Introduction to Machine Learning

COMP 6321 Machine Learning
Concordia University
Fall 2020

What is machine learning?

Machine learning can be understood from several perspectives. This course emphasizes a software perspective: that *machine learning is just an example-driven way to build programs.* Machine learning is an important approach to software development because it can build programs that no human could write by hand, and because those programs are *valuable*. Programs built by machine learning can do new things, earn more money, save more time, and even save more lives than programs built by traditional software engineering methods.

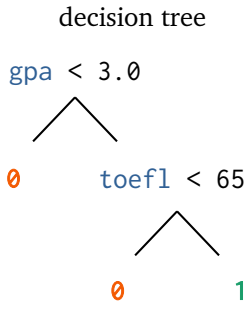
Machine learning is also hard to do correctly, and it is easy to fool yourself into thinking that you have succeeded in building a useful program—or a safe program—when in fact you have not. That is why software engineers should formally study machine learning!

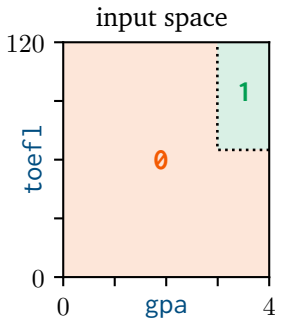
What is machine learning to a software engineer?

Humans are amazingly good at learning ideas from just a few examples. So, the best way for you to understand the idea of machine learning will be *by example*! Let's start with a simple example and then build on it.

To see what it means to "build a program" in an "example-driven" way, first try to imagine writing the example program below the usual way, by hand. The program either accepts (`1`) or rejects (`0`) a student from university based on grades (GPA) and language proficiency score (TOEFL). At left is the C code, and at right are two depictions of how it works: as a *decision tree* diagram, and as a plot of outputs (`0` or `1`) over the input space.

```
int accept(float gpa, int toefl) {
    if (gpa < 3.0)
        return 0;          // REJECT
    else if (toefl < 65)
        return 0;          // REJECT
    else
        return 1;          // ACCEPT
}
```



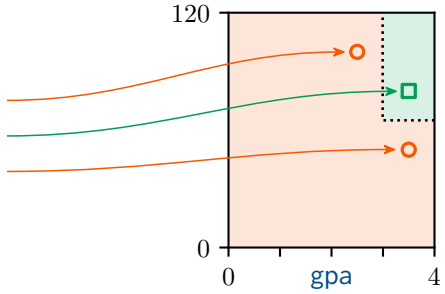


As a human, to create the above program you would have to: understand the goals and requirements; design steps that hopefully achieve those goals; express those steps in a human-readable programming language. That is how software is made, the usual way.

(Throughout this course, it's very important that you understand the visualizations used, especially plots that depict the outputs over input space like the previous figure. Study the rightmost plot carefully and make sure you understand why it captures program behaviour. We will use two-dimensional plots like this dozens of times in the course.)

To connect this example with machine learning, imagine writing a unit test for the `accept` function above. Assume that you decided on three test cases, shown below at left. If we plot these test cases in input space, we see that the expected output (○ for `0`, □ for `1`) matches the actual output (coloured regions) at all three locations, so `test_accept` passes.

```
void test_accept() {
    ASSERT(accept(2.5, 100) == 0);
    ASSERT(accept(3.5,  80) == 1);
    ASSERT(accept(3.5,  50) == 0);
}
```

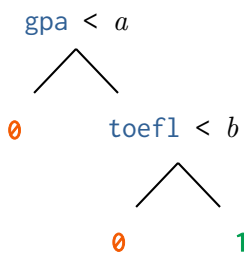


Each test is an *example* of what we consider to be 'correct' program behaviour. However, there exist many non-equivalent that pass all three test cases, and our `accept` function is just one of them. In other words, the test cases *underspecify* correct program behaviour.

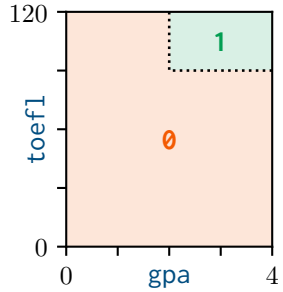
An important concept in machine learning is the *parametrization* of programs. Let's parametrize the `accept` function by replacing its hard-coded thresholds 3.0 and 65 with adjustable parameters a and b. Now we have a *family* of programs, where $(a, b) = (3.0, 65)$ identifies the original `accept` function, but other programs can be identified too, like the program for $(2.0, 90)$ shown below at right.

```
int accept(float gpa, int toefl) {
    if (gpa < a)
        return 0;
    else if (toefl < b)
        return 0;
    else
        return 1;
}
```

decision tree with parametrized thresholds

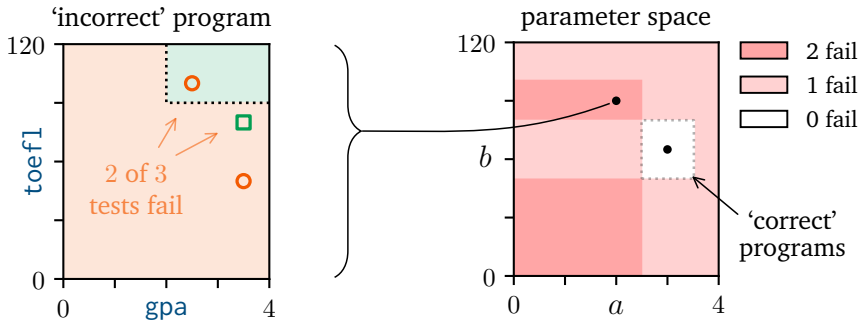


input space when $a = 2.0, b = 90$



Never confuse parametrization (a, b) with the program parameters (`gpa`, `toefl`). After we choose a good a and b (*i.e.* after we 'learn'), they become "baked in" as constants.

Now that we have parametrized the `accept` function, which members of this family can pass our unit test? If we choose $(a, b) = (2.0, 90)$ the unit test fails because, for 2 of 3 test cases, the actual output does not match the expected output; this can be seen by plotting the test cases in input space, shown below at left. To see the bigger picture, can plot the number of test failures over *all* possible choices of (a, b), shown below at right.

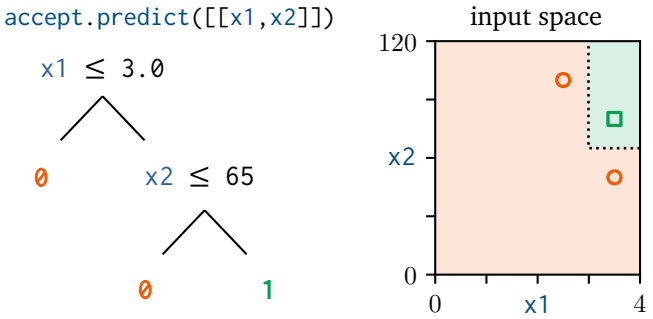


The parameter space plot shows that our original program at $(3.0, 65)$ has 0 failures, whereas the alternate program at $(2.0, 90)$ has 2 failures.

And now the main point: machine learning can *automatically* build a program that passes the tests, without needing to write the `accept` function by hand. A machine learning algorithm searches over a particular "space of programs," like the parameter space above, until it finds a program that is minimizes the number of failures.

Let's try to 'build' an `accept` function, using machine learning. By using the scikit-learn library (`import sklearn`), it only takes a few lines of Python. First we convert the three tests into a *training set*, organized into an array of example inputs (`x`) and an array of expected outputs (`y`). Then we create an instance of `sklearn.tree.DecisionTreeClassifier`, which is a general-purpose decision tree. Finally, we *train* (`fit`) the decision tree on our examples; this runs a particular machine learning algorithm designed to build decision trees. After running the Python script, shown below at left, the `accept` variable is a decision tree object that accepts two inputs and produces one output, as depicted at right.

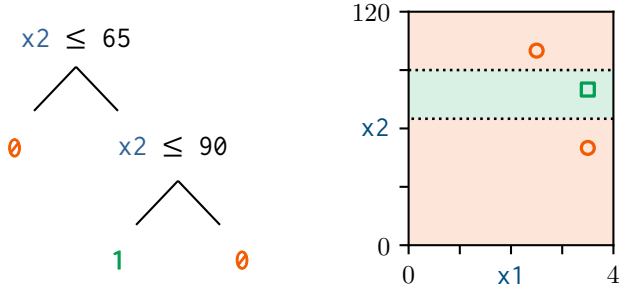
```
x = [[2.5,  100],  # 1st example
     [3.5,   80],  # 2nd example
     [3.5,   50]]  # 3rd example
y = [0, 1, 0]      # desired output

# Build a decision tree by example
accept = DecisionTreeClassifier()
accept.fit(x, y)
```



You can see that the 'learned' or 'fitted' decision tree passes all three tests. The important thing to notice is that scikit-learn's algorithm built a decision tree having the same structure (two internal nodes, three leaf nodes) and the same thresholds $(3.0, 65)$ as our hand-written version. The hand-written version in C can be invoked as `accept(3.5, 80)` which returns `1`. The scikit-learn version in Python can be invoked as `accept.predict([[3.5, 80]])` which likewise returns `1` because it is essentially the same function as the C version.

Since `DecisionTreeClassifier` is a general-purpose tool, not specific to one application, the names of its inputs are generic. So, whereas the inputs of the hand-written `accept` are named `gpa` and `toefl`, the inputs for the machine learning version are just `x1` and `x2`. You, as the machine learning practitioner, must keep track of which-input-means-what.

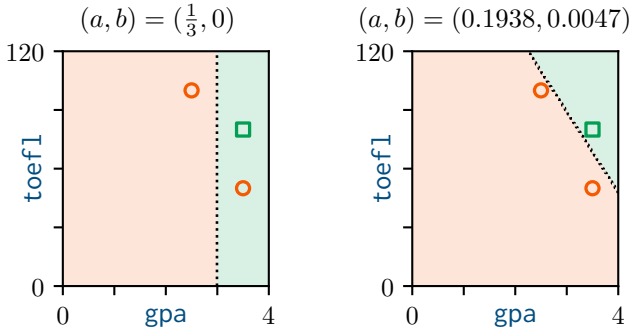
Recall that many different programs can pass all three test cases. In the above example, scikit-learn built the same decision tree that we programmed by hand, *but only by chance*. Run the exact same Python script a second time and it may build the decision tree below, which passes all the tests but does not even consult the GPA input (`x1`) when deciding!



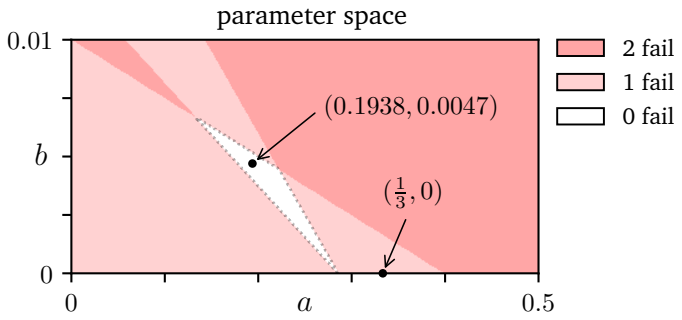
The reason for this variability is that the algorithm for building a decision tree has many arbitrary choices to make (*e.g.* should the root node check `x1`, or `x2`?) and so the algorithm breaks ties by consulting a random number generator. As a human adult, you know that GPA is important, and that the "ignore GPA" version of `accept` will admit *terrible* students. The machine learning algorithm does not know this, and is just as happy ignoring GPA.

Before we move on from the main example, there is another variation that is important to understand. Many programs that are *not* decision trees can also pass these three tests. A program could evaluate a weighted combination of the inputs like "`0.5*gpa + 0.1*toefl`" and return `0` or `1` depending on that sum. Does there exist a parameter setting (a, b) that passes all three tests? Let's see a parametrized version of `accept` where (a, b) are *weights* (not thresholds), shown below at left. Also shown is the *decision boundary* (dotted line) for two parameter settings.

```
int accept(float gpa, int toefl) {
    if (a*gpa + b*toefl < 1)
        return 0;
    else
        return 1;
}
```



As you can see, the decision boundary is always a single straight line, which makes sense. The line can be axis-aligned like decision tree boundary segments, or it can have slope $\frac{a}{b}$. For $(a, b) = (0.1938, 0.0047)$ the resulting program passes all three tests, but for $(\frac{1}{3}, 0)$ there is one failure. The two parameter settings above are depicted as black dots in the portion of parameter space plotted below, with the number of failures indicated in red.



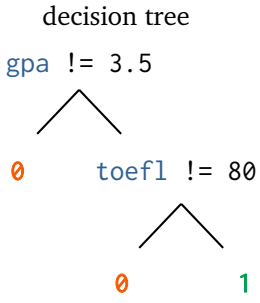
In machine learning, a weighted combination like "`a*gpa + b*toefl < 1`" is a simple kind of *linear model*. Scikit-learn makes it easy to build a linear model (*i.e.* to find a specific a and b) that can pass all three of our tests. Instead of a decision tree, we can create an instance of `sklearn.linear_model.LogisticRegression` and train it on the examples:

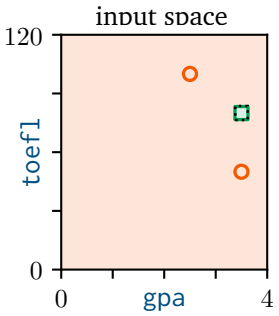
```
accept = LogisticRegression(penalty='none')  # Weighted combination of inputs
accept.fit(x, y)                             # Same x and y arrays from before
```

If you run the above script with the `x` and `y` containing our three training examples, the resulting `accept` function will return `0`/`1` based solely on a weighted combinations of inputs. Specifically, it will be equivalent to choosing parameter setting $(a, b) = (0.1938, 0.0047)$ in the hand-written version of `accept`, which we know has 0 failures on the tests. Logistic regression is an extremely useful and important method in its own right, and is important for understanding fancier methods too, so we will study it early on in the course.

Finally, we can here demonstrate an important concept in machine learning: *overfitting*. The most extreme form of overfitting is *memorization*. The program shown below passes all three tests but only because it 'memorized' the input for which it should return `1`, and returns `0` everywhere else. The resulting program rejects almost *everyone* and is useless.

```
int accept(float gpa, int toefl) {
    if (gpa != 3.5)
        return 0;
    else if (toefl != 80)
        return 0;
    else
        return 1;
}
```





"Memorize the training examples" may seem like a great idea for a machine learning algorithm—after all, it guarantees zero mistakes on the training examples!—but it is a terrible idea. Stop and think for a moment about *why* it's terrible. What if the perfect student applies, with GPA 4.0 and TOEFL 120? The program will reject this student, *merely because those scores are unfamiliar*. Memorization cannot generalize to unfamiliar inputs.

The 'memorization' example above is our strongest clue that machine learning is about building programs that do *more* than reproduce the training examples. Software only has value if it produces *useful* outputs for inputs that we *actually care about*, including inputs

that we've never encountered before. Software created through machine learning is no different. A breast cancer diagnostic must produce an accurate output for a new patient, even if that patient's mammogram and bloodwork are distinct from every patient that came before. Inferring general rules from specific examples is an inherently *ambiguous* exercise . In machine learning, overfitting is the main symptom of that ambiguity, and has huge practical consequences. Overfitting can be more nuanced than the "memorization problem," but it clearly demonstrates why "minimizing the number of training mistakes" is not the ultimate goal of machine learning, even if an algorithm strives for it.

Summary:

- ▷ Machine learning is an example-driven way to specify program behaviour.
- ▷ The training examples are often given as input-output pairs, a bit like unit tests.
- ▷ A machine learning algorithm searches a particular 'space' of programs (of decision trees, of linear functions, *etc*.) for a program that is consistent with the examples.
- ▷ For any set of examples, there exist *many* distinct programs that are consistent, and some of those programs have more value than others (*e.g.* due to overfitting).
- ▷ A program only has value if it provides *useful* outputs for inputs that *actually matter*.
- ▷ Some types of 'learned' programs are easier for a human to interpret than others.

Machine learning from other perspectives

Machine learning can be understood from other perspectives too. The goal of this section is to connect machine learning with as many 'familiar' concepts as possible. That way you can learn faster and gain insight from what you already know.

Test-driven development perspective. Any good software engineer thinks about tests, especially unit tests. Machine learning focuses on collecting examples of desired program behaviour before writing the program. In that sense, machine learning is like a form of "automated test-driven development," where the tests are written before the program is written. (This perspective was used to explain machine learning in the previous section.)

For example, suppose you plan to write a sorting algorithm `my_sort`. In test-driven development you write the tests *before* implementing, like this:

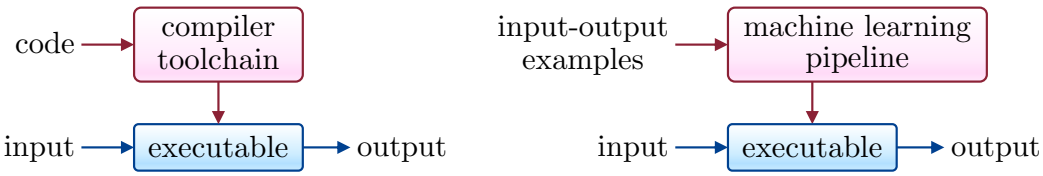
```
def my_sort(items):
    raise NotImplementedError()
```

```
def test_my_sort():
    assert my_sort([]) == []
    assert my_sort([1,3,2]) == [1,2,3]
    assert my_sort([3,1,2]) == [1,2,3]
```

The tests are expected to fail, initially. As you make progress on writing correct code for `my_sort`, you'll manage to get more tests to pass. So far, so good.

How would we get a machine learning algorithm to implement `my_sort` automatically, from examples alone? First we'd need a *lot* more examples of unsorted inputs and sorted outputs than shown above. Then we'd need an machine learning algorithm capable of producing a correct sorting algorithm based solely only on those examples. This is related to the concept of program induction, and it is not obvious how to do it in a general way. In fact, the ability to 'learn' a sorting algorithm from scratch was only recently achieved by advanced deep learning architectures like *Neural Turing Machines*.

Compiler toolchain perspective. You already know compiled programming languages such as C++ or Java. A compiler outputs an executable program. A machine learning algorithm *also* outputs an executable program, or at least a specific parameter setting for an existing parametrized program. The role of a machine learning algorithm is therefore like that of a compiler. The difference is that a compiler toolchain accepts *code*, whereas a machine learning pipeline accepts *input-output examples*:



In software development, source code is an upstream dependency of the executable, as is often expressed in a `makefile`. Changing your code means recompiling your program. In machine learning, the executable has two upstream dependencies: the *training data*, and the code for the *training algorithm* itself. Changing either of those means re-building (re-'training') your program. Again, any software engineer is accustomed to using "software tools that helps us write programs," and machine learning is one of those software tools.

There is another important analogy to make between compilers and machine learning. Software engineers know that the choice of programming language is important: it can make a given program easier or harder to write. Similarly, the choice of machine learning method is important: it can make a given program easier or harder to learn from examples. Anticipating which methods are likely to work best, and to not waste time on ill-suited methods, is a skill that takes time to develop.

Program template perspective. In machine learning, the programs we build are often called 'models' because they represent a mathematical approximation of some process or phenomenon. A model typically has two kinds of parameters: input parameters, and model parameters. For example, when you are fitting a line $y = ax + b$ to data, x is always an input parameter, whereas a and b are model parameters that become constants after training. The parameter space $(a, b) \in \mathbb{R}^2$ defines the family of lines that we can search over. This distinction between input parameters like x and model parameters like a and b is analogous to an important distinction in template programming.

If you are familiar with C++, you already know that a "function template" does not define an actual function, and cannot be called directly. Rather, a function template defines an abstract *family* of functions. Only by identifying a specific member of that family can we arrive at a callable function. Continuing the example of line fitting $y = ax + b$, below is a C++ function template representing the family of lines having integer coefficients; this example identifies a particular function having slope $a = 2$ and intercept $b = 1$, and then calls that function with input parameter $x = 3$ to compute output $y = 7$.

```
template <int a, int b>     // template parameters a and b
float f(float x)            // function parameter x
    { return a * x + b; }   // f is the family of all lines y = ax + b

auto  g = f<2, 1>;          // g is the family member y = 2x + 1
float y = g(3);             // y = 2*3 + 1 = 7
```

The point of the above example is to demonstrate how template parameters, like `a` and `b`, are conceptually distinct from input parameters, like `x`. In particular, this distinction in template programming is highly analogous to the distinction between model parameters and input parameters in machine learning.

For those unfamiliar with C++ templates, a similar analogy can be made with the *partial assignment* programming technique. Partial assignment takes a general-form function and 'specializes' it by fixing some input parameters to constants. Below is a demonstration of how this works in Python (`functools.partial`) and in C++ (`std::bind`), with both equivalent to the template programming example above.

```
def f(x, a, b):
    return a * x + b

g = partial(f, a=2, b=1)
y = g(3)
```

```
float f(float x, float a, float b)
    { return a * x + b; }

auto  g = bind(f, _1, 2, 1);  // a=2, b=1
float y = g(3);
```

In the above, think of `f` as the untrained 'line' where suitable values for a and b are yet unknown, and think of `g` as the trained line where $a = 2$ and $b = 1$ were chosen. Again, in template programming the template parameters like a and b must be specified by hand, like we did above, whereas in machine learning they are determined automatically by a learning algorithm.

Curve fitting perspective. It should be easy to see how our "accept/reject a student" scenario from earlier is essentially curve fitting: the decision boundary $ax_1 + bx_2 \leq 1$ is a straight line, which is just a simple kind of curve. Other familiar curves include a quadratic like $ax^2 + bx$, or splines, or other mathematical expressions that can be written in terms of a set of observed quantities $(x_1, \ldots, x_n)$. In other words, many forms of machine learning are analogous to fitting a curve to data. Even decision trees can be understood as a kind of unusual curve. This is why scikit-learn's training functions are all named `fit`!

Deep learning can be seen as "curve fitting on steroids," where each "deep architecture" is just a different family of curves. Until 2012 it was non-obvious that curve fitting (includ-

ing neural networks) could work so well. Read the excerpt below, from a 2018 interview with Turing Award winner Judea Pearl:

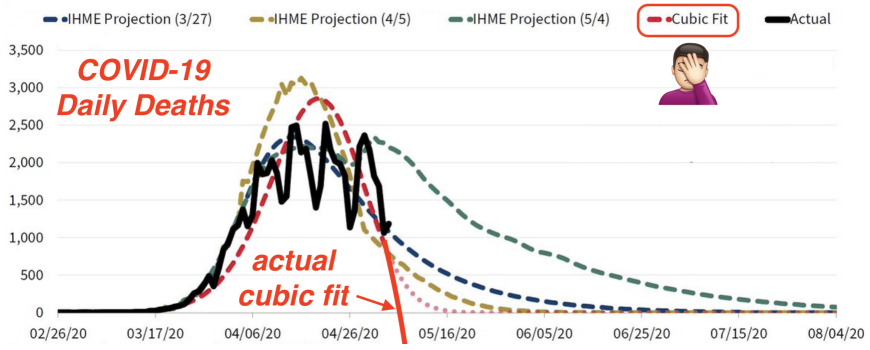
Judea Pearl: "As much as I look into what's being done with deep learning, I see they're all stuck there on the level of associations. **Curve fitting**. That sounds like sacrilege, to say that all the impressive achievements of deep learning amount to just fitting a curve to data. [But] it's still a curve-fitting exercise, albeit complex and nontrivial."

Interviewer: "The way you talk about curve fitting, it sounds like you're not very impressed with machine learning."

Judea Pearl: "No, I'm very impressed, because **we did not expect that so many problems could be solved by pure curve fitting**. It turns out they can."

Many top scientists, like Pearl, find something deeply unsatisfying about the curve fitting approach. For example, curve fitting has no explicit notion of cause and effect. That makes it easy to conflate correlation with causation. Mistaking correlation for causation is a classical fallacy and leads to bad predictions, by humans and computers alike. So, if your goal is to build software that makes *good* predictions, this is a real problem.

Curves often extrapolate data in ways that a human would recognize as totally wrong, and yet is hard to characterize all "obviously-bad curves" ahead of time. For example, in the midst of the unfolding COVID-19 crisis, the U.S. White House Council of Economic Advisors released an official forecast that predicted COVID-19 deaths would quickly plummet to zero. That official forecast, upon which actual human lives depended, was based on fitting (in Excel!!) the weights (a, b, c, d) of a cubic curve $ax^3 + bx^2 + cx + d$ to historical observations of daily deaths. Below is a marked-up version of the original chart (official tweet). Actual deaths are in black, and the now-infamous "cubic model" is in red.

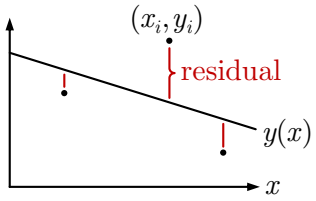


When this "cubic model" was extrapolated into the future (for x beyond the black curve), it predicted negative deaths. Any human would know that this is a nonsensical prediction. Indeed, the White House analyst noticed this too, so they truncated the cubic extrapolation and drew in a dotted line tapering off to zero deaths, by hand. The above "cubic model for COVID-19 deaths" model was rightly mocked but, if life-or-death policies are being made based on bad models, it is not funny at all.

Statistical perspective. The textbook for this course explains many machine learning concepts in statistical terms, like expected values $\mathbb{E}[x]$ or conditional probabilities $p(x \mid z)$. Why? Because statistics is a powerful framework for specifying what a machine learning system should do, and for reasoning about how the training algorithms should work. Major algorithms and theoretical results in machine learning have been justified entirely through statistical and probabilistic reasoning. Just as many compression algorithms would not exist without an information-theoretic justification, many machine learning algorithms would not exist without a statistical justification. In fact, many of the machine learning methods that you'll learn about in this course were developed from an approach that is explicitly called *statistical machine learning*.

If you just want to apply machine learning to some problem, a minimal understanding of the statistical viewpoint is probably enough to "get by" and to build useful programs. But the best machine learning practitioners all understand the statistical justifications for the methods they use, and they take this view seriously. So, take it seriously!

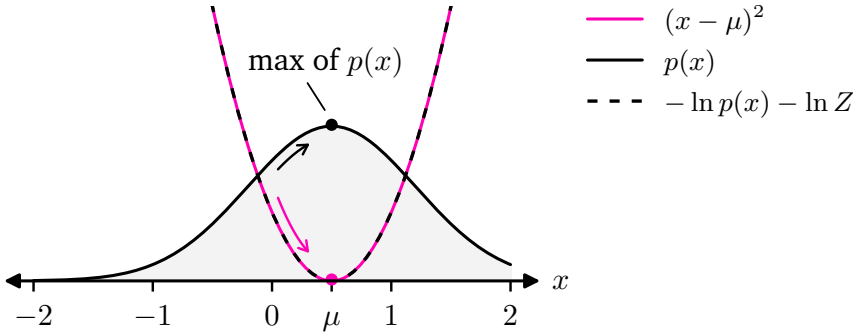
To see a rather simple connection, suppose you want to fit a line $y(x) = ax + b$ to a set of observations $(x_1, y_1), \ldots, (x_n, y_n)$. You should think of x as the input, $y(x)$ as the output[1] at x, and (x_i, y_i) as the training set. Fitting the line usually amounts to finding coefficients a and b that minimize $\sum_{i=1}^{n}(y_i - y(x_i))^2$, the sum of squared error residuals where $y_i - y(x_i)$ is the i^{th} residual, as shown at right. The statistical view is that each observed pair (x_i, y_i) is a random sample from an unknown joint probability distribution $p(x, y)$, and that our set of observations is just one of many that could have been randomly sampled. A more 'statistical' way to describe fitting a line is "find a and b that minimize the expected squared error residual $\mathbb{E}[(y_i - y(x_i))^2]$" where the expectation is taken over randomly samples $(x_i, y_i) \sim p(x, y)$ that we could have drawn. For a particular training set $\{(x_i, y_i)\}_{i=1}^{n}$, this expectation is can be approximated by $\frac{1}{n}\sum_{i=1}^{n}(y_i - y(x_i))^2$. In terms of fitting a and b, the scaling by $\frac{1}{n}$ doesn't matter, so this is equivalent to our original "sum of squared errors" formulation. Many machine learning formulations have equivalent statistical interpretations like this.



An important connection is also the correspondence of "maximizing a probability" and "minimizing an error." For example, recall the *univariate normal* (Gaussian) probability distribution: it has density function $p(x) = \frac{1}{Z}\exp(-\frac{(x-\mu)^2}{2\sigma^2})$ where μ is the mean, σ^2 is the variance, and $Z = \frac{1}{\sqrt{2\pi}\sigma}$ is the normalizing constant that ensures $\int p(x)\,\mathrm{d}x = 1$. The normal distribution is closely linked to squared error via the *negative logarithm*, because $-\ln p(x) = \frac{1}{2\sigma^2}(x-\mu)^2 + \ln Z$. (You should be able to easily derive this from $p(x)$ yourself.) You can see that the negative logarithm of the normal distribution corresponds to a kind of squared error between x and μ, which is then scaled by $\frac{1}{2\sigma^2}$ and has a constant $\ln Z$ added to it. The plot below shows that this correspondence between probability $p(x)$, negative

[1] Remember that $y(x)$ also depends on a and b. To make this explicit we could have written $y(x, a, b)$ or $y(x; a, b)$ or $y_{a,b}(x)$, but instead you should remember that $y(x)$ is "a function of x parametrized by a and b."

log probability[2] $-\ln p(x)$, and squared error for the case $\mu = \frac{1}{2}$ and $\sigma^2 = \frac{1}{2}$.



As you can see, in the above case finding x that maximizes probability $p(x)$ is exactly equivalent to finding x that minimizes error $(x-\mu)^2 = -\ln p(x) + \text{const}$. The connection between the normal distribution and squared error was established in 1795 by Carl Friedrich Gauss himself, and it is regularly applied in statistics and machine learning today.

Of course, the statistical perspective goes much deeper than these examples. Statistical reasoning gives a principled way to: express the goals of a machine learning system; compose machine learning systems together; deal with uncertainty in inputs and outputs; design training algorithms that are fast and correct; allow machine learning systems to generate ('dream') realistic data; and much more.

Optimization perspective. Machine learning is about building programs that minimize a measure of error, or that maximize a measure of accuracy. Optimization is about minimizing or maximizing functions. It should come as no surprise, therefore, that machine learning involves optimization. As we saw with the "memorization" example, machine learning is about more than just minimizing errors on the training data, but *a lot* of subproblems encountered in machine learning are still about optimization.

For example, consider a line fitting problem describes as follows: "fit $y(x) = ax + b$ to training examples $(x_1, y_1), \ldots, (x_n, y_n)$ but with non-negative slope a." We can express this as constrained optimization over a and b:

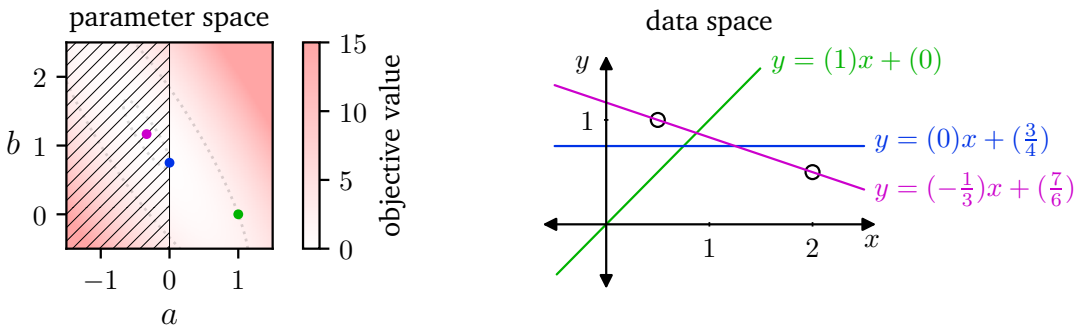
$$\begin{aligned} \underset{a,b}{\text{minimize}} \quad & \sum_{i=1}^{n} (y_i - y(x_i))^2 \\ \text{subject to} \quad & a \geq 0 \end{aligned}$$

If we substitute two specific training examples $(x_1, y_1) = (\frac{1}{2}, 1)$ and $(x_2, y_2) = (2, \frac{1}{2})$ into the abstract problem above, we get a concrete optimization problem where the objective is to minimize the sum of two squares (one for each training example) over a and b:

$$\begin{aligned} \underset{a,b}{\text{minimize}} \quad & (1 - \tfrac{1}{2}a - b)^2 + (\tfrac{1}{2} - 2a - b)^2 \\ \text{subject to} \quad & a \geq 0 \end{aligned} \tag{1}$$

[2]The plot subtracts off the $\ln Z$ term to make it clear that the dotted curve and the pink one have identical shape, and thus the same x minimizes both curves.

We can visualize this 'learning' problem as constrained optimization over a parameter space (a, b), where the objective value at each point is shown below, at left. Each point corresponds to a different member of the family of lines $y = ax + b$. The green dot is at $(a, b) = (1, 0)$, which is a poor fit to the two training examples. The purple dot is at $(0, \frac{3}{4})$, which is a perfect fit to the data (residual errors are zero!) but is 'forbidden' because of its negative slope (the hatched region violates $a \geq 0$). The blue dot is at $(-\frac{1}{3}, \frac{7}{6})$, which is not a perfect fit but is the solution to the optimization problem (1) and thereby a solution to the 'learning' problem that we originally expressed in English.



The point of formulating your learning problem as 'optimization' is that you can then use powerful optimization algorithms. For example, optimization problem (1) is easy to solve using a Python package like SciPy (`import scipy`). Specifically, because this objective is a "sum of squared errors" and the constraint $a \geq 0$ is a simple lower bound, we can use `scipy.optimize.least_squares` to solve problem (1) in a few lines. The algorithm starts at some initial point, like $(a, b) = (1, 0)$ (the green line), and then repeatedly evaluates residuals $y_i - y(x_i)$ for different a and b until an optimal value is found.

```
x = np.array([0.5, 2.0])  # training inputs [x_1, x_2]
y = np.array([1.0, 0.5])  # training outputs [y_1, y_2]

def f(ab):                # computes residuals for the given (a, b)
    a, b = ab             # unpack the a and b that least_squares is querying
    return y - (a*x + b)  # return an array of two residuals [y_1 - y(x_1), y_2 - y(x_2)]

bounds = ([0, -inf], [inf, inf])                    # 0 ≤ a ≤ ∞, −∞ ≤ b ≤ ∞
ab_ini = [1, 0]                                     # initialize search at (1, 0)
ab_opt = least_squares(f, ab_ini, None, bounds).x   # returns [0, 0.75]
```

Even if you don't know how the `least_squares` algorithm is defined, make sure you understand everything else that is going on in the code above, and why it works, because it'll be important for your success in machine learning and in this course. Remember, even though the above example has only two parameters, and was 'trained' on only two examples, it is representative of how very complex machine learning systems are trained.

Optimization has also become important for gaining insight into how an already-trained machine learning system makes decisions, especially for deep learning. A program built by deep learning will compute many intermediate values, sometimes called 'neurons'. The role of a particular neuron is often mysterious, even to the human who trained the model.

One way of gaining insight is to ask "what input would maximize the value of this neuron?" This question can be posed as an optimization problem.

For example, if you train a deep learning model to recognize objects in images (cat, dog, human, car, *etc*), a certain neuron might be more 'active' (take large positive value) when the input image is of a cat, and not of some other object. To study this neuron, you might collect example inputs (images) that 'activate' the neuron (cause the neuron's value to be large). Or, you could explicitly optimize the pixels of the image so as to maximize the neuron's activity, beyond what any natural image can produce. The two images below demonstrate the idea; they were taken from *Feature Visualization* by the Google Brain team, where you can find many more examples.

natural images that strongly activate the neuron

synthetic image that maximizes neuron activation

After using an optimization algorithm to "maximize the neuron's activity" we can see that, although this neuron responds to cats, its "idealized cat" is quite surreal. This neuron only 'understands' the appearance of a cat at the level of basic patterns (stripes, ears, fur, eyes).

In short, optimization is how we formalize machine learning, and is also a tool for gaining insight. These insights can help to debug machine learning models, to identify fundamental limitations, and to spur new ideas on how to improve machine learning.

Categories of machine learning

The goal of this section is to give you a rough sense of the major 'categories' of machine learning. The descriptions here are brief and high level. The machine learning methods that you will study in this course will be applicable to one or more of these categories.

Supervised learning. The machine learning problems described in this document have actually been *supervised learning*. A machine learning problem is 'supervised' if the training examples come in the form $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$, where each input $\boldsymbol{x}_i$ is directly paired with a desired output $\boldsymbol{y}_i$. The desired output $\boldsymbol{y}_i$ is called a *supervisory signal*, because it 'teaches' what to output. Supervised learning is the most straight-forward form of machine learning.

Unsupervised learning. When we have example inputs $\{\boldsymbol{x}_i\}_{i=1}^n$ but no example outputs, this is called *unsupervised learning* because the supervisory signal is absent. Building a

program from input examples alone may seem like an ill-posed problem, because it is! However, we can often discover patterns in input examples that are useful in later analysis and, furthermore, unsupervised data (no $\boldsymbol{y}_i$) is usually abundant and cheap to collect compared to data for supervised learning.

Semi-supervised learning. As the name suggests, when some training examples have supervisory signal and some do not, this is called *semi-supervised learning*. Think of it as a way to boost the performance of supervised learning (for which data is expensive) by adding in some unsupervised learning (abundant, cheap data).

Classification and regression. Roughly speaking, if you your program outputs a single category from a set like `{no, yes}` or from `{cat, dog, car}` then you are doing *classification*. If your program outputs a scalar value like temperature or a vector value like a point in 3D space, then you are probably doing *regression*. Some learning algorithms can be applied in either scenario, whereas some algorithms are designed for one or the other.

Structured learning. In regular machine learning, inputs $\boldsymbol{x}_i$ and outputs $\boldsymbol{y}_i$ are usually vectors, so you can do things like multiply vectors by a matrix, or subtract two vectors to compute their difference, *etc*. In *structured prediction*, the inputs and/or outputs are complex mathematical objects, like trees or graphs, with structure that makes it less clear how to compare them. For example, imagine the input $\boldsymbol{x}_i$ encodes an English sentence and the desired output $\boldsymbol{y}_i$ is a correct parse tree for that sentence. Structured prediction requires special training algorithms and special ways to measure the 'error' between outputs.

Reinforcement learning. If the program you are trying to build acts as some kind of "agent" that must output "actions" so as to achieve "goals" over time, like the control system of a robot, then you probably need *reinforcement learning*. Recall that, in supervised learning, each training input $\boldsymbol{x}_i$ is directly paired with a desired output $\boldsymbol{y}_i$. You can think of this as explicit and immediate feedback: "whenever the program sees $\boldsymbol{x}_i$ it is immediately rewarded for outputting action $\boldsymbol{y}_i$." In reinforcement learning there is still a supervisory signal, *but the feedback is neither direct, nor immediate*. Instead, there is an explicit notion of time, and there is a time gap between a program's action and its receiving any 'reward" for that action. Reinforcement learning is hard because it is so ambiguous: desired outputs for the program are only given *indirectly*, through subsequent rewards or a lack thereof.

Active learning. In standard machine learning, the training examples are pre-determined. In *active learning*, the training algorithm gets to 'ask' for a specific 'next' training example, guiding the data collection process in a smart and incremental way. Think of it as a kind of feedback loop, alternating between learning and data collection. When data is slow or expensive to collect, active learning can help by collecting only the most important data.

Transfer learning. When training data is limited, one of the most powerful techniques for improving performance is *transfer learning*. The idea is to first train a model to solve a prediction task for which data is abundant (task A), and then use patterns discovered

there to do a better job at the prediction task of interest (task B). If the two prediction tasks are related enough, then task B will benefit greatly from patterns discovered for task A. In computer vision and natural language processing, transfer learning from pre-trained models has become commonplace: practitioners regularly take large pre-trained models "off-the-shelf" and then use them to boost predictive performance for a different task.

Multi-task learning. Suppose that for a given input $\boldsymbol{x}_i$ you know the desired outputs for two separate tasks A and B. You could obviously train two independent models, one for task A and one for task B, but in *multi-task learning* you train a single model to predict all task outputs simultaneously. Think of it as a kind of simultaneous transfer learning, where each task benefits from the patterns discovered in solving the other tasks. Multi-task learning is very powerful when applicable, and is especially popular for neural networks. Multi-task learning was key to winning the $40,000 Merck Molecular Activity challenge.

Deep learning. Unlike the other versions of "learning" mentioned here, deep learning is a family of machine learning *techniques* rather than a category of machine learning *problem*. Deep learning can be applied to any of the categories mentioned above. Deep learning has a long history, but only in the last decade has it matured enough to become a uniquely successful approach to machine learning. There is, by now, a huge ecosystem of deep learning techniques, but you can just think of deep learning as "curve fitting on steroids."

How is artificial intelligence related?

Machine learning is not what most people mean when they talk of "artificial intelligence." Artificial intelligence is a goal, whereas machine learning is a strategy. Artificial intelligence is about imbuing computers with human-like abilities such as perception, reasoning, planning, and communication. Machine learning is just one strategy that is relevant to that endeavor, among many other strategies such as symbolic A.I. that have long histories.

Machine learning is useful in the pursuit of artificial intelligence for two main reasons: (1) there are many artificially-intelligent programs that we do not know how to write by hand, and (2) for those programs we often have access to many examples of the desired artificially-intelligent behaviour. As an example-driven approach to building software, machine learning it is well-suited to the pursuit of artificial intelligence.

Deep learning is just one approach to machine learning, but it is most closely associated with artificial intelligence for two reasons: (1) the techniques were inspired by neuroscience, and (2) it works well for tasks that we associate with human-like abilities, like perception. The inventors of deep learning were motivated by the pursuit of artificial intelligence, and not by the pursuit of any other kind of software. At the same time, these inventors were skeptical about symbolic A.I. because of its focus on logic and reasoning, which does not represent how the human mind works most of the time. For example, watch this brief YouTube interview (3:42) with neural network pioneer and Turing Award

winner Geoffrey Hinton, excerpted below.

Geoffrey Hinton: "In the old days, the basic idea was that human reasoning is the core of intelligence, and so to understand human reasoning we'd better get something like logic into the computer. Then the computer would 'reason away' and maybe we could make it reason like people. But it's quite tricky to reason like people, mainly because **people don't do most of their thinking by reasoning.**

So the alternative view was that we should look at biology and we should try to make systems that work roughly like the brain, and the brain doesn't do most of its thinking by reasoning; it uses things like analogies. It's a great big neural network that has huge amounts of knowledge in the connections. It's got so much knowledge in it that **you couldn't possibly program it all in by hand**."

Scientists like Hinton resorted to machine learning in their pursuit of artificial intelligence because the programs they wanted to write were inspired connections in the brain, and it was obvious to them that *no human could write those kinds of programs by hand*. That is why machine learning has an important role to play in artificial intelligence.

The goal of artificial intelligence precedes the development of machine learning. Alan Turing, considered the 'father' of artificial intelligence, developed his ideas about "intelligent machines" around 1950, including his now famous "Turing test." The idea that computers could serve as a substrate for intelligence, as an alternative to organic brains, captivated the public imagination. For a glimpse into this exciting period, listen to Turing's 1951 BBC broadcast (15 min) titled "Can digital computers think?", or you can read his original draft with hand-written corrections. Like the excerpt from Hinton above, Turing laments that he does not know how to program intelligence by hand, likening it to the task of "writing a treatise about family life on Mars." Turing also explains that the limited computer memory of his era is insufficient, likening it to "not having enough paper on which to write." Turing summarizes the situation:

Alan Turing: "Our problem of programming a computer to behave like a brain is something like trying to write this treatise from a desert island: we cannot get enough paper to write on, and in any case, we cannot know what to write down if we had it. This is a poor state of affairs.

I will not attempt to say much about how this procedure of 'programming machine to think' is to be done. The fact is that we know very little about it, and very little research has yet been done. There are plentiful ideas, but we do not yet know which of them are of importance.

I will only say this: that **I believe the process should bear a close resemblance to that of teaching**. "

Turing goes on to discuss the ramifications for our notions of free will, and he asks that "no great effort" be put into building machines that bear characteristics of humans unrelated to intelligence, saying that such machines would have "the unpleasant quality of artificial flowers." Turing was describing what is now known as the "uncanny valley," best exemplified by the 'creepy' appearance of humanoid robots, like those of Hiroshi Ishiguro.

When is machine learning a good idea?

Machine learning is example-driven software development. For much of the software that the world needs today, writing code by hand is better than trying to 'teach' by example. Powerpoint is a program that probably cannot be built "by example." A correct implementation of a blockchain protocol probably cannot be built "by example." That being said, even software that is mostly programmed by hand can benefit greatly from components that are implemented by machine learning techniques. For example, Powerpoint's grammar checking and image processing tools are greatly enhanced by machine learning.

Given a software component that you want to build, machine learning is worth considering if the following conditions hold, simultaneously:

- ☐ You know how compute a meaningful measure of success for any candidate program.
- ☐ You expect that hand-written programs will not succeed in solving your problem.
- ☐ You have access to many examples of correct program behaviour, or you have at least a few examples and a good transfer learning opportunity.
- ☐ You have access to computational resources sufficient for training and deploying.
- ☐ You understand the software maintenance burdens of a machine learning system and have deemed the potential benefits worthwhile; see "technical debt" discussion later.

For example, if you work for a company like Boston Dynamics (YouTube demo, 0:38), you can definitely check the box that says a purely "hand-written programs will not succeed" because, despite years of trying, nobody managed to write a satisfactory bipedal robot controller by hand—not until machine learning came along.

If you work on a real software project, bad reasons for using machine learning include: "because everyone is using it," or "because I want the experience for my *next* job," or "because then we can say that we're doing A.I."

Whether machine learning is a "good idea" also depends on what you mean by "good." Machine learning help you build a programs that achieve your goal. But, like for any software, this says nothing of whether the goal is itself a good idea. If your goal is to automatically detect breast cancer, and machine learning lets you do it, then machine learning is an unambiguously good idea. If your goal is to assassinate people with autonomous drones, and machine learning lets you do it, then this is still a terrible idea; for a glimpse

of how terrible, watch the *Slaughterbots* video (8 min) and the New York Times video piece *A.I. Is Making It Easier to Kill (You)* (20 min). At the outset of this document, we said that programs built by machine learning can "save more lives" than ordinary software. But machine learning can also build programs that *take* more lives, when applied for example to autonomous weapons systems.

What makes machine learning difficult?

Software engineering is already a difficult discipline. Machine learning is extra-difficult for a number of reasons, and understanding those reasons will help to focus your learning.

Extra skills are needed. Being a good software engineer helps *a lot* when it comes to implementing machine learning systems, but it is not enough. Machine learning also requires a wider set of skills that not all software engineers or computer scientists have:

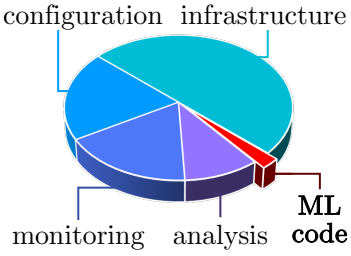
- *probability* is important because machine learning systems deal with uncertainty and because the goal is often to minimize the probability of an erroneous output;
- *optimization algorithms* are important because they let us formalize our goals (minimize errors) and because they provide powerful algorithms for achieving those goals;
- *multivariable calculus* and *linear algebra* are important because they let us express families of programs and derive training algorithms, especially for deep learning.

Just as it takes time and experience to become a good software engineer, it takes time and experience to become a good machine learning practitioner. And, like any discipline, the best way to learn is to tackle a tiny piece of a big problem that you care about solving; even if you fail to solve the subproblem to your satisfaction, you will learn much better!

Technical debt is harder to manage. Every experienced software engineer is familiar with the concept of technical debt. When too much technical debt accrues in a software system, that debt must be 'repaid' by refactoring code and fixing bugs. Such "debt maintenance" activities represent an opportunity cost, because they come at the expense of new features or capabilities. Managing technical debt is a key software engineering skill.

In a machine learning system, technical debt accumulates in new and insidious ways. Machine learning systems are therefore harder to manage and to maintain over time than traditional software systems; this is especially true for systems in production, like the programs that recognize speech in your smartphone or that make recommendations in your social media feeds.

Every software engineer who works with machine learning should read *Hidden Technical Debt in Machine Learning Systems*. The article explains several new sources of technical debt. One example is the tendency of machine learning projects degrade into unmaintainable "pipeline jungles." Another example is that



machine learning systems have "data dependencies," and these
are harder to track and to manage than code dependencies. It is also important to recognize that machine learning code represents only a small fraction of the total code needed to deploy a machine learning system; this is depicted at right. A tremendous and complex infrastructure must be built out and maintained, and new forms of technical debt accrue in that software, too.

Collecting data is hard. Most people building real-world machine learning systems quickly find that the hardest part is not the machine learning itself. The hardest part is getting lots of high-quality data, and in a form that is directly useful for the problem you want to solve. As they say: "garbage in, garbage out." Collecting data takes patience, persistence, and more often than not it also takes money. Many companies and academics have used crowdsourcing services like Amazon Mechanical Turk to hire an army of low-paid human labourers ('Turkers') who will add supervisory signal to data, *i.e.*, the human is presented with an example input $\boldsymbol{x}_i$ and is paid a few pennies to enter the corresponding $\boldsymbol{y}_i$ into the database; this has been called "artificial artificial intelligence." If you work for a company that relies on machine learning, you may find that investing in more data, or in better data, is a better way to add value than by experimenting with exotic algorithms.

Learning is computationally hard. Students of computer science or mathematical optimization will not be surprised that machine learning is computationally hard. The difficult depends somewhat on the family of 'programs' that we permit ourselves to search over. For example, polynomial fitting can be solved efficiently, whereas an optimal decision trees or neural networks are NP-complete to build. However, approximation algorithms can work surprisingly well in machine learning, despite the problems being formally 'hard'.

Program correctness is hard to verify. Correctness is paramount for software running in high-stakes setting, such as medical decisions, autonomous vehicles, cyber-security, financial markets, or criminal justice. Almost by definition, hand-written programs are understandable to a human, and so we can reason about how inputs could plausibly cause outputs, and about "how wrong" such a program might plausibly behave. Most machine learning systems offer no such reassurance, and could be "arbitrarily wrong" the more unfamiliar the input data. Empirically we might get lucky and find that the system works under all the conditions we care about. But we usually cannot prove that the machine learning system will never go haywire on certain inputs.

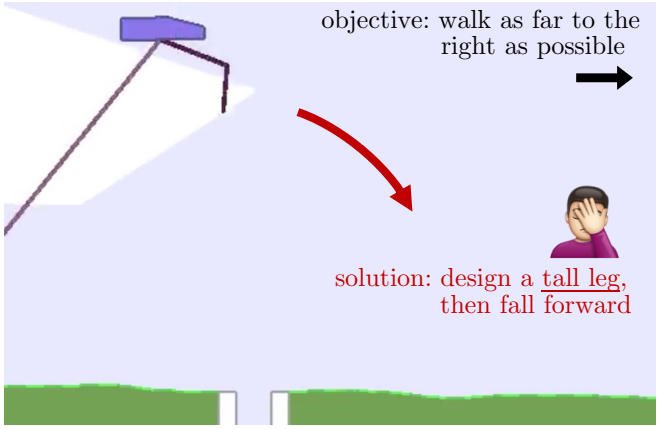
The study of "adversarial examples" is largely about finding inputs for which the output has gone haywire, *i.e.*, points in input space where the 'curve' is arbitrarily wrong. Finding adversarial inputs is the machine learning version of fuzz testing, which every software engineer should be familiar with. The reason for seeking such inputs may be to sanity-check program behaviour (think 'white hat'), or to maliciously exploit incorrect program behaviour (think 'black hat'). Either way, the curve fitting perspective is one way to understand why "software defects" are hard to "stamp out" in machine learning systems.

Again, the ability to identify adversarial inputs and then "stamp them out" has huge ramifications for safety-oriented systems like autonomous vehicles, or for high-stakes adversarial settings like cybersecurity and finance. Some machine learning scientists, like Cynthia Rudin, argue that high-stakes applications should use models whose correctness can be verified; if interested, you can read her article entitled *Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead*.

Measuring success is tricky. The most common failure mode for a machine learning project is to pursue a measure of success that is misleading, or that has no value, or that is hard to evaluate computationally, or that is easy to circumvent. This is a difficulty for software engineering generally, but it is especially hard for machine learning projects. Why? Because providing a measure of success that is explicit, relevant, computable, optimizable, and un-hackable, is harder than it seems.

A classic example of a misleading measure of success is to use "accuracy" on a highly imbalanced classification problem. For example, if you report "99.9% accuracy," this sounds impressive. But if 99.9% of your data is the same class (say, the "not breast cancer" class), then you have not actually succeeded in anything. If it is not yet clear why, it will be.

What does it mean to "circumvent" a measure of success? Well, if data is a teacher then machine learning algorithms are like lazy students: they will take any shortcut that permits success by your definition. When this happens, machine learning scientists will sometimes describe their system as having "cheated." It turns out that we humans are very bad at being expressing our objectives to a computer. George Dantzig, the inventor of linear programming, once tried to formally express the "the diet problem" in a computer, in an attempt to lose weight. The computer determined that, according to the constraints that he specified, he should drink 500 gallons of vinegar. Machine learning algorithms can take surprising shortcuts to solve prediction or design tasks. Below is a classic example, taken from *Reinforcement Learning for Improving Agent Design* (David Ha, Google Brain), where the machine learning algorithm is rewarded for designing an agent—both its body and its control system—that can walk "to the right," through a virtual obstacle course, as fast as possible. The learning algorithm discovered a shortcut: the *taller* robot is, the further it will *fall* to the right, and thereby "succeed" without needing a control system.



Similar shortcuts, or algorithmically-discovered hacks, have long been encountered by researchers working with evolutionary algorithms. For example, *The Surprising Creativity of Digital Evolution* describes virtual robots exploiting the same "tall robots fall over" strategy. That article also tells of an anecdote from 1997 where students competed to produce an agent that could play "infinite tic-tac-toe" by placing Xs or Os in an infinite grid. The winning team's algorithm discovered a surprising strategy: by placing its initial X at a huge coordinate in the "infinite grid," say at position $(2^{31}, 2^{31})$, many of the opposing agents would crash and lose by default, from trying to allocate memory for a bigger grid in memory.

Summary

From a software engineering standpoint, machine learning is akin to reverse-engineering a program from input-output examples, and doing so *automatically*. Like any software, a trained machine learning system must work for all inputs we might care about, not just for training inputs. Programs that work well on non-training inputs are said to *generalize well*.

Of the many programs that fit your training data, some will generalize well, and some will generalize poorly. A big part of your job as a machine learning practitioner will be to build systems that are likely to generalizes *well*, and to assess how well they are likely to generalize, even when you don't know all the conditions the system will be exposed to. People may some day depend on your system for their livelihoods, or even for their lives.