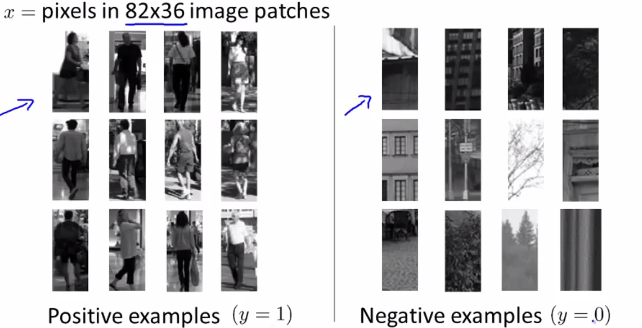
***Photo OCR***

**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**

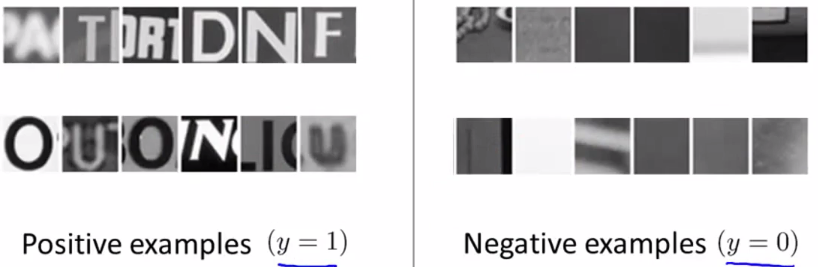
* The 1st stage of the photo OCR pipeline was Text detection, an unusual problem in CPU vision.
* Depending on the length of the text you're trying to find, rectangles that surround that text can have different aspects.
* Simpler example = pedestrian detection
* Want to take an image + find the individual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



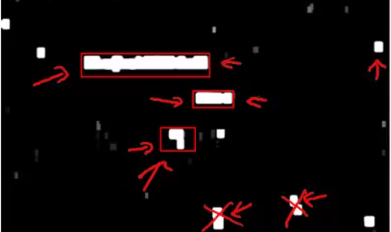
* Hopefully our classifier will return y = 0 for that patch, since there is no pedestrian.
* Next, take the green rectangle, slide it over a bit, + run the new image patch through our classifier to decide if there's a pedestrian there, and so on.
* The amount by which you shift the rectangle over each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

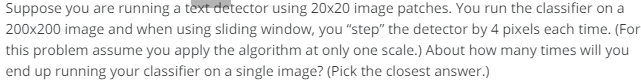


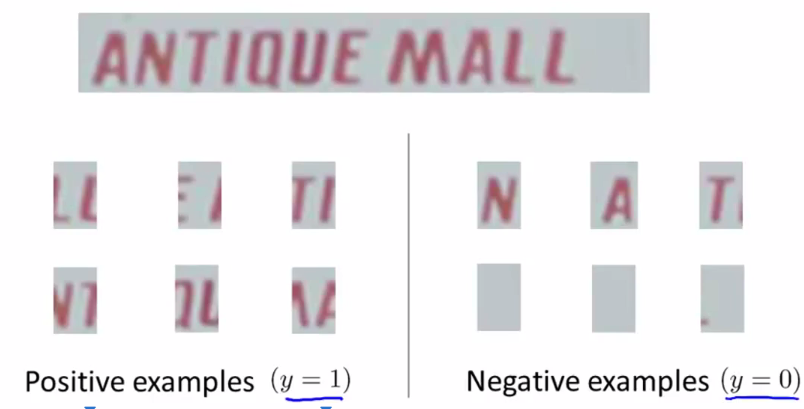
* So instead of trying to detect pedestrians, we're now trying to detect texts
* Using a fixed-scale window:

* 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**

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