CSCI 3340.01: Intro to Machine Learning with Applications to Chemistry

Jean-Baptiste Tristan

Fall 2021

Welcome to 3340.01: Introduction to Machine Learning with Applications to Chemistry. Here's some important information:

- The course webpage is:
  `https://bostoncollege.instructure.com/courses/1625020`
- Office hours are at the following times in Fulton 160:
  - Dr. Tristan: 6-8pm on Tuesday
  - Daniella: 12-2pm on Wednesday
  - Jetta: 5-7pm on Wednesday
- Everyone needs to serve as scribe for at least one lecture. The sign-up sheet is `here`.
- Relevant emails are {`tristanj, chujx, zunicd, asilisj`}`@bc.edu`. Email with questions, comments, or concerns.
- These notes were taken by Julian and have **not been carefully proofread** – they're sure to contain some typos and omissions, due to Julian.

Contents

§1 Tuesday, August 31

§1.1 What is machine learning?

We're going to start by discussing what machine learning is and explaining why we think it's a good idea to approach the subject with applications (such as chemistry) in mind.

So, what is machine learning? There are a couple of perspectives:

1. The engine of artificial intelligence (AI).
   - AI is a relatively old field, having commenced around the 50's, but ML is what's really given it success in the last few decades.
   - There's a famous quote from Andrew Ng:

     > "AI is the new electricity. It will transform every industry and create huge economic value. Technology like supervised learning is automation on steroids. It is very good at automating tasks and will have an impact on every sector – from healthcare to manufacturing, logistics and retail."

2. A significant source of sometimes meaningful, high paying jobs (somewhat more cynically).
3. A genuine source of technological and scientific innovation.
   - Recently, we've seen DeepMind's AI make advances in protein folding problems and defeat Go champions.
4. And (hopefully) ridiculous hype.
   - Elon Musk:

     > "With AI, we are summoning the demon."

All of this helps us see that ML is a pretty big deal, but it doesn't tell us much about what it actually *is*. Ultimately, machine learning is an interdisciplinary science combining computer science (via algorithms) and statistics (via data, often randomly generated). Let's make things a bit more concrete by looking at an example.

Example 1.1

Consider the problem of filtering email for spam. Nowadays, services like Gmail do the work for you, but a few decades ago it was a real problem. We can think about two approaches to filter email for spam:

1. Write an algorithm with lots of explicit rules (e.g., if the email contains the phrase 'amazing opportunity' or 'essential oils', file it as spam).
2. Have a learning algorithm which takes in many examples of emails labeled as spam and not-spam and produces its own rule for filtering.

The second approach turns out to be much more fruitful than the first. It might be hard to see how it would actually be implemented, but hopefully you can see why the first approach wouldn't work very well. It'd be extremely hard to exhaustively list all the rules defining spam without going overboard.

In classical computer science, the problem setup is to – for instance – take in a list of integers and output a list of sorted integers. In machine learning, the problem would be to take in a collection of integer lists with sorted copies, and to output a rule for sorting lists.

Returning to our previous example, imagine that there is a perfect spam filter f that maps an email to 1 if it is spam and 0 otherwise. Given sample data, i.e. a list of example emails with labels of 0 and 1, we'd like to output rule for predicting whether an email (without unknown label!) has label 0 or 1. The goal would be to have our rule behave as much as possible like the unknown function f.

Let's generalize this. In machine learning, the problem setting is defined by a universe X of **features** (e.g. possible emails) and Y of **labels** (e.g. $\{$*spam, not spam*$\}$, or $\{0, 1\}$). One is given a data set $D \subseteq X \times Y$ generated by some target function $f : X \to Y$, and the goal is to use D to output a prediction function as close to f as possible. That was very fast, but the goal was to give you a flavor of what machine learning is.

§1.2 Why focus on chemistry?

If you're any kind of natural scientist, it's useful to gain exposure to ML because it's an increasingly important part of scientific research. If you're a computer scientist, looking at applications of ML to chemistry/natural science gives you a great source of interesting problems and data sets, and you may find it more meaningful than working on problems in ad placement, for instance.

One curious thing that sometimes happens in chemistry and physics is that we have a perfect mathematical description of a system's behavior via an equation, but the equation is far too difficult to actually solve. In this kind of setting, you can see how machine learning might be of help. At a high level, combining ML and physics has lots of potential in science & tech.

§1.3 Course structure

We'll be thinking of this as a three part course.

1. Use ML models to learn from experimental data
   - We'll learn the fundamental of machine learning: bias-variance trade-off, the no free lunch theorem, etc.
2. Create data and bespoke ML models
   - Dive into linear regression
   - Graph Neural Networks
3. Combine ML and quantum chemistry to solve the Schrodinger equation
   - Look at Gaussian processes

See the syllabus on canvas for details on the course staff, the schedule of office hours, etc.

§2 Thursday, September 2

As I mentioned last time, the 4th homework will be larger/more significant and will concern predicting the solubility of molecules. The data set for that assignment is AqSolDb, and it might be worth giving it a look in the next few weeks. Anyway, today, I'll cover the machinery that you'll need to solve the first homework.

§2.1 Key libraries

There are four libraries that you should know about:

1. `pandas` (for analyzing data)
2. `matplotlib` (for plotting data)
3. `RDKit` (chemistry/machine learning)
4. `mordred` (wrapper around `RDKIT`)

It won't be necessary to go particularly deep into any of these libraries, and I'll try to cover all the major commands/ideas in each library.

So, the name of the game is to predict a molecule's solubility, which is the amount amount of the molecule that can dissolve into a given solvent (in our case, usually water). In order to use molecules in a machine learning algorithm, we need to represent the molecule in a format that the machine can understanding (e.g. as a string, graph, etc.). One of the most popular such methods is SMILE, which represents a molecule as a string. We won't dive into the details of SMILE, but it's our first example of a representation, and it also introduces some questions concerning feature engineering.

For instance, the energy of a molecule is invariant to rotation and translation of the molecule. So, when predicting molecules' energies, it would be ideal to use a representation which is invariant with respect to each of those transformations. That's a challenging problem in feature engineering – i.e., how objects are formatted to a machine learning algorithm – and it's of great practical importance.

For the rest of the class, we walked through examples of using these packages - bit thorny to write it all down here :/

§3 Tuesday, September 7

We're going to talk about lots of big ideas in machine learning today. We'll also be talking about matrices, and we'll often be thinking of their rows as points (or vectors) in n-dimensional space. The `numpy` and `pandas` packages have lots of built-in functionality for handling matrices.

§3.1 True and empirical risk

Let's say we have a collection of points $\{(x_i, f(x_i)\}_{i\in[n]}$ generated from some curve f. We'd like to deduce from these points a curve as 'close' to f as possible. In fact, this is precisely the notion of **generalization** – using the data to create a function h which mimics function f as closely as possible everywhere, not just at the $\{x_i\}$.

How do we formalize the notion of proximity in this context, i.e. what does it mean for h to be close to f? There are a couple of reasonable choices for our **true risk** $\mathcal{L}$, like

$$\mathcal{L}_f(h) = \int_{-\infty}^{\infty} |h(x) - f(x)| \mathrm{d}x \quad \text{or} \quad \mathcal{L}_f(h) = \int_{-\infty}^{\infty} (h(x) - f(x))^2 \mathrm{d}x.$$

The issue here is that we can't compute $\mathcal{L}_f$, since we don't actually know what f is (that's the whole point!). One natural proxy for the relative behaviors of f and h everywhere is their relative behaviors on the sample points $\{x_i\}$. That's known as the **empirical risk**, and it's formalized as follows:

$$\widehat{\mathcal{L}}_f(h) = \frac{1}{n}\sum_{i=1}^{n}(h(x_i) - f(x_i))^2.$$

Make sure to keep in mind the intuition here – $\mathcal{L}$ measures how closely h mimics f on all of $\mathbb{R}$, while $\widehat{\mathcal{L}}$ measures how closely h mimics f on the $\{x_i\}$. Another important part of the picture in machine learning is the **hypothesis class** $\mathcal{H}$, which is the collection of candidate functions h that we can use to guess f. In particular, we restrict focus to a collection of functions $\mathcal{H}$ at the outset, for the sake of making our problem more tractable, for expressing some prior belief about the function f, etc. In the case of linear regression, $\mathcal{H}$ will consist of linear functions, so the name of the game is to find a line (or hyperplane) that mimics f as closely as possible.

Remark 3.1. If the label set $\mathcal{Y}$ equals $\{0,1\}$, then the learning problem is one of *binary classification.* If it's $\{0,1,\dots,n\}$, then it's *multi-class classification.* If it's $\mathbb{R}$, then the problem is *regression.*

§3.2 Underfitting and overfitting

The restriction to hypotheses in $\mathcal{H}$ can be a useful way to express prior knowledge, but it can also leave you somewhat powerless (e.g. if $\mathcal{H}$ consists of linear functions and f is a degree 10 polynomial). This is referred to as **underfitting**, and it tends to result in high levels of both true and empirical risk. It's worth noting here, though, that even linear regression can be extremely powerful when you perform clever feature engineering.

One the other hand, if $\mathcal{H}$ allows for perverse functions, then you can end up with a function h that simply memorizes all the points $\{(x_i, f(x_i)\}$ and outputs 0 everywhere else, for instance. This phenomenon, of fitting the noise in the sample and generalizing poorly, is known as **overfitting**. On the ground, it's something of a tug of war between underfitting and overfitting – you can run into issues when $\mathcal{H}$ is too powerful as well as when it's too weak. This issue is known as the **bias-complexity tradeoff**. It actually has a precise (though trivial) mathematical formulation:

$$\mathcal{L}_f(h) \;=\; \underbrace{\min_{h\in\mathcal{H}} \mathcal{L}_f(h)}_{\text{Approximation error}} \;+\; \underbrace{\left(\mathcal{L}(h) - \min_{h\in\mathcal{H}} \mathcal{L}_f(h)\right)}_{\text{Estimation error}}.$$

The key takeaway here is that the approximation error decreases as $\mathcal{H}$ gets larger, but the estimation error increases in turn, since it's harder (i.e. requires more data) to single out the best hypothesis in $\mathcal{H}$. In English: increasing the size of $\mathcal{H}$ is a double-edged sword.

§4 Thursday, September 9

§4.1 Probability distributions

There's lots more ground to cover regarding the big ideas in ML. As we mentioned last time, there general setup is that an unknown function $f : \mathcal{X} \to \mathcal{Y}$ which generates our data set $\{(x_i, y_i) \mid x_i \in X, y_i = f(x_i)\}$. The goal is to 'guess' the function f from that data set.

We introduced notions of true risk $\mathcal{L}_f$ for measuring how closely a candidate function h mimics f. More explicitly, we considered

$$\mathcal{L}_f(h) = \int_{-\infty}^{\infty} |h(x) - f(x)| \mathrm{d}x \quad \text{and} \quad \mathcal{L}_f(h) = \int_{-\infty}^{\infty} (h(x) - f(x))^2 \mathrm{d}x.$$

These are both pretty reasonable candidates for true risk (can you see why?), but it turns that we'd like something more sophisticated. In particular, we may to weight the performance of h relative to f more greatly on certain regions of the domain $\mathcal{X}$ than others. For instance, the failure (or even success) of h on regions of $\mathcal{X}$ that are very rarely witnessed in practice shouldn't matter very much. One way to formalize this is to introduce a probability distribution $\mathcal{D}$ on $\mathcal{X}$, and to define

$$\mathcal{L}_{f,\mathcal{D}}(h) = \mathbb{E}[(h(x) - f(x))^2]$$

where the expectation on the right hand side is taken over $\mathcal{D}$. Now that probability is coming into play, it'll be useful to have some inequalities in our toolkit. One important one is Hoeffding's inequality.

Lemma 4.1 (Hoeffding's inequality)

Let $\Theta_1, \ldots, \Theta_n$ be i.i.d. random variables with $\mathbb{E}(\Theta_i) = \mu$ and $\mathrm{P}(\Theta_i \in [a,b]) = 1$. Then

$$\mathrm{P}(|\frac{1}{n}\sum_{i=1}^{n}\Theta_i - \mu| > \epsilon) \le 2e^{-n\epsilon^2/(b-a)^2}.$$

In English, the inequality is telling us about how closely the sample mean of the Θ_i tends to approximate their true mean μ. One crucial application of the inequality is that it helps us describe how quickly empirical risk comes to approximate true risk. Crucially, empirical risk becomes a greater estimator of true risk as sample size increases but suffers as $|\mathcal{H}|$ increases. This takes us back to the bias-variance trade-off we were discussing earlier.

§4.2 Model validation

Here's a question: after picking a model, how can we measure its performance to make sure we didn't go wrong? One important observation is that we can't just see how it performs on the data that we trained with. Simply put, this is cheating: we're testing the model with the data we already used to construct it (i.e. with the data whose behavior is already 'baked into' the parameters/structure of that model).

Fortunately, there's a relatively simply solution you may have heard of: split the data into a training set and a test set. As their names suggest, the training set is used to pick a model (perhaps via empirical risk minimization), and the test set is used to give it an honest evaluation of its performance.

One important visual tool for contrasting the training and testing losses is the **validation curve**, which plots those losses with respect to the choice of model. If the training loss is high, that's usually a sign of underfitting; if testing loss is very high (relative to training loss), that's often a sign of overfitting. A similar tool for examining the effect of data quantity is the **learning curve** – it plots the training and test loss of a particular model with respect to the size of the data set.

§5 Tuesday, September 14

§5.1 Cross-validation

Last week, we talked about the importance of splitting the data into two groups – the testing and training data sets. In particular, it's bankrupt to assess your model using the data that you used to train it, as it's already seen that data before.

In reality, this idea is often taken even further, though. It's difficult to have a test set that's big enough to be meaningful without detracting from the precious training set.

One solution for this quandary is **cross-validation**. Rather than splitting the data into testing and training tests, we split it into k many folds. For now, say we have 5 folds. Then we can train the data on folds $\{1,2,3,4\}$ and test it on the 5th fold. We can also train on $\{1,2,3,5\}$ and test on fold 4. By doing this $k = 5$ many times, we can get much more information than a mere training/testing split.

Remark 5.1. There's an important warning here: if you use cross-validation on your data set to pick parameters/build your model, then you *can't* use any of that data to evaluate your model's performance. Your model has already seen the data, so it's bankrupt to then test it using that data.

§5.2 Occam's razor

We've been talking about empirical risk minimization, i.e. the rule of picking a hypothesis h from a class $\mathcal{H}$ which minimizes empirical risk on the provided data set. Generally speaking, there's no reason for this h to be unique – there may be a variety of hypotheses $h_{i_1}, \ldots, h_{i_N}$ attaining an empirical risk of zero, for instance. In this setting, we'd like some criterion for breaking ties.

The idea that shows up is **Occam's razor**, which says that we should give preference to the simplest candidate hypothesis in the above case. Philosophically, if two theories explain some evidence equally well, then we should prefer the simpler one.

Example 5.2

Say two polynomials f, g fit our data equally well, and $\deg(f) = 3$ while $\deg(g) = 5$. Then Occam's razor says we should prefer f, as it's the simpler explanation. Perhaps more realistically, say polynomials of degree 3 and higher (including polynomials of arbitrarily high degree) can fit our data. Then we should select a polynomial of degree 3.

§5.3 Structural risk minimization

Occam's razor is nice, but it only applies in the case of perfect ties in empirical risk. If we have some prejudice against polynomials of high degree (for instance), then really we should bake that into our loss function. In particular, rather than minimizing $\frac{1}{n}\sum(h(x_i)-y_i)^2$, it may be more sensible to minimize $\frac{1}{n}\sum(h(x_i)-y_i)^2 + c(h)$, where $c(h)$ is a measure of the 'complexity' of h.

In this way, we're constantly expressing our preference for simpler hypotheses. The paradigm of minimizing this new quantity (with the $c(h)$ term) is known as **structural risk minimization**. Note that it's just the analogue of empirical risk minimization in the case where we punish complicated hypotheses using $c(h)$.

§5.4 Stability

One idea to be aware of is a learner's **stability**, i.e. its sensitivity to the training data. A highly unstable learner can output a considerably different hypothesis when a single point is added or removed from its training data, for instance. A stable learner, on the other hand, won't change its predictions all that much given a small change in its training data.

Stability and generalization tend to go hand in hand, i.e. unstable learners tend to overfit/generalize poorly, while stable learners have a better shot at generalizing well. So the stability idea gives us another form of bias when breaking ties between hypotheses. This gives rise to **Tikhonov regularization**, which – in the case of polynomials – tells us to favor polynomials with low weights (i.e. coefficients). This is really just a particular instance of structural risk minimization, in which the complexity function $c(-)$ equals (for instance) the sum of a polynomial's squared coefficients.

§5.5 Regression

Ordinary linear regression is just an ERM (empirical risk minimization) learner over the hypothesis class of linear functions. By introducing the Tikhonov regularization term (i.e. adding the sum of squared weights to the empirical risk), you arrive at what's called **ridge regression**. If you used the absolute sum of weights, rather than squared sum, in the complexity penalization term, you'd get **lasso regression**. By adding both of those regularization functions, you get **elastic net regression**.

§6 Thursday, September 16

Couple key topics to cover today:

- Curse of dimensionality
- Blessing of dimensionality
- Principal component analysis (PCA)

After today's lecture, we'll have covered everything that will be tested on the midterm. Let's go.

§6.1 Curse (and blessing!) of dimensionality

When we refer to **dimensionality** today, we'll mean the dimension of the space in which our features live. It's importance to distinguish this from the size (number of observations) in our dataset. For instance, if a feature in our dataset is a tuple of 3 numbers, then the dimensionality is 3, regarldess of whether our dataset has size 3, 10, or 10 million.

So why can dimensionality be a curse? At first glance, it might seem like more feature dimensions are always better – after all, features are the ingredients we can use to predict labels, meaning richer/higher-dimensional features should give us a better shot at guessing labels. One reason why dimensionality can be a curse is that high-dimensional spaces like $\mathbb{R}^n$ (for large n) can have strange geometric properties.

Example 6.1

The volume of a unit sphere in $\mathbb{R}^n$ decreases as n increases. In particular, as you go into higher dimensions, the unit sphere occupies an arbitrarily small percentage of the unit cube.

So geometric intuition tends to fail in high dimensions, and you need to be careful when analyzing your data. On the other hand, dimensionality can have its benefits. For instance, it's easier to separate collections of points using planes in higher-dimensional space than in lower-dimensional space. This is formalized in part by Cover's theorem.

Next we'll talk about PCA, but before that, a lemma.

Lemma 6.2 (Johnson-Lindenstrauss)

For any $\epsilon \in (0,1)$, dataset X of m points in $\mathbb{R}^N$, and number $n > 8\log(m)/e^2$, there exists a linear map $f : \mathbb{R}^N \to \mathbb{R}^n$ with the following property:

$$(1-\epsilon)\|u-v\|^2 \leq \|f(u)-f(v)\|^2 \leq (1+\epsilon)\|u-v\|^2 \text{ for all } u, v \in X.$$

In English, the lemma tells us that data points can always be mapped linearly to high-dimensional space in a way that *almost* preserves distances.

§6.2 Principal Component Analysis (PCA)

PCA is a famous tool in **dimensionality reduction**, meaning it compresses the information of high-dimensional features into a lower-dimensional form. We're not going to go deep into the math of it right now, but the idea is that it picks a few orthogonal vectors in the feature space that capture the most variance/information in the features. Those vectors are known as the **principal components**, and they describe as much of the dataset's variance as possible.

When principal components start explaining very little of the data's variance, that's a sign that you've extracted most of the information from your data and don't need any more of its dimensions.

§7 Tuesday, September 21

§7.1 From shallow to deep learning

We're going to transition to deep learning from the kinds of method's we've discussed so far, which you might call shallow machine learning. In particular, the learning processes we've considered have consisted of two primary steps:

1. Featurization: turning some objects into a representation that computers can handle (e.g., SMILES for molecules). There are several kinds of features you can work with:
   - OD Features (e.g., for molecules, number of atoms of each type, number of bonds of each type, etc.)
   - 1D Features (e.g., number of rings of some type (length, aromaticity))
   - 2D Features (e.g. Wiener index, Zagreb index, adjacency matrix)
   - 3D Features (e.g. surface area, moment of inertia, geometrical index, gravitational index, MORSE).
   - 'Bespoke features' (e.g. Lipinski's rule of 5, FilterItLogS).
   - Electronic features: electronic properties of molecules can be obtained as perturbations of external electric/magnetic field, nuclear/electron spin, and geometry. (e.g., Dipole moment (1st derivative for electric field perturbation), Harmonic vibrational frequencies (2nd derivative for geometric perturbation)).
   - Thermodynamic features: can be estimated using the Boltzmann distribution (reaction rates, protein-ligand affinity, solubility).
2. Use an explicit learning algorithm (e.g., ERM over a hypothesis class).

An issue with featurization is that it can destroy information about the object you began with – for instance, structural isomers can be indistinguishable under 0D and 1D featurizations. In addition, as features get more sophisticated, they can get more powerful but also demand more domain expertise, become increasingly expensive/approximate, etc.

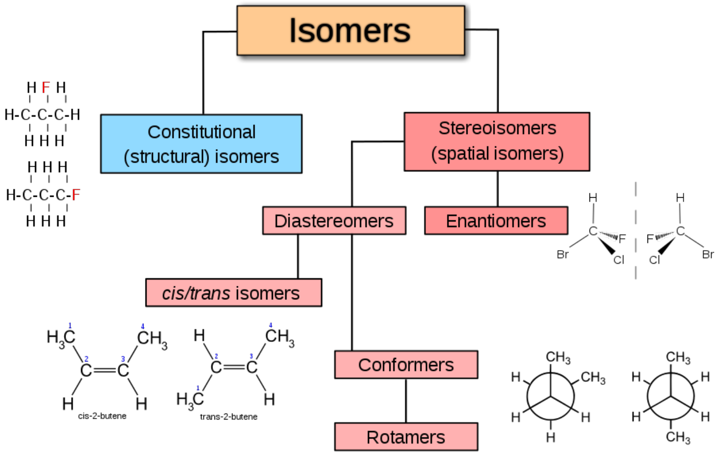


Figure 1: Isomers for a given molecular formula, from here.

Remark 7.1. The Born-Oppenheimer **potential energy surface** (PES) calculates a molecule's energy as a function of its state/geometry. Often in chemistry one wants to find the geometry that minimizes a molecule's energy. This amounts to locating a minimum of the PES, but an issue is that there may be several local minima which are not global minima.

The alternative to what we've been calling 'shallow learning' is a modern method known as **deep learning**. Informally, it can take away lots of the trouble of featurization/needing domain-specific knowledge when performing machine learning. Instead, deep learning can use large amounts of raw data without any guidance to then construct a model which performs remarkably well. And, in fact, traditional models (e.g. physics-based or chemistry-based) can sometimes be used to help generate the large amounts of data required by these deep learners. In setting like video game playing, you can even have the model play against itself (at high speeds) in order to generate data and learn from experience!

§8 Midterm Review

Quick administrative note: we're moving the midterm back to next Thursday (Sept 30), and next Tuesday will instead be a linear algebra review.

§8.1 Useful coding tools

Recall that if `A` is a tensor/matrix, then you can get it's shape using `A.shape` and display `A` in a Jupyter notebook by simply writing `A` in its own cell.

Indexing and slicing are crucial tools when operating with arrays. For instance, `A[4,3]` will fetch the value in the 4th row and 3rd column of `A`, while `A[:,3]` will fetch the entirety of the 3rd column (`:` is similar to 'give me everything'). To compress all of `A` to a single number, you can write `A.sum()` to get the sum of all of its entries. Alternatively, to get the sums over each of its rows, you can write `A.sum(axis=1)`.

Remark 8.1. Professor Tristan is sharing a Jupyter notebook with all of these coding examples, as well as links to pages describing many of the most important tools that will be covered on the midterm.

Quick overview of some key topics we'll expect you to know about on Thursday (not exhaustive!):

- Pre-processing (e.g. `PolynomialFeatures`)
- Model validation and selection
  - Expect questions about this!
- Models (e.g. `DummyRegressor`, linear regression (including Ridge))
- Plotting (e.g. `plot, scatter`)
- Numpy (e.g. creating tensors, indexing, array manipulation, mathematical functions)
- Definitions of key terms (check the Jupyter notebook for a long list)

§8.2 Key concepts

There are several central concepts we've been stressing over the past couple weeks:

- Validation: in order to get a real sense of the efficacy of your model, you need testing data that your model was not trained on/has not seen.
- Now say you're going to have several models competing. Then you don't want to test all of your 20 models on the testing data because it's a recipe for overfitting (i.e., picking the model that performs well on the testing set by pure luck, and whose performance you overestimate).

  So you need a training set to train all your models, a **validation set** to test all of them and pick your favorite, and then a testing set to get a fair assessment of that model's accuracy.
- There's an even higher level to this idea, which is cross validation. You split your training data into k folds, and then use the ith fold as the testing data (and everything else as the training data) for all $1 \le i \le k$. So this is like a classic training/testing split you perform k many times, with training data that varies over your whole data set (`GridSearchCV` helps take care of all this work). Then you can test your final model on the testing data you originally set aside, which you've never touched.

§9 Tuesday, October 5

The content of the class is going to change quite a bit now. So far we've been thinking about data science and the big picture of machine learning. Now we'll get deeper into the details of training learners. Understanding the training process – beginning with linear regression – will be crucial as we get to fancier methods like neural networks.

§9.1 Arg maxes

First a quick intro on arg mins and arg maxes, which will be useful today: for a function $f\colon X \to \mathbb{R}$, **arg min**$_x$ $f(x)$ denotes the set of inputs $x \in X$ attaining the minimal value of $f(x)$ over all of X. Note that in general, arg mins may be empty (e.g., for the identity function on $(0,1) \subseteq \mathbb{R}$) or may be a set of size far greater than one (e.g., for a constant function $x \mapsto 3$). We will usually consider – or assume – functions with non-empty arg mins. An analogous definition and remark holds for **arg max**es.

Lemma 9.1

Let X be a set and $f\colon X \to \mathbb{R}$ a function. Then the following hold:

1. $\arg\min_x f = -\arg\min(-f)$, where $(-f)\colon X \to \mathbb{R}$ is defined by $x \mapsto -(f(x))$.
2. $\arg\min_x (cf) = \arg\min_x f$ for all $c > 0$, and $\arg\min_x (f + c) = \arg\min_x f$ for all $c \in \mathbb{R}$.
3. $\arg\min_x f = \arg\min_x (\log \circ f)$ and $\arg\max_x f = \arg\max_x (\log \circ f)$, where $\log \circ f\colon X \to \mathbb{R}$ is defined by $x \mapsto \log(f(x))$.

§9.2 Linear regression, more seriously

Let's recall the basic setup. There's an unknown function $f : \mathbb{R}^n \to \mathbb{R}$ that's actually labeling the data (i.e. some feature $x \in \mathbb{R}^n$ arises in nature and it carries the label $f(x) \in \mathbb{R}$). If we're in the setting of linear regression, then our hypothesis class consists of (affine) linear functions, i.e.,

$$\mathcal{H} = \{h(x) = w_0 + \sum_{i=1}^{n} w_i x_i \mid w_i \in \mathbb{R}\} = \{h(x) = \langle w, x_+ \rangle \mid w \in \mathbb{R}^{n+1}\}$$

where x_+ is just x with a 1 tacked onto the end (which allows us to capture the translation term w_0). So let $X \in \mathbb{R}^{k \times n}$ be a collection of k randomly chosen features, and $Y \in \mathbb{R}^k$ be defined by $Y_i = f(X_i)$. In particular, X and Y just define the features and labels in our sample, respectively.

Here's a problem: find the value of w that minimizes the empirical risk with respect to X and Y. Ideally, we'd like to exhibit weights w with $Xw = Y$, i.e., weights that perfectly interpolate on our sample points.

Example 9.2

Say $X = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$ and $Y = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$. Then $\mathcal{H} = \{h(x) = ax + b \mid a, b \in \mathbb{R}\}$, and we actually can find a hypothesis that perfectly interpolates on these points. In particular, that's $h_{\mathrm{ERM}}(x) = \frac{1}{2} + \frac{1}{2}x$. Note the importance of that translation term a. Geometrically, all that's happening here is that any set of two points in $\mathbb{R}^2$ can be connected by a line (not necessarily passing through the origin). For larger sets of points, this is usually not possible, and you may be obligated to incur some training error (i.e., fail to interpolate).

There's an important modification to this setting that takes the fuzziness of the real world into account – that is, incorporate noise into the labels. So we'll have $X \in \mathbb{R}^{k \times n}$ and $Y \in \mathbb{R}^k$ where the X_i are still drawn from distribution (perhaps normal), but $Y_i = f(X_i) + \epsilon_i$

where the ϵ_i are i.i.d. $\mathcal{N}(0, \sigma^2)$. In words, there's some uncertainty on the labels that get observed with a feature; the world is no longer perfectly deterministic.

Now how do we select hypotheses from samples? Empirical risk minimization doesn't really make sense anymore, as it doesn't take the randomness on labels into account. It's instead more sensible to perform **maximum likelihood estimation**. That is, what is the choice of labeling function h that would maximize the likelihood of observing the given data? Try to think a bit about why this is a more clever way to test a hypothesis' compatibility with the data than ERM in this setting.

Formally, the maximum likelihood estimator for observations X and Y with labeling function $\langle w, X_i \rangle$ and variance σ^2 on noise terms ϵ_i is:

$$\begin{aligned}
\arg\max_w P(Y \mid X, w, \sigma^2) &= \arg\max_w \prod_{i=1}^{k} \mathcal{N}(Y_i \mid X, w, \sigma^2) \\
&= \arg\max_w \prod_{i=1}^{k} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}((Y_i - \langle w, X_i \rangle)/\sigma)^2} \\
&= \arg\max_w \log\left(\frac{1}{\sigma\sqrt{2\pi}}\right) - \frac{1}{2}\left(\frac{Y_i - \langle w, X_i \rangle}{\sigma}\right)^2 \\
&= \arg\min_w \sum_{i=1}^{k} \left(Y_i - \langle w, X_i \rangle\right)^2
\end{aligned}$$

Corollary 9.3

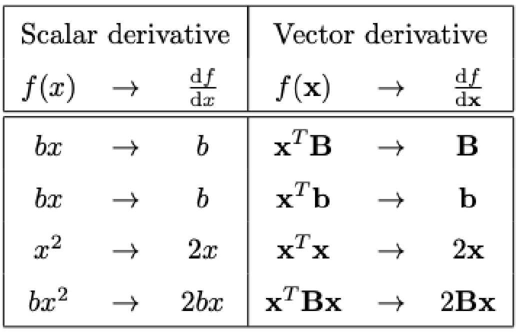
Doing ERM with squared loss on a deterministic model is the same as doing MLE on a model with Gaussian noise! Likewise, attaching a prior distribution to the weights w and calculating the MLE estimator recovers ridge regression. In fact, all the empirical risk minimizers in (deterministic) linear regression have a probabilistic counterpart!

Furthermore, note that the term $\sum_{i=1}^{k} \left(Y_i - \langle w, X_i \rangle\right)^2$ is a polynomial of degree 2 in the variable w (recall that X and Y are the known constants corresponding to our sample points). Since it's of degree 2, its global extremum is guaranteed to happen at the point with derivative 0. Note also that $\sum_{i=1}^{k} \left(Y_i - \langle w, X_i \rangle\right)^2 = ||Xw - Y||^2$. So we have:

$$\begin{aligned}
\frac{\partial}{\partial w}||Xw - Y||^2 &= \frac{\partial}{\partial w}(Xw - Y)^T(Xw - Y) \\
&= \frac{\partial}{\partial w}(w^T X^T - Y^T)(Xw - Y) \\
&= \frac{\partial}{\partial w} w^T X^T X w - Y^T X w - w^T X^T Y + Y^T Y \\
&= \frac{\partial}{\partial w} w^T X^T X w - 2\frac{\partial}{\partial w} w^T X^T Y \\
&= 2X^T X w - 2X^T Y = 0.
\end{aligned}$$

This amounts precisely to $\boxed{X^T X w = X^T Y}$, which is known as the **normal equation**. We'll spend lots of time talking about how to solve this equation. Methods include LU decomposition, QR decomposition, Cholesky decomposition, and SVD (we won't talk very much about the first two).

Notably, the previous derivation made use of a few results in matrix calculus that you should be aware of.



Scalar derivative			Vector derivative		
$f(x)$	→	$\frac{df}{dx}$	$f(\mathbf{x})$	→	$\frac{df}{d\mathbf{x}}$
bx	→	b	$\mathbf{x}^T\mathbf{B}$	→	$\mathbf{B}$
bx	→	b	$\mathbf{x}^T\mathbf{b}$	→	$\mathbf{b}$
x^2	→	$2x$	$\mathbf{x}^T\mathbf{x}$	→	$2\mathbf{x}$
bx^2	→	$2bx$	$\mathbf{x}^T\mathbf{B}\mathbf{x}$	→	$2\mathbf{B}\mathbf{x}$

Figure 2: Common vector derivatives, from here.

§9.3 Multivariate normal distribution

A quick detour on the multivariate normal distribution, which we've touched on before. The **multivariate normal distribution** (MVN) is the higher-dimensional analogue of the normal distribution. In particular, a k-dimensional MVN distribution is determined by k many mean parameters and k^2 covariance parameters (one mean for each dimension and one covariance for each pair of dimensions, satisfying symmetry rules).

Definition 9.4 — A k-dimensional random vector $\mathbf{X} = (X_1, \ldots, X_k)$ following a multivariate normal distribution is written as follows:

$$\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}).$$

In particular, $\boldsymbol{\mu} = \mathbb{E}(\boldsymbol{X}) = \big(\mathbb{E}(X_1), \ldots, \mathbb{E}(X_k)\big)^T$ and $\Sigma_{i,j} = \mathrm{Cov}(X_i, X_j)$. Its density is

$$f_{\mathbf{X}}(x_1, \ldots, x_k) = \frac{\exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right)}{\sqrt{(2\pi)^k|\boldsymbol{\Sigma}|}}.$$

Notably, a 1-dimensional MVN is simply a normal distribution (with single mean and covariance parameters), and marginalizing/conditioning an MVN produces another MVN. More explicitly, if $(X_1, \ldots, X_k)$ follows a k-dimensional MVN, then $(X_1, \ldots, X_{k-1})$ follows a $(k-1)$-dimensional MVN, and $\big((X_1, \ldots, X_k) \mid X_k = c\big)$ does as well.

§9.4 Cholesky decomposition

The statement at the center of **Cholesky decomposition** is the following: if X is a symmetric, square, positive-definite matrix, then $X = LL^T$ for some lower-diagonal matrix L. The importance of the decomposition is that it makes it much easier to pre-image under X, i.e. to solve equations of the form $Xa = b$, where b is known and a is not.

To see the relevance for the normal equation, let's again say we have $X^TXw = X^TY$ where X and Y are known. Suppose furthermore that we have the Cholesky decomposition for X^TX, i.e., $X^TX = LL^T$. Then the normal equation reduces to

$$LL^Tw = X^TY$$

which can be solved efficiently by leveraging lower-diagonality of L via two forward-substitution. More explicitly,

$$L\alpha = X^T Y$$
$$L^T w = \alpha.$$

§9.5 SVD

Here's the statement of **SVD**, or singular value decomposition: if $X \in \mathbb{R}^{a\times b}$, then it admits a decomposition as $X = U\Sigma V^T$ where $U \in \mathbb{R}^{a\times a}$ is orthogonal, $V \in \mathbb{R}^{b\times b}$ is orthogonal, and $\Sigma \in \mathbb{R}^{a\times b}$ is diagonal (with the 'singular values' of X on its diagonal). Informally, all linear transformations consist of a rotation, followed by a projection with scaling in each dimension, followed by another rotation.[1]

§10 Thursday, October 7

§10.1 Solving the normal equation with . . .

Quick recap of last time: we started studying how to calculate ERM hypotheses for linear regression, meaning we want $\arg\min_w ||Xw - Y||^2$. We also saw, remarkably, that this is the same thing as finding the MLE weights w for a linear model with Gaussian noise. It can sometimes be useful to be move between these interpretations of the same process: ERM in a deterministic world and MLE in a stochastic world.

We also saw that setting the derivative of $||Xw - Y||^2$ with respect to w equal to 0 recovers the normal equation, which is $X^T Xw = X^T Y$. We saw that the normal equation can be solved using linear algebra, along with a few tricks to speed things up.

1. Cholesky decomposition: $X = LL^T$, where L is lower-diagonal.

2. SVD: $X = U\Sigma V^T$, where U, V are orthogonal and Σ is diagonal. Orthogonal matrices can be thought of as changes of bases, so SVD really says that all linear transformations consist of changing basis, projecting/scaling, and changing basis one more time. In other words, all linear transformations are just scalings in each dimension, up to changes of basis.

We've talked about Cholesky, so now we'll dive into using SVD to solve the normal equation.

[1] Really, orthogonal matrices correspond to more general functions than rotations (e.g., reflections), but that's the idea.

§10.1.1 SVD

So say we have the normal equation in the setting where X has gone through SVD. Then we have:

$$\begin{aligned}
X^TXw &= X^TY \\
(U\Sigma V^T)^T(U\Sigma V^T)w &= (U\Sigma V^T)^TY \\
V\Sigma^TU^TU\Sigma V^Tw &= V\Sigma^TU^TY \\
V\Sigma^T\Sigma V^Tw &= V\Sigma^TU^TY \\
\Sigma^T\Sigma V^Tw &= \Sigma^TU^TY \\
\Sigma V^Tw &= U^TY \\
V^Tw &= \Sigma^\dagger U^TY \\
w &= V\Sigma^\dagger U^TY.
\end{aligned}$$

Along the way, we made use of the fact that orthonormal matrices are invertible (and their inverses are their transposes), and that Σ isn't quite invertible but has a Moore-Penrose pseudoinverse $\Sigma^\dagger$.[2]

§10.1.2 Root finding (via Newton's method)

Newton's method is a powerful tool for finding zeroes (or roots) of polynomials. It works by randomly picking a point x_0 and computing its image under the polynomial p. Then, in general, it takes the derivative of p at x_n and takes x_{n+1} to be the point at which that derivative has a zero. We're not going to go into the mathematical details here, but this algorithm usually tends to a zero of p, which is what we're after.

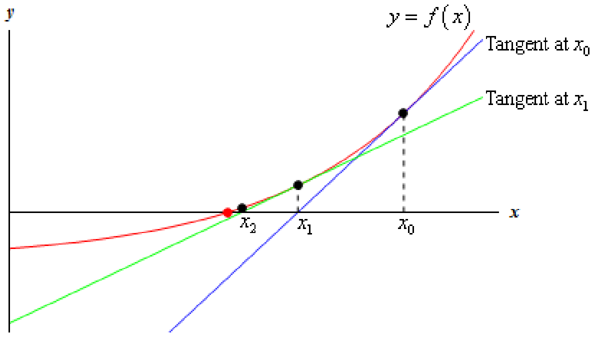


Figure 3: Newton's method, from Paul's online math notes.

This is a neat method with good foundations, but it's not used all too often in practice, since its complexity suffers in high dimensions. It's useful to know, though, and it can work well in low-dimensional settings like the ones we work in. To solve the normal equation, we just apply it to the derivative of $||Xw - Y||^2$.

§10.1.3 Gradient descent

gradient descent is a legendary technique in optimization – the idea is to evaluate the derivative of the function at a given point, move your point away from the derivative (or

[2] There may be a typo here – be careful.

toward it, if maximizing), and repeat. Formally, given a function f, you would define your estimates x_i via:

$$\begin{cases} x_0 = v, \\ x_{m+1} = x_m + f'(x_m). \end{cases}$$

This is fairly simple and intuitive, but it also has a pretty big pitfall – it will very often fail to converge, especially when derivatives are large (and the steps you're taking are huge). To remedy this, you usually put a coefficient λ on $f'(x_m)$:

$$\begin{cases} x_0 = v, \\ x_{m+1} = x_m + \lambda f'(x_m). \end{cases}$$

In higher dimensions, you do the analogous thing using the gradient ∇f of f.

§10.1.4 Stochastic gradient descent

stochastic gradient descent (SGD) is probably the most important algorithm of the 21st century, and lies right at the center of modern machine learning (i.e., deep learning). The idea behind SGD is to randomly pick some of your data points and take your derivative with respect to them, rather than using all of your data (which is much more computationally expensive, of course).

So, if you're minimizing $||Xw - Y||^2$, then in SGD X and Y really won't be fixed; at any given step, you'll be calculating the gradient using only some subset of the data in X and Y. This actually works a lot of the time (with the right parameters), and it's much cheaper than ordinary gradient descent.

So this is the fundamental tradeoff being faced: GD takes very precise steps at high cost, while SGD takes noisier steps at lower expense.

§11 Thursday, October 14

Last time we talked about implementing empirical risk minimization in the setting of linear regression. That is, how can we find the coefficients (or weights) in a linear function that produce the best fit for the data we've seen? We saw that minimizing that empirical risk is equivalent to solving the normal equation, and saw several methods for doing this concretely.

In that whole setup, the only function we had access to were of the form $\{f(x) = \langle w, x\rangle \mid w \in \mathbb{R}^n\}$, which is not so sophisticated. There are definitely settings in which we'll want to model nonlinearities, and the feature engineering might not be so obvious if we're restricted to linear hypotheses. So we'll be looking at more powerful classes of hypotheses, via **neural networks**. These maps can be so complex that most of our previous optimization techniques won't apply, aside from (S)GD. Those require derivatives, though, which brings us to the topic of automatic differentiation.

§11.1 Automatic differentiation

An important package for machine learning is `torch`, which has the functionality for automatic differentiation built in. **automatic differentation** is just the process of taking in a program for a function and outputting a program for its derivative. As you can imagine, having this kind of tool will be really important for (S)GD, which involves repeatedly calculating gradients. Roughly speaking, this done by breaking up a function into sums and compositions of elementary functions (like multiplication, trig functions, exponentiation, etc.), calculating their derivatives, and using linearity of the derivative along with the chain rule.

Here are some common autodiff mistakes:

1. Calling `grad` on a non-leaf. In particular, you should call `grad` on the variables that are used to build the final expression, not the final expression itself.

2. Calling `backward` multiple times before calling `grad`. This warps the derivative that gets calculated, so be careful. If you're not sure how many times you've called `backward`, you can just set `x.grad = None`, then call `backward` once and use `grad`.

§11.2 Neural networks

At a high level, **neural networks** work by repeatedly passing scaled sums of simple functions through nonlinear **activation functions**. This gives rise to an extremely flexible class of functions that almost make feature engineering, model selection, and clever thinking obsolete. It's hard to overstate how powerful these neural networks can be, and how much deep learning has come to dominate contemporary machine learning.

In addition to being extremely successful, these networks also resemble somewhat the structure of the brain (hence the name) and how it performs visual processing, with neurons connected to each other and the earliest ones working to recognize lines and quadrants.

§12 Tuesday, October 19

§12.1 Neural networks, again

Last time we talked about neural networks. We saw that a central component of their success is automatic differentiation, which allows for relatively easy implementation of (S)GD. We also talked about the role of activation functions (like sigmoid and ReLU), which allow us to bake nonlinearities into the networks.[3]

By tradition, layers in neural networks that are neither input nor output layers – and are simply used for intermediate manipulations – are referred to as **hidden layers**.

[3] If the activation functions are linear, then the entire neural network will actually end up encoding a linear function. This is a consequence of the fact that linear functions are closed under scaling, addition, and composition (i.e., doing any of those things to a set of linear functions will give you another linear function).

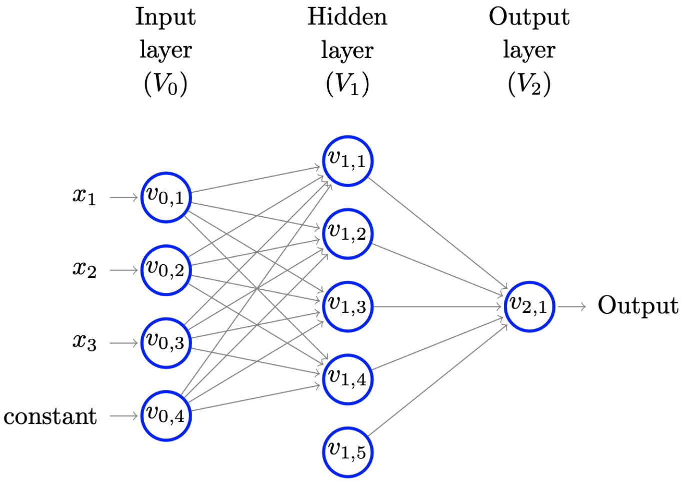


Figure 4: Neural network, from *Understanding Machine Learning: From theory to algorithms.*

An extremely important point is that neural networks are not understood all that well. It can be shown that they have extraordinary flexibility, i.e., can be used to approximate almost any function arbitrarily well. As a result, they fly somewhat in the face of the usual bias-variance tradeoff, as they often generalize well when trained with (S)GD despite representing an extremely complex hypothesis class.

It's worth noting that there are only a couple important activation functions:

1. sigmoid: it used to be very popular, but not so much anymore – we've found better functions.
2. tanh: the hyperbolic tangent function.
3. ReLU: it's just $x \mapsto \max(0, x)$, but it's become the most popular activation function, and it's very effective.

Recall the structure of gradient descent; we select initial parameters for our model, compute the gradient of the empirical risk with respect to those parameters, and move to the parameters in the direction of decreasing gradient. The geometric intuition is clear: we pick a point, calculate the slope, and follow the slope downwards (only slightly) in the hopes of reaching the minimum. Stochastic gradient descent is identical but takes the gradient of the empirical risk calculated on some randomly chosen subset of our data, rather than the entire dataset (the idea being that our dataset might just be too huge).

There are lots of degrees of freedom here, though; when to stop gradient descent, how much to move in the direction of decreasing gradient, where to initialize your parameters, etc. Training these networks effectively can be extremely difficult, and both an art and a science. There are optimizers that are slightly more clever than (S)GD, though, which – for instance – try to use information about the second derivative. Perhaps the most famous is Adam, and others include Adadelta, RMSProp, etc.

§13 Thursday, October 21

Homework 8 is being built right now, and it's not going to be easy – you'll need to find your own dataset, clean it and preprocess it, etc. It's *really* important to start early on this one, and you may want to assemble into a larger group if you're in a small group right now.

Another logistical point: the final may end up counting for less than it's currently stated to count for (30%), with more weight going toward some of the homeworks, which have been pretty hard. Everyone will have to vote in an anonymous poll (and unanimously agree on the new grading system) if we're going to change the syllabus, though.

§13.1 Neural networks

So, thanks to autodiff, (S)GD, and `torch`, it's not so hard to play around with neural networks. At a basic level, we've seen everything we need in order to design and train neural networks. But there are other cool and useful facets of `torch` that are worth touching on; they can make a huge difference in the efficiency of training a neural network, for instance.

A couple of points we'll touch on today:

1. Defining neural networks using modules, which are flexible and highly optimized (i.e., efficient). It's also time-saving (and tear-saving) to work at a higher level of abstraction, with built-in functionality that pros have built.

2. Using pre-defined optimizers.

3. Datasets of DataLoaders. These can have huge performance benefits from clever work under the hood (i.e., bringing data close to the CPU before you need to use it).

4. Tips for successfully training neural networks.

When it comes to optimizers, a rule of thumb is to use Adam when you're not quite sure what to do. It's not particularly well-understood, but it's usually effective. Generally speaking, it's worth reiterating that training neural networks is really hard – there are people whose full-time job is to train networks, it's a very active area of research, etc. The upside to this is that if you practice training them and build intuition for this, you'll have an extremely valuable skill.

As we've mentioned, separating the data into training, validation, and testing is crucial. So you'll want to have a DataLoader for each one. One useful tip is to use `tqdm` when running `for` loops, as they give you a progress bar for `for` loops and an estimated time until completion. The syntax is

```
for i in tqdm(range(n)):
    do_thing(i)
```

Another useful package is `wandb`, which helps you keep track of analytics while performing ML. For instance, you can make sure that you train until you get a training loss of 0 (i.e., perfectly interpolate on the sample). In fact, this is often a desirable thing to do. Notably, that flies entirely in the face of the bias-variance tradeoff that we covered only a few weeks ago. Unfortunately, this is just one of the central mysteries of deep learning – neural networks often interpolate on sample data yet generalize extremely well.[4] Whoever figures this out will probably be a legend in the field.

That's all for today: next time, we'll talk about neural networks that take in graphs describing molecular systems.

[4] You can look into 'benign overfitting' or 'double descent' to learn more.

§14 Tuesday, October 26

§14.1 Vanishing and exploding gradients

A quick detour on a common mistake from the last homework. A couple weeks ago we talked about how autodiff uses the chain rule to calculate the derivatives of programs. In particular, this results in the multiplication of lots of derivatives, which can tend to result in the vanishing/exploding of the overall product (especially when you're reusing weights in your network). For instance, $\frac{1}{2}^n$ very quickly tends to 0 while 2^n quickly explodes.

A corollary of this is that deeper networks (i.e., with more hidden layers) are generally harder to train, as the n in the previous sentence is larger.

Example 14.1

Facebook sometimes gets around this issue while using very deep networks (e.g., with dozens of hidden layers) by adding connections directly from earlier layers to later layers. This helps avoid the failure of backpropagation for such deep networks.

§14.2 Graph neural networks

Today we'll be talking about **graph neural networks**, which are neural networks that take in graphs as input. They'll come up in this upcoming homework, where the name of the game will be to predict the energy of a molecule's ground state. Furthermore, you'll need to create your own model on this homework, rather than using black-box technologies from popular libraries. Important packages for graph neural networks are `DGL`, which stands for Deep Graph Library, and `dgllife`.

To give some context here, previously we've been ignoring the graph structure of molecular systems, which has made prediction fairly difficult. In particular, even with the knowledge of a system's atoms and bonds, lots of information about the actual geometry of the molecule is being destroyed (e.g., the location of the bonds). So the goal here is to feed an actual graph describing the molecular system to a neural network, which is where these graph neural networks come in.

The way this is actually done involved the notion of **convolution**, which – informally – featurizes a node using information from itself and its neighbors, so that information about relationships in the graph is captured. A similar idea is used in neural networks for image processing, where relationships between nearby pixels matter.

§14.2.1 Some syntax

When you have a graph `g`, `g.adj()` gives you the adjacency matrix of `g` (in sparse matrix form), which contains the information of `g`'s edges. In order to actually add structure to your graph (e.g., labels for edges and vertices), you will need to the `node_featurizer` and `edge_featurizer` arguments in `smiles_to_bigraph()`; more details in the jupyter notebook for today's lecture.

§15 Thursday, October 28

Some people didn't vote for the syllabus change so we had a big discussion about contract law and the European Union – gnarly.

§15.1 Geometry and potential energy

Back to machine learning and chemistry; a couple notions to introduce that will be useful for the remainder of the course (and the final library we'll be introducing). The library we'll look at today is `ase`, which stands for atomic simulation environment. It allows you to work with molecules by considering the positions of their atoms' nuclei in space.

> **Definition 15.1** — The **structure** or **geometry** of a molecule refers to the spatial arrangement of its nuclei.

The structure/geometry of a molecule is so crucial because of the famous saying: structure determines property. In other words, everything we care about is a function of a molecule's geometry.

> **Example 15.2**
>
> The energy of a molecule is a function of its geometry. So are the forces that act on its nuclei and the bonds that it has.

Another crucial concept in chemistry is that of the **potential energy surface** (PES), which is the graph of a molecule's geometry with its potential energy. Note for instance, that the global minimum on the PES corresponds to the ground state of the molecule. Speaking more generally, PES's contain an enormous amount of information about a molecular system, and can tell us about its reactions, transition structures, conformers, etc.

§15.2 Homework

The goal of this homework is to predict the ground state energy of a molecule. In particular, you'll be taking in the smiles representation of a molecule and outputting (a prediction of) its energy in the ground state. So you'll have to put together a dataset whose elements take the form (smiles string, ground state energy).[5] Once you've put together this dataset, you'll be able to use something like a graph neural network to predict the energies (possibly using `smiles_to_bigraph`).

§15.3 ASE package

The `ase` package will be really useful here. If `mol` is of type `ase.Atoms`, then you can run `mol.get_positions()` to see the positions of its atoms. It'll be key to give `mol` a calculator attribute, by writing something like

```
mol.calc = MOPAC(label='TMP', task='1SCF UHF BONDS GRADS').
```

Once `mol` has the calculator attribute, you can write `mol.get_potential_energy()` to get the potential energy of its current geometry.[6] It's that simple! We'll also get an output file from MOPAC with much more detail on the chemical properties of `mol`, like the gradients of its PES.

[5] Strictly speaking, we're only asking you to predict the energy at one of the local minima on the PES, not the global minimum.

[6] If we had given MOPAC a different `task` argument when setting it as `mol`'s calculator, we could have gotten another energy, like one if its ground state energies.

Warning. Be careful with units here; keep in mind that MOPAC outputs energies in units of eV (electron volts).

In particular, MOPAC will give you the forces acting on the molecules, which equal the negative derivatives of the PES. The punchline is that you can use this to perform S(GD) with your neural network!

§16 Midterm 2 review

§16.1 Linear regression

The setup of linear regression $\mathbb{R}^n \to \mathbb{R}$ is that $f(x) = \langle w, x \rangle$ for some (unknown) $w \in \mathbb{R}^n$. We saw that one algorithm for learning linear regression is ERM where the loss function is squared error, i.e., to output $h(x) = \langle w_0, x \rangle$ where $w_0 = \arg\min_w ||Xw - Y||^2$. We also saw that calculating $\arg\min_w ||Xw - Y||^2$ could be done by setting its derivative with respect to w equal to 0, i.e. $\frac{\partial}{\partial w} ||Xw - Y||^2$.

We won't go through the algebra again, but we derived earlier that setting that derivative equal to zero recovered the normal equation, i.e.,

$$\frac{\partial}{\partial w} ||Xw - Y||^2 = 2X^T Xw - 2X^T Y.$$

And thus we want to pick w satisfying $X^T Xw = X^T Y$.

§16.1.1 Cholesky and SVD

To actually solve this, we talked about the Cholesky and SVD decompositions of matrices. For instance, when X and Y are fixed samples, the Cholesky decomposition of $X^T X$ takes the form $X^T X = LL^T$ for a lower-diagonal matrix L. Then the normal equation reduces to

$$LL^T w = X^T Y$$

which can be solved efficiently by leveraging lower-diagonality of L via two forward-substitution. More explicitly,

$$\begin{aligned} L\alpha &= X^T Y \\ L^T w &= \alpha. \end{aligned}$$

§16.1.2 (Stochastic) gradient descent

We saw that you can also minimize a function using (stochastic) gradient descent, which is often much cheaper than using an exact solver like Cholesky or SVD. The big idea of gradient descent is to calculate the derivative of your function at a point x_i (using your sample points X and Y), move your point away from the derivative to get x_{i+1}, and repeat. Stochastic gradient descent is a close analogue where you only use a subset of the data in X and Y to calculate your gradient at each step.

This only makes use of the first derivative of a function; more sophisticated solvers like Adam and RMSProp actually use information about the function's higher derivatives.

§16.1.3 Stochastic linear regression

There's another way of thinking about linear regression, where you actually have noise on your labels, i.e. $f(x) = \langle w, x \rangle + \epsilon_x$ for i.i.d. noise terms $\epsilon_x \sim \mathcal{N}(0,1)$. Now we'd like to pick a candidate w that maximizes the likelihood of having observed the sample X, Y (in statistical language, a maximum likelihood estimator). More explicitly, we want

$$\arg\max_w P(Y \mid X, w) = \arg\max_w \prod_{i=1}^{n} \mathcal{N}(Y_i \mid X, w).$$

Remarkably, we saw that this recovers the exact same arg min as performing ERM with deterministic linear regression and squared loss.

§16.2 Q & A

Question: Do any of these techniques come up in the packages we use?
Answer: Definitely! Packages like `torch` (and most of the fancy machine learning technologies you hear about in the news) use SGD all the time. Since it found its way into machine learning around a decade ago, it's been used almost everywhere.

Question: Why do we use noise terms?
Answer: We do this in practical settings where we think the true data generating process is of the form $\langle w, x \rangle$, e.g. $y = \langle w, x \rangle$ *but* we have noisy observations $\tilde{y} = y + \epsilon_x$ due to imprecision in our actual measuring devices (like telescopes, or something). Alternatively, there could be some additional factors beyond x that very slightly affect y, but we ignore them in favor of a simpler, more tractable model that's still useful (e.g., maybe y actually also depends very slightly on the distance between the moon and the Earth at the time of measurement, but we aren't really going to worry about thousands of such factors).

Question: Any multiple choice questions this time?
Answer: No, no multiple choice. This midterm will be more mathematical than the first one (though it'll also have a few programming problems).

Question: What's the breakdown of the test?
Answer: Here's the full breakdown:

- 3 coding questions, worth 4 points each
- 3 'math' questions, worth 2 points each
- 2 chemistry/machine learning questions, worth 1 point each

§16.3 Homework 6

Let's go over Homework 6. It dealt with automatic differentation, which concerns how computers calculate gradients for programs, and is at the heart of techniques like (S)GD in practice. The actual math isn't too fancy: a program is a composition of lots of very simple functions, and you can use the chain rule to calculate its derivative, piece by piece.

In this homework, you optimized once by calculating the derivative explicitly by hand, and another time using autodiff. In each case, the idea was just to iterate a couple thousand times, calculating the derivative each time and moving your weights very slightly away from the derivative (i.e., with a coefficient of something like `1e-7`).

§17 Tuesday, November 9

§17.1 Midterm post-mortem

Remember to sign up for a final project if you'd prefer it to a final exam. Now a quick remark on the midterm: it ended up being a little harder than expected, but don't be terribly worried about it. The current intention isn't to fail anyone, or even to give out any D's. It really looks like everyone is trying hard on the homework and the class overall, and the plan is to overwhelmingly give A's and B's.

§17.1.1 Question 1

Most of you did really well on this, though there was one common hiccup: the loss function is not just $||Xw - Y||^2$, but $||Xw - Y||^2 + ||w||^2$, so you need to add that second term when calculating your error. Almost everyone made that mistake, but it really does matter for what your algorithm does.

§17.1.2 Question 2

You only needed one linear layer here, nothing terribly fancy.

§17.1.3 Question 3

This one was a bit tricky; it was again important to include the $||w||^2$ term in your mse calculation. This could be done by feeding a tensor purely of ones to your model, which would recover its coefficients.

§17.1.4 Question 4

People did great on this question – nicely done.

§17.1.5 Question 5

Now to the theory: this question asked you to calculate

$$\frac{\partial}{\partial w}||Xw - Y||^2 + ||w||^2.$$

You already know the derivative of the first term with respect to w, so you only need the second one. That comes out to

$$\frac{\partial}{\partial w}||w||^2 = \frac{\partial}{\partial w}w^T w = 2w.$$

Most people got this, as the matrix calculus rules are in the notes.

§17.1.6 Question 6

So the equation you need to solve is this one:

$$2X^T Xw - 2X^T Y + 2w = 0.$$

That just reduces to $(XX^T + I)w = X^T Y$. You can then use Cholesky decomposition on $(XX^T + I)$, and the usual forward substitution trick. Other people used SVD on X, which is doable but much more difficulty as there's an added identity term.

§17.1.7 Question 7

Only one person got this one; it was pretty hard. Recall that there's the connection between least squares $\arg\min_w ||Xw - Y||^2$ and MLE (maximum likelihood estimation) for the probabilistic model $y_i = \langle x_i, w\rangle + \epsilon_i$. In this question, we furthermore imposed a multivariate normal prior distribution on w (with identity covariance matrix). When calculating the likelihood of observing the sample, you get the usual likelihood (without the prior term on w) multiplied by something proportional to $e^{-||w||^2}$. You then take logs, which turns $c \times e^{-||w||^2}$ to $\log(c) - ||w||^2$, where c is the likelihood expression from the usual case.

§17.1.8 Question 8

Few people got this: C_4H_10 has 14 molecules, and each of them are specified by 3 coordinates in space, so the PES is from $\mathbb{R}^{42}$ (geometry of the system) to $\mathbb{R}$ (the energy of the system).

§17.2 Kernel methods

The goal of this class is really to get you to understanding machine learning deeply enough (no pun intended) that you can define your own methods. Learning about kernel methods will be a key step of that; they're fairly advanced topics in learning, and also quite mathematical. We'll get in-depth on this on Thursday, and today we'll discuss some of the high-level ideas.

Traditional machine learning in chemistry takes the following form: data is collected experimentally or through expensive calculations, it's 'featurized' into feature vectors (possibly using quantum chemistry), and off-the-shelf models are used on these features. As we've discussed, this is somewhat limited because you need to cast your data as vectors of real numbers.

The modern approach to using machine learning in chemistry is that the data is collected (possibly even created from scratch, using quantum chemistry software), and then the model input actually matches the structure of the data. Two differences between the modern and traditional camps that we should highlight here:

1. Featurization is replaced with 'feature learning'

2. Vectorization is replaced with 'structured ML', that actually respects the structure of the input in its usual form.

In many cases, this modern approach has proven to work far better than the traditional approach. One somewhat invisible cost to this is that the models are much more complicated, hard to interpret, and sometimes black-box. For this reason, chemists sometimes don't love the modern approach.

There's a third approach, though, in which machine learning and quantum chemistry work in tandem. In this case, there's not any pre-existing data; it's actually collected 'on the fly' to improve a model of the PES. In particular, an ML model serves as a model of the PES. Then methods from quantum chemistry are used to (approximately) calculate the value of the PES at a point p, and the ML model uses the information at p to update its model of the PES. Informally, you explore quantum chemistry to explore the PES and incrementally update your ML model of it.

This can be a powerful technique, but it's worth noting some of the difficulties (and some of the promise) when modeling the PES:

1. The PES cannot be computed in closed form.
2. The PES can have practically unlimited complexity, for increasingly large systems.
3. The PES contains an incredible amount of important information concerning a chemical system, about almost anything we would want to know.

Now recall some of our previous discussion about linear regression: it's applicable not only 'directly', i.e., to model a line in your feature space $\mathcal{X}$, but also applicable with an embedding function on your feature space. More explicitly, you can have a linear model with an embedding function ϕ of the form $f(x) = \langle w, \phi(x) \rangle$. With a map like $\phi(x) = (x, x^2, x^3, x^4)$, this let us fit polynomials in the framework of linear regression.

Neural networks arose as a further generalization of this, where you have something close to a multi-layered linear regression (with these nonlinear activation functions and possibly with an embedding at the start). Kernel methods will arise as yet another form of generalization, which we'll get into on Thursday.

§18 Thursday, November 11

§18.1 Kernel methods

We'll keep moving with kernel methods today; the hope is to give you a sense of where they come from and what they're supposed to do. As we mentioned last time, we'll begin by considering a function of the form $f(x) = \langle w, x \rangle$, with $x, w \in \mathbb{R}^d$. Furthermore, we assume we have some dataset $X \in \mathbb{R}^{N \times d}$ and $Y = f(X) \in \mathbb{R}^N$. Nothing fancy here – this is just linear regression.

As we've mentioned, something you can do here is to actually perform an embedding ϕ on x (perhaps into some higher-dimensional space), and then consider linear regression. Formally, this comes out to $f(x) = \langle w, \phi(x) \rangle$. One example we saw on the homework was $\phi(x) = (x, x^2, x^3, x^4)$, which let us fit degree 4 polynomials in the framework of regression. So we have the dataset X and $Y = \phi(X)$, by which we mean ϕ applied to each row of X. So solving for the w that attains least squares is just

$$\arg\min_w \|\phi(X) - Y\|^2.$$

With the result about the normal equation, that just amounts to solving

$$\phi(X)^T \phi(X) w = \phi(X) Y.$$

We've seen how to solve this with Cholesky/SVD/etc; if the inverse of $\phi(X)^T \phi(X)$ exists[7], then you can trivially solve this with

$$\begin{aligned} w &= \left(\phi(X)^T \phi(X)\right)^{-1} \phi(X) Y \\ &= \phi(X)^T \left(\phi(X)\phi(X)^T\right)^{-1} Y. \end{aligned}$$

Definition 18.1 — The matrix $\phi(X)^T \phi(X)$ is known as the **design matrix**, while $\phi(X)\phi(X)^T$ is the **Gram matrix**.

[7] One way to make sure the inverse exists is to add the identity matrix to this term, which (as you'll remember from the midterm) recovers regularized least squares. One reason why we don't like to do this in practice is that inverting a matrix is numerically unstable.

Lets think about the dimensions of the Gram and design matrices for a moment. The Gram matrix is NxN (where N is the number of data points you have), while the design matrix is ExE (where E is the dimension of the codomain of ϕ, i.e., the dimension of the space where you embedded your features). So, crucially, the size of the Gram matrix does not depend on the size of your ambient embedding space, while the design matrix does not depend on the size of your data set.

Going even deeper, note that the entries of the Gram matrix G are the dot products of our (embedded) feature vectors, i.e., $G_{i,j} = \langle \phi(x_i), \phi(x_j) \rangle$. The entries of the design matrix, are dot products over all of the examples' coordinates in a certain dimension. In short, the Gram matrix has dot products of examples while the design matrix has dot products of coordinates.

You can see now that it would be useful to consider a function $k(a,b) = \langle \phi(a), \phi(b) \rangle$, as it would recover the entries of the Gram matrix (which are used in solving the normal equation!). In particular, if we could compute $k(a,b)$ without explicitly computing $\phi(a)$ and $\phi(b)$, that would be really useful as we move into higher-dimensional spaces (where taking the dot product of $\phi(a)$ and $\phi(b)$ directly would be expensive).

Definition 18.2 — For a set X, a function $k\colon X \times X \to \mathbb{R}$ is referred to as a **kernel**.

So this is where kernel methods come from; it's really just a way of rethinking this normal equation as we bring in these high-dimensional embeddings ϕ. The key observation is just that if $k(a,b)$ were easy to compute (even when $\phi(a)$ and $\phi(b)$ are 10,000-dimensional), that would be really, really useful. The same reasoning would hold anytime your learning algorithm makes use of the inner products of your embedded features, i.e., $\langle \phi(a), \phi(b) \rangle$. Taking advantage of such an easy-to-compute function k is known as the **kernel trick**, and it's really powerful.

Remark 18.3. The kernel trick works for the version of the normal equation that uses the Gram matrix, but not the one that uses the design matrix, as the design matrix has inner products of coordinates, not of entire examples/features.

That was fairly theoretical – let's look at some examples.

Example 18.4

Let's say $x \in \mathbb{R}$ and $\phi(x) = \begin{pmatrix} 1 \\ \sqrt{2}x \\ x^2 \end{pmatrix}$. Then $\langle \phi(x), \phi(y) \rangle = 1 + 2xy + x^2y^2 = (1+xy)^2$. That's easier to compute than actually embedding x and y into higher dimensions and then taking dot products. It's pretty remarkable; there turns out to be an efficient way of computing dot products in high-dimensional space that looks like a much simpler algebraic manipulation (seemingly having nothing to do with dot products).

Conversely, if we had $k(x,y) = (1+xy)^d$, then it would recover $k(x,y) = \langle \phi(x), \phi(y) \rangle$ where $\phi(x)$ is an embedding of x into $(d+1)$-dimensional space with all relevant powers of x, i.e, x^0, x^1, $\dots$, x^d. That's pretty incredible.

The previous example gives rise to a question; when do we know that a kernel function $k\colon X \times X \to \mathbb{R}$ actually recovers the dot product from some embedding, i.e., that $k(x,y) = \langle \phi(x), \phi(y) \rangle$ for some ϕ? We're not going to go into the details here, but this turns out to be a solved problem (Answer: k needs to be symmetric and positive-definite).

Now, let's take this to infinite-dimensions. Say we want $\phi(x)$ to be the infinite-dimensional vector (of length $|\mathbb{R}|$) containing $e^{-(x-r)^2}$ for all $r \in \mathbb{R}$. An $|\mathbb{R}|$-dimensional vector is the same thing as a function out of $\mathbb{R}$, in the same way a 5-dimensional vector v is the same thing as a function f out of $\{1, 2, \ldots, 5\}$, with $v_i = f(i)$. So say ϕ is such an embedding into (uncountably!) infinite-dimensional space. What would its 'dot product' look like? We would actually need to speak of a slightly more general idea, known as an *inner product*, but the idea is very similar, i.e., to measure the similarity between vectors.

A natural candidate in this case would be[8]

$$\langle \phi(x), \phi(y) \rangle = \int_{-\infty}^{\infty} e^{-(x-\mu)^2} e^{-(y-\mu)^2} \mathrm{d}\mu.$$

This turns out to be computable in closed form, as $\sqrt{\frac{\pi}{2}} e^{-\frac{1}{2}(x-y)^2}$. So, we can actually perform infinite-dimensional embeddings with our data! Geometrically, we can also think about this as $\phi(x)$ and $\phi(y)$ being bell curves centered at x and y, respectively. Then $\langle \phi(x), \phi(y) \rangle$ measures the area under the curve of their product, which will be bigger the more their peaks overlap. This is a super meaningful notion of similarity!

§19 Thursday, November 18

§19.1 Predicting energy and force

Last lecture (which isn't covered in these notes) involved further discussion of kernel methods. We talked about the mean function μ and the kernel k, which intuitively calculates a similarity score between its two arguments. The key property was that $k(x, y) = \langle \phi(x), \phi(y) \rangle$ for some embedding ϕ.

We also talked about the Gaussian kernel $k(x, y) = e^{-\|x-y\|^2/2\ell^2}$, where ℓ plays the role of the **length-scale**, which – informally – determines a point's sphere of influence. Now we're going to talk about multi-label learning, in which we're aiming to predict two things rather than one, i.e., both the energy and force on a chemical system.

One thing we could is to just built two separate models: one for predicting energy and another for predicting force. That's not unreasonable, but it misses out on the fact that we have prior knowledge for this problem, i.e., of the fact that force is the (negative) gradient of energy. Something more clever we can do is to have a single set of weights for predicting energy and force, and to have the prediction for force be simply the (negative) gradient of the prediction for energy. The details here rely crucially on linearity of the derivative. More explicitly, let f and g be the models outputting the energy and force of a system.

Then we have $f(x) = \langle w, \phi(x) \rangle$ and $g(x) = f'(x) = \langle w, \phi'(x) \rangle$. Now, it's almost as if we have we have an optimization with two objects rather than one (i.e., to have f fit well and to have g fit well). Fortunately, we can glue these together into a single optimization problem in a fairly straightforward way, i.e.,

$$\arg\min_{w} \left\| \begin{pmatrix} \phi(x) \\ \phi'(x) \end{pmatrix} w - \begin{pmatrix} y \\ y' \end{pmatrix} \right\|^2.$$

This just recovers the standard equation, with slightly different values of x and y than we're used to. In particular, we end up with

$$w = \begin{pmatrix} \phi(x) \\ \phi'(x) \end{pmatrix}^T \left(\begin{pmatrix} \phi(x) \\ \phi'(x) \end{pmatrix} \begin{pmatrix} \phi(x) \\ \phi'(x) \end{pmatrix}^T \right)^{-1} \begin{pmatrix} y \\ y' \end{pmatrix}.$$

[8] To motivate this, think about how the integral $\int f(x)g(x)\mathrm{d}x$ is the generalization of $\sum_i f(i)g(i)$.

The point here is that your data on both energy and force works toward your prediction on a single set of weights showing up in the equations energy and force. In particular, we just made use of linearity of the derivative:

$$\begin{aligned} g(x) &= \frac{\mathrm{d}}{\mathrm{d}x} f(x) \\ &= \frac{\mathrm{d}}{\mathrm{d}x} \langle w, \phi(x) \rangle \\ &= \frac{\mathrm{d}}{\mathrm{d}x} \sum_i w_i \cdot \phi(x)_i \\ &= \sum_i w_i \cdot \frac{\mathrm{d}}{\mathrm{d}x} \phi(x)_i \\ &= \sum_i w_i \cdot \phi'(x)_i \\ &= \langle w, \phi'(x) \rangle \end{aligned}$$

In fact, this same argument would have worked for any linear operator by the above reasoning (including, for instance, integration). So you can predict force, infer weights w, and use those same weights to predict energy in a sensible way (or vice versa) without doing any extra work!

§20 Tuesday, November 30

§20.1 Capturing invariances

We're back from Thanksgiving. For the past couple lectures, we've been talking about designing kernels to capture important properties and invariances. In physics and chemistry, for instance, this shows up because the energy of a molecular system is invariant to the rotation or translation of the molecule. That should be intuitive – if I just move all my atoms one meter to the left, the energy of the system won't change (and likewise if I rotate it by 90 degrees).

The goal today is to delve deeper into these topics, since they can be pretty important.

§20.1.1 Translation

By **translation**, we mean moving all of the objects in a system in the same direction and over the same distance. A given translation can be represented by a vector, for instance, which captures the difference between each object's final location and its original location. In our case, the idea is that each nucleus in the molecule will be moved in exactly the same way.

If $X \in \mathbb{R}^{N \times 3}$ is our system of N nuclei, then a translation represented by the vector $v \in \mathbb{R}^3$ sends X to $v \oplus X$, where

$$v \oplus X = \begin{pmatrix} v + X_1 \\ \vdots \\ v + X_N \end{pmatrix}.$$

A couple straightforward observations:

- $v_1 \oplus (v_2 \oplus X) = (v_1 + v_2) \oplus X$
- $(v_1 + v_2) + v_3 = v_1 + (v_2 + v_3)$

- $v + 0 = 0 + v$
- $v + (-v) = 0$

So, here's a recap of what we've seen so far about translations.

1. Translations can be parameterized by vectors in $\mathbb{R}^3$
2. The action of a translation $v \in \mathbb{R}^3$ is denoted by $v \oplus X$
3. Vector addition corresponds to composing translations. Furthermore, such translations (parameterized by vectors), have an associative addition, inverses, and a 0 element.

We're getting equipped to make the notion of translation invariance precise. We'll need to introduce the notion of an orbit.

Definition 20.1 — The **orbit** of a system $X \in \mathbb{R}^{N\times 3}$ is the set $\{g \oplus X : g \in \mathbb{R}^3\}$.

Definition 20.2 — We will say that two systems $X, Y \in \mathbb{R}^{N\times 3}$ are **equivalent**, and write $X \sim Y$, if their orbits are the same.

Definition 20.3 — A map $f{:}\mathbb{R}^{N\times 3} \to \mathbb{R}$ is **translation invariant** if $f(X) = f(Y)$ whenever $X \sim Y$.

Now we've made everything precise :)

§20.1.2 Permutation

The idea here is that we want our model to send $X = \begin{pmatrix} X_1 \\ \vdots \\ X_n \end{pmatrix}$ to the same place as $X_\sigma = \begin{pmatrix} X_{\sigma(1)} \\ \vdots \\ X_{\sigma(n)} \end{pmatrix}$ whenever σ is a permutation. We can again describe this using permutation matrices, which are matrices with exactly one 1 in each row/column and all zeroes elsewhere.

These matrices correspond to permutations of basis vectors, so they model precisely those transformation that shuffle the order of our data points X_i. Furthermore, they have the group structure[9] that we discussed in the previous section, so we can speak of their orbits in a sensible way.

In particular, a system $X \in \mathbb{R}^{N\times 3}$ is acted upon by a permutation matrix $P \in \mathbb{R}^{N\times N}$ via $X \mapsto PX$, i.e., using matrix multiplication. Just as in the previous section, we can consider composition of permutations via $P_1(P_2X) = (P_1P_2)X$ and observe that these permutations have inverses and a null/identity element.

The big picture is again that:

1. Permutations are represented using permutation matrices.
2. Permutation matrices act on molecules by multiplying from the left.

[9] Don't worry if you haven't seen group theory before; the idea is just that these permutation matrices associate, have inverses, and have an identity element.

3. Permutations are combined by matrix multiplication.
4. This gives rise to orbits and an equivalence relation $\sim_P$.
5. A **permutation invariant** map f is one that satisfies $f(X) = f(Y)$ when $X \sim_P Y$.

§20.1.3 Rotation and reflection

You've got the hang of this by now: we represent rotations and reflection using **orthonormal matrices**. They act on the data via matrix multiplication (on the right this time!), give rise to orbits/an equivalence relation, and we can ask for invariance by demanding that the function's output be equal when two systems are equivalent.

§21 Thursday, December 2

§21.1 Designing kernels

Last time, we got precise about the ideas of rotation and translation invariance for models. The point to all this was that we know that we're in a setting where the thing that we're predicting (the energy of a chemical system) is indeed rotation- and translation-invariant. That is, simply translating or rotating a system doesn't change its energy at all.

So the goal is for our model f to have the property that $f(X) = f(Y)$ when $X \sim_T Y$. Recall that $X \sim_T Y$ means that $X = v \oplus Y$ for a vector $v \in \mathbb{R}^3$, with $X, Y \in \mathbb{R}^{N\times 3}$. Now say we have a training set $X = \{X^{(1)}, \dots, X^{(n)}\}$. Then, by our previous discussion about kernels, our model f will take the form

$$f(X) = \sum_{i=1}^{n} \alpha_i K(X, X^{(i)}).$$

Now how do we define kernels that act between molecular systems living in $\mathbb{R}^{N\times 3}$? One way to do this is to sum up kernels acting on each of the coordinates of the systems, i.e.,

$$K(X^{(a)}, X^{(b)}) = \sum_{i=1}^{N}\sum_{j=1}^{N} \text{Gaussian}(X_i^{(a)}, X_j^{(b)}).$$

We're not saying that is the best way to do it, but it's an option. What we really want to keep in mind is that our kernel should be invariant to the action at hand (e.g., rotation or translation).

Definition 21.1 — A kernel K is invariant to a relation $\sim$ if $K(X, Z) = K(Y, Z)$ for all Z when $X \sim Y$.

Lemma 21.2

If a kernel K is invariant to an equivalence relation $\sim$, then so is the model f it induces.

Proof.

$$f(X) = \sum_{i=1}^{n} \alpha_i K(X, X^{(i)}) = \sum_{i=1}^{n} \alpha_i K(Y, X^{(i)}) = f(Y)$$

□

Remark 21.3. The task of producing a model with certain invariances now reduces to the task of producing kernels with the desired invariances!

Example 21.4

One fairly natural way to attain translation-invariance is to center your data before doing anything with it. In particular, we could define

$$\Phi(X) = (-\overline{X}) \oplus X.$$

Then any kernel of the form $K(X,Y) = \langle \Phi(X), \Phi(Y) \rangle$ would have the translation-invariance that we're after.

Lemma 21.5

Φ, as defined in the previous example, is such that $\Phi(X) = \Phi(Y)$ if and only if $X \sim_T Y$.

Proof. Say $\Phi(X) = \Phi(Y)$. Then

$$X = \overline{X} \oplus \Phi(X) = \overline{X} \oplus \Phi(Y) = \overline{X} \oplus (-\overline{Y} \oplus Y) = (\overline{X} \oplus -\overline{Y}) \oplus Y.$$

Conversely, suppose $X \sim_T Y$. Then $X = v \oplus Y$ for some v. We then have

$$\Phi(X) = \Phi(v \oplus Y) = -\overline{v \oplus Y} \oplus (v \oplus Y) = -\overline{Y} \oplus Y = \Phi(Y).$$

□

There's a modern field of deep kernel learning in which deep learners are used to construct the maps Φ and kernels are then constructed as $K_I(X,Y) = K(\Phi(X), \Phi(Y))$. This can be really successful, performing much better than manually designed kernels when things go right.

One last comment: if you were interested in making a kernel that's permutation invariant, then the sum of (for instance) Gaussian kernels we discussed earlier could be useful, as addition is commutative.

That's everything – thanks for joining us, and best of luck on the final project!

Index