Draft Version

MACHINE LEARNING YEARNING

Technical Strategy for AI Engineers, In the Era of Deep Learning

ANDREW NG

deeplearning.ai

Machine Learning Yearning is a
deeplearning.ai project.

Error analysis by parts

53 Error analysis by parts

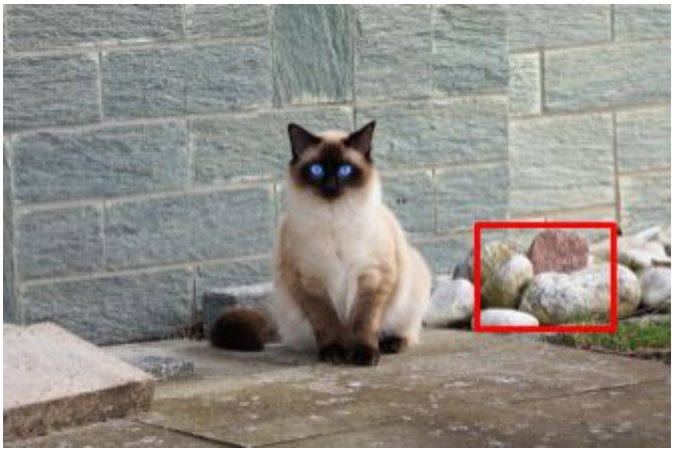
Suppose your system is built using a complex machine learning pipeline, and you would like to improve the system's performance. Which part of the pipeline should you work on improving? By attributing errors to specific parts of the pipeline, you can decide how to prioritize your work.

Let's use our Siamese cat classifier example:

The first part, the cat detector, detects cats and crops them out of the image. The second part, the cat breed classifier, decides if it is a Siamese cat. It is possible to spend years working on improving either of these two pipeline components. How do you decide which component(s) to focus on?

By carrying out **error analysis by parts**, you can try to attribute each mistake the algorithm makes to one (or sometimes both) of the two parts of the pipeline. For example, the algorithm misclassifies this image as not containing a Siamese cat (y=0) even though the correct label is y=1.

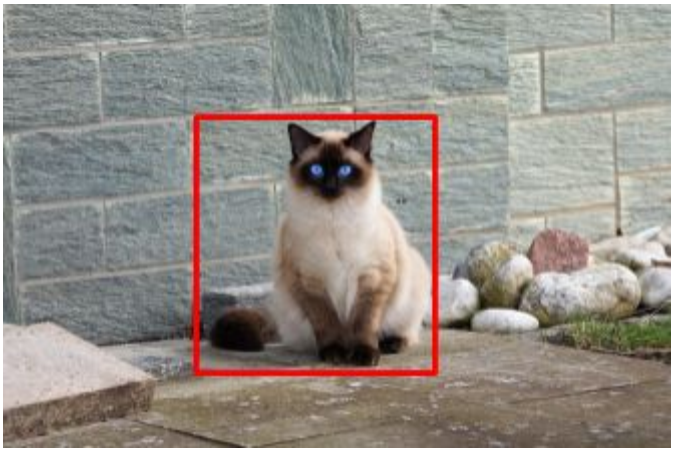
Let's manually examine what the two steps of the algorithm did. Suppose the Siamese cat detector had detected a cat as follows:



This means that the cat breed classifier is given the following image:

The cat breed classifier then correctly classifies this image as not containing a Siamese cat. Thus, the cat breed classifier is blameless: It was given of a pile of rocks and outputted a very reasonable label y=0. Indeed, a human classifying the cropped image above would also have predicted y=0. Thus, you can clearly attribute this error to the cat detector.

If, on the other hand, the cat detector had outputted the following bounding box:



then you would conclude that the cat detector had done its job, and that it was the cat breed classifier that is at fault.

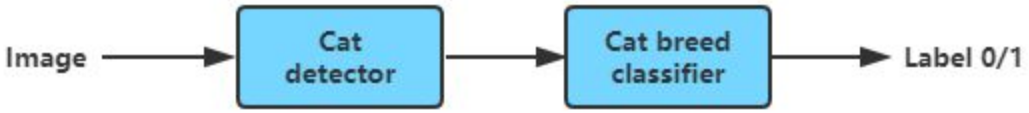
Say you go through 100 misclassified dev set images and find that 90 of the errors are attributable to the cat detector, and only 10 errors are attributable to the cat breed classifier. You can safely conclude that you should focus more attention on improving the cat detector.

Further, you have now also conveniently found 90 examples where the cat detector outputted incorrect bounding boxes. You can use these 90 examples to carry out a deeper level of error analysis on the cat detector to see how to improve that.

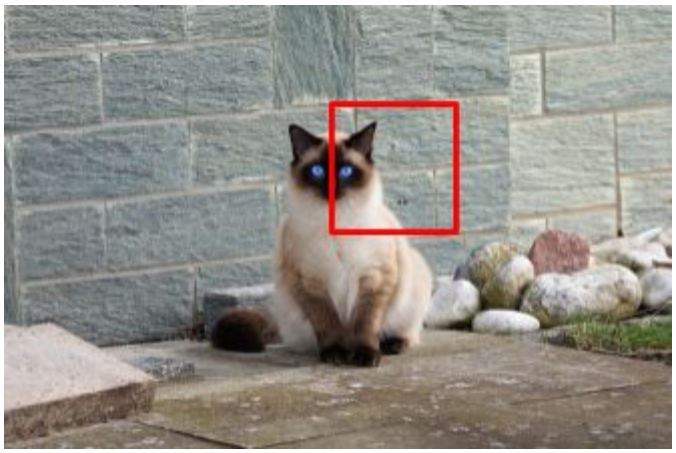
Our description of how you attribute error to one part of the pipeline has been informal so far: you look at the output of each of the parts and see if you can decide which one made a mistake. This informal method could be all you need. But in the next chapter, you'll also see a more formal way of attributing error.

54 Attributing error to one part

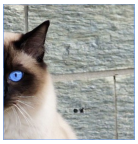
Let's continue to use this example:



Suppose the cat detector outputted this bounding box:



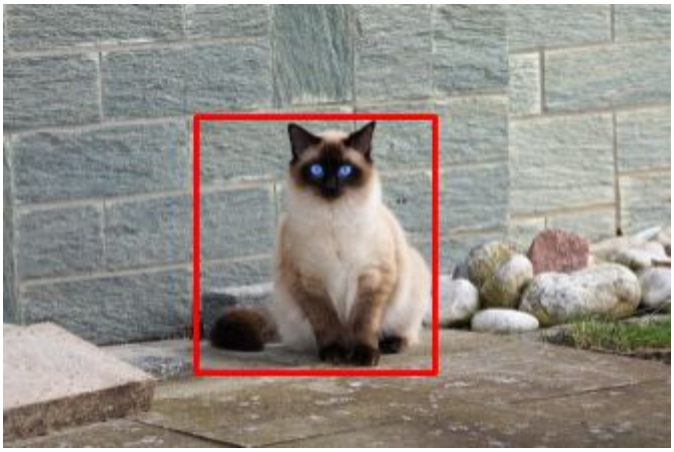
The cat breed classifier is thus given this cropped image, whereupon it incorrectly outputs y=0, or that there is no cat in the picture.



The cat detector did its job poorly. However, a highly skilled human could arguably still recognize the Siamese cat from the poorly cropped image. So do we attribute this error to the cat detector, or the cat breed classifier, or both? It is ambiguous.

If the number of ambiguous cases like these is small, you can make whatever decision you want and get a similar result. But here is a more formal test that lets you more definitively attribute the error to exactly one part:

1. Replace the cat detector output with a hand-labeled bounding box.



2. Run the corresponding cropped image through the cat breed classifier. If the cat breed classifier still misclassifies it, attribute the error to the cat breed classifier. Otherwise, attribute the error to the cat detector.

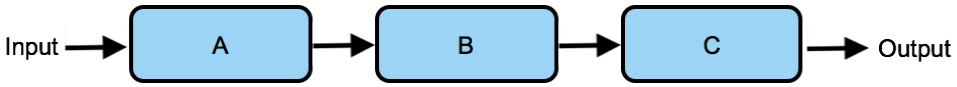
In other words, run an experiment in which you give the cat breed classifier a "perfect" input. There are two cases:

- Case 1: Even given a "perfect" bounding box, the cat breed classifier still incorrectly outputs y=0. In this case, clearly the cat breed classifier is at fault.
- Case 2: Given a "perfect" bounding box, the breed classifier now correctly outputs y=1. This shows that if only the cat detector had given a more perfect bounding box, then the overall system's output would have been correct. Thus, attribute the error to the cat detector.

By carrying out this analysis on the misclassified dev set images, you can now unambiguously attribute each error to one component. This allows you to estimate the fraction of errors due to each component of the pipeline, and therefore decide where to focus your attention.

55 General case of error attribution

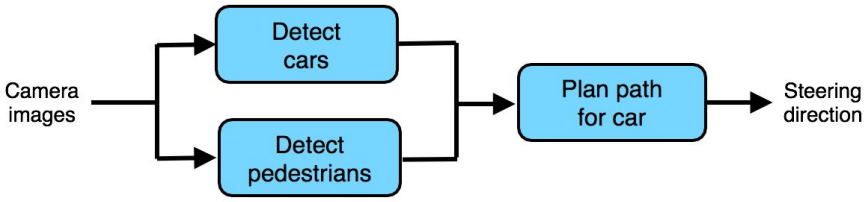
Here are the general steps for error attribution. Suppose the pipeline has three steps A, B and C, where A feeds directly into B, and B feeds directly into C.



For each mistake the system makes on the dev set:

1. Try manually modifying A's output to be a "perfect" output (e.g., the "perfect" bounding box for the cat), and run the rest of the pipeline B, C on this output. If the algorithm now gives a correct output, then this shows that, if only A had given a better output, the overall algorithm's output would have been correct; thus, you can attribute this error to component A. Otherwise, go on to Step 2.
2. Try manually modifying B's output to be the "perfect" output for B. If the algorithm now gives a correct output, then attribute the error to component B. Otherwise, go on to Step 3.
3. Attribute the error to component C.

Let's look at a more complex example:



Your self-driving car uses this pipeline. How do you use error analysis by parts to decide which component(s) to focus on?

You can map the three components to A, B, C as follows:

A: Detect cars
B: Detect pedestrians
C: Plan path for car

Following the procedure described above, suppose you test out your car on a closed track and find a case where the car chooses a more jarring steering direction than a skilled driver would. In the self-driving world, such a case is usually called a **scenario**. You would then:

1. Try manually modifying A (detecting cars)'s output to be a "perfect" output (e.g., manually go in and tell it where the other cars are). Run the rest of the pipeline B, C as before, but allow C (plan path) to use A's now perfect output. If the algorithm now plans a much better path for the car, then this shows that, if only A had given a better output, the overall algorithm's output would have been better; Thus, you can attribute this error to component A. Otherwise, go on to Step 2.
2. Try manually modifying B (detect pedestrian)'s output to be the "perfect" output for B. If the algorithm now gives a correct output, then attribute the error to component B. Otherwise, go on to Step 3.
3. Attribute the error to component C.

The components of an ML pipeline should be ordered according to a Directed Acyclic Graph (DAG), meaning that you should be able to compute them in some fixed left-to-right order, and later components should depend only on earlier components' outputs. So long as the mapping of the components to the A->B->C order follows the DAG ordering, then the error analysis will be fine. You might get slightly different results if you swap A and B:

A: Detect pedestrians (was previously *Detect cars*)
B: Detect cars (was previously *Detect pedestrians*)
C: Plan path for car

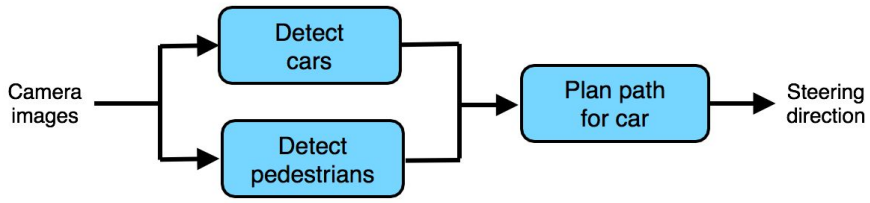
But the results of this analysis would still be valid and give good guidance for where to focus your attention.

56 Error analysis by parts and comparison to human-level performance

Carrying out error analysis on a learning algorithm is like using data science to analyze an ML system's mistakes in order to derive insights about what to do next. At its most basic, error analysis by parts tells us what component(s) performance is (are) worth the greatest effort to improve.

Say you have a dataset about customers buying things on a website. A data scientist may have many different ways of analyzing the data. She may draw many different conclusions about whether the website should raise prices, about the lifetime value of customers acquired through different marketing campaigns, and so on. There is no one "right" way to analyze a dataset, and there are many possible useful insights one could draw. Similarly, there is no one "right" way to carry out error analysis. Through these chapters you have learned many of the most common design patterns for drawing useful insights about your ML system, but you should feel free to experiment with other ways of analyzing errors as well.

Let's return to the self-driving application, where a car detection algorithm outputs the location (and perhaps velocity) of the nearby cars, a pedestrian detection algorithm outputs the location of the nearby pedestrians, and these two outputs are finally used to plan a path for the car.



To debug this pipeline, rather than rigorously following the procedure you saw in the previous chapter, you could more informally ask:

1. How far is the Detect cars component from human-level performance at detecting cars?
2. How far is the Detect pedestrians component from human-level performance?

3. How far is the overall system's performance from human-level performance? Here, human-level performance assumes the human has to plan a path for the car given only the outputs from the previous two pipeline components (rather than access to the camera images). In other words, how does the Plan path component's performance compare to that of a human's, when the human is given only the same input?

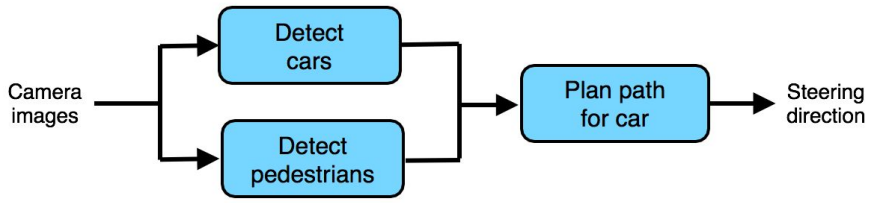
If you find that one of the components is far from human-level performance, you now have a good case to focus on improving the performance of that component.

Many error analysis processes work best when we are trying to automate something humans can do and can thus benchmark against human-level performance. Most of our preceding examples had this implicit assumption. If you are building an ML system where the final output or some of the intermediate components are doing things that even humans cannot do well, then some of these procedures will not apply.

This is another advantage of working on problems that humans can solve--you have more powerful error analysis tools, and thus you can prioritize your team's work more efficiently.

57 Spotting a flawed ML pipeline

What if each individual component of your ML pipeline is performing at human-level performance or near-human-level performance, but the overall pipeline falls far short of human-level? This usually means that the pipeline is flawed and needs to be redesigned. Error analysis can also help you understand if you need to redesign your pipeline.



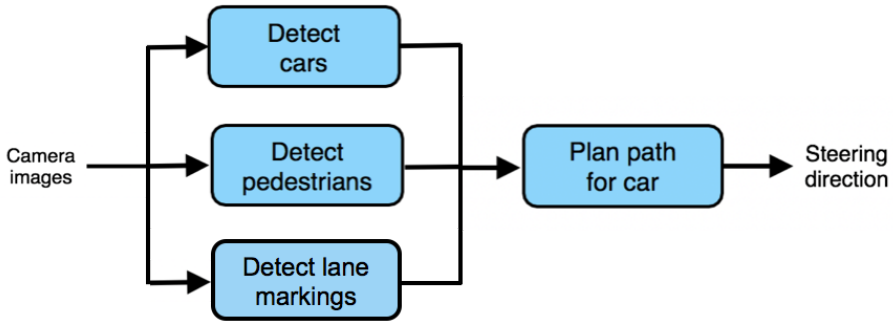
In the previous chapter, we posed the question of whether each of the three components' performance is at human level. Suppose the answer to all three questions is yes. That is:

1. The Detect cars component is at (roughly) human-level performance for detecting cars from the camera images.
2. The Detect pedestrians component is at (roughly) human-level performance for detecting cars from the camera images.
3. *Compared to a human that has to plan a path for the car given only the outputs from the previous two pipeline components (rather than access to the camera images),* the Plan path component's performance is at a similar level.

However, your overall self-driving car is performing significantly below human-level performance. I.e., humans given access to the camera images can plan significantly better paths for the car. What conclusion can you draw?

The only possible conclusion is that the ML pipeline is flawed. In this case, the Plan path component is doing as well as it can *given its inputs*, but the inputs do not contain enough information. You should ask yourself what other information, other than the outputs from the two earlier pipeline components, is needed to plan paths very well for a car to drive. In other words, what other information does a skilled human driver need?

For example, suppose you realize that a human driver also needs to know the location of the lane markings. This suggests that you should redesign the pipeline as follows[1]:



Ultimately, if you don't think your pipeline as a whole will achieve human-level performance, even if every individual component has human-level performance (remember that you are comparing to a human who is given the same input as the component), then the pipeline is flawed and should be redesigned.

[1] In the self-driving example above, in theory one could solve this problem by also feeding the raw camera image into the planning component. However, this would violate the design principle of "Task simplicity" described in Chapter 51, because the path planning module now needs to input a raw image and has a very complex task to solve. That's why adding a Detect lane markings component is a better choice--it helps get the important and previously missing information about lane markings to the path planning module, but you avoid making any particular module overly complex to build/train.

Conclusion

58 Building a superhero team - Get your teammates to read this

Congratulations on finishing this book!

In Chapter 2, we talked about how this book can help you become the superhero of your team.

The only thing better than being a superhero is being part of a superhero team. I hope you'll give copies of this book to your friends and teammates and help create other superheroes!