



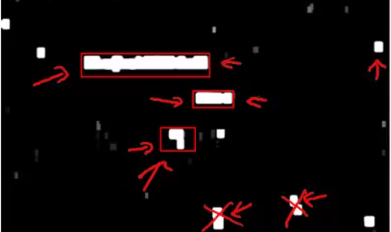
* Hopefully our classifier will return y = 0 for that patch, since there is no pedestrian.
* Next, take the green rectangle, slide it over a bit, + run the new image patch through our classifier to decide if there's a pedestrian there, and so on.
* The amount by which you shift the rectangle over each time is a parameter = the **step size** or **slide parameter**
* If you step 1 pixel at a time (step size or stride = 1), that usually performs best but is more computationally-expensive
* A step size of maybe 4 or 8 pixels at a time might be more common,
* Eventually, as you slide this window over the different locations in the image, you’d have run all of these different image patches through the classifier.
* This was a pretty small rectangle + would only detect pedestrians of 1 specific size.
* Next we start to look at *larger* image patches + run them through the classifier as well by taking the larger + *resizing it down to 82x36* to pass through the classifier
* After this whole process hopefully the algorithm will detect whether there’s pedestrian in the image
* That’s how you use a **sliding windows detector** in order to train to find pedestrians in an image.
* Similar to pedestrian detection you can come up w/ a labeled training set w/ positive examples + negative examples w/ examples corresponding to regions where text appears.

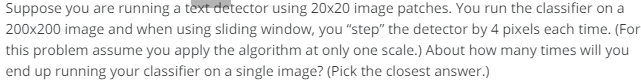


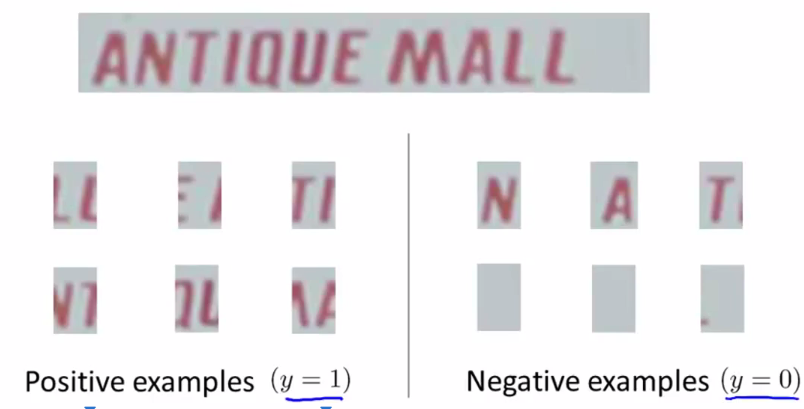
* So instead of trying to detect pedestrians, we're now trying to detect texts
* Using a fixed-scale window:

* White regions = where my text detection system has found text
* black = classifier does not think it's found any text up there,
* different shades of grey = probability output by the classifier of where it thinks it might have found text but has lower confidence the bright whites
* Now draw rectangles around all regions where there is text in the image via 1 more step of applying an **expansion operator** to the output of the classifier
* 
* Takes each of the white blobs in the image + expands the white region.
* Mathematically, the way you implement that is, for every pixel we ask if its w/in some distance of a white pixel
* If a specific pixel is w/in 5 or 10 pixels of a white pixel, we'll also color that pixel white in
* The effect = each of the white blobs is expanded a bit by coloring nearby pixels white as well
* Finally, look at the connecting components/continuous white regions + draw bounding boxes around them
* If we use a simple heuristic to rule out rectangles whose aspect ratios look funny (boxes around text should be much wider than they are tall so we ignore thin, tall blobs)



* This example by the actually misses 1 piece of text written against a transparent window.
* Having found these rectangles w/ text, we can now just cut out these image regions + then use later stages of the pipeline to try to read the text.
* 
* 
* The 2nd stage of pipeline was **character segmentation** 🡪 given an image of a word, segment out the individual characters
* Can use a supervised learning algorithm w/ some set of positive + some set of negative examples to look in an image patch + try to decide if there is split between 2 characters



* We want to split the positive examples where the middle of the image = a gap between 2 distinct characters + we don’t want to split the negative examples b/c they don't represent the midpoint between 2 characters.
* We train a classifier using a NN or a different learning algorithm to try to classify between positive + negative examples.
* Having trained such a classifier, we can then run it on text our text detection system has pulled out.
* Start by looking at a rectangle + ask, "Does the middle of that rectangle look like the midpoint between 2 characters?"
* Then slide the window over in a 1-dimensional sliding window classifier 🡪 slide the window only in 1 straight line from left to right (no rows here).
* When the classifier outputs y =1 one, we draw a line to try to split 2 characters
* Overall photo OCR pipeline



**Getting Lots of Data + Artificial Data**

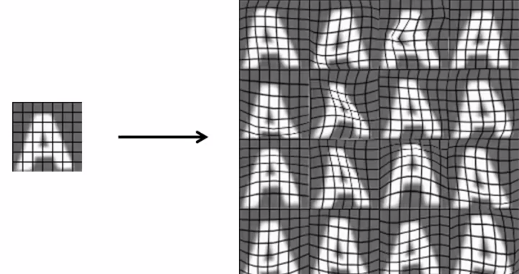
* 1 of the most reliable ways to get a high performance ML system is to take a low bias learning algorithm + train it on a massive training set.
* But where did you get so much training data from?
* In ML there's a fascinating idea called **artificial data synthesis** 🡪 doesn't apply to every problem, + to apply to a specific problem, often takes some thought + innovation + insight.
* But if this idea applies to your ML problem, it can sometimes be an easy way to get a huge training set to give to your learning algorithm.
* The idea of **artificial data synthesis** comprises of 2 variations
* 1) Essentially creating new data from scratch
* 2) We already have a small labeled training set + somehow amplify it/turn it into a larger set
* To talk about artificial data synthesis, look at the character recognition portion of the photo OCR pipeline 🡪 take an input image + recognize what character it is.

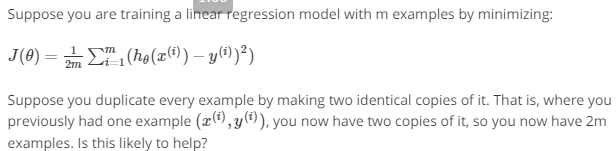


* Goal = Take image patches in this square + recognize the character in the middle of that image patch
* Using color doesn't seem to help that much for *this* particular problem.
* Modern CPUs often have a huge font library + if you want more training examples, 1 thing you can do = take characters from different fonts + paste them against different random backgrounds
* If you do 🡪 now have a training example of an image of whatever character you chose
* After some amount of work, you can get a **synthetic training set**

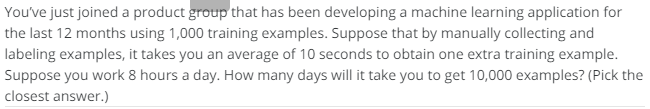


* Take a font 🡪 paste an image of one/a few character(s) from that font against a random background image 🡪 maybe apply a little blurring operators + distortions
* It takes thought in order to make the synthetic data look realistic, + if you do a sloppy job in terms of how you create synthetic data, it actually won't work well.
* By using synthetic data you have essentially an unlimited supply of training examples for artificial training synthesis/to create a supervised learning algorithm for the character recognition problem.
* This was an example of artificial data synthesis of creating new data from scratch
* The other main approach to artificial data synthesis is to take examples you currently have + create additional data so as to amplify your training set.
* Can take an image + introduce artificial warping/distortions into the image + it into 16 new examples



* Again, in order to do this, it does take thought + insight to figure out reasonable sets of distortions or other ways to amplify + multiply a training set
* For the specific example of character recognition, introducing warping seems like a natural choice, but for a different ML application, there may be different the distortions that might make more sense
* Ex: Speech recognition.
* Have audio clips + want to learn from the audio clip to recognize words spoken in that clip.
* Have 1 labeled training example of someone saying a few specific words + you want to try to apply a learning algorithm to try to recognize the words said in that.
* To amplify the data set, 1 thing is to introduce additional audio distortions into the data set.
* Can add background sounds to maybe simulate a bad cell phone connection, noisy backgrounds due to a crowd or machinery
* We amplify an original *clean* audio clip training example into multiple different training examples by adding different background sounds to the clean audio
* 1 word of warning about synthesizing data by introducing distortions: *If you try to do this yourself, the distortions you introduce should be representative the source of noises/distortions you might see in the test set.*
* Usually does not help to add purely random/meaningless noise to data.
* The process of artificial data synthesis it a little bit of an art as well + sometimes you just have to try it + see if it works.
* But if trying to decide what sorts of distortions to add, think about *meaningful* distortions that will generate additional training examples that are at least somewhat representative of the sorts of images you expect to see in your test sets.
* 



* As always, before expending a lot of effort figuring out how to create artificial training examples:
* It's often a good practice is to make sure you have **a low-bias classifier** = which means having a lot more training data will actually help
* Standard way 🡪 plot the learning curves + make sure you have a low-bias, high-variance **classifier**
* If you *don't* have a **low-bias classifier**, 1 other thing worth trying = keep increasing the number of features your classifier has (increasing # of hidden units in a NN) until you actually have a low bias classifier + only *then* put the effort into creating a large, artificial training set
* Want to avoid spending weeks/months figuring out how to get a great artificially synthesized data set only to realize afterward that your learning algorithm’s performance doesn't improve that much even given a huge training set.
* When working on ML problems, 1 question to often ask = “How much work would it be to get 10X as much date as we currently have?”
* It's really not that hard, maybe a few days of work at most
* Very often if you can get 10X as much data, there will be a way to make your algorithm do much better.
* Several ways to do so:
* **Artificial data synthesis =** generating data from scratch or taking an existing example + introducing distortions that amplify/enlarge a training set
* Collect the data/label them yourself 🡪 ask how many minutes/hours it takes to get a certain number of examples
* Might take 10 seconds to label 1 new example, + so if I want 10X as many examples from our original 1K examples 🡪 need 10K examples \* 10 seconds per example
* Sometimes a few days of work + many teams are surprised at how little work it could be sometimes to just get a lot more data + give a learning algorithm a huge boost in performance
* **Crowdsourcing** 🡪 hire people on the web to label large training sets for you
* This idea has an entire academic literature + it's own complications (pertaining to labeler reliability)
* Amazon Mechanical Turk systems is probably the most popular crowd sourcing option right now.
* Often quite a bit of work to get to work to get very high quality labels, but is sometimes an option worth considering
* 
* 

**Ceiling Analysis: What Part of the Pipeline to Work on Next**

* In earlier videos,
* I've said over + over that, when you're developing a ML
* system, one of the most valuable resources is your time as the developer,
* in terms of picking what to work on next. Or, if you have a team of developers or a team of engineers working together
* on a ML system. Again, one of the most valuable resources
* is the time of the engineers or the developers working on the system. + what you really want
* to avoid is that you or your colleagues your friends spend a lot
* of time working on some component. Only to realize after weeks or
* months of time spent, that all that worked just doesn't make a huge difference on
* the performance of the final system. In this video what I'd like to do is
* something called ceiling analysis. When you're the team working on
* the pipeline machine on your system, this can sometimes give you a very
* strong signal, a very strong guidance on what parts of the pipeline might be
* the best use of your time to work on. To talk about ceiling analysis I'm
* going to keep on using the example of the photo OCR pipeline. + see right here each of these boxes,
* text detection, character segmentation, character recognition, each of these boxes can have even
* a small engineering team working on it. Or maybe the entire system is
* just built by you, either way. But the question is where
* should you allocate resources? Which of these boxes is most worth
* your effort of trying to improve the performance of. In order to explain the idea
* of ceiling analysis, I'm going to keep using the example
* of our photo OCR pipeline. As I mentioned earlier, each of these
* boxes here, each of these machines + components could be the work of
* a small team of engineers, or the whole system could be
* built by just one person. But the question is, where should
* you allocate scarce resources? That is, which of these components, which
* one or two or maybe all three of these components is most worth your time,
* to try to improve the performance of. So here's the idea of ceiling analysis. As in the development process for
* other ML systems as well, in order to make decisions on what
* to do for developing the system is going to be very helpful to have a single
* rolled number evaluation metric for this learning system. So let's say we pick
* character level accuracy. So if you're given a test set image,
* what is the fraction of alphabets or characters in a test image
* that we recognize correctly? Or you can pick some other single road
* number evaluation that you could, if you want. But let's say for
* whatever evaluation measure we pick, we find that the overall system
* currently has 72% accuracy. So in other words,
* we have some set of test set images. + from each test set images,
* we run it through text detection, then character segmentation,
* then character recognition. + we find that on our test set the
* overall accuracy of the entire system was 72% on whatever metric you chose. Now here's the idea behind ceiling
* analysis, which is that we're going to go through, let's say the first module of our
* machinery pipeline, say text detection. + what we're going to do, is we're
* going to monkey around w/ the test set. We're gonna go to the test set. For every test example, which is going
* to provide it the correct text detection outputs, so in other words, we're going to
* go to the test set + just manually tell the algorithm where the text is
* in each of the test examples. So in other words gonna simulate
* what happens if you have a text detection system w/
* a hundred percent accuracy, for the purpose of detecting text in an image. + really the way you do
* that's pretty simple, right? Instead of letting your learning
* algorhtim detect the text in the images. You wouldn't say go to the images + just manually label what is the location
* of the text in my test set image. + you would then let these correct or let these ground truth labels of where
* is the text be part of your test set. + just use these ground truth
* labels as what you feed in to the next stage of the pipeline, so
* the character segmentation pipeline. Okay?
* So just to say that again. By putting a checkmark over here, what I mean is I'm going to go to my test
* set + just give it the correct answers. Give it the correct labels for
* the text detection part of the pipeline. So that as if I have a perfect test
* detection system on my test set. What we need to do then is run this
* data through the rest of the pipeline. Through character segmentation +
* character recognition. + then use the same
* evaluation metric as before, to measure what was the overall
* accuracy of the entire system. + w/ perfect text detection,
* hopefully the performance will go up. + in this example, it goes up by by 89%. + then we're gonna keep going, let's
* got o the next stage of the pipeline, so character segmentation. So again, I'm gonna go to my test set,
* + now I'm going to give it the correct text detection output + give it
* the correct character segmentation output. So go to the test set + manually label
* the correct segmentations of the text into individual characters,
* + see how much that helps. + let's say it goes up to 90%
* accuracy for the overall system. Right? So as always the accuracy
* of the overall system. So is whatever the final output of
* the character recognition system is. Whatever the final output
* of the overall pipeline, is going to measure the accuracy of that. + finally I'm going to build a character
* recognition system + give that correct labels as well, + if I do that too then
* no surprise I should get 100% accuracy. Now the nice thing about having done this
* analysis is, we can now understand what is the upside potential of improving
* each of these components? So we see that if we get
* perfect text detection, our performance went up from 72 to 89%. So that's a 17% performance gain. So this means that if we take our current
* system we spend a lot of time improving text detection, that means that we could potentially
* improve our system's performance by 17%. It seems like it's well worth our while. Whereas in contrast, when going from text detection when we
* gave it perfect character segmentation, performance went up only by 1%, so
* that's a more sobering message. It means that no matter how much time
* you spend on character segmentation. Maybe the upside potential is going to be
* pretty small, + maybe you do not want to have a large team of engineers
* working on character segmentation. This sort of analysis shows that even
* when you give it the perfect character segmentation, you performance
* goes up by only one percent. That really estimates what is the ceiling,
* or what is an upper bound on how much you can improve the performance of your system
* + working on one of these components. + finally, going from character, when we get better character recognition
* w/ the forms went up by ten percent. So again you can decide is ten percent
* improvement, how much is worth your while? This tells you that maybe w/ more effort
* spent on the last stage of the pipeline, you can improve the performance
* of the systems as well. Another way of thinking about this, is that by going through these sort
* of analysis you're trying to think about what is the upside potential of
* improving each of these components. Or how much could you
* possibly gain if one of these components became
* absolutely perfect? + this really places an upper bound
* on the performance of that system. So the idea of ceiling analysis is pretty
* important, let me just answer this idea again but w/ a different example but
* more complex one. Let's say that you want to do
* face recognition from images. You want to look at the picture +
* recognize whether or not the person in this picture is
* a particular friend of yours, + try to recognize the person
* Shown in this image. This is a slightly artificial example, this isn't actually how face
* recognition is done in practice. But we're going to set for an example,
* what a pipeline might look like to give you another example of how
* a ceiling analysis process might look. So we have a camera image, + let's say
* that we design a pipeline as follows, the first thing you wanna do is
* pre-processing of the image. So let's take this image like we
* have shown on the upper right, + let's say we want to
* remove the background. So do pre-processing +
* the background disappears. Next we want to say detect
* the face of the person, that's usually done on the learning So we'll run a sliding Windows crossfire
* to draw a box around a person's face. Having detected the face, it turns out
* that if you want to recognize people, it turns out that the eyes
* is a highly useful cue. We actually are, in terms of recognizing
* your friends the appearance of their eyes is actually one of the most
* important cues that you use. So lets run another crossfire to
* detect the eyes of the person. So the segment of the eyes + then since this will give us useful
* features to recognize the person. + then other parts of
* the face of physical interest. Maybe segment of the nose,
* segment of the mouth. + then having found the eyes, the nose,
* + the mouth, all of these give us useful features to maybe feed into
* a logistic regression classifier. + there's a job w/ a cost priority,
* they'd give us the overall label, to find the label for who we think
* is the identity of this person. So this is a kind of complicated pipeline,
* it's actually probably more complicated than you should be using if you actually
* want to recognize people, but there's an illustrative example that's useful
* to think about for ceiling analysis. So how do you go through ceiling
* analysis for this pipeline. Well se step through these
* pieces one at a time. Let's say your overall
* system has 85% accuracy. The first thing I do is
* go to my test set + manually give it the full
* background segmentation. So manually go to the test set. + use Photoshop or something to just
* tell it where's the background + just manually remove the graph background,
* so this is a ground true background, + see how much the accuracy changes. In this example the accuracy
* goes up by 0.1%. So this is a strong sign that even if you
* have perfect background segmentation, the form is, even w/ perfect
* background removal the performance or your system isn't going
* to go up that much. So it's maybe not worth a huge
* effort to work on pre-processing on background removal. Then quickly goes to test set give
* it the correct face detection images then again step though the eyes nose + mouth segmentation in some
* order just pick one order. Just give the correct
* location of the eyes. Correct location in noses,
* correct location in mouth, + then finally if I just give it the correct
* overall label I can get 100% accuracy. + so as I go through the system +
* just give more + more components, the correct labels in the test set, the
* performance of the overall system goes up + you can look at how much the
* performance went up on different steps. So from giving it
* the perfect face detection, it looks like the overall performance
* of the system went up by 5.9%. So that's a pretty big jump. It means that maybe it's worth quite
* a bit effort on better face detection. Went up 4% there, it went up 1% there. 1% there, + 3% there. So it looks like the components
* that most work are while are, when I gave it perfect face
* detection system went up by 5.9 performance when given perfect eyes
* segmentation went to four percent. + then my final which is cost for
* well there's another three percent, gap there maybe. + so this tells maybe whether the
* components are most worthwhile working on. + by the way I want to tell
* you a true cautionary story. The reason I put this is
* in this in preprocessing background removal is b/c I actually
* know of a true story where there was a research team that actually literally
* had to people spend about a year + a half, spend 18 months working
* on better background removal. But actually I'm obscuring the details for
* obvious reasons, but there was a computer vision application where there's a team of
* two engineers that literally spent about a year + a half working on better
* background removal, actually worked out really complicated algorithms +
* ended up publishing one research paper. But after all that work they found that
* it just did not make huge difference to the overall performance of the actual
* application they were working on + if only someone were to do
* ceiling analysis before hand maybe they could have realized. + one of them said to me afterward. If only you've did this sort of analysis
* like this maybe they could have realized before their 18 months of work. That they should have spend their effort
* focusing on some different component then literally spending 18 months
* working on background removal. So to summarize, pipelines are pretty pervasive in
* complex ML applications. + when you're working on a big
* ML application, your time as developer is so
* valuable, so just don't waste your time working on something that
* ultimately isn't going to matter. + in this video we'll talk about
* this idea of ceiling analysis, which I've often found to be a very good
* tool for identifying the component of a video as you put focus on that
* component + make a big difference. Will actually have a huge effect on the
* overall performance of your final system. So over the years working machine
* learning, I've actually learned to not trust my own gut feeling
* about what components to work on. So very often, I've work on machine
* learning for a long time, but often I look at a ML problem, +
* I may have some gut feeling about oh, let's jump on that component +
* just spend all the time on that. But over the years, I've come to
* even trust my own gut feelings + learn not to trust gut feelings that much. + instead, if you have a sort of machine
* learning problem where it's possible to structure things + do a ceiling
* analysis, often there's a much better + much more reliable way for
* deciding where to put a focused effort, to really improve the performance
* of some component. + be kind of reassured that,
* when you do that, it won't actually have a huge effect on the final

performance of the overall system.