Introduction to PyTorch

Deep Learning Framework

MISiS Mega Science, Spring Semester

Andrey Ustyuzhanin

National Research University Higher School of Economics

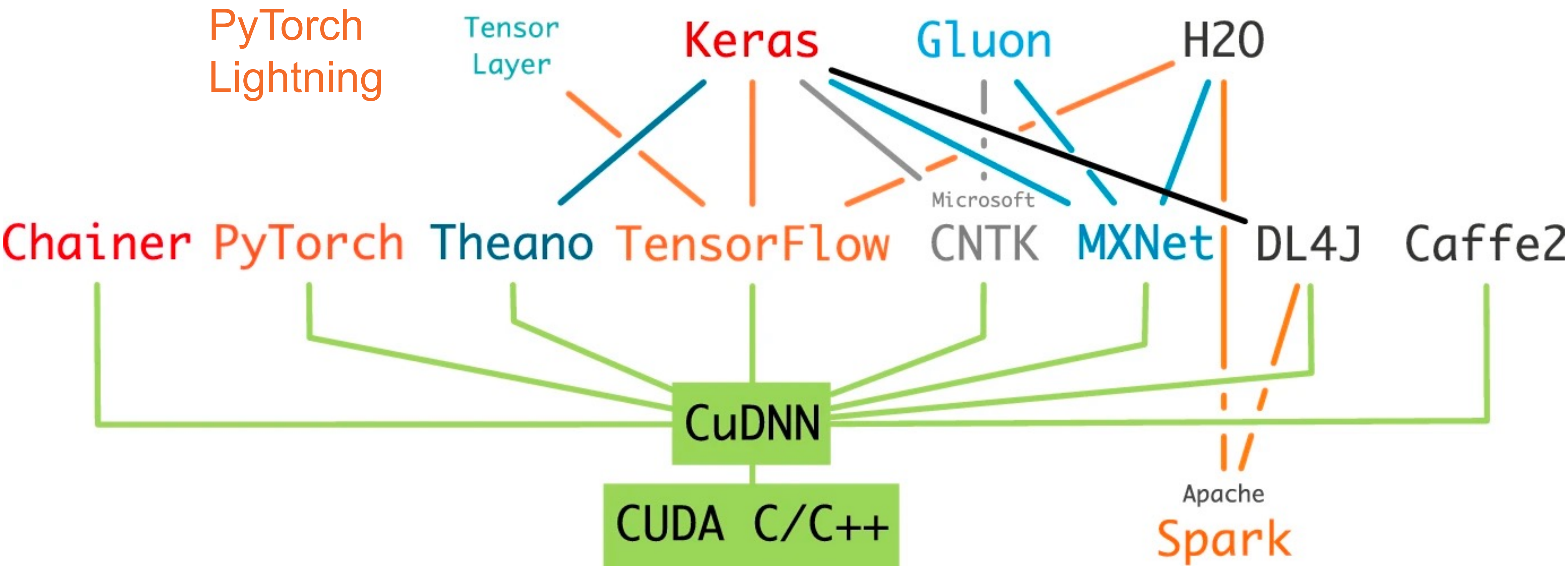
MISiS National University of Science and Technology

LAMBDA • HSE

March, 2021

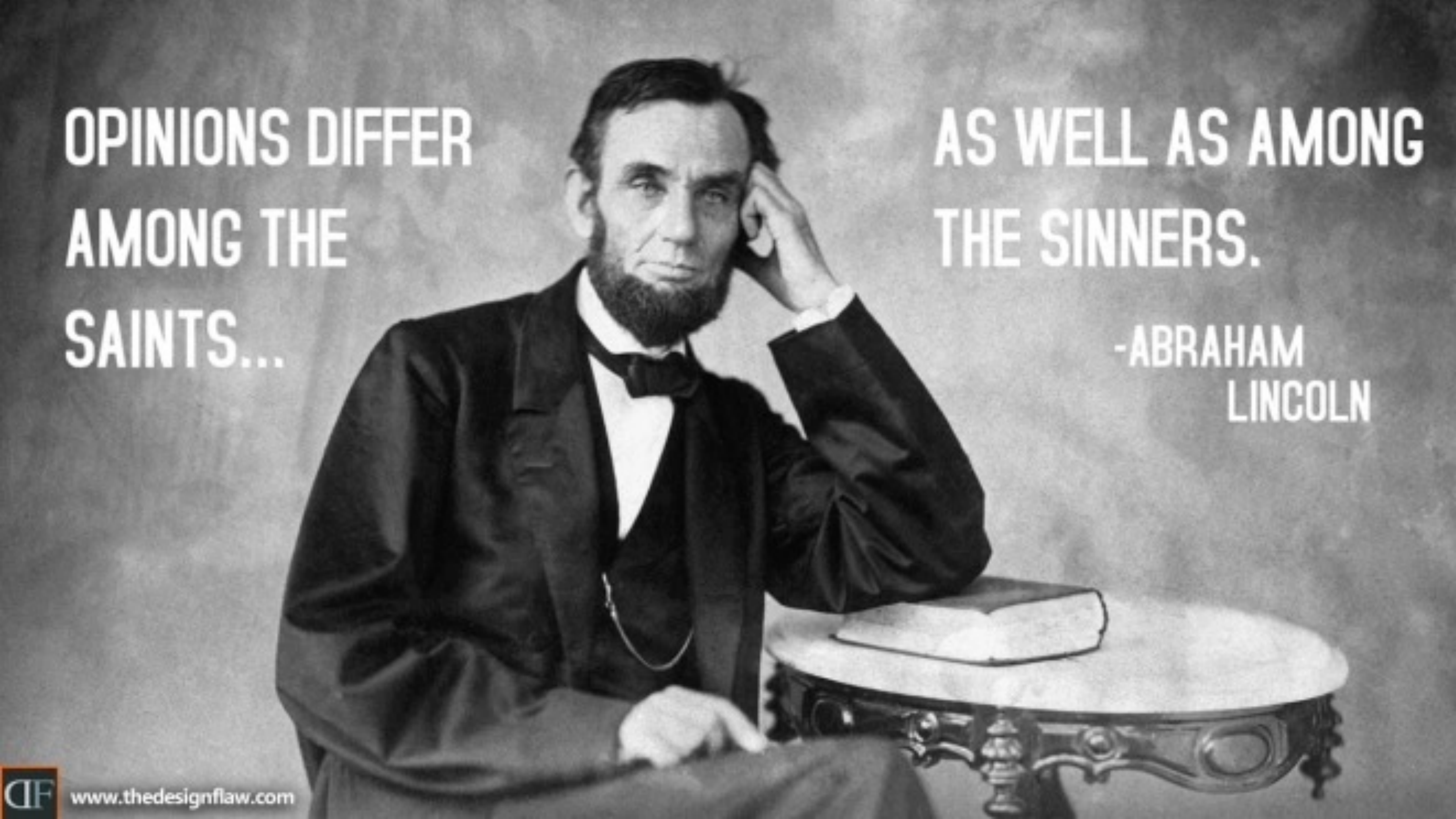
MISIS National University of Science and Technology

DL frameworks in various abstraction levels



source

Comparison, moar comparison

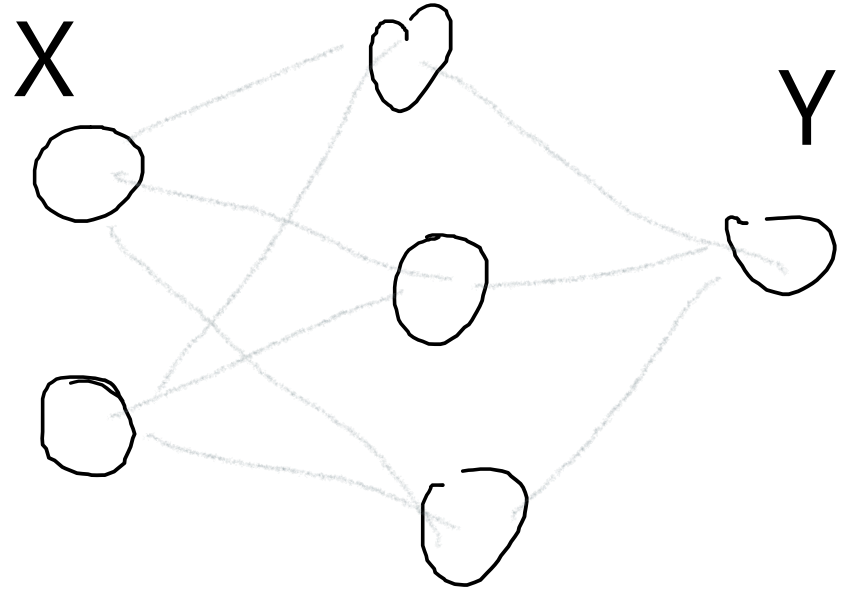

OPINIONS DIFFER
AMONG THE
SAINTS...
AS WELL AS AMONG
THE SINNERS.
-ABRAHAM
LINCOLN
www.thedesignflaw.com

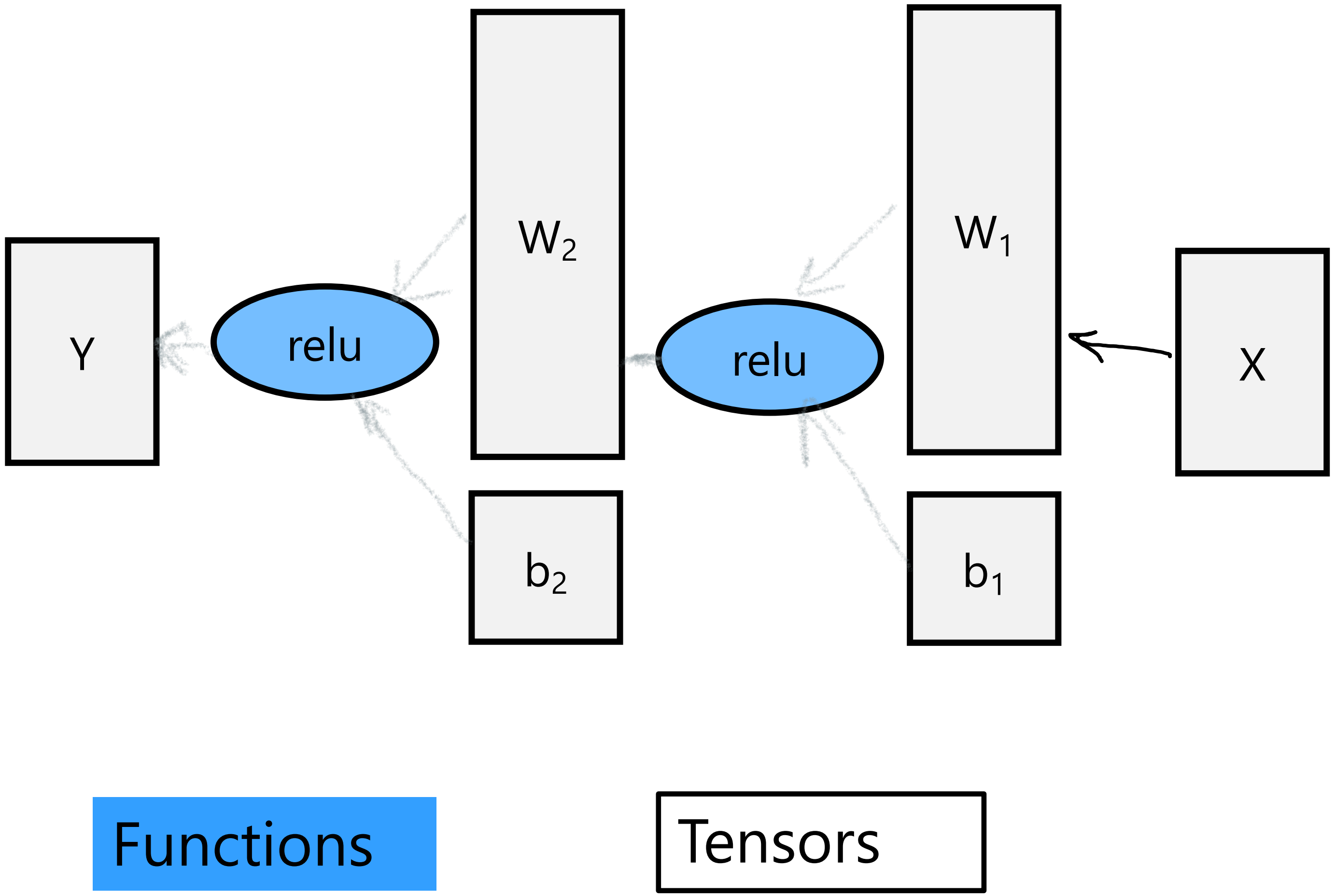
PyTorch highlights

- Automatic differentiation engine (autograd)
- Simple, transparent development/ debugging
- Rich Ecosystem:
  - Plenty of pretrained models
  - NLP, Vision, …
  - Interpretation
  - Hyper-optimization
- Production Ready (C++, ONNX, Services)
- Distributed Training, declarative data parallelism
- Cloud Deployment support
- Choice of many industry leaders and researchers

facebook Artificial Intelligence

Neural network representation





$$Y = \mathrm{relu}(W_2 \times \mathrm{relu}(W_1 X + b_1) + b_2)$$

Building blocks, tensors

```
torch.randn(*size)             # tensor with independent N(0,1) entries
torch.[ones|zeros](*size)      # tensor with all 1's [or 0's]
torch.Tensor(L)                # create tensor from [nested] list or ndarray L
x.clone()                      # clone of x
with torch.no_grad():          # code wrap that stops autograd from tracking tensor history
requires_grad=True             # arg, when set to True, tracks computation
                               # history for future derivative calculations
```

```
x.size()                       # return tuple-like object of dimensions
torch.cat(tensor_seq, dim=0)   # concatenates tensors along dim
x.view(a,b,...)                # reshapes x into size (a,b,...)
x.view(-1,a)                   # reshapes x into size (b,a) for some b
x.transpose(a,b)               # swaps dimensions a and b
x.permute(*dims)               # permutes dimensions
x.unsqueeze(dim)               # tensor with added axis
x.unsqueeze(dim=2)             # (a,b,c) tensor -> (a,b,1,c) tensor
```

Building blocks, graph

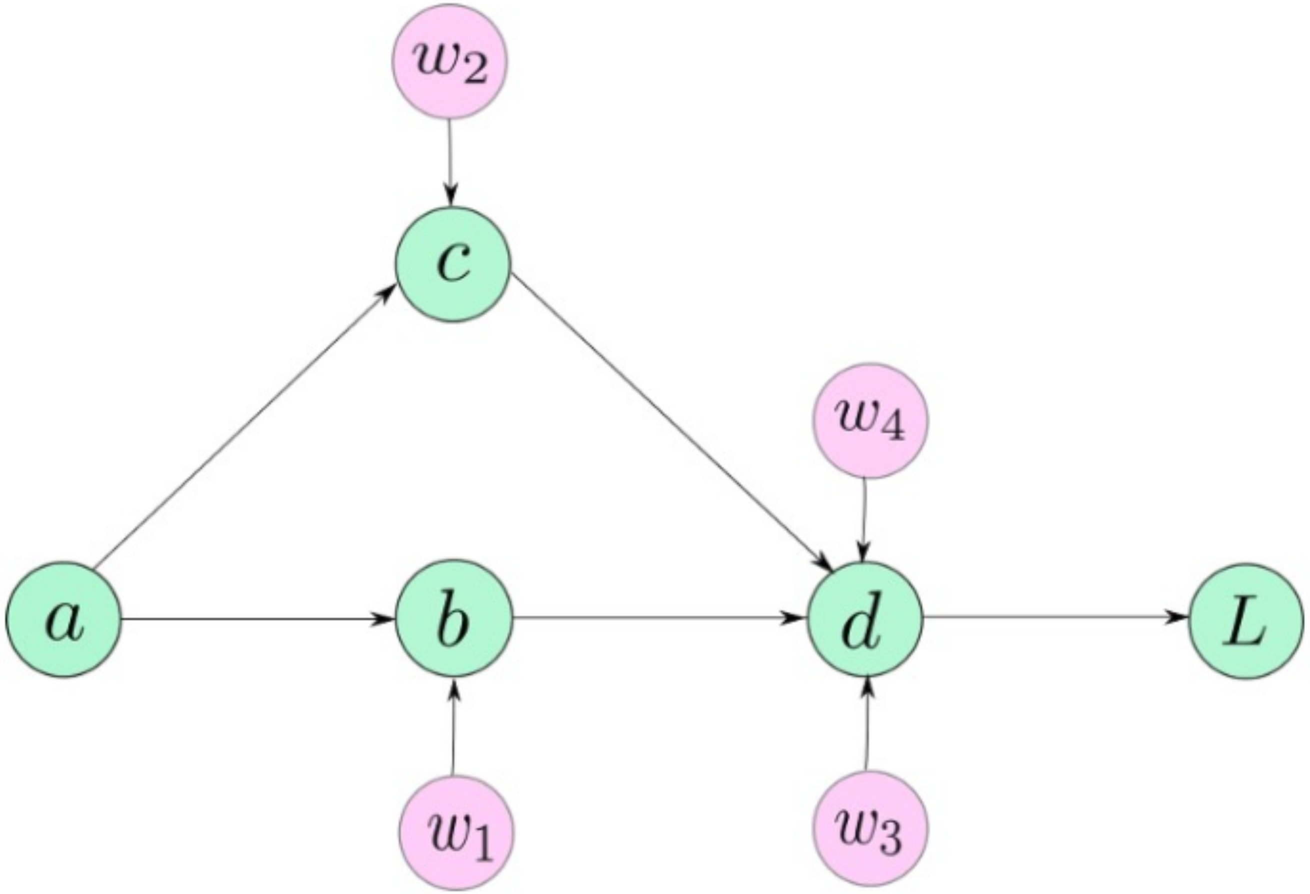
Toy example:

$$b = w_1 * a$$

$$c = w_2 * a$$

$$d = w_3 * b + w_4 * c$$

$$L = 10 - d$$



source

Math operations

```
A.mm(B)         # matrix multiplication
A.mv(x)         # matrix-vector multiplication
x.t()           # matrix transpose
```

- https://pytorch.org/docs/stable/torch.html?highlight=mm#math-operations

Building blocks, computational graph

$$b = w_1 * a$$

$$c = w_2 * a$$

$$d = w_3 * b + w_4 * c$$

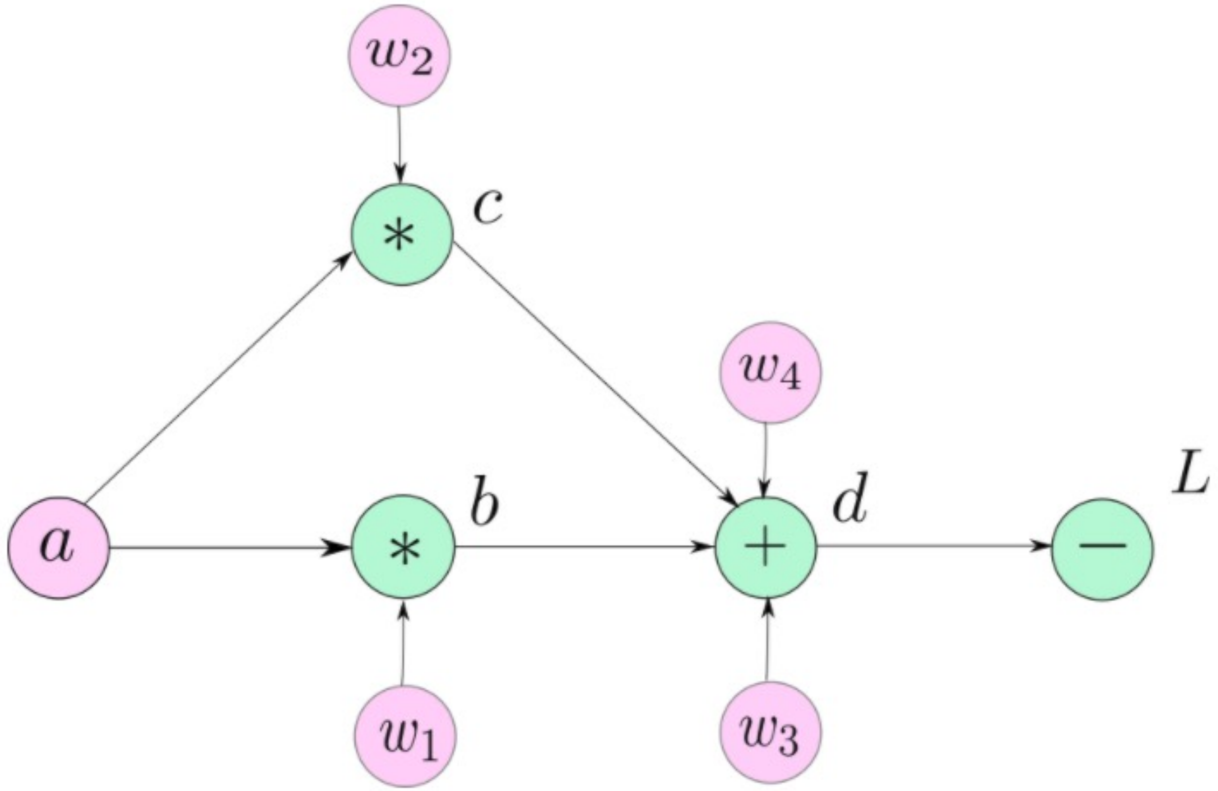
$$L = 10 - d$$

$$\frac{\partial L}{\partial w_4} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial w_4}$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial w_3}$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial c} * \frac{\partial c}{\partial w_2}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial b} * \frac{\partial b}{\partial w_1}$$



Building blocks, computational graph

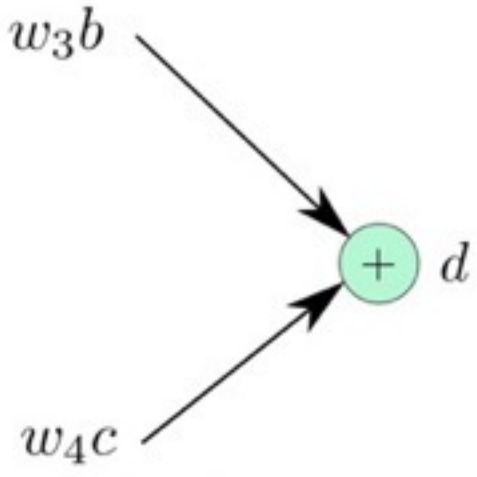
$$b = w_1 * a$$

$$c = w_2 * a$$

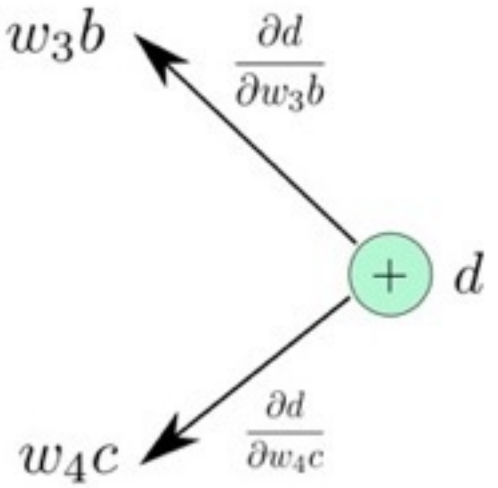
$$d = w_3 * b + w_4 * c$$

$$L = 10 - d$$

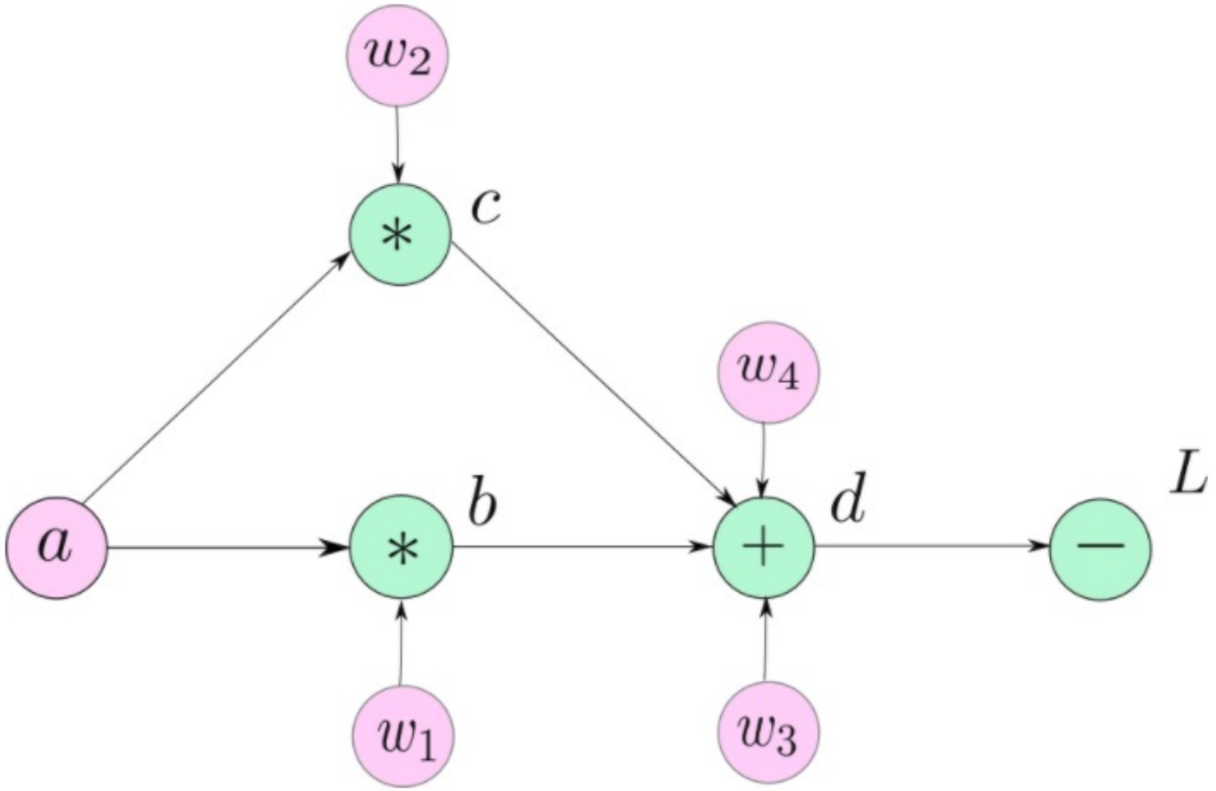
$d = f(w_3 b, w_4 c)$



d is output of function f(x,y) = x + y



Local Gradients



$$\frac{\partial f}{\partial w_4} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial b} * \frac{\partial b}{\partial a} + \frac{\partial L}{\partial d} * \frac{\partial d}{\partial c} * \frac{\partial c}{\partial a}$$

Gradient and tensors

```
>> t1 = torch.randn((3,3), requires_grad = True)

>> t2 = torch.FloatTensor(3,3) # No way to specify requi
>> t2.requires_grad = True
```

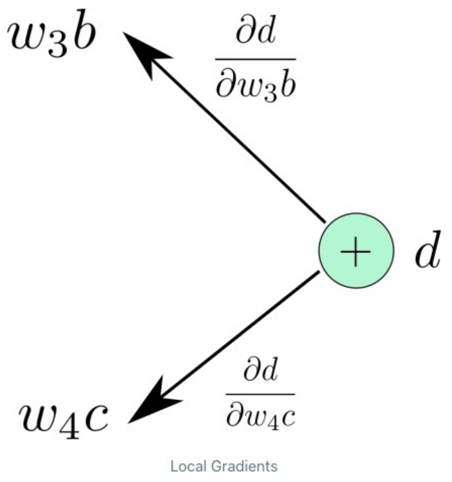
```
import torch

a = torch.randn((3,3), requires_grad = True)

w1 = torch.randn((3,3), requires_grad = True)
w2 = torch.randn((3,3), requires_grad = True)
w3 = torch.randn((3,3), requires_grad = True)
w4 = torch.randn((3,3), requires_grad = True)

b = w1*a
c = w2*a

d = w3*b + w4*c

L = 10 - d

print("The grad fn for a is", a.grad_fn)
print("The grad fn for d is", d.grad_fn)
```

Each Tensor has
- an attribute **grad_fn**, which refers to the mathematical operator that create the variable;
- an attribute **grad,** which contains gradient value per tensor element

If Tensor is a leaf node (initialised by the user), then the **grad_fn** is also **None**.

```
The grad fn for a is None
The grad fn for d is <AddBackward0 object at 0x1033afe48>
```

Functions



- All math operations performed by *torch.autograd.Function* children
  - *forward*, computes node output and buffers it
  - *Backward,* stores incoming **gradient** and passes further up

```
def backward (incoming_gradients):
        self.Tensor.grad = incoming_gradients

        for inp in self.inputs:
                if inp.grad_fn is not None:
                        new_incoming_gradients = //
                          incoming_gradient * local_grad(self.Tensor, inp)

                        inp.grad_fn.backward(new_incoming_gradients)
                else:
                        pass
```

From backward() to gradient descent

```
# Replace L.backward() with
L.backward(torch.ones(L.shape))
```

```
w1 = w1 - learning_rate * w1.grad
```

Dynamic graph

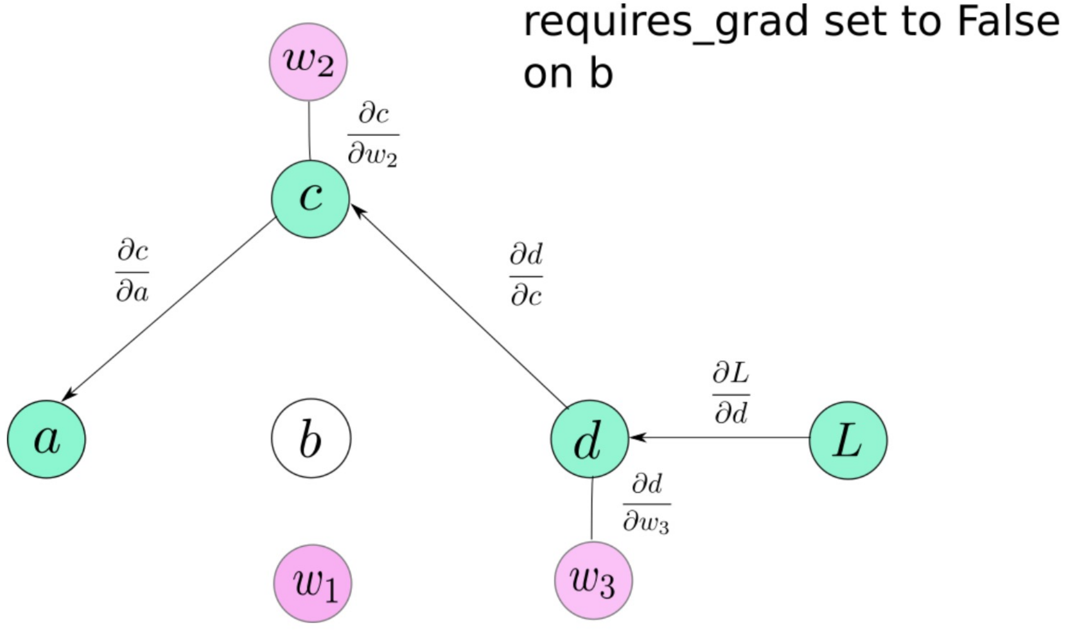
- After calling *forward* the intermediate node variables are created;
- Then, the buffers for the non-leaf nodes allocated for the graph and intermediate values used for computing gradients later. When you call *backward*, as the gradients are computed, these buffers are essentially freed, and the graph is destroyed;
- Next time, you call *forward* on the same set of tensors, **the leaf node buffers from the previous run will be shared, while the non-leaf nodes buffers will be created again.**
- If *retain_graph = True* passed to the *backward* function, the graph is not recreated, and the computed gradients will be added to the previous iteration values.

Gradient cleaning

- Due to the flexibility of the network architecture, it is not obvious when does iteration of a gradient descent stops, so *backward's* gradients are accumulated each time a variable (Tensor) occurs in the graph;
- It is usually desired for RNN cases;
- If you do not need to accumulate those, you must clean previous gradient values at the end of each iteration:
  - Either by x.data.zero_() for every model tensor x;
  - Or by optimizers's *zero_grad*() method (preferred).

Freezing weights

- **Requires_grad** attribute of the *Tensor* class. By default, it's **False**. It comes handy when you must freeze some layers and stop them from updating parameters while training.
- Thus, no gradient would be propagated to them, or to those layers which depend upon these layers for gradient flow **requires_grad**.
- When set to **True**, **requires_grad** is contagious: even if one operand of an operation has **requires_grad** set to **True**, so will the result.



Pre-trained models enhancement

```
model = torchvision.models.resnet18(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
# Replace the last fully-connected layer
# Parameters of newly constructed modules have requires_grad=True by
default
model.fc = nn.Linear(512, 100)

# Optimize only the classifier
optimizer = optim.SGD(model.fc.parameters(), lr=1e-2, momentum=0.9)
```

Inference

- When we are computing gradients, we need to cache input values, and intermediate features as they maybe required to compute the gradient later.The gradient of $b = w1 * a$ w.r.t it's inputs $w1$ and a is a and $w1$ respectively. We need to store these values for gradient computation during the backward pass. This affects the memory footprint of the network.
- While, we are performing inference, we don't compute gradients, and thus, don't need to store these values. Infact, no graph needs to be create during inference as it will lead to useless consumption of memory.

```
with torch.no_grad:
        inference code goes here
```

GPU, TPU support

```
torch.cuda.is_available                              # check for cuda
x.cuda()                                             # move x's data from
                                                     # CPU to GPU and return new object

x.cpu()                                              # move x's data from GPU to CPU
                                                     # and return new object

if not args.disable_cuda and torch.cuda.is_available(): # device agnostic code
    args.device = torch.device('cuda')               # and modularity
else:                                                #
    args.device = torch.device('cpu')                #

net.to(device)                                       # recursively convert their
                                                     # parameters and buffers to
                                                     # device specific tensors

mytensor.to(device)                                  # copy your tensors to a device
                                                     # (gpu, cpu)
```

- https://pytorch.org/docs/stable/cuda.html
- http://pytorch.org/xla/release/1.5/index.html

torch.nn.Module

```python
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Loss functions

```
nn.X                                        # where X is BCELoss, CrossEntropyLoss,
                                            # L1Loss, MSELoss, NLLLoss, SoftMarginLoss,
                                            # MultiLabelSoftMarginLoss, CosineEmbeddingLoss,
                                            # KLDivLoss, MarginRankingLoss, HingeEmbeddingLoss
                                            # or CosineEmbeddingLoss
```

- https://pytorch.org/docs/stable/nn.html#loss-functions

Activation functions

```
nn.X                                # where X is ReLU, ReLU6, ELU, SELU, PReLU, LeakyReLU,
                                    # Threshold, HardTanh, Sigmoid, Tanh,
                                    # LogSigmoid, Softplus, SoftShrink,
                                    # Softsign, TanhShrink, Softmin, Softmax,
                                    # Softmax2d or LogSoftmax
```

- https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity

Optimizers

```
opt = optim.x(model.parameters(), ...)      # create optimizer
opt.step()                                  # update weights
optim.X                                     # where X is SGD, Adadelta, Adagrad, Adam,
                                            # SparseAdam, Adamax, ASGD,
                                            # LBFGS, RMSProp or Rprop
```

- https://pytorch.org/docs/stable/optim.html

Data Utils

Datasets

```
Dataset                    # abstract class representing dataset
TensorDataset              # labelled dataset in the form of tensors
Concat Dataset             # concatenation of Datasets
```

- https://pytorch.org/docs/stable/data.html?highlight=dataset#torch.utils.data.Dataset

Dataloaders and DataSamplers

```
DataLoader(dataset, batch_size=1, ...)      # loads data batches agnostic
                                            # of structure of individual data points

sampler.Sampler(dataset,...)                # abstract class dealing with
                                            # ways to sample from dataset

sampler.XSampler where ...                  # Sequential, Random, Subset,
                                            # WeightedRandom or Distributed
```

- https://pytorch.org/docs/stable/data.html?highlight=dataloader#torch.utils.data.DataLoader

Ecosystem

- PyTorch lightning
- PyTorch geometric
- Hydra
- Horovod
- Skorch
- Captum
- And many others, see https://pytorch.org/ecosystem/

Moar stuff

- https://pytorch.org/docs/stable/index.html
- https://pytorch.org/tutorials/beginner/ptcheat.html
- http://neuralnetworksanddeeplearning.com/chap2.html
- https://www.khanacademy.org/math/differential-calculus/dc-chain
- https://blog.paperspace.com/pytorch-101-understanding-graphs-and-automatic-differentiation/
- https://github.com/yandexdataschool/mlhep2019/blob/master/notebooks/day-3/seminar_pytorch.ipynb

Conclusion

- PyTorch is a solid, flexible, production-ready foundation for real-life deep-learning applications
- Building blocks:
  - Tensors
  - Functions
- Dynamic graph automatic differentiation
  - CPU, GPU, TPU
- Rich ecosystem

Thank you!

austyuzhanin@hse.ru

anaderiRu

hse_lambda

Andrey Ustyuzhanin