Generalization in Machine Learning

Sam, Joe, and Arthur 2020 Summer

The Cake Problem

A Tempting Pattern

We'll start out with a fun puzzle that's not directly related to machine learning but whose math will later help us. Imagine n people are seated around a large disk-shaped cake. Each pair of people will use a two-person handsaw to cut the cake. For example, the picture below shows a group of 4 people; there are 6 pairs of people, so the group makes 6 cuts total. When 4 people encircle the cake, it ends up cut into 8 pieces. We wonder: how many pieces arise when a different number of people sit around the cake?

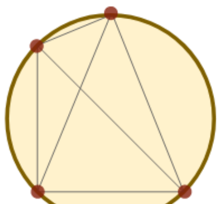

Figure 1: *With $n = 4$ people, we make $\binom{n}{2} = 6$ cuts, which gives 8 pieces in total (4 outside and 4 inside).*

Here are some drawings for possibilities ranging from $n = 1$ people to $n = 6$ people. Try counting the pieces and guessing a pattern! Can you find a formula for how many pieces arise when there are n people?

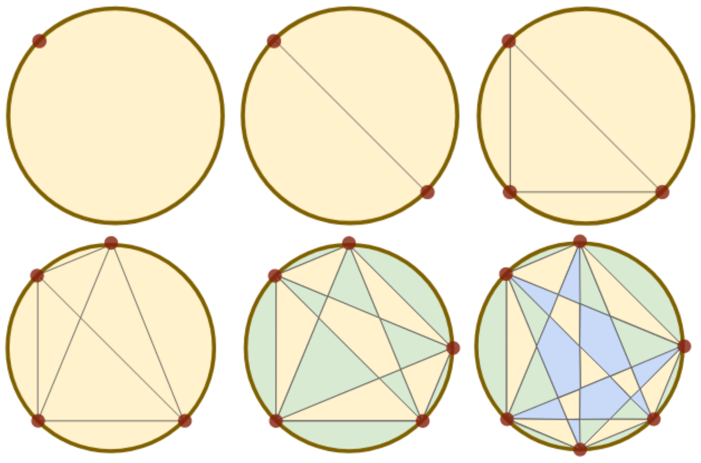

Figure 2: *Here, we color some of the cake pieces just to make them easier to see and to count. The $n = 1$ case doesn't have any cuts. The $n = 5$ case has 5 outside green pieces, 5 inside green pieces, 5 yellow triangles — so far this is three fives, which makes fifteen — and what's left is 1 yellow center piece. The $n = 6$ case has 6 outside green pieces, 6 inside green pieces, 6 outer yellow triangles, 6 blue triangles — so far this is four sixes, which makes twenty four — and what's left are the 7 central blue and yellow shapes.*

Well, it seems that with $n = 1, 2, 3, 4$ people, there are $p(n) = 1, 2, 4, 8$ pieces. (Here, p stands for "pieces", and we'll use this way of writing just to save time). It seems that $p(n)$ doubles for each next n, meaning that p looks like powers of two. In symbols, our guess is: $p(n) \overset{?}{=} 2^{n-1}$.

Let's check this guess. Does it continue to $n = 5$? We expect the next power of two, namely $8 + 8 = 16$. And yes: $p(5)$ really is 16! How about $n = 6$? We expect $16+16=32$. But — uh oh! — it seems that there are only 31 pieces. So the pattern breaks.

An Explanation

Let's now figure out $p(n)$ for any n; along the way, we'll see why the powers-of-two pattern seemed to work until $n=6$. There are two steps: we'll relate cake to constellations and then relate constellations to counting.

We'll use **Euler's polyhedron formula**. This formula says that if we connect a bunch of dots by edges to form some enclosed regions, then the numbers of dots, edges, and regions are related:

$$\text{Regions} - \text{Edges} + \text{Dots} = 1$$

For example, the constellation that we build up in stages below has 3 regions (one pentagon and two triangles), 13 edges, and 11 dots. And $3-13+11=1$, just like the formula says. Why is the formula true? Well, we can build up our

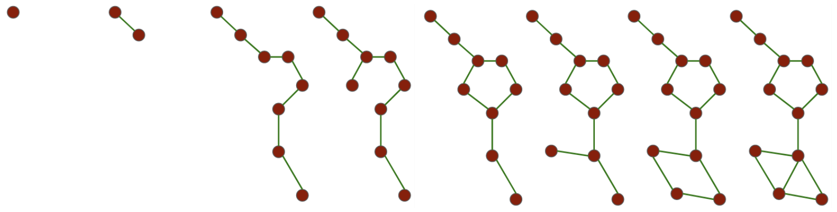

Figure 3: *Steps to build an example constellation (right) starting from a single dot (left). By the way, this method only helps us build constellations that don't have crossing edges, and each two of whose dots are connected by a path of edges. Euler's formula only applies to constellations that follow these rules.*

constellation starting from a single dot. The single dot follows the formula (since there are no regions and no edges, and $0-0+1=1$). And each step of building preserves the formula: **either** we connect a new dot to an old dot (so both $-$Edges and $+$Dots change by one, meaning that the total stays the same) **or** we connect two old dots to create a new region (so both Regions and $-$Edges change by one, meaning that the total stays the same). This logic proves Euler's formula.

To wrap up, let's think of a cake as a constellation as shown below. In addition to n outer dots, there are $\binom{n}{4}$ inner dots, because from each inner dot emanate 4 rays pointing toward 4 outer dots. By similar logic, the number of edges is $n+2\binom{n}{2}/2+4\binom{n}{4}/2$, since there are n outer arcs (green), $2\binom{n}{2}$ straight-half edges (blue) between outer dots, and 4 half-edges (orange) emanating from each of $\binom{n}{4}$ inner dots.

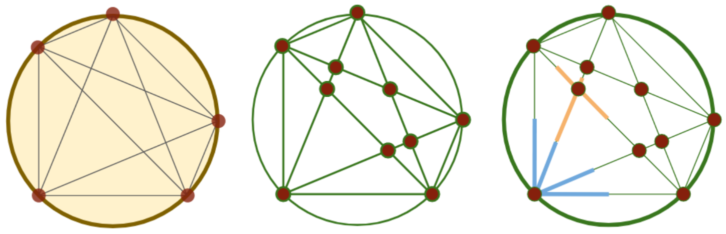

Figure 4: *We can think of a cake (left) as a constellation (middle) by adding inner dots. We can count the half-edges of the constellation (right) by binning them into three groups: the outer arcs (green), the straight half-edges between outer dots (blue), and the straight half-edges emanating from inner dots (orange).*

Putting all the pieces together, we find that Regions $= 1 + \text{Edges} - \text{Dots} = 1 + \binom{n}{2} + \binom{n}{4}$. We can simplify this by using the facts that $1 = \binom{n-1}{0}$ and $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$:

$$\text{Regions} = \binom{n-1}{0} + \binom{n-1}{1} + \binom{n-1}{2} + \binom{n-1}{3} + \binom{n-1}{4}$$

Since $\binom{n-1}{0} + \cdots + \binom{n-1}{k} = 2^{n-1}$ as long as $k \leq n-1$, this looks like powers of two as long as $n \leq 5$. So we have solved the mystery. With n people, the number $p(n)$ of pieces equals the number of subsets of $n-1$ people with at most 4 members.

Learning to Classify

What is Learning?

Today, we'll analyze machines that learn to classify images into two possible buckets such as Cow and Dog. To benefit from math, we need a precise set-up. What do we mean by "learning to classify"?

Say $\mathcal{X}$ contains all possible images and $\mathcal{Y} = \{\text{Cow}, \text{Dog}\}$ is the set of buckets. We posit a probability distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$ that says which pairs $(x, y) \in \mathcal{X} \times \mathcal{Y}$ are more likely and which are less likely. For instance: we might have , , $\cdots \in \mathcal{X}$, and $\mathcal{D}$ might say that (, Cow) is more likely to occur in nature than (, Cow), which is more likely to occur than (, Dog), which in turn is more likely than (, Dog).

A **classifier** is then a function from images to buckets, that is, some $f \in \mathcal{Y}^{\mathcal{X}}$. By **learning**, we mean acquiring information about $\mathcal{D}$ from "training samples" $\mathcal{S} \in (\mathcal{X} \times \mathcal{Y})^N$ drawn independently from $\mathcal{D}$. So "learning to classify" means mapping $\mathcal{S}$s to fs. In particular, an **algorithm** is a function $\mathcal{L} : (\mathcal{X} \times \mathcal{Y})^N \to \mathcal{Y}^{\mathcal{X}}$.[1]

[1] By the way, we don't have to know $\mathcal{D}$ in order to apply our theory; we just named it in order to reason about it.

Instead of considering all possible classifiers, we usually limit ourselves to some subset $\mathcal{H} \subseteq \mathcal{Y}^{\mathcal{X}}$ of especially nice, "candidate" classifiers. As an example, $\mathcal{H}$ might have just three elements:

$\mathcal{H} = \{$always say Cow,

say Cow if x is brown on average and otherwise say Dog,

say Cow if $x \in \{$, , $\}$ and otherwise say Dog$\}$

In practice, $\mathcal{H}$ might actually be a much bigger set of neural networks.

An algorithm $\mathcal{L}$ is good when $\mathcal{L}(\mathcal{S})$ accurately classifies images freshly drawn from $\mathcal{D}$. Notice that this is different from merely asking that $\mathcal{L}(\mathcal{S})$ accurately

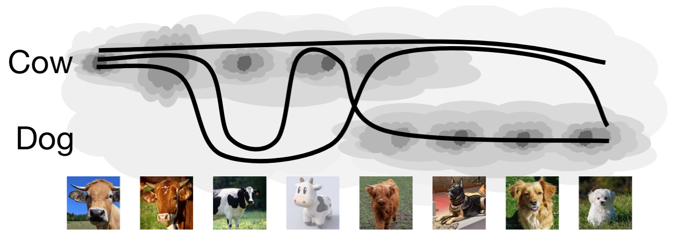

Figure 5: *A cartoon of $\mathcal{D}$ (clouds) and $\mathcal{H}$ (curves) with the $\mathcal{X}$ axis horizontal and the $\mathcal{Y}$ axis vertical.*

classifies the elements of $\mathcal{S}$! To notate this difference, we define the **out-error** and **in-error** (also called the test and train errors) of a candidate $f \in \mathcal{H}$ as

$$\mathcal{E}_{\text{out}}(f) = \mathbb{P}_{(x,y)\sim\mathcal{D}}\left[f(x) \neq y\right] \qquad \mathcal{E}_{\text{in},\mathcal{S}}(f) = \mathbb{P}_{(x,y)\sim\mathcal{S}}\left[f(x) \neq y\right]$$

Likewise, the out- and in-errors of a algorithm are $\mathcal{E}_{\dots}(\mathcal{L}) = \mathcal{E}_{\dots}(\mathcal{L}(\mathcal{S}))$ where $\mathcal{S} \sim \mathcal{D}$.

We hope $\mathcal{E}_{\text{out}}$ is low, but we don't know $\mathcal{E}_{\text{out}}$: from $\mathcal{S}$, we can directly calculate only $\mathcal{E}_{\text{in},\mathcal{S}}$. Intuiting that $\mathcal{E}_{\text{gap},\mathcal{S}} = \mathcal{E}_{\text{out}} - \mathcal{E}_{\text{in},\mathcal{S}}$ tends to be small, we might design $\mathcal{L}$ to minimize $\mathcal{E}_{\text{in},\mathcal{S}}$. That is, $\mathcal{L}$ computes $\mathcal{E}_{\text{in},\mathcal{S}}(f)$ for each $f \in \mathcal{H}$, then (breaking ties arbitrarily) to settle on an f with the least $\mathcal{E}_{\text{in},\mathcal{S}}$.

This $\mathcal{L}$ will work when $\mathcal{E}_{\text{gap},\mathcal{S}}$ is small. But is $\mathcal{E}_{\text{gap},\mathcal{S}}$ actually small? In particular, if $\mathcal{L}(\mathcal{S})$ accurately classifies its training samples, will it also accurately classify test samples it has never before seen?

Sometimes, the answer is **yes**. If there is 1 candidate $f \in \mathcal{H}$ and a thousand datapoints, then $\mathcal{E}_{\dots}(\mathcal{L}) = \mathcal{E}_{\dots}(f)$, and $\mathcal{E}_{\text{in},\mathcal{S}}(f)$ will very probably be very close to $\mathcal{E}_{\text{out}}(f)$. A technical reason for this is that $\mathcal{L}(\mathcal{S})$ and $\mathcal{S}$ are independent, and we may apply the law of large numbers. It's as if we roll a fair die a thousand times: we'd expect about a sixth of the rolls to land on ⚄.

Other times, the answer is **no**. If every possible classifier is a candidate (so $\mathcal{H}$ is very infinite) and only a few datapoints, then for any training sequence $\mathcal{S}$, many candidates will perfectly classify the training samples. Say that $\mathcal{L}$ breaks ties by settling on

$$f(x) = \text{say Cow if } (x, \text{Cow}) \in \mathcal{S}\text{; otherwise, say Dog}$$

The problem (assuming each image appears with negligible chance and that Cows occur with non-negligible chance) is f will misclassify every fresh image of a Cow as a Dog, so it will have a huge out-error! It's as if we do a trillion experiments where in each experiment we roll a fair die a thousand times: in any one experiment, we'd expect about a sixth of the rolls to land on ⚄, but if we focus on the experiment that has the most ⚄s (which is like focusing on the hypotheses that do the best on the training data), we are likely to focus on an outlier that has disproportionately many ⚄s!

These two examples illustrate that $\mathcal{L}$'s training performance generalizes to test time when $\mathcal{L}$ uses lots of datapoints to select from only a small set of candidates. How do these forces balance, for example if there are a lot of candidates and also a lot of datapoints? Let's find out.

Is Learning Possible?

Let's analyze the case $|\mathcal{H}| = 1$ more closely. We write p as shorthand for $\mathcal{E}_{\text{out}}(f)$; then $\mathcal{E}_{\text{in},\mathcal{S}}$ counts the fraction of heads that appear in N independent flips of a coin that has chance p of landing heads. Intuitively $\mathcal{E}_{\text{in},\mathcal{S}}$ will usually be close to p when N is big. Let's make "usually", "close to", and "big" precise.

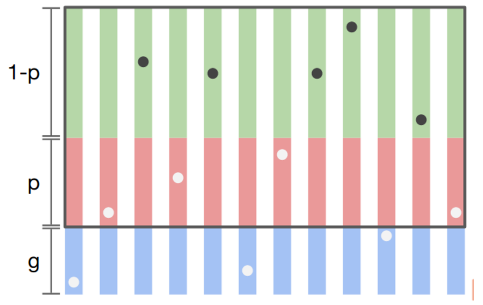

Figure 6: *We randomly select points on N vertical sticks. Each stick has three parts:* ***green*** *with length $1-p$,* ***red*** *with length p, and* ***blue*** *with length g. We call non-blue points* ***boxed*** *and non-green points* ***hollow****.*

We'll switch viewpoints: flipping a coin is like choosing a boxed point on a stick where green means tails and red means heads. We'll show that probably at most $M_0 = (p+g)N$ heads appear. That is, we want to show — given that all points are boxed — that probably at most M_0 points are red. For any $M \geq M_0$:

$$\begin{aligned}
&\mathbb{P}[M \text{ are red} \mid \text{all are boxed}] \\
=\ &\mathbb{P}[M \text{ are red and all are boxed}] \ / \ \mathbb{P}[\text{all are boxed}] \\
=\ &\mathbb{P}[M \text{ are hollow}] \cdot \mathbb{P}[\text{all hollows are red} \mid M \text{ are hollow}] \ / \ \mathbb{P}[\text{all are boxed}] \\
=\ &\mathbb{P}[M \text{ are hollow}] \cdot (1+g/p)^{-M} \ / \ (1+g)^{-N} \\
\leq\ &\mathbb{P}[M \text{ are hollow}] \cdot (1+g/p)^{-M_0} \ / \ (1+g)^{-N}
\end{aligned}$$

Since the above holds for all $M \geq M_0$, we conclude:

$$\begin{aligned}
&\mathbb{P}[\text{at least } M_0 \text{ are red} \mid \text{all are boxed}] \\
\leq\ &\mathbb{P}[\text{at least } M_0 \text{ are hollow}] \cdot (1+g/p)^{-M_0}/(1+g)^{-N} \\
\leq\ &(1+g/p)^{-M_0}/(1+g)^{-N} && \text{probabilities are at most 1} \\
\leq\ &\exp(-M_0 g/p)\exp(Ng) && 1+x \leq \exp(x) \\
=\ &\exp(-(p+g)Ng/p + Ng) && \text{substitute } M_0 = (p+g)N \\
=\ &\exp(-Ng^2/p) && \text{simplify} \\
\leq\ &\exp(-Ng^2) && \text{probabilities are at most 1}
\end{aligned}$$

This is the **Chernoff bound** for coin flips.[2]

[2] There are lots of versions of this bound. Google up "multiplicative Chernoff bound" or "Bernstein inequality" for some fun examples!

What have we learned? We have learned that if we flip a fair coin N times, the odds of deviating by more than 1% from the expected value of 50% heads decays very quickly with N! Tying back to learning, if hypothesis f accurately classifies 80% of a large training set ($E_{\text{in},\mathcal{S}} = 0.2$), then f will probably accurately classify about 80% of its test set ($E_{\text{out}} \approx 0.2$). For example, if there are $N = 300$ training points, then the odds of f suffering less than 70% test accuracy are less than

$$\exp(-Ng^2) = \exp(-300 \cdot 10\%^2) \approx 5.0\%$$

Notice the exponential dependence on N! Intuitively, exponential dependencies are strong dependencies. For example, if we add 10^4 to N, then we get

$$\exp(-Ng^2) = \exp(-400 \cdot 10\%^2) \approx 1.8\%$$

And if we add yet another 10^4 to N, then we get

$$\exp(-Ng^2) = \exp(-500 \cdot 1\%^2) \approx 0.67\%$$

The chance of an inacceptable test accuracy decays quickly with N. Data is useful!

We can likewise play with g instead of N. Since g is squared in the formula, it can have a huge effect on the probability of success. What if we think that a generalization gap of $g = 10\%$ test accuracy is too much? For $g = 8\%$, we find:

$$\exp(-Ng^2) = \exp(-500 \cdot 6\%^2) \approx 4.1\%$$

For $g = 6\%$:

$$\exp(-Ng^2) = \exp(-500 \cdot 6\%^2) \approx 16\%$$

The chance of an inacceptable test accuracy increases very quickly quickly as our notion of acceptability becomes more stringent. Tolerance is useful!

Now, instead of trying different values of g, we might want to solve for g in terms of a probability of failure that we name δ. We find that with probability at most δ will the generalization gap exceed $\sqrt{\log(1/\delta)/N}$. Call this $\mathcal{G}(\delta)$.

Multiple candidates

What if $\mathcal{H}$ contains multiple (say, H many) candidates? Well, each candidate $f \in \mathcal{H}$ has $\mathcal{E}_{\text{gap},\mathcal{S}}(f) \geq \mathcal{G}(\delta/H)$ with probability at most $\delta/|\mathcal{H}|$. Therefore, the chance that some $f \in \mathcal{H}$ has $\mathcal{E}_{\text{gap},\mathcal{S}}(\mathcal{L}) \geq \mathcal{G}(\delta/H)$ is at most H times δ/H. In sum, with probability at least $1-\delta$ the gap will be small:

$$\mathcal{E}_{\text{gap},\mathcal{S}}(\mathcal{L}) \leq \max_{f\in\mathcal{H}} \mathcal{E}_{\text{gap},\mathcal{S}}(f) < \mathcal{G}(\delta/H) = \sqrt{\log(H/\delta)/N}$$

What we have shown is that, if our algorithm $\mathcal{L}$ selects from among a finite number of candidates, then learning is possible: the more data points we have, the smaller will be the generalization gap bound $\mathcal{G}(\delta/H)$.

Here's an example. Suppose we train a neural net that has 10^3 parameters, each a floating point number. Since there are at most 2^{32} possible floating point numbers, H is at most $2^{32\cdot 10^3}$. If we train on 10^6 images, then with probability $99.99\% = 1 - 10^{-4}$, the generalization gap is less than

$$\sqrt{\log(2^{32\cdot 10^3}/10^{-4})/10^6} \approx 9.8\% \approx 10\%$$

So if after training, the neural net's training accuracy is 80%, we can expect its testing accuracy to be at least 70%. We proved this without knowing $\mathcal{D}$; we used simply that $\mathcal{D}$ exists at all! Ain't that cool?