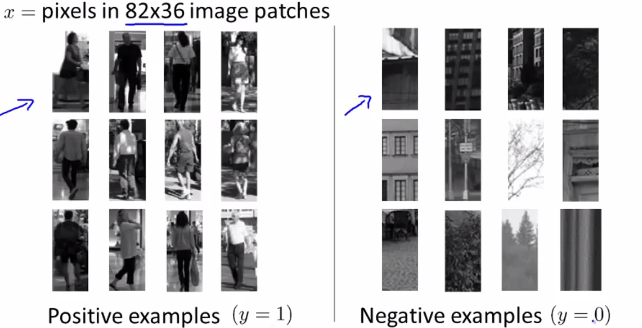
***Photo OCR***

**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**

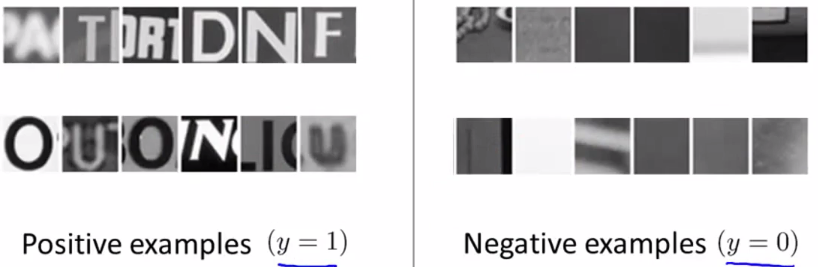
* The 1st stage of the photo OCR pipeline was Text detection, an unusual problem in CPU vision.
* Depending on the length of the text you're trying to find, rectangles that surround that text can have different aspects.
* Simpler example = pedestrian detection
* Want to take an image + find the individual 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



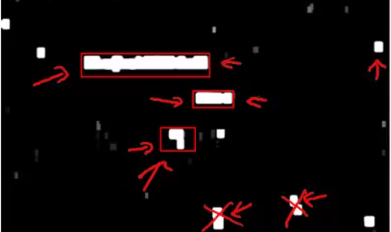
* Hopefully our classifier will return y = 0 for that patch, since there is no pedestrian.
* Next, take the green rectangle, slide it over a bit, + run the new image patch through our classifier to decide if there's a pedestrian there, and so on.
* The amount by which you shift the rectangle over each 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.

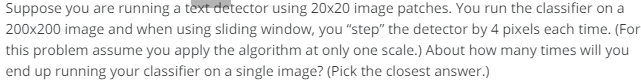


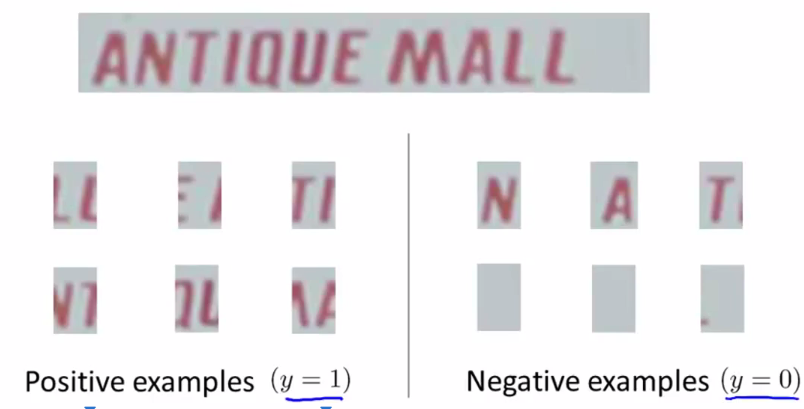
* So instead of trying to detect pedestrians, we're now trying to detect texts
* Using a fixed-scale window:

* 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**

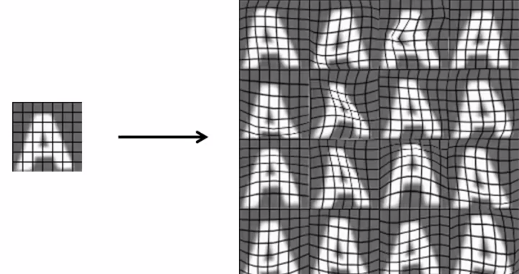
* 1 of the most reliable ways to get a high performance ML system is to take a low bias learning algorithm + train it on a massive training set.
* But where did you get so much training data from?
* In ML there's a fascinating idea called **artificial data synthesis** 🡪 doesn't apply to every problem, + 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.

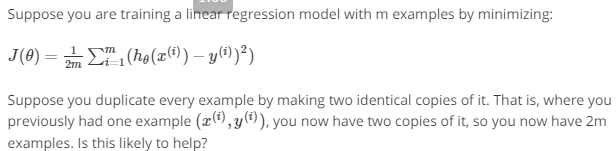


* Goal = Take image patches in this square + recognize the character in the middle of that image patch
* Using color doesn't seem to help that much for *this* particular problem.
* Modern CPUs often have a huge font library + if you want more training examples, 1 thing you can do = take characters from 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**

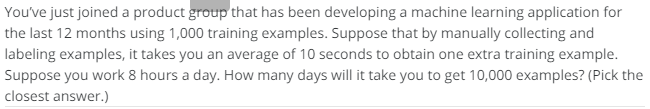


* Take a font 🡪 paste an image of one/a few character(s) from that font against a random background image 🡪 maybe apply a little blurring operators + distortions
* It takes thought in order to make the synthetic data look realistic, + if you do a sloppy job in terms of how you create synthetic data, 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



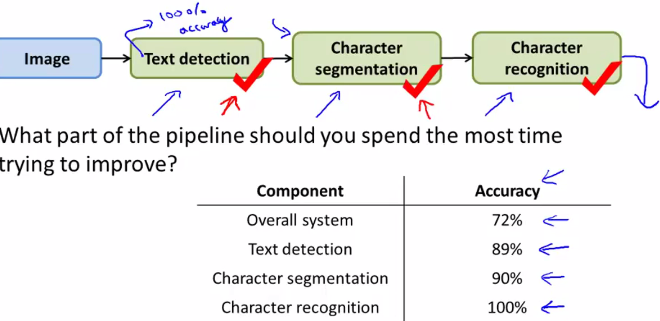
* Again, in order to do this, it does take thought + insight to figure out reasonable sets of distortions or other ways to amplify + multiply a training set
* For the specific example of character recognition, introducing warping seems like a natural choice, but for a different ML application, there 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.
* 

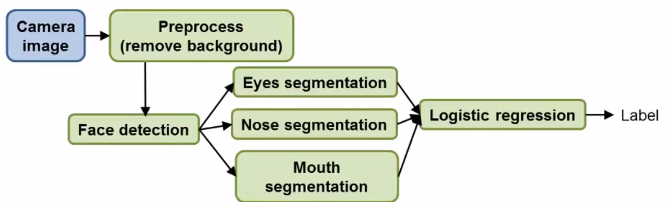


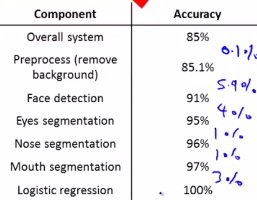
* As always, before expending a lot of effort figuring out how to create artificial training examples:
* It's often a good practice is to make sure you have **a low-bias classifier** = which means having a lot more training data will actually help
* Standard way 🡪 plot the learning curves + make sure 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
* 
* 

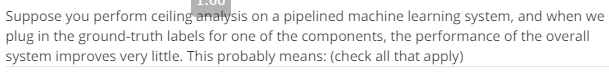
**Ceiling Analysis: What Part of the Pipeline to Work on Next**

* When developing a ML system, 1 of the most valuable resources = time as the developer in terms of picking what to work on next
* Or w/ a team of developers/engineers working together on a ML system = the time of the engineers/developers working on the system
* Want to avoid spending a lot of time working on some component only to realize weeks/months later that all that work just doesn't make a huge difference on performance of the final system.
* **Ceiling Analysis** can sometimes give you a very strong signal/guidance on what parts of the pipeline might be the best use of your time to work on.
* Ex: photo OCR pipeline
* 
* The question is *where should you allocate scarce resources*? Which of these boxes/components is most worth the time to try to improve the performance of?
* As in the development process for other ML systems, in order to make decisions on what to do next for developing a system, it’s helpful to have a **single-row number evaluation metric** for a learning system.
* Say we pick character-level accuracy 🡪 if given a test set image, what is the fraction of characters in a test image we recognize correctly?
* Can pick some other metric
* Say we find that the overall system currently has 72% accuracy on our test set
* Now go to the 1st module of our pipeline (text detection) + monkey around w/ the test set
* For every test example, provide the algorithm w/ the correct text detection outputs
* In other words, use the test set + just manually tell the algorithm where text is in each test examples
* This simulates what happens w/ a text detection system w/ 100% accuracy of detecting text in an image
* Instead of letting a learning algorithm detect text in the test images, you go to the test images + manually label the location of text in them + feed these ground-truth labels into the next stage of the pipeline (character segmentation)
* Then is run this data through the rest of the pipeline + use the same evaluation metric as before to measure the overall accuracy of the entire system
* W/ perfect text detection, hopefully performance will go up to say 89%
* Then keep going to the next stage of the pipeline, character segmentation.
* Go to test set, manually label correct segmentations of text into individual characters, + give the algorithm the correct character segmentation output + see how much that helps
* Finally go to the character recognition system
* Give that correct labels as well +, no surprise, get 100% accuracy for that component



* The nice thing about having done this analysis is we can now understand the upside potential of improving each of these components
* If we get perfect text detection, our performance went up from 72 to 89%, a 17% performance gain.
* So if we take our current system + spend a lot of time improving text detection, we could potentially improve our system's performance by 17% 🡪 seems well-worth our while.
* Whereas in contrast, in giving the algorithm perfect character segmentation, performance went up only by 1%, a more sobering message.
* It means no matter how much time spent on character segmentation, upside potential is going to be pretty small 🡪 maybe don’t want a large team of engineers working on character segmentation
* This sort of analysis shows that even w/ perfect character segmentation, performance goes up by only 1%.
* Finally, when we get better character recognition performance went up by 10%.
* You can decide if 10% improvement is worth your while?
* This says that maybe w/ more effort spent on the last stage of the pipeline, you can improve performance of the system as well.
* This analysis estimates the **ceiling**/upper bound on how much you can improve performance of a system + working on one of these components
* Another way of thinking 🡪 by going through these sort of analysis, you're trying to think about the upside potential of improving each of these components.
* How much could you possibly gain if 1 of these components became absolutely perfect?
* This really places an upper bound on the performance of that system
* The idea of **ceiling analysis** is pretty important
* Say that you want to do face recognition from images
* We have a camera image + design a pipeline as follows:
* 1) Pre-processing of the image 🡪 remove the background
* 2) Detect face of the person 🡪 run a sliding windows classifier
* turns out that eyes is a highly useful cue in terms of recognizing faces
* 3) Run another classifier to detect the eyes of the person (eyes segmentation)
* this will give us useful features to recognize the person
* 4) Segment out other parts of the face that may be of physical interest.
* Maybe segment out the nose + the mouth
* Having found the eyes, nose, + mouth, we get useful features to feed into a logistic regression classifier
* 5) Logistic regression
* Job of the classifier = give us overall final label for what we think who this person is.
* 
* To go through ceiling analysis for *this* pipeline, step through these pieces 1 at a time.
* Say the overall system has 85% accuracy
* 1) Go to the test set + manually give the system the ground-truth background segmentation (ex: Photoshop to tell it where the background is + to manually remove the background)
* 2) See how much the accuracy changes, maybe by only 0.1% = a strong sign that even w/ perfect background segmentation, system performance isn't going to go up that much
* So it's maybe not worth a huge effort to work on pre-processing/background removal.
* 3) Go to test set give the system the correct face detection images + run again
* 4) Do the above steps through the eyes nose + mouth segmentation in some order (correct location of eyes + run, then noses + run, then mouth + run)
* 5) Finally give the system the correct overall label to get 100% accuracy



* As I go through the system + give more + more components the correct labels from the test set, performance of the overall system goes up
* Can look at how much the performance went up on different steps.
* From giving perfect face detection, it looks like overall system performance went up by 5.9%, a pretty big jump = might be worth quite a bit effort to work on better face detection.
* Looks like the components most worthwhile to work on are face detection + eyes segmentation
* Cautionary story: A research team had 2 people spend about a year + a half working on better background removal for a computer vision application
* They actually worked out really complicated algorithms + ended up publishing a research paper.
* But after all that work they found it just did not make huge difference to the overall performance of the actual application they were working on
* If only someone did ceiling analysis beforehand, maybe they could have realized before their 18 months of work that they should have spent their effort focusing on some different component
* To summarize, pipelines are pretty pervasive in complex ML applications + when working on a big ML application, time as developer is so valuable
* Just don't waste your time working on something that ultimately isn't going to matter
* 
* 
* 
* **Ceiling analysis** can be a very good tool for IDing components that if you actually focus on, makes a big difference/has a huge effect on the overall performance of your final system
* Don’t trust gut feelings about what components to work on
* If you have a ML problem where it's possible to structure things to do a ceiling analysis, it’s a much better + much more reliable way for deciding where to put a focused effort to improve the performance of some component + be reassured it will actually have a significant effect on the final performance of the overall system.