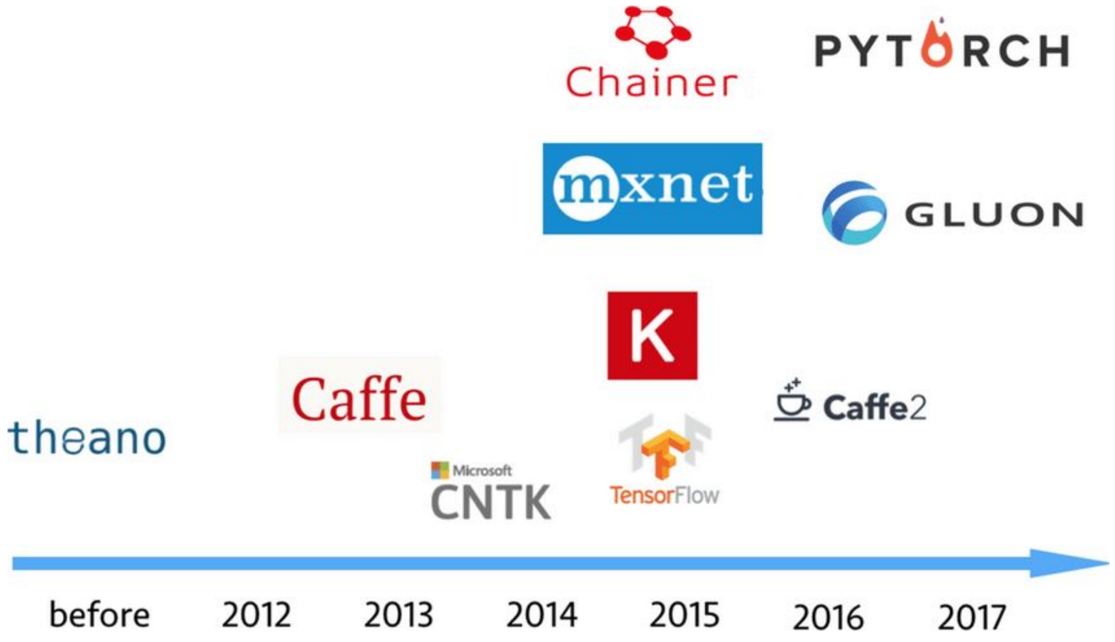
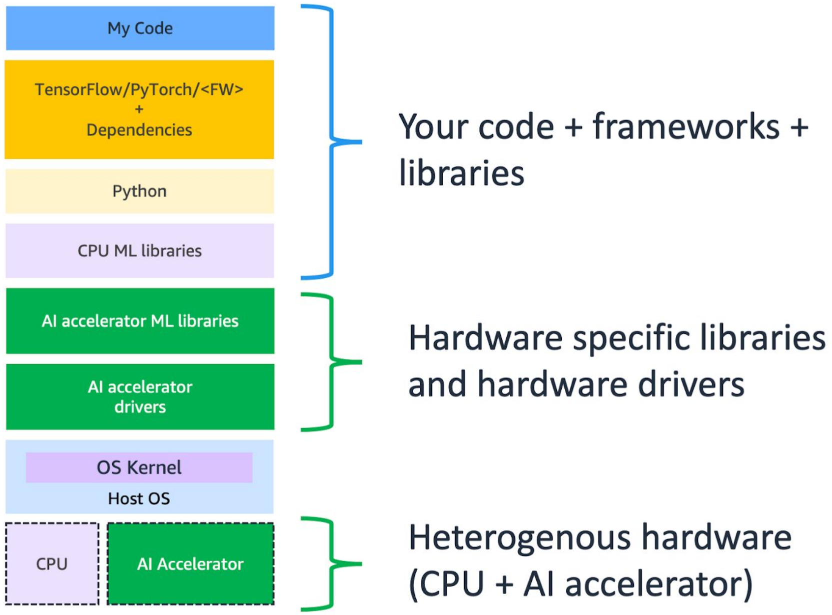
Heuristic optimization with heterogeneous AI frameworks

Zsolt Kisander
PTE-MIK, Department of Automation

Timeline of “new wave” AI frameworks

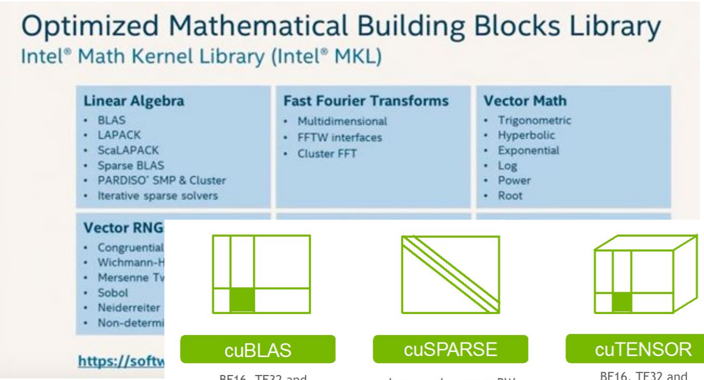


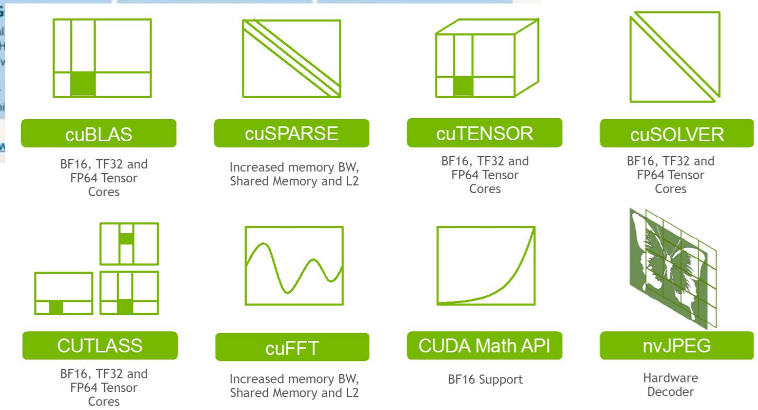
Typical AI framework architecture



Out of box hardware acceleration

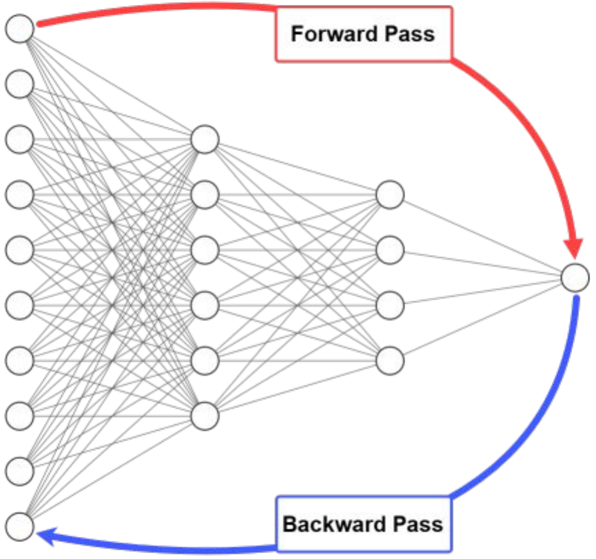
- Modern AI frameworks are built on top of hardware accelerated math libraries (MKL, CUDA, etc.)
- They provide easy to use, high level APIs
- Drop-in replacement for popular Python math and ML libraries
- Seamless transition between different computing architectures (deployment)



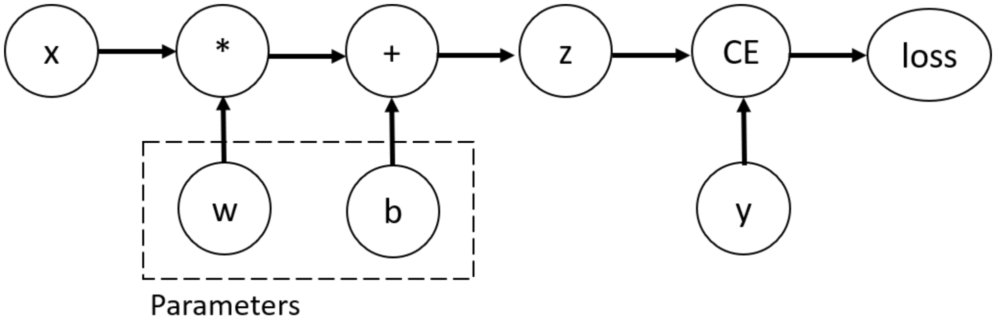


Autodiff and autograd

- Serves as the basis of the backpropagation algorithm, but not restricted to neural networks
- PyTorch module: torch.autograd
- Automatic differentiation of arbitrary scalar valued functions (loss fn)
- Supports automatic computation of gradient for any computational graph
- If we define a computational model with a scalar valued performance metric, then we can use autograd to tune the model parameters



Autodiff and autograd



$$z = wx + b$$

$$y = \frac{1}{1 + \exp(-z)}$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

$$t_1 = wx$$

$$z = t_1 + b$$

$$t_3 = -z$$

$$t_4 = \exp(t_3)$$

$$t_5 = 1 + t_4$$

$$y = 1/t_5$$

$$t_6 = y - t$$

$$t_7 = t_6^2$$

$$\mathcal{L} = t_7/2$$

Motivation - Summary

- AI frameworks provide high-level abstraction over mathematical libraries
- Uniform API across different computing architectures
- Not restricted to neural network models
- Hardware acceleration can be used to solve non-AI problems

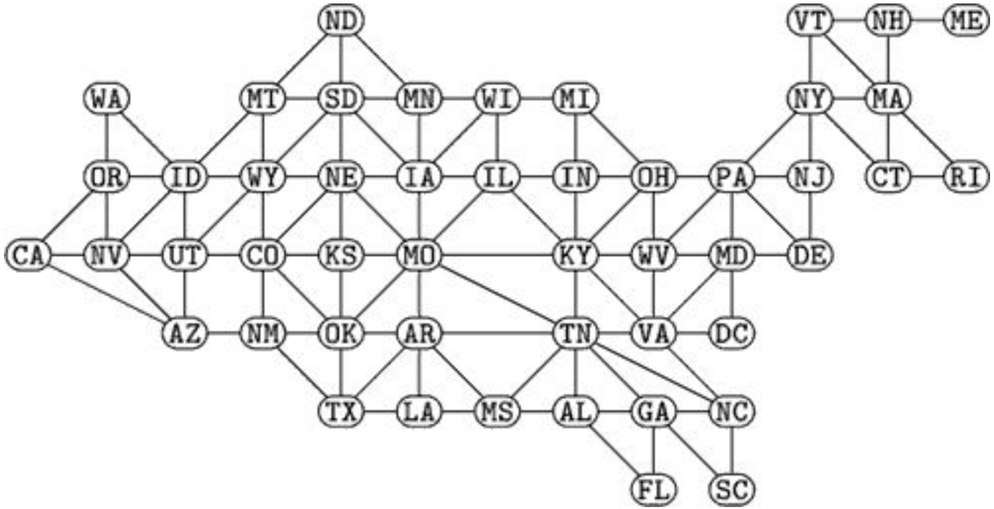
Example - Four coloring

- No more than four colors are required to color the regions of any map so that no two adjacent regions have the same color
- In graph-theoretic terms, the theorem states that for loopless planar graph, its chromatic number is less than or equal four

Example - US states

Graph of US states

	Alabama	Arizona	Arkansas	California	Colorado	Connecticut	Delaware	District of Columbia	Florida	Georg
Alabama	0	0	0	0	0	0	0	0	1	
Arizona	0	0	0	1	1	0	0	0	0	
Arkansas	0	0	0	0	0	0	0	0	0	
California	0	1	0	0	0	0	0	0	0	
Colorado	0	1	0	0	0	0	0	0	0	
Connecticut	0	0	0	0	0	0	0	0	0	
Delaware	0	0	0	0	0	0	0	0	0	
District of Columbia	0	0	0	0	0	0	0	0	0	
Florida	1	0	0	0	0	0	0	0	0	
Georgia	1	0	0	0	0	0	0	0	1	
Idaho	0	0	0	0	0	0	0	0	0	
Illinois	0	0	0	0	0	0	0	0	0	
Indiana	0	0	0	0	0	0	0	0	0	
Iowa	0	0	0	0	0	0	0	0	0	
Kansas	0	0	0	0	1	0	0	0	0	
Kentucky	0	0	0	0	0	0	0	0	0	
Louisiana	0	0	1	0	0	0	0	0	0	
Maine	0	0	0	0	0	0	0	0	0	
Maryland	0	0	0	0	0	0	1	1	0	
Massachusetts	0	0	0	0	0	1	0	0	0	
Michigan	0	0	0	0	0	0	0	0	0	



Problem formulation in PyTorch I.

Let A be the m×m adjacency matrix of a finite, loopless planar graph. Let C be an m×n matrix, where each row represents a vertex and each row vector is a one-hot encoded color vector candidate. In Torch, the default variable is the “tensor”.

Initially, fill C with i.i.d. random numbers.
Apply row-wise softmax normalization after each update on C to ensure one-hot encoding for row vectors.

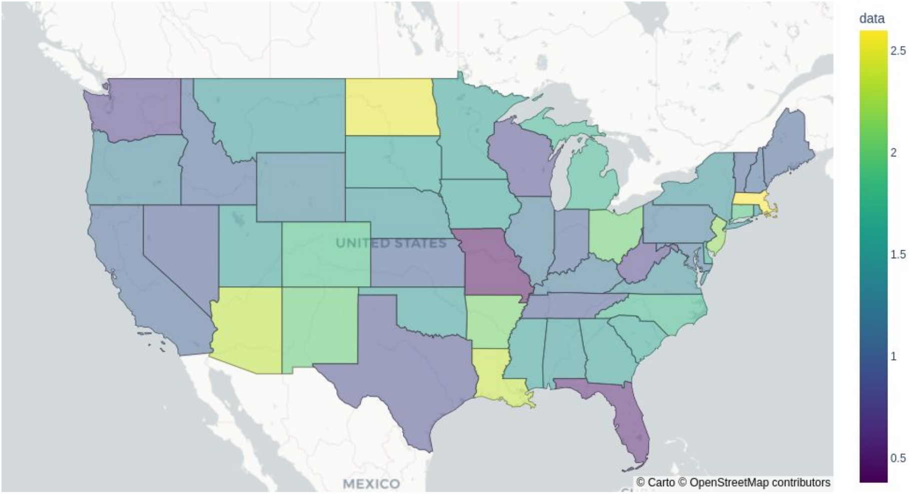
$$softmax(r)_i = \frac{\exp(r_i)}{\sum_j \exp(r_j)}$$

```
tensor([[0.0145, 0.1547, 0.2028, 0.6280],
        [0.5330, 0.1573, 0.1309, 0.1788],
        [0.3996, 0.4051, 0.0592, 0.1361],
        [0.0380, 0.3634, 0.3822, 0.2164],
        [0.4776, 0.0215, 0.0645, 0.4364],
        [0.5077, 0.1789, 0.0715, 0.2420],
        [0.0919, 0.4041, 0.2066, 0.2973],
        [0.0605, 0.0551, 0.5995, 0.2849],
        [0.1468, 0.2879, 0.5467, 0.0187],
        [0.1868, 0.0508, 0.2139, 0.5485],
        [0.1415, 0.1145, 0.1192, 0.6248],
        [0.0386, 0.3112, 0.3116, 0.3386],
        [0.1767, 0.1166, 0.2088, 0.4979],
        [0.0604, 0.0554, 0.8131, 0.0710],
        [0.6682, 0.0555, 0.0716, 0.2048],
        [0.2157, 0.5020, 0.2151, 0.0671],
        [0.7103, 0.0238, 0.0353, 0.2306],
        [0.2241, 0.1507, 0.3075, 0.3178],
        [0.2021, 0.2389, 0.0416, 0.5173],
        [0.4137, 0.0380, 0.0844, 0.4639]], grad_fn=<SoftmaxBackward0>)
```

```
tensor([[0, 0, 0, 1],
        [0, 0, 0, 1],
        [0, 0, 1, 0],
        [0, 1, 0, 0],
        [0, 0, 1, 0],
        [0, 1, 0, 0],
        [1, 0, 0, 0],
        [1, 0, 0, 0],
        [1, 0, 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1],
        [1, 0, 0, 0],
        [0, 1, 0, 0],
        [0, 0, 1, 0],
        [1, 0, 0, 0],
        [0, 0, 1, 0],
        [1, 0, 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1],
        [1, 0, 0, 0],
        [1, 0, 0, 0],
        [1, 0, 0, 0],
```

Problem formulation in PyTorch II.

1. Take the product C^*C^T (**matmul**). In the m×m result, each element can be interpreted as a "color similarity"
2. Mask the resulting matrix with the adjacency matrix by calculating the elementwise product
3. Sum all the elements to get a scalar valued metric of coloring error (loss)
4. Minimize the error w.r.t C, using torch.autograd



$$L(C) = \sum\sum[(CC^T) \circ A]$$

Problem formulation in PyTorch III.

```
class Colors(nn.Module):
    def __init__(self, initial_t=1, t_reduce_factor=0.99):
        super(Colors, self).__init__()
        self.colors = torch.randn((len(counties), nb_colors), requires_grad=True, device="cuda:0")
        self.T = initial_t
        self.factor = t_reduce_factor

    def forward(self):
        return nn.functional.softmax((1.0/self.T)*self.colors, dim=1)

    def reduce_T(self):
        self.T *= self.factor
```

```
def loss_fn(co, adj):
    return torch.mul(torch.matmul(co, co.T), adj).sum().sum()
```

“Training” loop

- In the forward step, we evaluate the model (simply outputting the color candidates)
- In the backward step, we calculate the loss gradients and modify the model parameters
- Repeat until an acceptable result

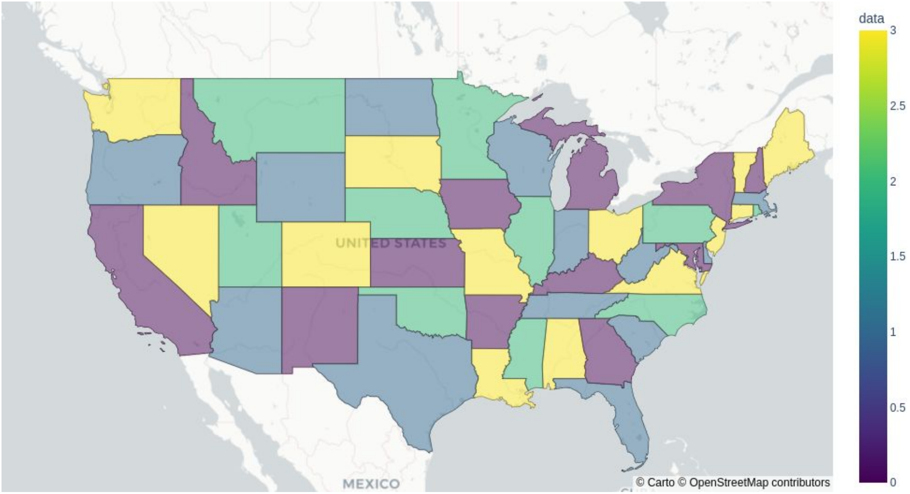
```
zero = 1e-12
loss = loss_fn(cc(), adjacency)

while loss > zero:
    loss = loss_fn(cc(), adjacency)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

- Due to its heuristic nature, the algorithm does not guarantee convergence to an optimal coloring, therefore restart conditions should be defined
    - patience - restart if the loss fails to decrease after some number of steps

Results I.

- Using the %%timeit magic function from Jupyter
- Devices
  - i5-5300u CPU
  - Nvidia GTX 1050 Ti GPU
  - Google Colab Nvidia T4 GPU instance
- D-Wave Leap QPU as “baseline” using the hybrid CQM solver and the “graph-coloring” example https://github.com/dwave-examples/graph-coloring



Results II. - xPU time

	i5-5300u CPU	GTX 1050 Ti GPU	T4 GPU instance**	D-Wave QPU
mean	24.1 s	17.3 s	9.2 s	QPU_ACCESS_TIME 0.032 s CHARGE_TIME 5.000 s RUN_TIME 5.214 s
std-dev	0.32 s	0.47 s	3.75 s	-

Remarks

- An AI framework was successfully used to solve a non-AI problem
- The solution can be executed on a diverse set of hardwares, without any modification in the code
- Moving the computation between CPU and GPU is a one line command; torch_tensor.to(device) or setting the “device=” argument

Thank you for your attention!