JAX.
AN INTRODUCTION TO DEEP LEARNING PROGRAMMING PRINCIPLES

SAMUELE PAPA

WHO AM I?

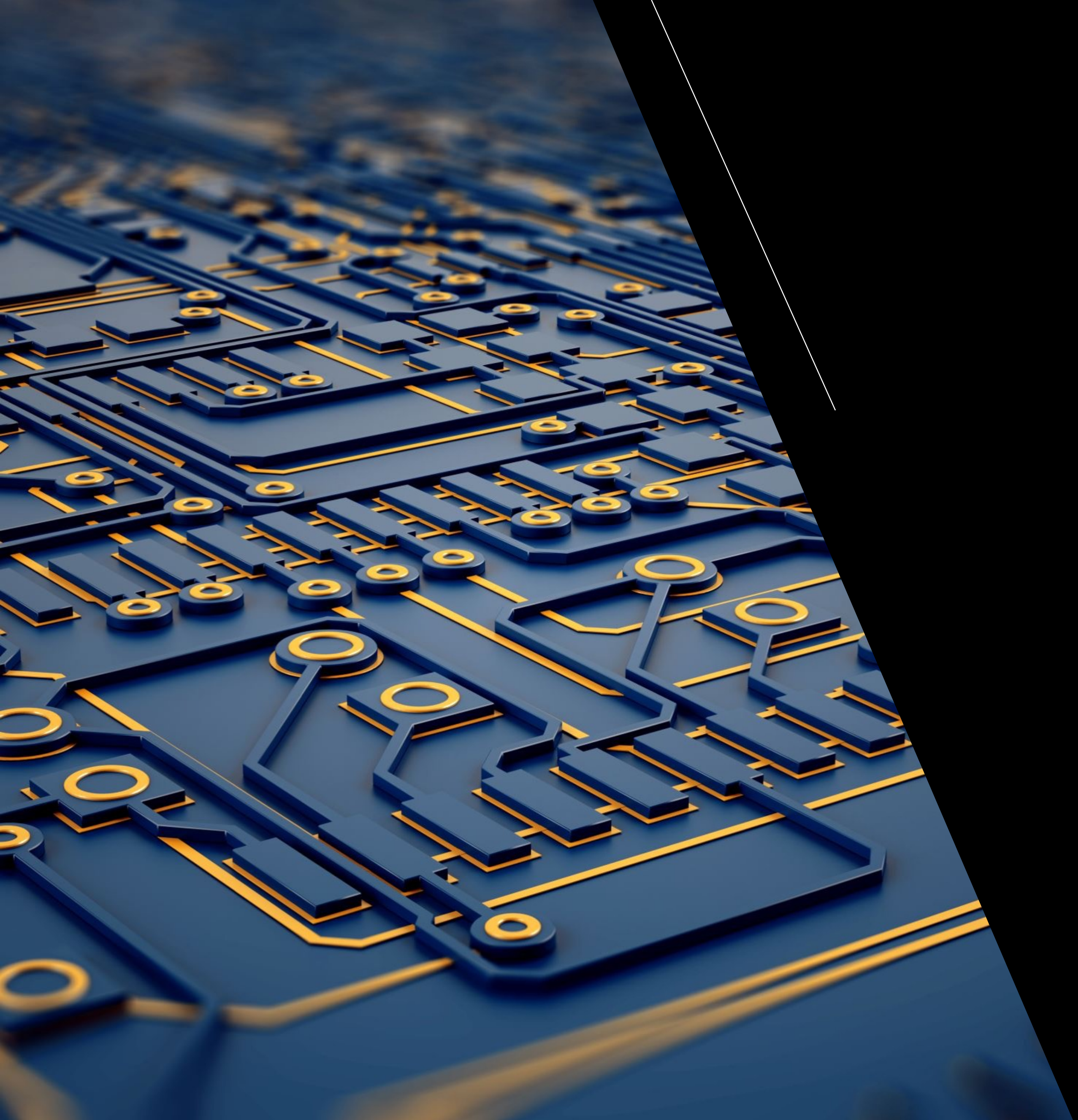
PhD student @ UvA & NKI

- Real-world applications of Deep Learning
- Scientific modeling
- Medical imaging

Contact: s.papa@uva.nl

INSPIRATION AND SOURCES

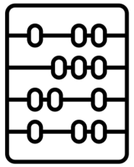
- Phillip Lippe: notebooks, presentations, and more.
- Official JAX documentation.
- Other sources.



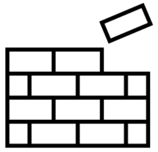
KEY ASPECTS OF MODERN COMPUTING

KEY ASPECTS OF MODERN COMPUTING

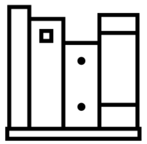
Parallelization



Compute



Memory



HOW CAN WE IMPROVE?

Manual optimization

Use **algorithms** and **data structures**.

↓

e.g. async loading and preprocessing of data on CPU

e.g. hash maps for spatial embeddings

Leverage compilers

HOW CAN COMPILERS HELP?

T4 RAM Disk

[4]

```
import jax
import jax.numpy as jnp

def selu(x, alpha=1.67, lambda_=1.05):
  return lambda_ * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)

x = jnp.arange(1000000)
%timeit selu(x).block_until_ready()
```

```
1.02 ms ± 324 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

HOW CAN COMPILERS HELP?

T4 RAM Disk

[2]

```python
selu_jit = jax.jit(selu)

# Pre-compile the function before timing...
selu_jit(x).block_until_ready()

%timeit selu_jit(x).block_until_ready()
```

```
264 µs ± 64.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

WHY?

T4 RAM Disk

Each operation (approximately) calls a new kernel

[3]

```
def selu(x, alpha=1.67, lambda_=1.05):
    original_x = x
    x = jnp.exp(x)
    x = alpha * x
    x = x - alpha
    x = jnp.where(original_x > 0, original_x, x)
    x = lambda_ * x
    return x

x = jnp.arange(1000000)
%timeit selu(x).block_until_ready()
```

```
922 µs ± 114 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

JUST-IN-TIME COMPILATION

JIT. Compile code during execution.

Simply use `jax.jit()`

Compiles the function by converting to intermediate `jaxprs` language.

Tracks usage and optimizes also memory.

FUNCTIONAL PROGRAMMING

PURE FUNCTIONS

Example of **classic object-oriented** design pattern.
Often encountered when using classes.

```
[4]
counter = 0
def increase_counter_by(x):
    return counter + x

print(increase_counter_by(12))
```

```
12
```

```
[5]
counter = 10
print(increase_counter_by(12))
```

```
22
```

PURE FUNCTIONS

Now we compile it.

```
[6]  jit_increase_counter_by = jax.jit(increase_counter_by)
     print(jit_increase_counter_by(10))
```

```
20
```

```
[7]  counter = 0
     print(jit_increase_counter_by(12))
```

```
22
```

PURE FUNCTIONS

A pure function is a function that, given the **same input**, will always return the **same output** and does not have any observable **side effect**.

A side effect *is e.g. something that is performed in-place, affects something outside the scope of the function.*

WHY PURE FUNCTIONS?

1. Makes your code more maintainable.
2. Makes compilation possible and simple.
3. Makes parallelization easier.
4. You can replace the whole function with its outputs when necessary.
5. Functional composition makes math-to-code easier.

HOW TO WRITE JAX

JAX IS NUMPY

`array` concept, just as in `numpy`.

```
[9]  1 import jax.numpy as jnp
     2 a = jnp.zeros((2, 5), dtype=jnp.float32)
     3 print(a)
```

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

All `numpy` functions are available.

API matches.

```
[10]  1 b = jnp.arange(6)
      2 print(b)
```

```
[0 1 2 3 4 5]
```

JAX IS NUMPY

`Array` objects are always placed directly on the available accelerators

```
[13]  1 b.devices()
```

```
{cuda(id=0)}
```

When we retrieve from device, it becomes a `numpy` array.

```
[14]  1 b_cpu = jax.device_get(b)
      2 print(b_cpu.__class__)
```

```
<class 'numpy.ndarray'>
```

PARALLELIZATION

JAX HAS AUTOMATIC VECTORIZATION

Simple parallelization of operations
using `jax.vmap()`

An example of simple element-wise operation.

[13]

```python
def single_linear(x, w, b):
    return (x[:,None] * w).sum(axis=0) + b

# Example inputs
x_in = jnp.ones((4,))
w_in = jnp.ones((4, 3))
b_in = jnp.ones((3,))

single_linear(x_in, w_in, b_in)
```

```
Array([5., 5., 5.], dtype=float32)
```

JAX HAS AUTOMATIC VECTORIZATION

Input shapes

[5,4] - batched

[4,3] - shared

[4] - shared

Output shapes

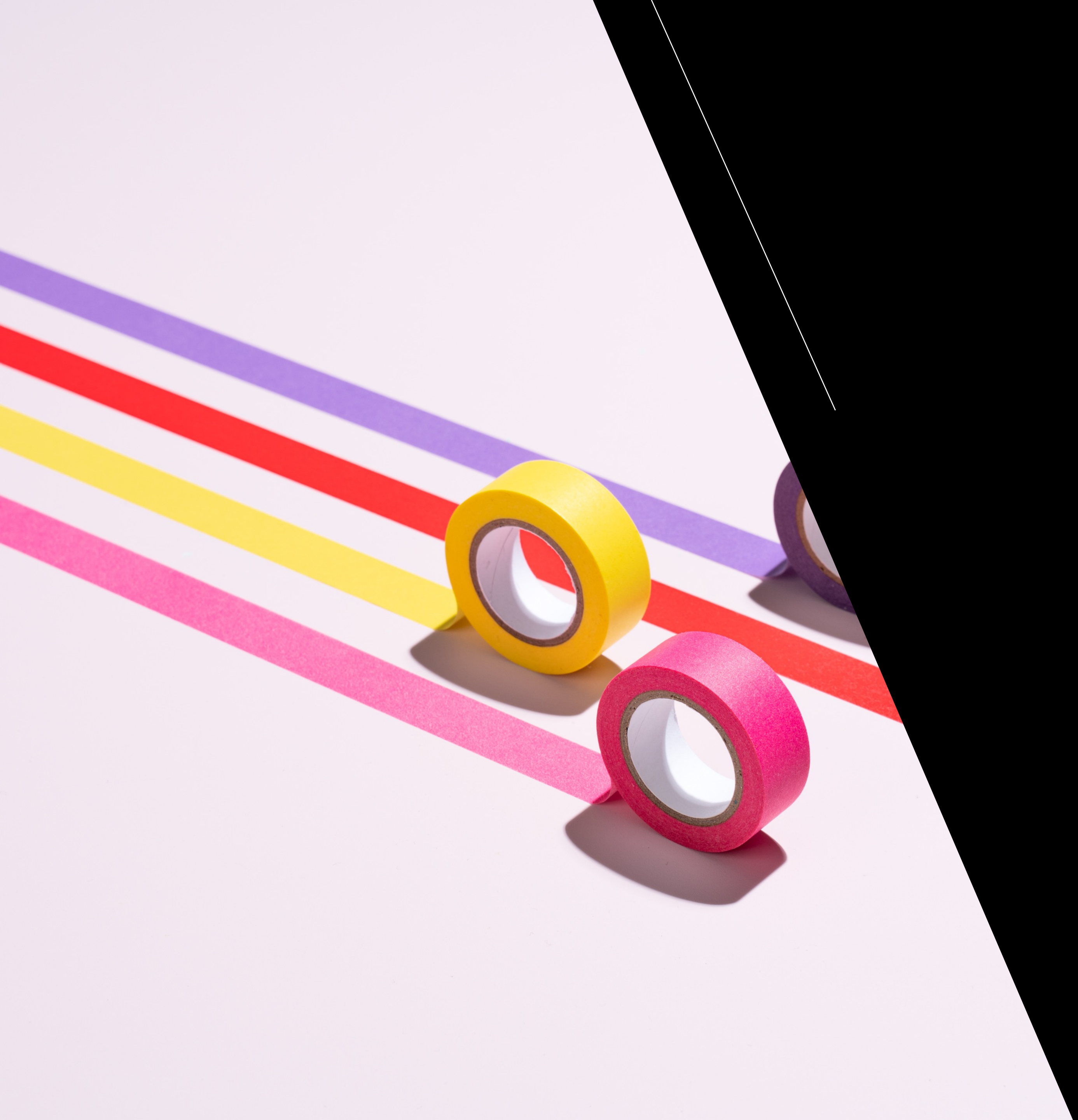
[4,3] - batched

```
[14]
batched_linear = jax.vmap(
    single_linear,
    in_axes=(0,None,None),
    out_axes=0
)

# Example batched inputs
batched_x_in = jnp.ones((5, 4,))

batched_linear(batched_x_in, w_in, b_in)
```

```
Array([[5., 5., 5.],
       [5., 5., 5.],
       [5., 5., 5.],
       [5., 5., 5.],
       [5., 5., 5.]], dtype=float32)
```



PARALLELIZATION

FUNCTIONAL COMPUTATION OF GRADIENTS

The `jax.grad()` function returns the function that evaluates the derivative at any given input

```
[15]
def rms_error(x, y):
    return jnp.sqrt(jnp.mean((x-y)**2))

x_in = jnp.array([1.2,3.2,4], dtype=jnp.float32)
y_target = jnp.array([0,5.,10.], dtype=jnp.float32)

grad = jax.grad(rms_error)
grad(x_in, y_target)
```

```
Array([ 0.1086251 , -0.16293764, -0.54312545], dtype=float32)
```

GRADIENT DESCENT

Very intuitive implementation from math to code.

[22]

```
lambda_ = 0.05
x_new = x_in
for i in range(200):
    x_new = x_new - lambda_ * grad(x_new, y_target)
    if i % 10 == 0:
        print(rms_error(x_new, y_target))
```

```
3.6657236
3.4990568
3.3323894
3.1657221
2.9990551
2.8323882
2.665721
2.4990537
2.3323865
2.1657195
1.9990524
1.8323854
1.6657186
1.4990516
1.3323847
1.1657186
0.9990542
0.8323898
0.6657253
0.49906078
```

[23]

```
x_new
```

```
Array([0.11374928, 4.829371  , 9.431246  ], dtype=float32)
```

GRADIENT DESCENT

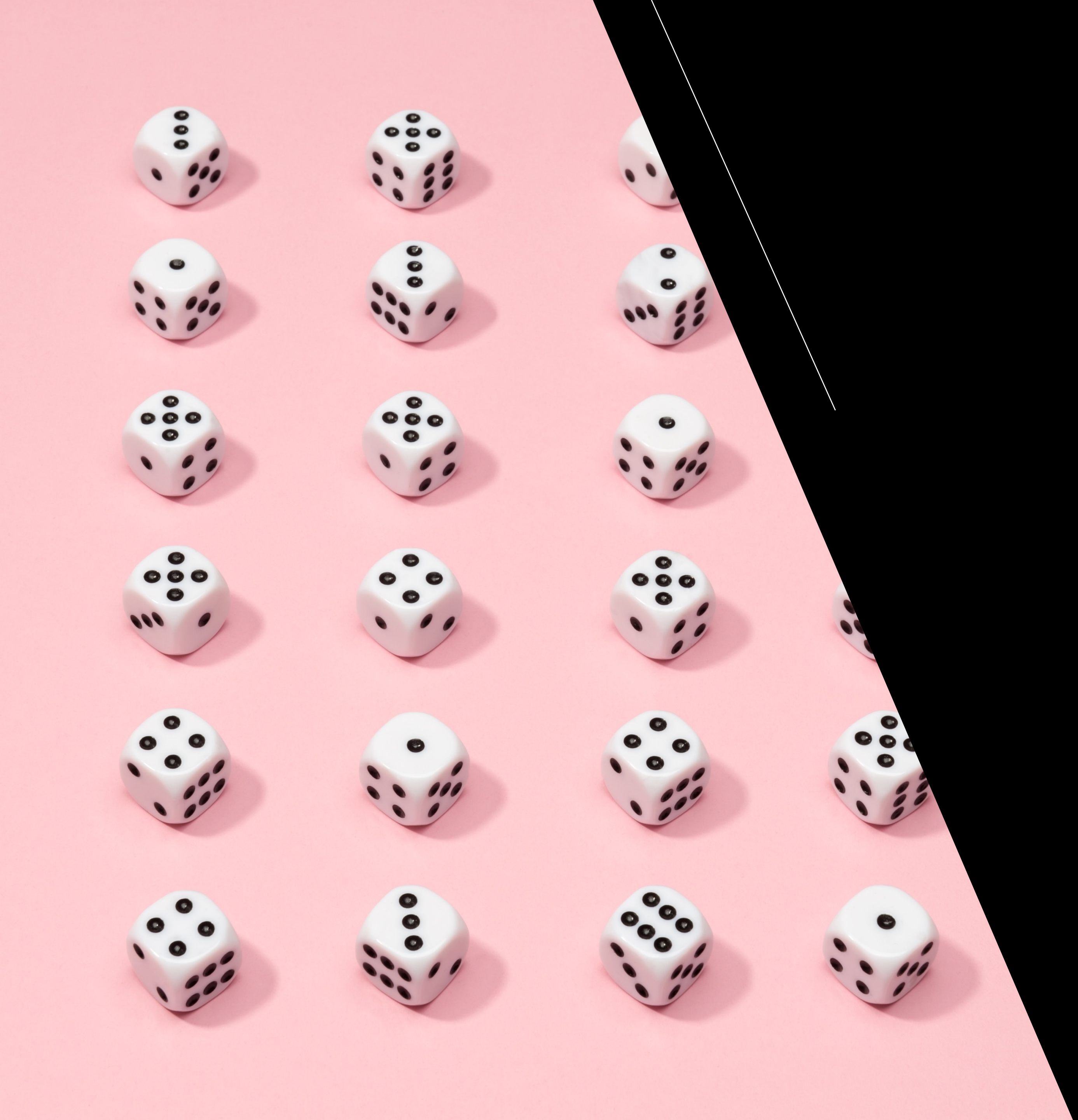
Improving the performance a bit with some heuristic annealing

```
[18]
lambda_ = 0.1
annealing = 0.75
steps = 50
x_new = x_in
for i in range(1000):
    x_new = x_new - lambda_ * grad(x_new, y_target)
    if (i + 1) % steps == 0:
        lambda_ *= annealing
        print(rms_error(x_new, y_target))
```

```
2.0157194
0.7657221
0.015718736
0.012406095
0.008687421
0.0071328944
0.0047321706
0.0041667605
0.002507655
0.0024983105
0.0012557198
0.0012558495
0.0008558435
0.0007279681
0.00046012658
0.00043075677
0.00023763097
0.00023762452
0.00013843694
0.00013850731
```

```
[19]
x_new
```

```
Array([4.142728e-05, 4.999917e+00, 9.999779e+00], dtype=float32)
```

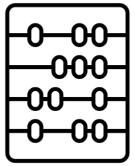


PSEUDO-RANDOM NUMBER GENERATION

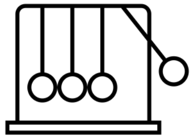
THE GOALS OF PSEUDO-RANDOM NUMBER GENERATOR

Reproducible

Parallelizable



Vectorizable



HOW IT IS USUALLY DONE

Set a global seed.

How does it behave on multiple devices?

What happens with intermediate steps of random sampling?

```
[21]
import numpy as np
import torch
np.random.seed(0)
torch.manual_seed(0)
```

```
<torch._C.Generator at 0x7f32c06c33d0>
```

WHEN DOES THE GLOBAL SEED FAIL?

Order of operations is not guaranteed.

Especially in parallel computations.

```
[22]
import numpy as np

np.random.seed(0)

def bar(): return np.random.uniform()
def baz(): return np.random.uniform()

def foo(): return bar() + 2 * baz()

print(foo())
```

```
1.979192236721637
```

WHEN DOES THE GLOBAL SEED FAIL?

[26]

```python
import numpy as np

np.random.seed(0)

def bar(): return np.random.normal()
def baz(): return np.random.uniform()

def foo(): return bar() + baz()

print(foo())
```

```
2.366815722039308
```

[27]

```python
import numpy as np

np.random.seed(0)

def bar(): return np.random.uniform()
def baz(): return np.random.normal()

def foo(): return bar() + baz()

print(foo())
```

```
1.290405244736486
```

Same operation, different results.

USE PRNG KEYS

Key: used by pseudo-random number generator to actually create randomness.

```
[28]
from jax import random

key = random.key(42)
print(key)
```

```
Array((), dtype=key<fry>) overlaying:
[ 0 42]
```

Given a key, the output of the random operation is always the same.

```
[29]
print(random.uniform(key))
print(random.uniform(key))
```

```
0.42672753
0.42672753
```

Same is possible in `numpy` and `torch` using generators.

DID WE SOLVE THE PROBLEM?

[31]

```python
key = random.key(42)

def bar(key): return random.uniform(key)
def baz(key): return random.normal(key)

def foo(key): return bar(key) + baz(key)

print(foo(key))
```

```
0.24201576
```

[32]

```python
key = random.key(42)

def bar(key): return random.normal(key)
def baz(key): return random.uniform(key)

def foo(key): return bar(key) + baz(key)

print(foo(key))
```

```
0.24201576
```

SUMMARY

1. **Compilation** = free code optimization.
2. **Functional** programming is powerful.
3. **Vectorization** to explicitly batch operations.
4. JAX at the core is numpy with **autograd.**
5. Reliable **pseudo-RNG** with keys.

THANK YOU!

Samuele Papa.
Open to chat and collaborate!
Looking for internship opportunities.

samuele.papa@gmail.com
samuelepapa.github.io