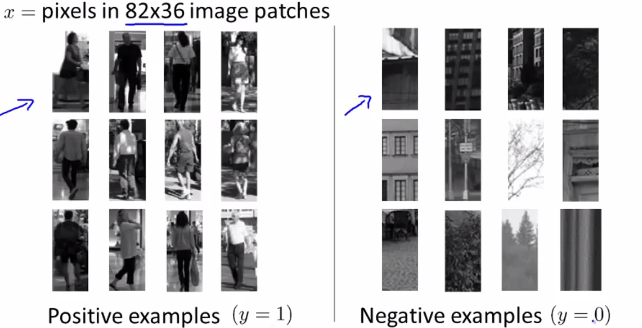
***Photo OCR***

**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 **crossfire** = 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



* . Here are examples of 82 X 36 image patches that do contain pedestrians + here are examples of images that do not. On this slide I show 12 positive examples of y1 + 12 examples of y0. In a more typical pedestrian detection application, we may have anywhere from a 1,000 training examples up to maybe 10,000 training examples, or even more if you can get even larger training sets. + what you can do, is then train in your network or some other learning algorithm to take this input, an MS patch of dimension 82 by 36, + to classify 'y' + to classify that image patch as either containing a pedestrian or not. So 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. Now, lets say we get a new image, a test set image like this + we want to try to find a pedestrian's picture image. What we would do is start by taking a rectangular patch of this image. Like that shown up here, so that's maybe a 82 X 36 patch of this image, + run that image patch through our classifier to determine whether or not there is a pedestrian in that image patch, + hopefully our classifier will return y equals 0 for that patch, since there is no pedestrian. Next, we then take that green rectangle + we slide it over a bit + then run that new image patch through our classifier to decide if there's a pedestrian there. + having done that, we then slide the window further to the right + run that patch through the classifier again. The amount by which you shift the rectangle over each time is a parameter, that's sometimes called the step size of the parameter, sometimes also called the slide parameter, + if you step this one pixel at a time. So you can use the step size or stride of 1, that usually performs best, that is more cost effective, + so using a step size of maybe 4 pixels at a time, or eight pixels at a time or some large number of pixels might be more common, since you're then moving the rectangle a little bit more each time. So, using this process, you continue stepping the rectangle over to the right a bit at a time + running each of these patches through a classifier, until eventually, as you slide this window over the different locations in the image, first starting w/ the first row + then we go further rows in the image, you would then run all of these different image patches at some step size or some stride through your classifier. Now, that was a pretty small rectangle, that would only detect pedestrians of one specific size. What we do next is start to look at larger image patches. So now let's take larger images patches, like those shown here + run those through the crossfire as well. + by the way when I say take a larger image patch, what I really mean is when you take an image patch like this, what you're really doing is taking that image patch, + resizing it down to 82 X 36, say. So you take this larger patch + re-size it to be smaller image + then it would be the smaller size image that is what you would pass through your classifier to try + decide if there is a pedestrian in that patch. + finally you can do this at an even larger scales + run that side of Windows to the end + after this whole process hopefully your algorithm will detect whether theres pedestrian appears in the image, so thats how you train a the classifier, + then use a sliding windows classifier, or use a sliding windows detector in order to find pedestrians in the image. Let's have a turn to the text detection example + talk about that stage in our photo OCR pipeline, where our goal is to find the text regions in unit. similar to pedestrian detection you can come up w/ a label 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. + so positive examples are going to be patches of images where there is text. + negative examples is going to be patches of images where there isn't text. Having trained this we can now apply it to a new image, into a test set image. So here's the image that we've been using as example. Now, last time we run, for this example we are going to run a sliding windows at just one fixed scale just for purpose of illustration, meaning that I'm going to use just one rectangle size. But lets say I run my little sliding windows classifier on lots of little image patches like this if I do that, what Ill end up w/ is a result like this where the white region show where my text detection system has found text + so the axis' of these two figures are the same. So there is a region up here, of course also a region up here, so the fact that this black up here represents that the classifier does not think it's found any texts up there, whereas the fact that there's a lot of white stuff here, that reflects that classifier thinks that it's found a bunch of texts. over there on the image. What i have done on this image on the lower left is actually use white to show where the classifier thinks it has found text. + different shades of grey correspond to the probability that was output by the classifier, so like the shades of grey corresponds to where it thinks it might have found text but has lower confidence the bright white response to whether the classifier, up w/ a very high probability, estimated probability of there being pedestrians in that location. We aren't quite done yet because what we actually want to do is draw rectangles around all the region where this text in the image, so were going to take one more step which is we take the output of the classifier + apply to it what is called an expansion operator. So what that does is, it take the image here, + it takes each of the white blobs, it takes each of the white regions + it expands that white region. Mathematically, the way you implement that is, if you look at the image on the right, what we're doing to create the image on the right is, for every pixel we are going to ask, is it w/ing some distance of a white pixel in the left image. + so, if a specific pixel is w/in, say, five pixels or ten pixels of a white pixel in the leftmost image, then we'll also color that pixel white in the rightmost image. + so, the effect of this is, we'll take each of the white blobs in the leftmost image + expand them a bit, grow them a little bit, by seeing whether the nearby pixels, the white pixels, + then coloring those nearby pixels in white as well. Finally, we are just about done. We can now look at this right most image + just look at the connecting components + look at the as white regions + draw bounding boxes around them. + in particular, if we look at all the white regions, like this one, this one, this one, + so on, + if we use a simple heuristic to rule out rectangles whose aspect ratios look funny because we know that boxes around text should be much wider than they are tall. + so if we ignore the thin, tall blobs like this one + this one, + we discard these ones because they are too tall + thin, + we then draw a the rectangles around the ones whose aspect ratio thats a height to what ratio looks like for text regions, then we can draw rectangles, the bounding boxes around this text region, this text region, + that text region, corresponding to the Lula B's antique mall logo, the Lula B's, + this little open sign. Of over there. This example by the actually misses one piece of text. This is very hard to read, but there is actually one piece of text there. That says [xx] are corresponding to this but the aspect ratio looks wrong so we discarded that one. So you know it's ok on this image, but in this particular example the classifier actually missed one piece of text. It's very hard to read because there's a piece of text written against a transparent window. So that's text detection using sliding windows. + having found these rectangles w/ the text in it, we can now just cut out these image regions + then use later stages of pipeline to try to meet the texts. Now, you recall that the second stage of pipeline was character segmentation, so given an image like that shown on top, how do we segment out the individual characters in this image? So what we can do is again use a supervised learning algorithm w/ some set of positive + some set of negative examples, what were going to do is look in the image patch + try to decide if there is split between two characters right in the middle of that image match. So for initial positive examples. This first cross example, this image patch looks like the middle of it is indeed the middle has splits between two characters + the second example again this looks like a positive example, because if I split two characters by putting a line right down the middle, that's the right thing to do. So, these are positive examples, where the middle of the image represents a gap or a split between two distinct characters, whereas the negative examples, well, you know, you don't want to split two characters right in the middle, + so these are negative examples because they don't represent the midpoint between two characters. So what we will do is, we will train a classifier, maybe using new network, maybe using a different learning algorithm, to try to classify between the positive + negative examples. Having trained such a classifier, we can then run this on this sort of text that our text detection system has pulled out. As we start by looking at that rectangle, + we ask, "Gee, does it look like the middle of that green rectangle, does it look like the midpoint between two characters?". + hopefully, the classifier will say no, then we slide the window over + this is a one dimensional sliding window classifier, because were going to slide the window only in one straight line from left to right, theres no different rows here. There's only one row here. But now, w/ the classifier in this position, we ask, well, should we split those two characters or should we put a split right down the middle of this rectangle. + hopefully, the classifier will output y equals one, in which case we will decide to draw a line down there, to try to split two characters. Then we slide the window over again, optic process, don't close the gap, slide over again, optic says yes, do split there + so on, + we slowly slide the classifier over to the right + hopefully it will classify this as another positive example + so on. + we will slide this window over to the right, running the classifier at every step, + hopefully it will tell us, you know, what are the right locations to split these characters up into, just split this image up into individual characters. + so thats 1D sliding windows for character segmentation. So, here's the overall photo OCR pipe line again. In this video we've talked about the text detection step, where we use sliding windows to detect text. + we also use a one-dimensional sliding windows to do character segmentation to segment out, you know, this text image in division of characters. The final step through the pipeline is the character qualification step + that step you might already be much more familiar w/ the early videos on supervised learning where you can apply a standard supervised learning w/in maybe on your network or maybe something else in order to take it's input, an image like that + classify which alphabet or which 26 characters A to Z, or maybe we should have 36 characters if you have the numerical digits as well, the multi class classification problem where you take it's input + image contained a character + decide what is the character that appears in that image? So that was the photo OCR pipeline + how you can use ideas like sliding windows classifiers in order to put these different components to develop a photo OCR system. In the next few videos we keep on using the problem of photo OCR to explore somewhat interesting issues surrounding building an application like this.

**Getting Lots of Data + Artificial Data**

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