An AI-Driven Revolution in Scientific Computing

John Stachurski

RBA Nov 2024

Topics

Part 1: Slides

- AI-driven scientific computing
- Applications

Part 2: Hands on coding

`https://github.com/QuantEcon/rba_workshop_2024`

AI-driven scientific computing

AI is changing the world

- image processing / computer vision
- speech recognition, translation
- scientific knowledge discovery
- forecasting and prediction
- generative AI

Plus killer drones, skynet, etc....

AI-driven scientific computing

AI is changing the world

- image processing / computer vision
- speech recognition, translation
- scientific knowledge discovery
- forecasting and prediction
- generative AI

Plus killer drones, skynet, etc.…

Projected spending on AI in 2024:

- Google: \$50 billion
- Microsoft: \$60 billion
- Meta: \$40 billion
- etc.

Software component is mainly open source

What does this software do?

Deep learning in two slides

Aim: approximate an unknown functional relationship

$$y = f(x) \qquad (x \in \mathbb{R}^k,\ y \in \mathbb{R})$$

Examples.

- $x =$ cross section of returns, $y =$ return on oil futures tomorrow
- $x =$ weather sensor data, $y =$ max temp tomorrow

Problem:

- observe $(x_i, y_i)_{i=1}^n$ and seek f such that $y_{n+1} \approx f(x_{n+1})$

Nonlinear regression: choose model $\{f_\theta\}_{\theta\in\Theta}$ and minimize the empirical loss

$$\ell(\theta) := \sum_{i=1}^{n}(y_i - f_\theta(x_i))^2 \quad \text{s.t.} \quad \theta \in \Theta$$

In the case of ANNs, we consider all f_θ having the form

$$f_\theta = \sigma \circ A_m \circ \cdots \circ \sigma \circ A_2 \circ \sigma \circ A_1$$

where

- $A_j x = W_j x + b_j$ is an affine map
  - `output = dot(kernel, input) + bias`
- σ is a nonlinear "activation" function

Nonlinear regression: choose model $\{f_\theta\}_{\theta\in\Theta}$ and minimize the empirical loss

$$\ell(\theta) := \sum_{i=1}^{n}(y_i - f_\theta(x_i))^2 \quad \text{s.t.} \quad \theta \in \Theta$$

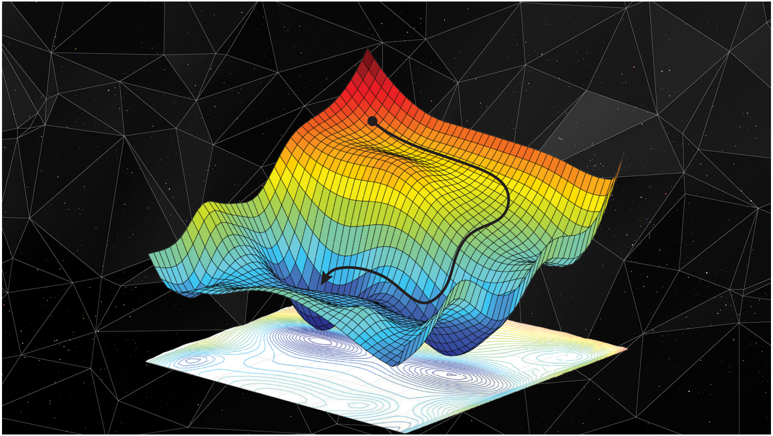
In the case of ANNs, we consider all f_θ having the form

$$f_\theta = \sigma \circ A_m \circ \cdots \circ \sigma \circ A_2 \circ \sigma \circ A_1$$

where

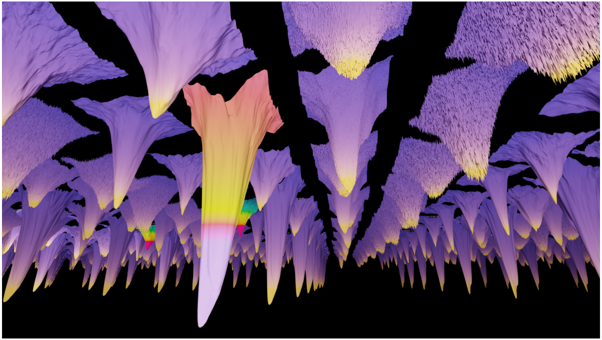
- $A_j x = W_j x + b_j$ is an affine map
    - `output = dot(kernel, input) + bias`
- σ is a nonlinear "activation" function

Minimizing a smooth loss functions – what algorithm?



Source: https://danielkhv.com/

Deep learning: $\theta \in \mathbb{R}^d$ where $d = ?$



Source: https://losslandscape.com/gallery/

How does it work?

Why is it possible to minimize over $\theta \in \mathbb{R}^d$ when $d = 10^{12}$?!?

Core elements

- automatic differentiation (for gradient descent)
- parallelization (GPUs or TPUs)
- Compilers / JIT-compilers

How does it work?

Why is it possible to minimize over $\theta \in \mathbb{R}^d$ when $d = 10^{12}$?!?

Core elements

- automatic differentiation (for gradient descent)
- parallelization (GPUs or TPUs)
- Compilers / JIT-compilers

Automatic differentiation

"Exact numerical" differentiation

```
def loss(θ, x, y):
  return jnp.sum((y - f(θ, x))**2)

loss_gradient = grad(loss)
```

Now use gradient descent...

Parallelization

```
outputs = pmap(f, data)
```

- multithreading over GPU cores (how many?)
- multiprocessing over accelerators in a GPU farm / supercomputing cluster (how many?)

Just-in-time compilers

```python
@jit
def f(x):
    return jnp.sin(x) - jnp.cos(x**2)
```

Advantages:

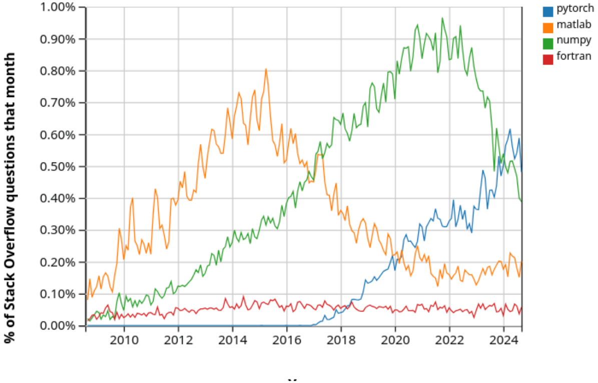
- compiler needs less type information
- can specialize based on parameter types / shapes
- can automatically specialize to CPUs / GPU / TPU

Platforms

Platforms that support AI / deep learning:

- Tensorflow
- PyTorch (Llama, ChatGPT)
- Google JAX (Gemini, DeepMind)
- Keras (backends = JAX, PyTorch)
- Mojo? (Modular (Python))
- MATLAB?

Popularity


1.00%
0.90%
0.80%
0.70%
0.60%
0.50%
0.40%
0.30%
0.20%
0.10%
0.00%
% of Stack Overflow questions that month
2010
2012
2014
2016
2018
2020
2022
2024
pytorch
matlab
numpy
fortran

Focus on JAX

https://jax.readthedocs.io/en/latest/

- Just-in-time compilation
- Automatic differentiation
- Xccelerated linear algebra

```python
import jax.numpy as jnp
from jax import grad, jit

def f(θ, x):
  for W, b in θ:
    w = W @ x + b
    x = jnp.tanh(w)
  return x

def loss(θ, x, y):
  return jnp.sum((y - f(θ, x))**2)

grad_loss = jit(grad(loss))  # Now use gradient descent
```

Example. AlphaFold3 (Google JAX)

Highly accurate protein structure prediction with AlphaFold

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool,...

<u>Nature</u> Vol. 596 (2021)

- Citation count = 30K
- Nobel Prize in Chemistry 2024

AI tools for economic modeling

Let's say that you want to do computational macro rather than deep learning per se

Can these new AI tools be applied?

Answer: Yes!

- fast matrix algebra
- fast solutions to linear systems
- fast nonlinear system solvers
- fast optimization, etc.

AI tools for economic modeling

Let's say that you want to do computational macro rather than deep learning per se

Can these new AI tools be applied?

Answer: Yes!

- fast matrix algebra
- fast solutions to linear systems
- fast nonlinear system solvers
- fast optimization, etc.

Advantages of JAX (vs PyTorch / Numba / etc.) for economists:

- exposes low level functions
- elegant functional programming style – close to maths
- elegant autodiff tools
- array operations follow standard NumPy API
- automatic parallelization
- same code, multiple backends (CPUs, GPUs, TPUs)

Case Study

The CBC uses the "overborrowing" model of Bianchi (2011)

- credit constraint loosens during booms
- bad shocks $\rightarrow$ sudden stops

CBC implementation in MATLAB

- runs on \$10,000 mainframe with 356 CPUs and 1TB RAM
- runtime $=$ 12 hours

Rewrite in Python + Google JAX

- runs on \$400 gaming GPU with 10GB RAM
- runtime $=$ 4.17 seconds

Case Study

The CBC uses the "overborrowing" model of Bianchi (2011)

- credit constraint loosens during booms
- bad shocks $\rightarrow$ sudden stops

CBC implementation in MATLAB

- runs on $10,000 mainframe with 356 CPUs and 1TB RAM
- runtime $=$ 12 hours

Rewrite in Python $+$ Google JAX

- runs on $400 gaming GPU with 10GB RAM
- runtime $=$ 4.17 seconds

Live coding

See notebook `fun_with_jax.ipynb` in

`https://github.com/QuantEcon/rba_workshop_2024`

Steps

1. Go to Google Colab (`https://colab.google/`)
2. Open notebook → GitHub → quantecon → `rba_workshop_2024` → `fun_with_jax.ipynb`
3. Edit → Notebook settings → select a GPU
4. Runtime → Run all