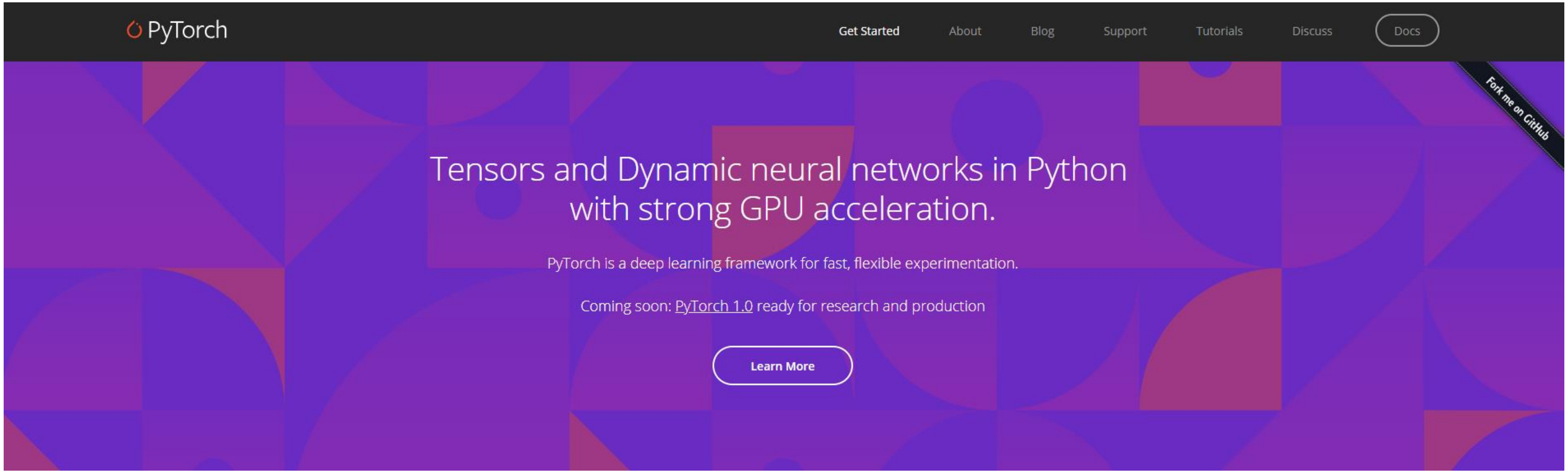
PyTorch

AImage Lab

UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

PYTORCH



Tensors and Dynamic neural networks in Python with strong GPU acceleration.

PyTorch is a deep learning framework for fast, flexible experimentation.

Coming soon: PyTorch 1.0 ready for research and production

Features:
Tensor computations (Ndarray support, like Numpy) on GPU
Automatic Differentiation
Gradient-based optimization
Neural Networks and vision support

PYTORCH

AImage Lab

Developed at Facebook AI Research, and used by…

Facebook Open Source

NVIDIA.

salesforce

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIE
PARIS INSTITUTE OF TECHNOLOGY

Carnegie Mellon University

UNIVERSITE PIERRE & MARIE CURIE
LA SCIENCE A PARIS

UBER

Stanford University

UNIVERSITY OF OXFORD

NYU

inria

ENS
ÉCOLE NORMALE SUPÉRIEURE

VisionLabs
MACHINES CAN SEE

Digital Reasoning

and by AImage Lab **as well ;)**

RECALL….

```
ssh –p port user@YourAzureVM
pip freeze | grep torch==0.4.0
git pull https://github.com/aimagelab/aidlda_tutoral
```

You have all the slides and the code in the Github repo!

Don’t get lost:

- if you don’t understand something: **ask**!
- If you can’t do something: **the solution is in the repo**!

NDARRAY LIBRARY

np.ndarray <-> torch.Tensor
200+ operations numpy-style
very fast acceleration on NVIDIA GPUs

A FAST CALCULATOR

```
import torch
import numpy as np
import time

d = 3000
a = np.random.rand(d, d)
b = np.random.rand(d, d)
start = time.time()
c = np.matmul(a,b)
end = time.time()
print("Elapsed time Numpy: %fs" % (end-start))

gpu = torch.device('cuda:0')
a = torch.rand(d,d, device=gpu)
b = torch.rand(d,d, device=gpu)
start = time.time()
c = torch.mm(a, b)
end = time.time()
print("Elapsed time PyTorch: %fs" % (end-start))
```

```
Elapsed time Numpy: 0.338300s
Elapsed time PyTorch: 0.003510s
```

TENSORS

A torch.tensor is a multi-dimensional matrix containing elements of a single data type which live on a device (either a CPU or a GPU).

Easy: a replacement of NumPy ndarrays, which can also go on GPU.

Different datatypes: float32, float64, float16, uint8, int8, int16, int32, int64

Constructors

From a Python list or sequence, using torch.tensor()

Using Numpy-style constructors (e.g. ones, zeros, …)

From Numpy (see next slides)

TENSORS

Construct a 5x3 matrix, uninitialized

```python
import torch
x = torch.Tensor(5,3)
print(x)
```

```
tensor([[ 2.9669e-34,  3.0711e-41,  6.0256e-44],
        [ 0.0000e+00,         nan,  0.0000e+00],
        [ 1.3733e-14,  9.5680e+20,  7.2065e+31],
        [ 2.6301e+20,  1.4601e-19,  6.4069e+02],
        [ 4.3066e+21,  1.1824e+22,  4.3066e+21]])
```

TENSORS

Construct a randomly initialized matrix

```
x = torch.rand(5,3)
print(x)
```

```
tensor([[ 0.6181,  0.1883,  0.8612],
        [ 0.4102,  0.5542,  0.7740],
        [ 0.2829,  0.1896,  0.5191],
        [ 0.8135,  0.9329,  0.1325],
        [ 0.9554,  0.5377,  0.0733]])
```

Get its size

```
print(x.size())
print(x.shape)
```

```
torch.Size([5, 3])
torch.Size([5, 3])
```

TENSORS

Standard numpy-like slicing and indexing is supported

```
print(x[:,1])
```

```
tensor([ 0.1883,  0.5542,  0.1896,  0.9329,  0.5377])
```

Standard operations between tensors are supported

```
y = torch.rand(5,3)
print(x+y)
```

```
tensor([[ 0.8932,  0.8877,  0.8732],
        [ 0.8380,  0.7639,  0.8646],
        [ 0.3234,  1.0936,  0.6914],
        [ 1.7137,  1.6938,  0.5719],
        [ 0.9686,  0.9211,  0.4722]])
```

TENSORS

Numpy bridge

Converting a Torch Tensor to a NumPy array and vice versa is a breeze.

The Torch Tensor and NumPy array will share their underlying memory locations, and changing one will change the other.

NUMPY BRIDGE

Zero memory-copy

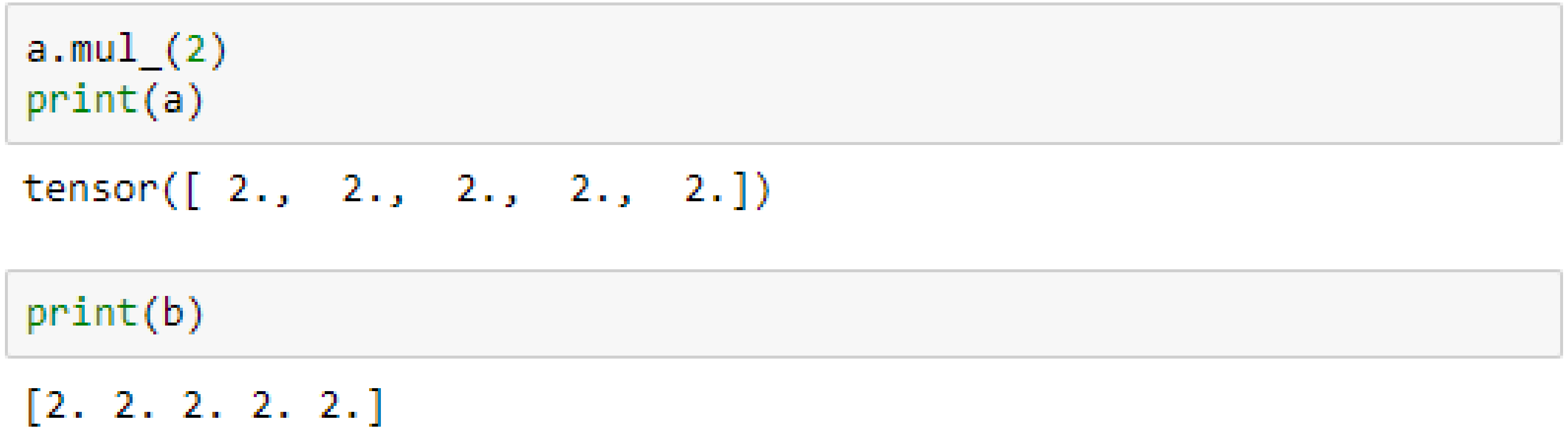
```
a = torch.ones(5)
print(a)
```

```
tensor([ 1.,  1.,  1.,  1.,  1.])
```

```
b = a.numpy()
print(b)
```

```
[1. 1. 1. 1. 1.]
```

NUMPY BRIDGE



```
a.mul_(2)
print(a)
```

```
tensor([ 2.,  2.,  2.,  2.,  2.])
```

```
print(b)
```

```
[2. 2. 2. 2. 2.]
```

NUMPY BRIDGE

Transferring from Numpy to PyTorch

```
import numpy as np
a = np.ones((5, ))
b = torch.from_numpy(a)

np.multiply(a, 2, out=a)
print(a)
print(b)
```

```
[2. 2. 2. 2. 2.]
tensor([ 2.,  2.,  2.,  2.,  2.], dtype=torch.float64)
```

TORCH.AUTOGRAD

“PyTorch has a unique way of building neural networks: using and replaying a tape recorder”

WHAT IS AUTOGRAD?

Autograd is reverse automatic differentiation system.

- Define a symbolic graph
- Compile it
- Feed data from entry points
- Changing the graph -> start from scratch

- Define graphs online, performing forward operations on data
- Evaluate the graph in the backward pass to compute gradients

NO OVERHEADS!

WHAT IS AUTOGRAD?

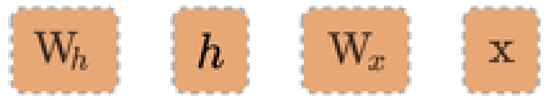
torch.autograd is a reverse automatic differentiation system.

- As you stack forward operations, autograd records a DAG
- Leaves are the input tensors and roots are the output tensors
- Each node is a torch.autograd.Function. (we'll delve into this later)
- During forward, autograd performs quietly operations allowing the computation of backward

The graph is recreated from scratch at every iteration!

GRAPH CREATION

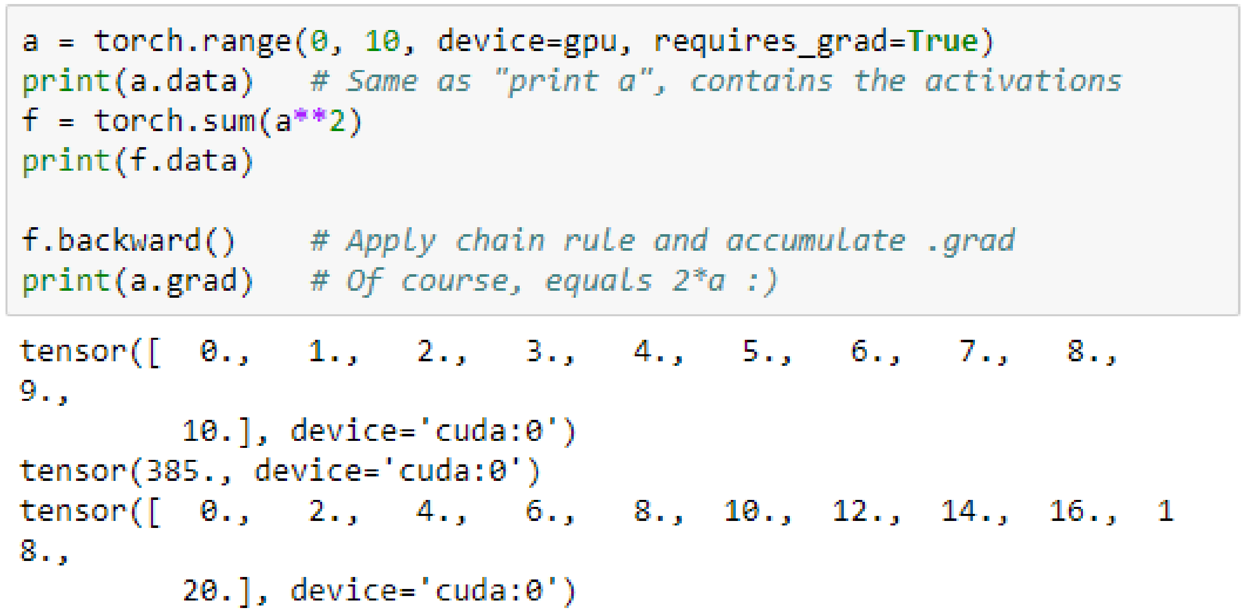
A graph is created on the fly



```
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
W_h = torch.randn(20, 20)
W_x = torch.randn(20, 10)
```

AUTO-DIFFERENTIATION

The autograd package provides automatic differentiation for all operations on Tensors. It is a **define-by-run** framework, which means that your backprop is defined by how your code is run, and that every single iteration can be different.



```
a = torch.range(0, 10, device=gpu, requires_grad=True)
print(a.data)   # Same as "print a", contains the activations
f = torch.sum(a**2)
print(f.data)

f.backward()    # Apply chain rule and accumulate .grad
print(a.grad)   # Of course, equals 2*a :)
```

```
tensor([  0.,    1.,    2.,    3.,    4.,    5.,    6.,    7.,    8.,
9.,
         10.], device='cuda:0')
tensor(385., device='cuda:0')
tensor([  0.,    2.,    4.,    6.,    8.,   10.,   12.,   14.,   16.,   1
8.,
         20.], device='cuda:0')
```

From here to building a complex NN… it is just a matter of scale! ☺

AUTO-DIFFERENTIATION

Some new things we saw:

x.data: returns a tensor that shares the same data with x, but is unrelated to the computational history of x. Any changes to x.data will not be tracked by autograd.

x.grad: a tensor with the same shape of x, containing its gradient

requires_grad: if a tensor has requires_grad=True, its computation will be tracked by autograd, and its .grad will be filled. In general, each tensor that results from an operation with a tensor having requires_grad=True, will have requires_grad=True

AUTO-DIFFERENTIATION

Conceptually, autograd records a graph recording all of the operations that created the data as you execute operations, giving you a directed acyclic graph whose leaves are the input tensors and roots are the output tensors. By tracing this graph from roots to leaves, you can automatically compute the gradients using the chain rule.

Internally, autograd represents this graph as a graph of **Function objects** (really expressions), which can be apply() ed to compute the result of evaluating the graph. When computing the forwards pass, autograd simultaneously performs the requested computations and builds up a graph representing the function that computes the gradient (*the .grad_fn attribute of each torch.Tensor is an entry point into this graph*). When the forwards pass is completed, we evaluate this graph in the backwards pass to compute the gradients.

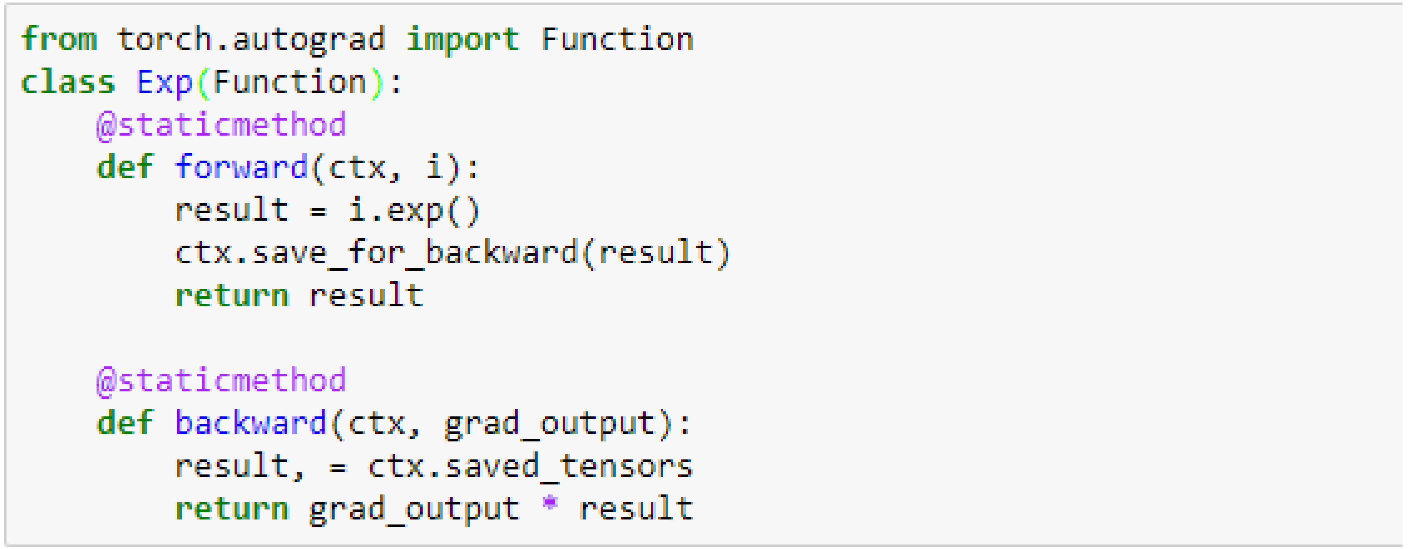
AUTO-DIFFERENTIATION

An important thing to note is that the graph is recreated from scratch at every iteration, and this is exactly what allows for using arbitrary Python control flow statements, that can change the overall shape and size of the graph at every iteration. You don’t have to encode all possible paths before you launch the training - what you run is what you differentiate.

FUNCTIONS

Every operation performed on Tensors creates a new **function** object, **that performs the computation, and records that it happened**.

The history is retained in the form of a DAG of functions.

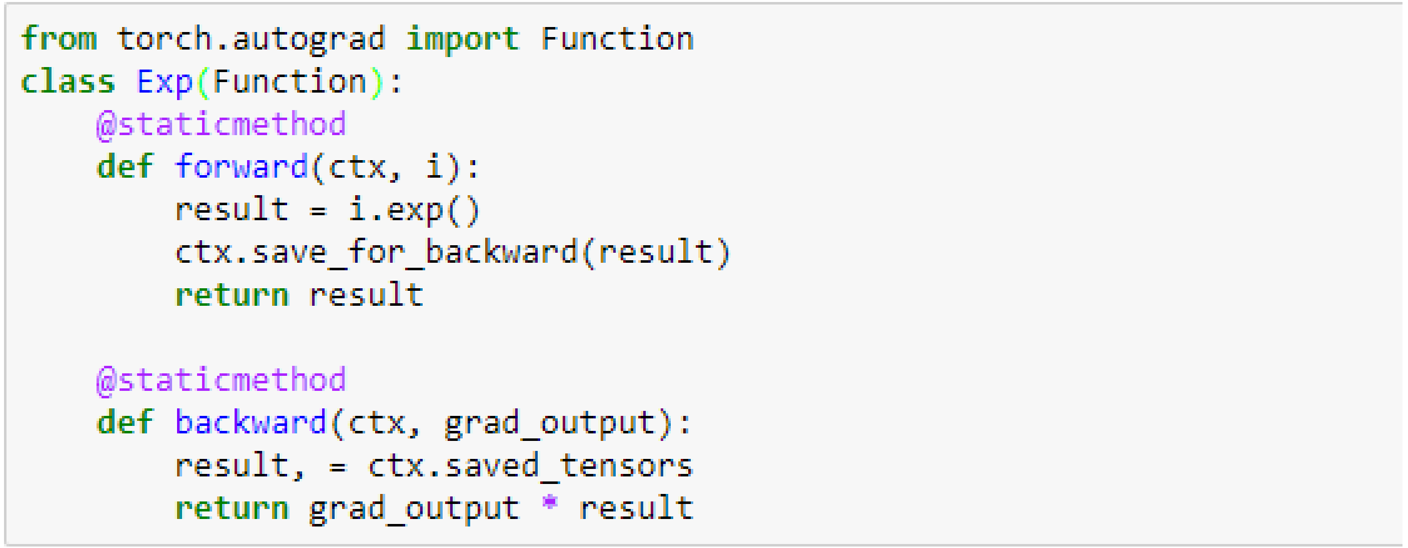


```python
from torch.autograd import Function
class Exp(Function):
    @staticmethod
    def forward(ctx, i):
        result = i.exp()
        ctx.save_for_backward(result)
        return result

    @staticmethod
    def backward(ctx, grad_output):
        result, = ctx.saved_tensors
        return grad_output * result
```

FUNCTIONS

They have two (static) methods:

- **forward**: performs the operations given the input(s).
  The first parameter, the context, can be used to store tensors for the backward pass.
- **backward**: defines the formulas for differentiating the operation
  accepts as many grad_output as many output the forward function defines. Each of these is the gradient w.r.t. the given output.
  returns as many outputs as many input the forward function has. Each of these is the gradient w.r.t. the given input.
  The context can be used to retrieve tensors saved during the forward pass.



```python
from torch.autograd import Function
class Exp(Function):
    @staticmethod
    def forward(ctx, i):
        result = i.exp()
        ctx.save_for_backward(result)
        return result

    @staticmethod
    def backward(ctx, grad_output):
        result, = ctx.saved_tensors
        return grad_output * result
```

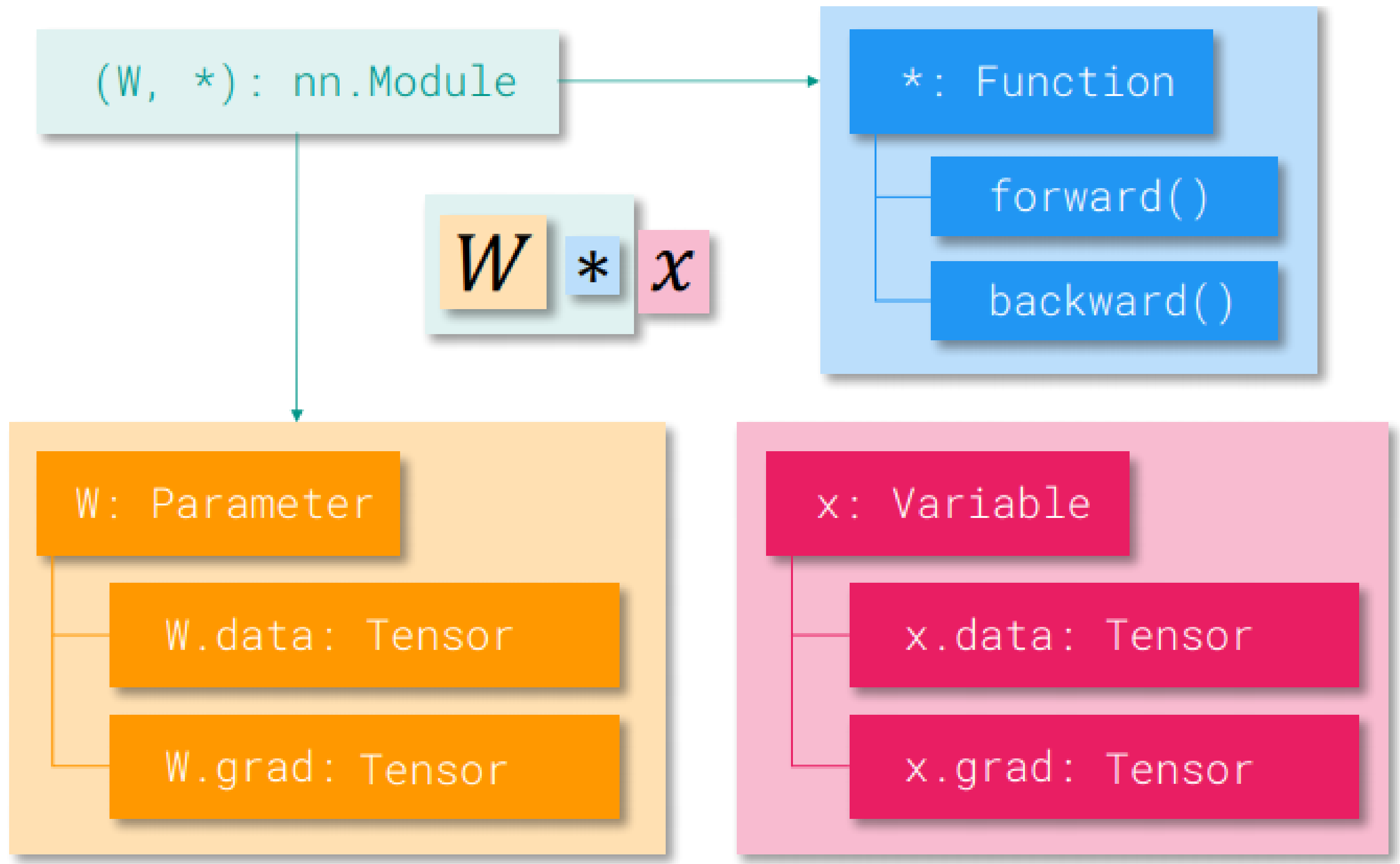
MODULES

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)

model = Net()
input = Variable(torch.randn(10, 20))
output = model(input)
```

BASIC ELEMENTS

- `torch.Tensor` – similar to numpy.array, with GPU
- `autograd.Function` – operate on `torch.Tensor` Implement forward and backward.
- `nn.Parameter` – a special `torch.Tensor`
- `nn.Module` – contain Parameters and define functions on input Variables

ANATOMY OF A FUNCTION



NEURAL NETWORKS 101

Neural networks can be constructed using the torch.nn package.

Now that you had a glimpse of autograd, nn depends on autograd to define models and differentiate them. An nn.Module contains layers, and a method forward(input) that returns the output.

Building a Neural Network:

- Define the neural network that has some learnable parameters (or weights)
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss (how far is the output from being correct)
- Propagate gradients back into the network's parameters
- Update the weights of the network, typically using a simple update rule:

OPTIMIZATION PACKAGE

torch.optim is a package implementing various optimization algorithms (SGD, Adagrad, RMSProp, LBFGS, etc.)

```
net = Net()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01, momentum=0.9)

for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = F.cross_entropy(output, target)
    loss.backward()
    optimizer.step()
```