ENM 3600: Introduction to Data-driven Modeling

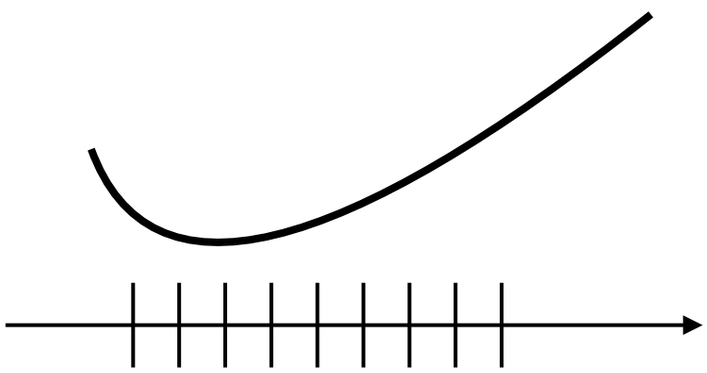
Lecture #9: Automatic Differentiation

Computing Derivatives

$$f : \mathbb{R}^d \to \mathbb{R}^s, \quad \nabla f = ?$$

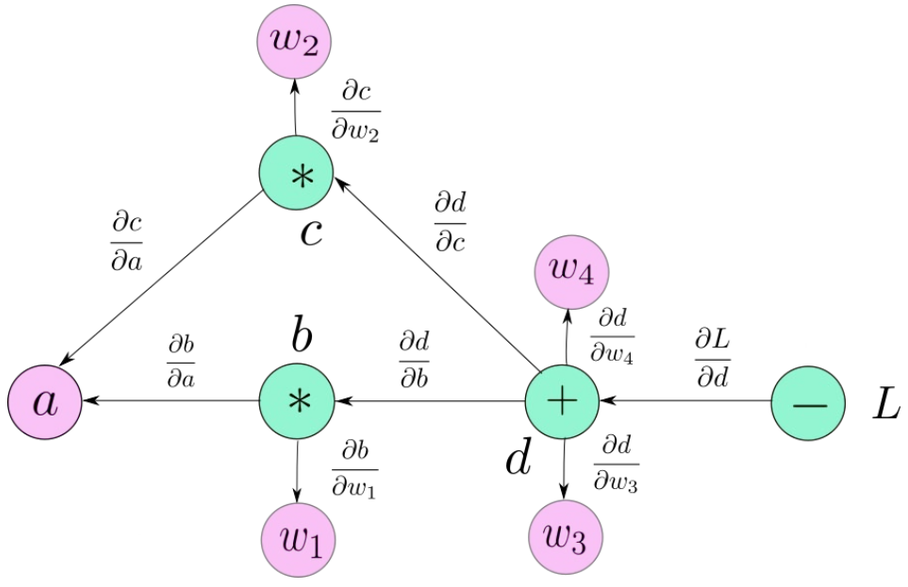
Analytic: $f(x) = 5x^5 + 4x^3 - 5 \quad \Rightarrow \quad f'(x) = 25x^4 + 12x^2$

Numerical:



$$f''(a) \approx \frac{f(a+h) - 2f(a) + f(a-h)}{h^2}$$

Algorithmic:



$$\frac{\partial h}{\partial x_i} = \sum_j \frac{\partial h}{\partial u_j} \cdot \frac{\partial u_j}{\partial x_i}$$

Automatic differentiation

https://matt-graham.github.io/slides/ad/index.html#/

6.5.2 Chain Rule of Calculus

The chain rule of calculus (not to be confused with the chain rule of probability) is used to compute the derivatives of functions formed by composing other functions whose derivatives are known. Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.

Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{6.44}$$

We can generalize this beyond the scalar case. Suppose that $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$, g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}. \tag{6.45}$$

In vector notation, this may be equivalently written as

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z, \tag{6.46}$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ Jacobian matrix of g.

From this we see that the gradient of a variable $\boldsymbol{x}$ can be obtained by multiplying a Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ by a gradient $\nabla_{\boldsymbol{y}} z$. The back-propagation algorithm consists of performing such a Jacobian-gradient product for each operation in the graph.

Automatic differentiation

Many contemporary algorithms require the evaluation of a derivative of a given differentiable function, f, at a given input value, $(x_1, \ldots, x_N)$, for example a gradient,

$$\left(\frac{\partial f}{\partial x_1} (x_1, \ldots, x_N) , \cdots , \frac{\partial f}{\partial x_N} (x_1, \ldots, x_N) \right) ,$$

or a directional derivative,[1]

$$\vec{v}(f) (x_1, \ldots, x_N) = \sum_{n=1}^{N} v_n \frac{\partial f}{\partial x_n} (x_1, \ldots, x_N) .$$

In its most basic description, automatic differentiation relies on the fact that all numerical computations are ultimately compositions of a finite set of elementary operations for which derivatives are known. Combining the derivatives of the constituent operations through the chain rule gives the derivative of the overall composition. This allows accurate evaluation of derivatives at machine precision with ideal asymptotic efficiency and only a small constant factor of overhead.

Carpenter, B., Hoffman, M. D., Brubaker, M., Lee, D., Li, P., & Betancourt, M. (2015). The stan math library: Reverse-mode automatic differentiation in C++. arXiv preprint arXiv:1509.07164.

Automatic differentiation

The chain rule, forward and reverse accumulation [edit]

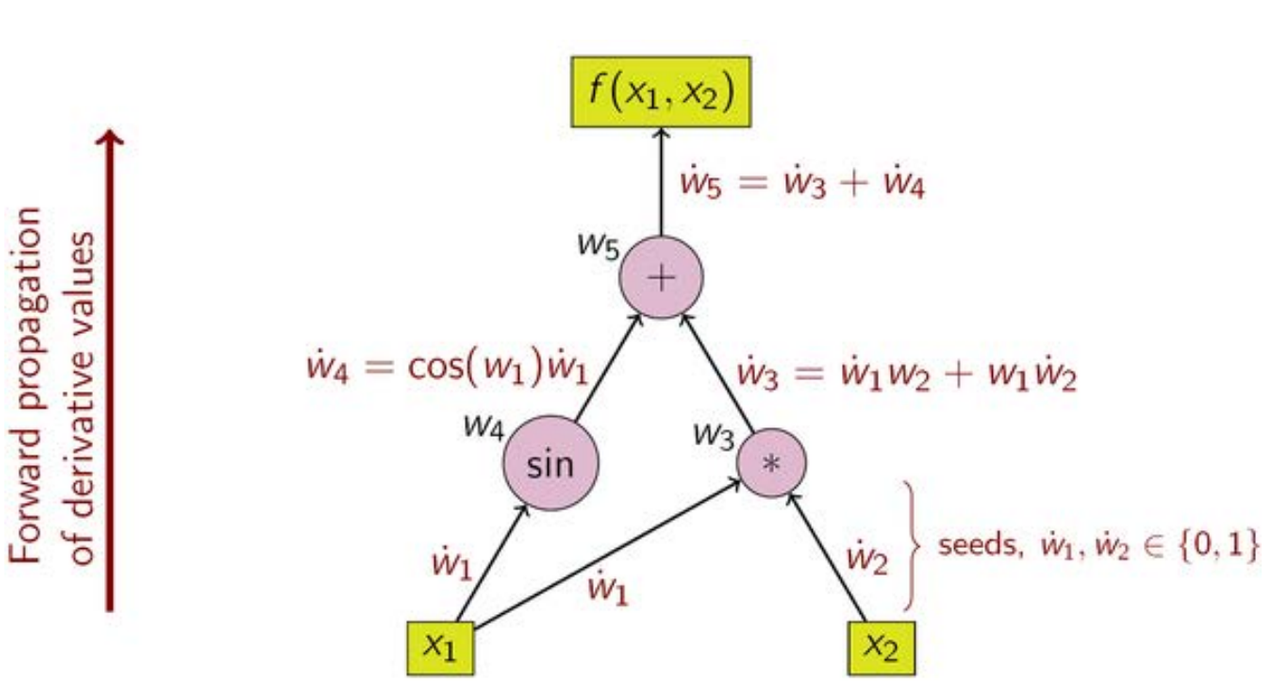
Fundamental to AD is the decomposition of differentials provided by the chain rule. For the simple composition $y = f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3$ the chain rule gives

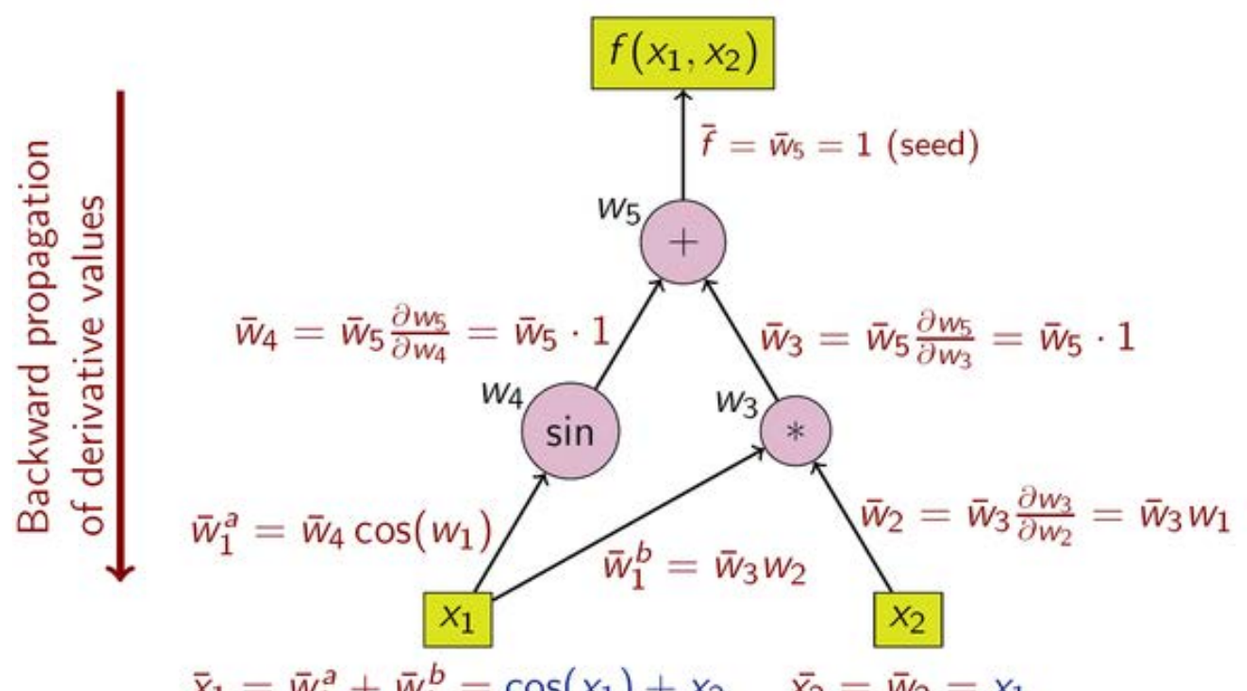
$$\frac{dy}{dx} = \frac{dy}{dw_2}\frac{dw_2}{dw_1}\frac{dw_1}{dx}$$

Usually, two distinct modes of AD are presented, **forward accumulation** (or **forward mode**) and **reverse accumulation** (or **reverse mode**). Forward accumulation specifies that one traverses the chain rule from inside to outside (that is, first compute dw_1/dx and then dw_2/dx and at last dy/dx), while reverse accumulation has the traversal from outside to inside (first compute dy/dw_2 and then dy/dw_1 and at last dy/dx). More succinctly,

1. **forward accumulation** computes the recursive relation: $\frac{dw_i}{dx} = \frac{dw_i}{dw_{i-1}}\frac{dw_{i-1}}{dx}$ with $w_3 = y$, and,
2. **reverse accumulation** computes the recursive relation: $\frac{dy}{dw_i} = \frac{dy}{dw_{i+1}}\frac{dw_{i+1}}{dw_i}$ with $w_0 = x$.

Example: $z = f(x_1, x_2) = x_1 x_2 + \sin x_1$





Automatic differentiation

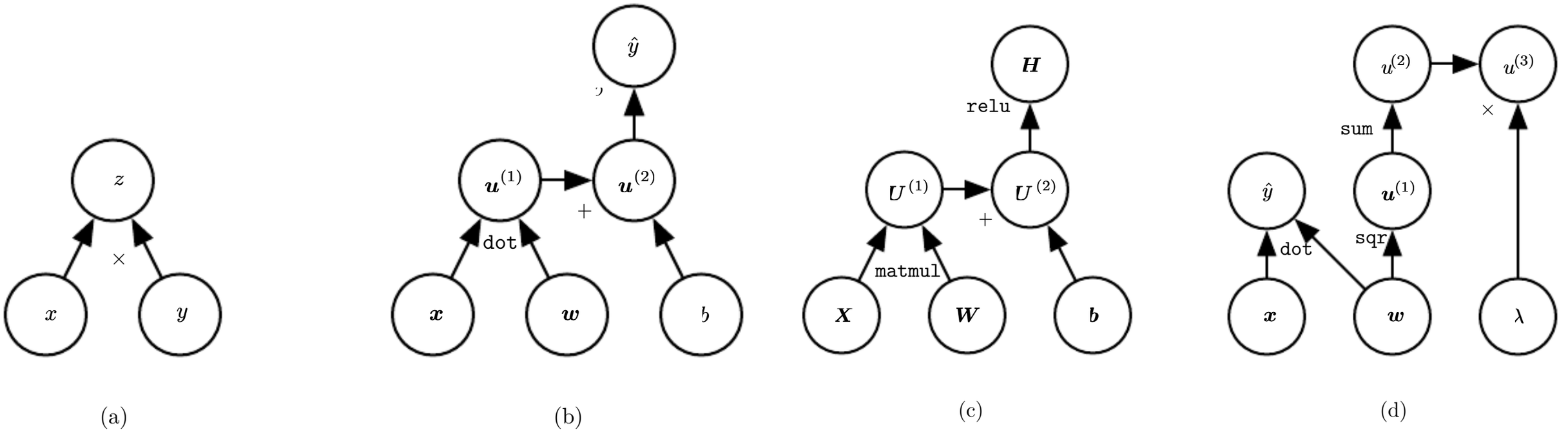


Figure 6.8: Examples of computational graphs. *(a)*The graph using the $\times$ operation to compute $z = xy$. *(b)*The graph for the logistic regression prediction $\hat{y} = \sigma\left(\boldsymbol{x}^\top \boldsymbol{w} + b\right)$. Some of the intermediate expressions do not have names in the algebraic expression but need names in the graph. We simply name the i-th such variable $\boldsymbol{u}^{(i)}$. *(c)*The computational graph for the expression $\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}\}$, which computes a design matrix of rectified linear unit activations $\boldsymbol{H}$ given a design matrix containing a minibatch of inputs $\boldsymbol{X}$. *(d)*Examples a–c applied at most one operation to each variable, but it is possible to apply more than one operation. Here we show a computation graph that applies more than one operation to the weights $\boldsymbol{w}$ of a linear regression model. The weights are used to make both the prediction $\hat{y}$ and the weight decay penalty $\lambda \sum_i w_i^2$.

https://colah.github.io/posts/2015-08-Backprop/

Automatic differentiation

It is one of the most useful - and perhaps underused - tools in modern scientific computing!

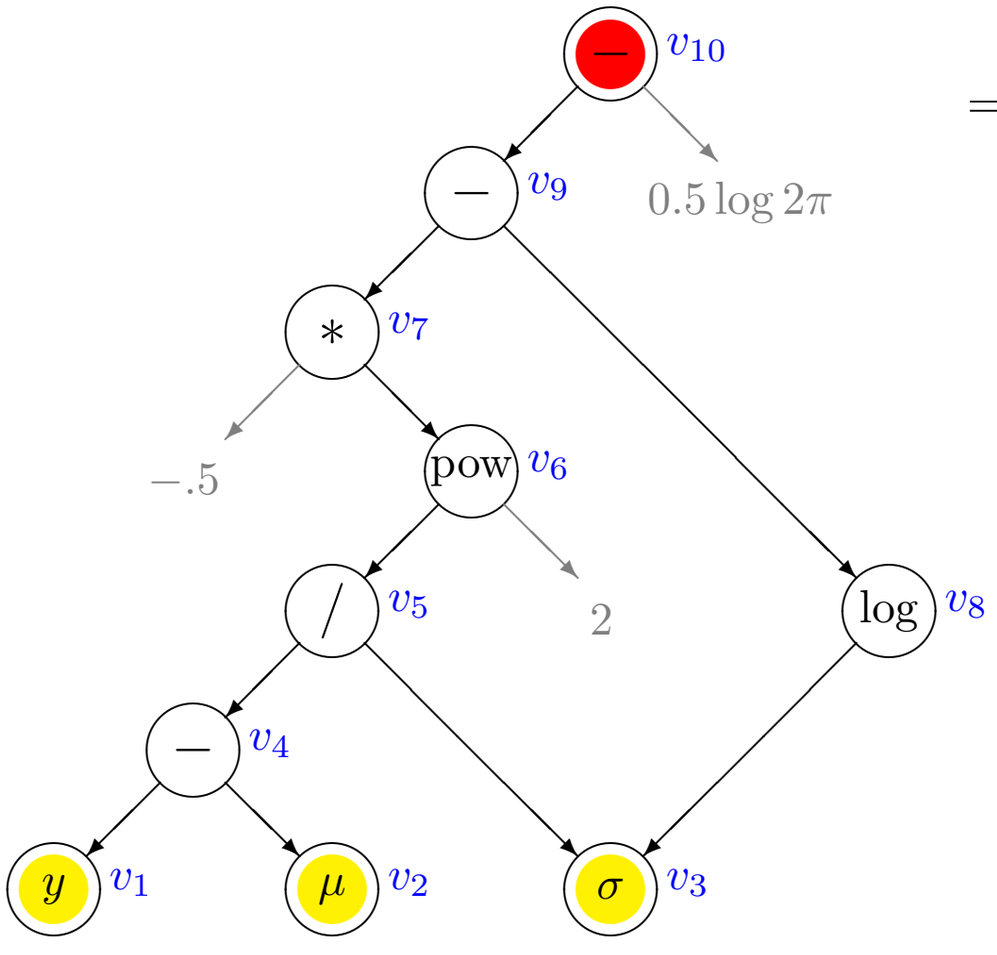
Applications:

- real-parameter optimization (many good methods are gradient-based)
- sensitivity analysis (local sensitivity = ∂(result)/∂(input))
- physical modeling (forces are derivatives of potentials; equations of motion are derivatives of Lagrangians and Hamiltonians; etc.)
- probabilistic inference (e.g., Hamiltonian Monte Carlo)
- machine learning
- and who knows how many other scientific computing applications.

Automatic differentiation

As an example, consider the log of the normal probability density function for a variable y with a normal distribution with mean μ and standard deviation σ,

$$f(y, \mu, \sigma) = \log\left(\mathsf{Normal}(y|\mu, \sigma)\right) = -\frac{1}{2}\left(\frac{y-\mu}{\sigma}\right)^2 - \log\sigma - \frac{1}{2}\log(2\pi) \qquad (1)$$



var	*value*	*partials*
v_1	y	
v_2	μ	
v_3	σ	
v_4	$v_1 - v_2$	$\partial v_4/\partial v_1 = 1$ $\quad$ $\partial v_4/\partial v_2 = -1$
v_5	v_4/v_3	$\partial v_5/\partial v_4 = 1/v_3$ $\quad$ $\partial v_5/\partial v_3 = -v_4 v_3^{-2}$
v_6	$(v_5)^2$	$\partial v_6/\partial v_5 = 2v_5$
v_7	$(-0.5)v_6$	$\partial v_7/\partial v_6 = -0.5$
v_8	$\log v_3$	$\partial v_8/\partial v_3 = 1/v_3$
v_9	$v_7 - v_8$	$\partial v_9/\partial v_7 = 1$ $\quad$ $\partial v_9/\partial v_8 = -1$
v_{10}	$v_9 - (0.5 \log 2\pi)$	$\partial v_{10}/\partial v_9 = 1$

Carpenter, B., Hoffman, M. D., Brubaker, M., Lee, D., Li, P., & Betancourt, M. (2015). The stan math library: Reverse-mode automatic differentiation in C++. arXiv preprint arXiv:1509.07164.

Introduction to JAX

JAX is a Python library which augments numpy and Python code with *function transformations* which make it trivial to perform operations common in machine learning programs. Concretely, this makes it simple to write standard Python/numpy code and immediately be able to:

- Compute the derivative of a function via a successor to autograd
- Just-in-time compile a function to run efficiently on an accelerator via XLA
- Automagically vectorize a function, so that e.g. you can process a "batch" of data in parallel

https://jax.readthedocs.io/en/latest/notebooks/quickstart.html