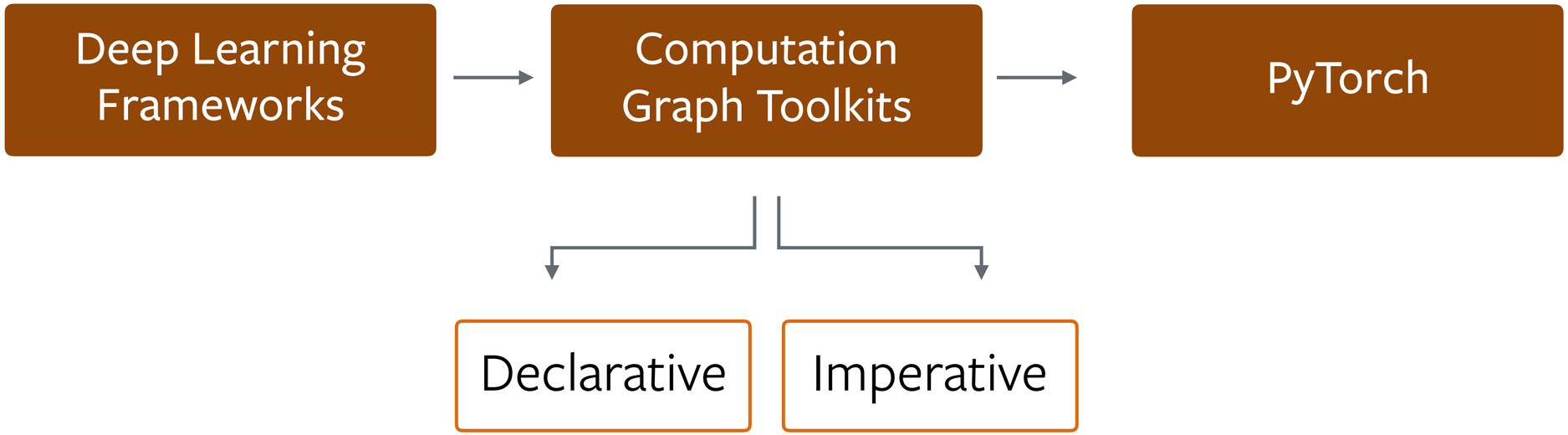

PYTORCH

# An overview of Deep Learning frameworks

# & an introduction to PYTORCH

*Adam Paszke, Sam Gross, **Soumith Chintala**, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Christian Sarofeen, Alban Desmaison, Andreas Kopf, Edward Yang, Zach Devito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy*

# Overview of the talk




Implementation

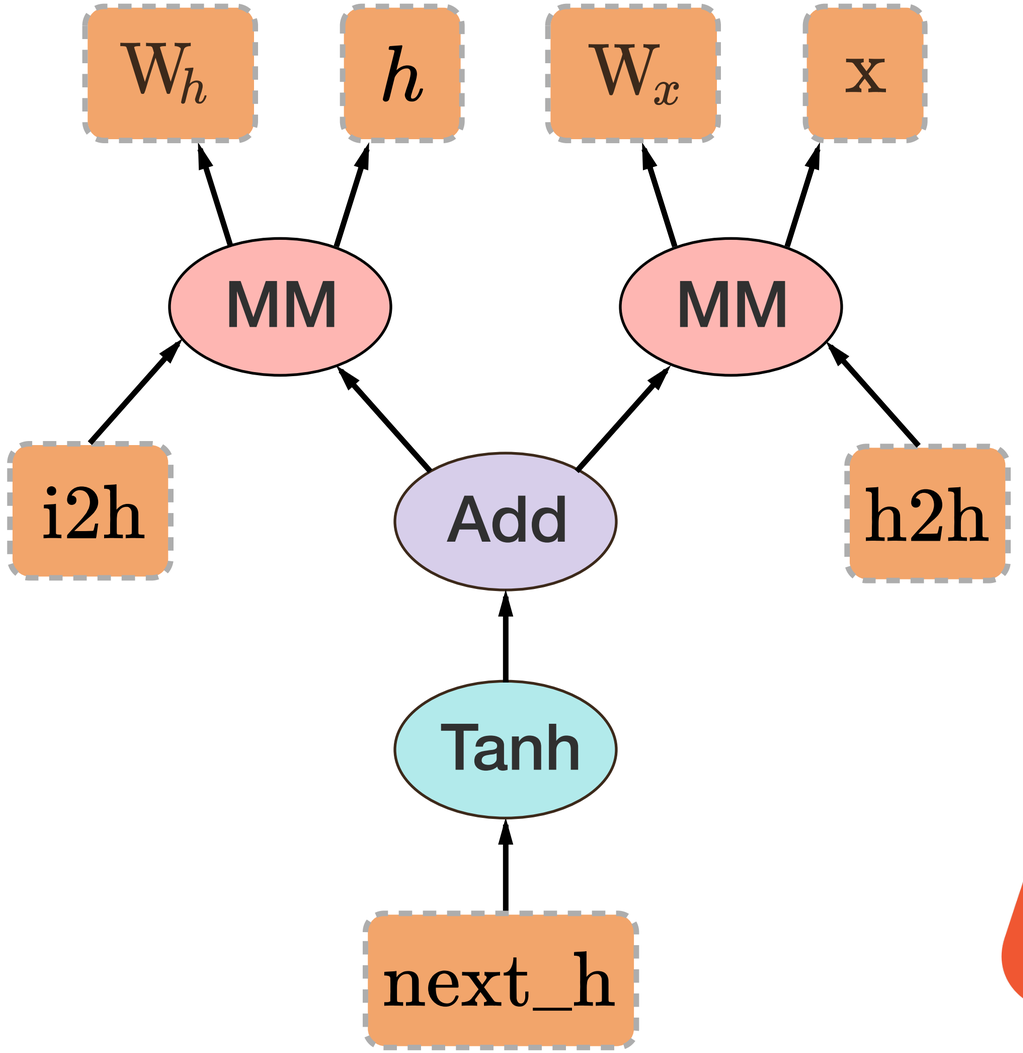
Advantages & Disadvantages

# Deep Learning Frameworks

# Deep Learning Frameworks

## In addition to Computation Graph Toolkits

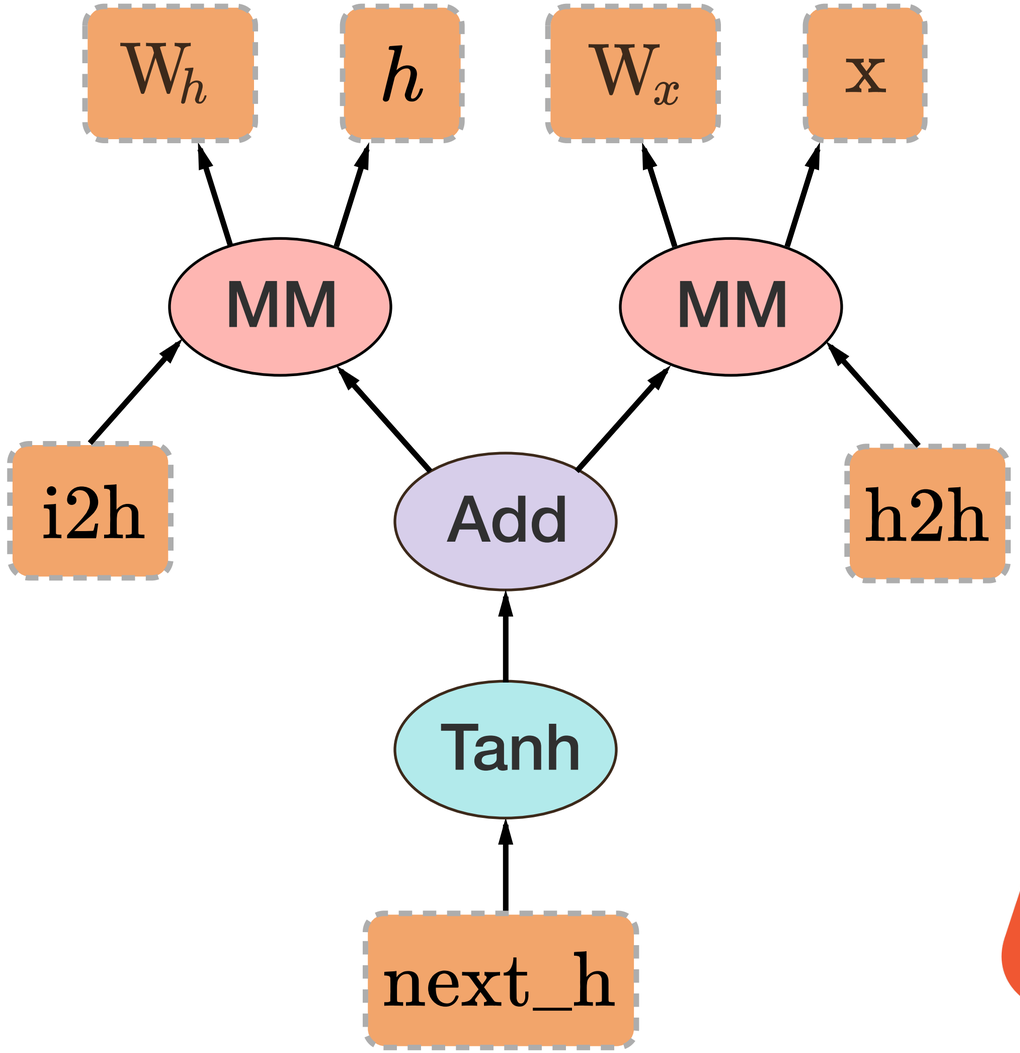
- Provide gradient computation
  - Gradient of one variable w.r.t. any variable in graph



# Deep Learning Frameworks

## In addition to Computation Graph Toolkits

- Provide gradient computation
  - Gradient of one variable w.r.t. any variable in graph
  - For example, can compute: $\frac{dnext_h}{dW_h}$

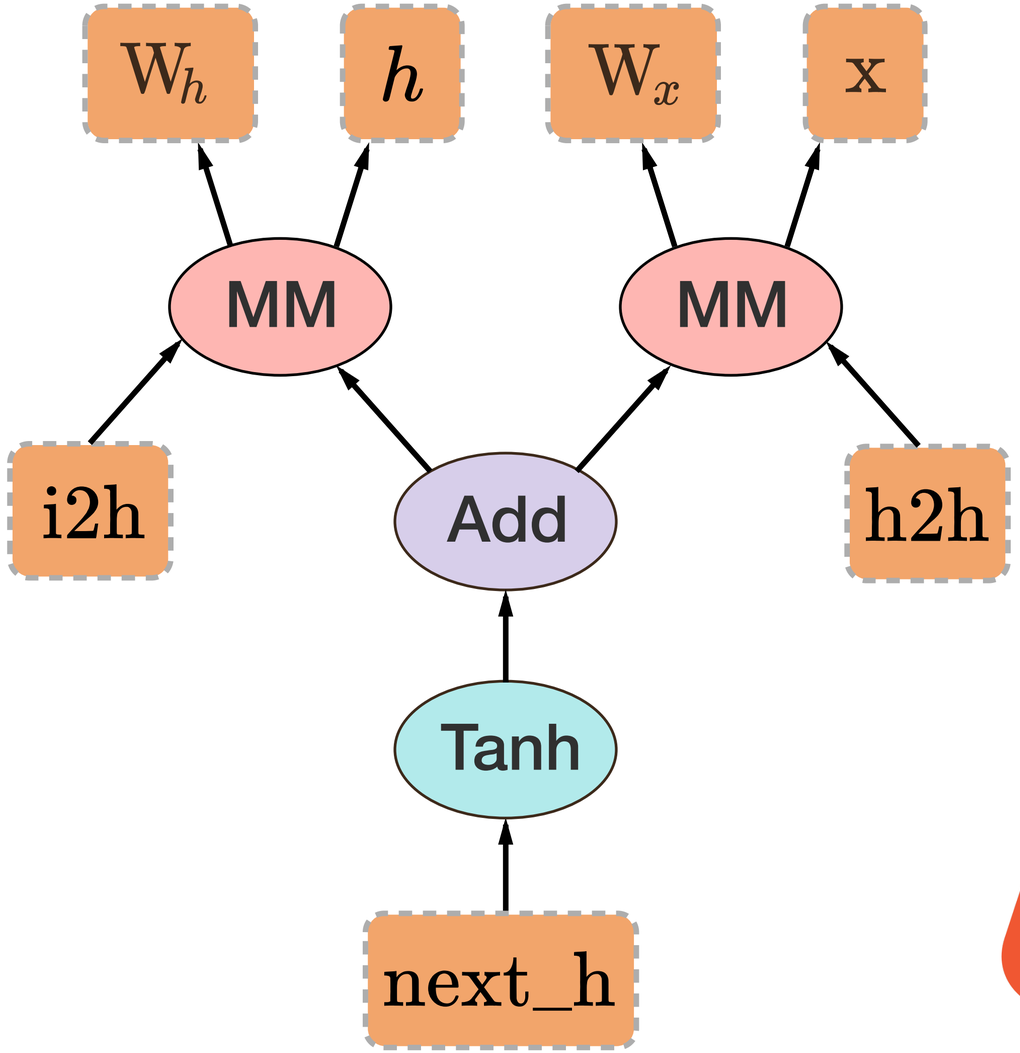


# Deep Learning Frameworks

## In addition to Computation Graph Toolkits

- Provide gradient computation
  - Gradient of one variable w.r.t. any variable in graph
  - For example, can compute: $\frac{dnext_h}{dW_h}$
  - And then you usually do SGD step:

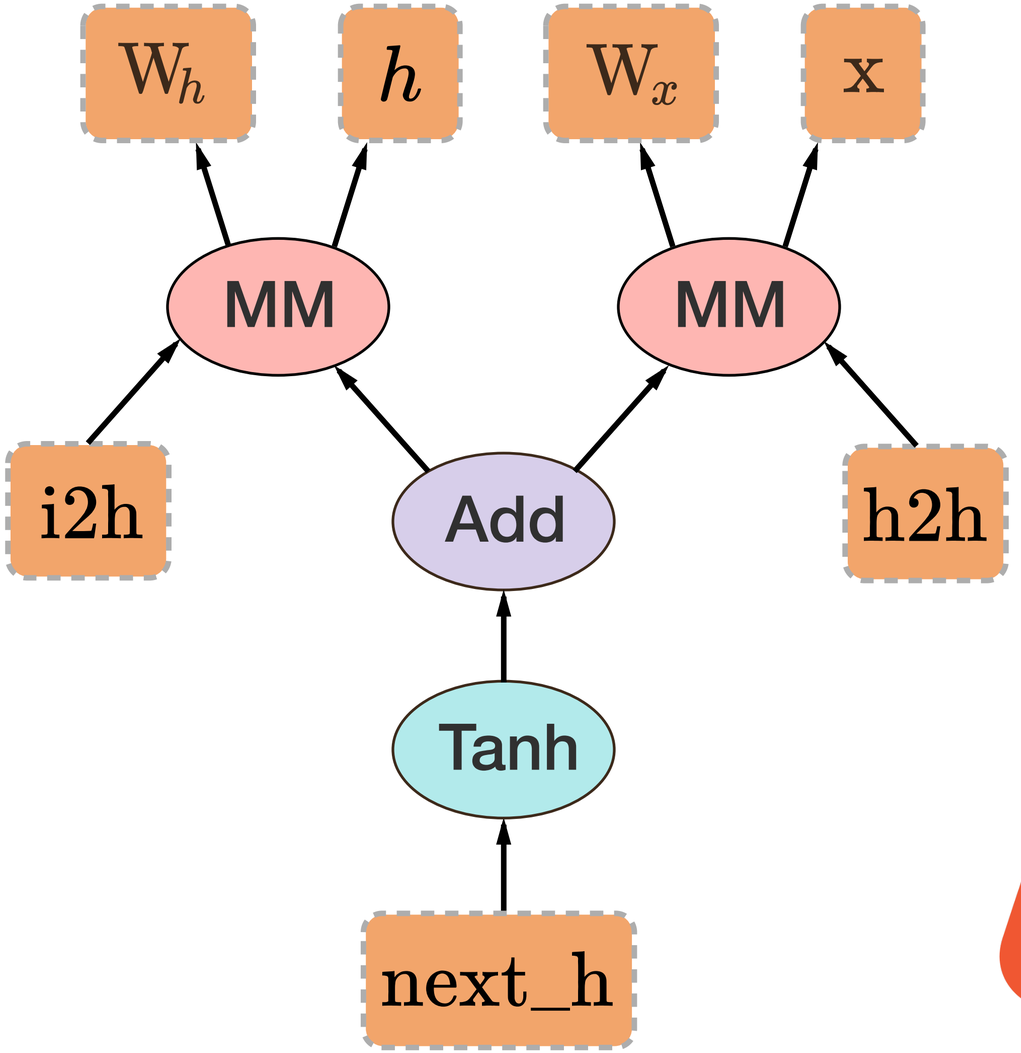
$$W_h = W_h - \alpha * \frac{dnext_h}{dW_h}$$



# Deep Learning Frameworks

## In addition to Computation Graph Toolkits

- Provide integration with high performance DL libraries like CuDNN




# Computation Graph Toolkits

# Computation Graph Toolkits

## Declarative Toolkits

theano

Caffe

# Computation Graph Toolkits

## Declarative Toolkits

- Declare a computation
  - with placeholder variables
- Compile it
- Run it in a Session

# Computation Graph

## Declarative Toolkits

- Declare a computation
  - with placeholder variables
- Compile it
- Run it in a Session

```
1  import tensorflow as tf
2  import numpy as np
3
4  trX = np.linspace(-1, 1, 101)
5  trY = 2 * trX + np.random.randn(*trX.shape) * 0.33
6
7  X = tf.placeholder("float")
8  Y = tf.placeholder("float")
9
10 def model(X, w):
11     return tf.multiply(X, w)
12
13 w = tf.Variable(0.0, name="weights")
14 y_model = model(X, w)
15
16 cost = tf.square(Y - y_model)
17
18 train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
19
20 with tf.Session() as sess:
21     tf.global_variables_initializer().run()
22
23     for i in range(100):
24         for (x, y) in zip(trX, trY):
25             sess.run(train_op, feed_dict={X: x, Y: y})
26
27     print(sess.run(w))
```

# Computation Graph

## Declarative Toolkits

- Declare a computation
  - with placeholder variables
- Compile it
- Run it in a Session

**A separate turing-complete Virtual Machine**

```
1  import tensorflow as tf
2  import numpy as np
3
4  trX = np.linspace(-1, 1, 101)
5  trY = 2 * trX + np.random.randn(*trX.shape) * 0.33
6
7  X = tf.placeholder("float")
8  Y = tf.placeholder("float")
9
10 def model(X, w):
11     return tf.multiply(X, w)
12
13 w = tf.Variable(0.0, name="weights")
14 y_model = model(X, w)
15
16 cost = tf.square(Y - y_model)
17
18 train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
19
20 with tf.Session() as sess:
21     tf.global_variables_initializer().run()
22
23     for i in range(100):
24         for (x, y) in zip(trX, trY):
25             sess.run(train_op, feed_dict={X: x, Y: y})
26
27     print(sess.run(w))
```

# Computation Graph Toolkits

## Declarative Toolkits

- Declare a computation
  - with placeholder variables
- Compile it
- Run it in a Session

**Can handle loops, conditionals (if, scan, while, etc.)**

```
from __future__ import division, print_function
import tensorflow as tf

def fn(previous_output, current_input):
    return previous_output + current_input

elems = tf.Variable([1.0, 2.0, 2.0, 2.0])
elems = tf.identity(elems)
initializer = tf.constant(0.0)
out = tf.scan(fn, elems, initializer=initializer)

with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    print(sess.run(out))
```

# Computation Graph

## Declarative Toolkits

- Declare a computation
  - with placeholder variables
- Compile it
- Run it in a Session

**Has it's own execution engine**

```
1  import tensorflow as tf
2  import numpy as np
3
4  trX = np.linspace(-1, 1, 101)
5  trY = 2 * trX + np.random.randn(*trX.shape) * 0.33
6
7  X = tf.placeholder("float")
8  Y = tf.placeholder("float")
9
10 def model(X, w):
11     return tf.multiply(X, w)
12
13 w = tf.Variable(0.0, name="weights")
14 y_model = model(X, w)
15
16 cost = tf.square(Y - y_model)
17
18 train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
19
20 with tf.Session() as sess:
21     tf.global_variables_initializer().run()
22
23     for i in range(100):
24         for (x, y) in zip(trX, trY):
25             sess.run(train_op, feed_dict={X: x, Y: y})
26
27     print(sess.run(w))
```

# Computation Graph

## Declarative Toolkits

- Declare a computation
  - with placeholder variables
- Compile it
- Run it in a Session

**Has it's own compiler**

- **fuse operations**
- **reuse memory**
- **do optimizations**

```
1  import tensorflow as tf
2  import numpy as np
3
4  trX = np.linspace(-1, 1, 101)
5  trY = 2 * trX + np.random.randn(*trX.shape) * 0.33
6
7  X = tf.placeholder("float")
8  Y = tf.placeholder("float")
9
10 def model(X, w):
11     return tf.multiply(X, w)
12
13 w = tf.Variable(0.0, name="weights")
14 y_model = model(X, w)
15
16 cost = tf.square(Y - y_model)
17
18 train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
19
20 with tf.Session() as sess:
21     tf.global_variables_initializer().run()
22
23     for i in range(100):
24         for (x, y) in zip(trX, trY):
25             sess.run(train_op, feed_dict={X: x, Y: y})
26
27     print(sess.run(w))
```

# Computation Graph

## Declarative Toolkits

- Declare a computation
  - with placeholder variables
- Compile it
- Run it in a Session

**Graph can be serialized easily**

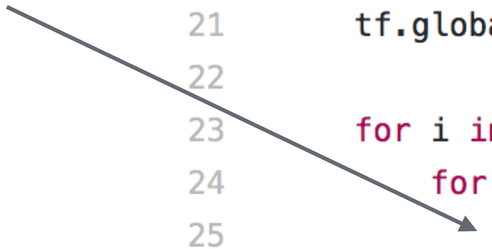
```
1  import tensorflow as tf
2  import numpy as np
3
4  trX = np.linspace(-1, 1, 101)
5  trY = 2 * trX + np.random.randn(*trX.shape) * 0.33
6
7  X = tf.placeholder("float")
8  Y = tf.placeholder("float")
9
10 def model(X, w):
11     return tf.multiply(X, w)
12
13 w = tf.Variable(0.0, name="weights")
14 y_model = model(X, w)
15
16 cost = tf.square(Y - y_model)
17
18 train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
19
20 with tf.Session() as sess:
21     tf.global_variables_initializer().run()
22
23     for i in range(100):
24         for (x, y) in zip(trX, trY):
25             sess.run(train_op, feed_dict={X: x, Y: y})
26
27     print(sess.run(w))
```

# Computation Graph

## Declarative Toolkits

- Declare a computation
  - with placeholder variables
- Compile it
- Run it in a Session

## Own Virtual Machine



```
1  import tensorflow as tf
2  import numpy as np
3
4  trX = np.linspace(-1, 1, 101)
5  trY = 2 * trX + np.random.randn(*trX.shape) * 0.33
6
7  X = tf.placeholder("float")
8  Y = tf.placeholder("float")
9
10 def model(X, w):
11     return tf.multiply(X, w)
12
13 w = tf.Variable(0.0, name="weights")
14 y_model = model(X, w)
15
16 cost = tf.square(Y - y_model)
17
18 train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
19
20 with tf.Session() as sess:
21     tf.global_variables_initializer().run()
22
23     for i in range(100):
24         for (x, y) in zip(trX, trY):
25             sess.run(train_op, feed_dict={X: x, Y: y})
26
27     print(sess.run(w))
```

# Computation Graph

## Declarative Toolkits

- Declare a computation
  - with placeholder variables
- Compile it
- Run it in a Session

**Own Virtual Machine**

- **separate debugging tools**

```
1  import tensorflow as tf
2  import numpy as np
3
4  trX = np.linspace(-1, 1, 101)
5  trY = 2 * trX + np.random.randn(*trX.shape) * 0.33
6
7  X = tf.placeholder("float")
8  Y = tf.placeholder("float")
9
10 def model(X, w):
11     return tf.multiply(X, w)
12
13 w = tf.Variable(0.0, name="weights")
14 y_model = model(X, w)
15
16 cost = tf.square(Y - y_model)
17
18 train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
19
20 with tf.Session() as sess:
21     tf.global_variables_initializer().run()
22
23     for i in range(100):
24         for (x, y) in zip(trX, trY):
25             sess.run(train_op, feed_dict={X: x, Y: y})
26
27     print(sess.run(w))
```

# Computation Graph

## Declarative Toolkits

- Declare a computation
  - with placeholder variables
- Compile it
- Run it in a Session

**Own Virtual Machine**

- **separate debugging tools**
- **non-linear thinking for user**

```
1  import tensorflow as tf
2  import numpy as np
3
4  trX = np.linspace(-1, 1, 101)
5  trY = 2 * trX + np.random.randn(*trX.shape) * 0.33
6
7  X = tf.placeholder("float")
8  Y = tf.placeholder("float")
9
10 def model(X, w):
11     return tf.multiply(X, w)
12
13 w = tf.Variable(0.0, name="weights")
14 y_model = model(X, w)
15
16 cost = tf.square(Y - y_model)
17
18 train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
19
20 with tf.Session() as sess:
21     tf.global_variables_initializer().run()
22
23     for i in range(100):
24         for (x, y) in zip(trX, trY):
25             sess.run(train_op, feed_dict={X: x, Y: y})
26
27     print(sess.run(w))
```

# Imperative Toolkits

# Computation Graph Toolkits

## Imperative Toolkits

- Run a computation
- computation is run


**HIPS Autograd**

**Dynet**


# Computation Graph

## Imperative Toolkits

- Run a computation
- computation is run!

```
1   import torch
2   from torch.autograd import Variable
3
4   trX = torch.linspace(-1, 1, 101)
5   trY = 2 * trX + torch.randn(*trX.size()) * 0.33
6
7   w = Variable(trX.new([0.0]), requires_grad=True)
8
9   for i in range(100):
10    for (x, y) in zip(trX, trY):
11      X = Variable(x)
12      Y = Variable(y)
13
14      y_model = X * w.expand_as(X)
15      cost = (Y - Y_model) ** 2
16      cost.backward(torch.ones(*cost.size()))
17
18      w.data = w.data + 0.01 * w.grad.data
19
20    print(w)
```

# Computation Graph

## Imperative Toolkits

- Run a computation
- computation is run!

```
1   import torch
2   from torch.autograd import Variable
3
4   trX = torch.linspace(-1, 1, 101)
5   trY = 2 * trX + torch.randn(*trX.size()) * 0.33
6
7   w = Variable(trX.new([0.0]), requires_grad=True)
8
9   for i in range(100):
10    for (x, y) in zip(trX, trY):
11      X = Variable(x)
12      Y = Variable(y)
13
14      y_model = X * w.expand_as(X)
15      cost = (Y - Y_model) ** 2
16      cost.backward(torch.ones(*cost.size()))
17
18      w.data = w.data + 0.01 * w.grad.data
19
20    print(w)
```

# Computation Graph

## Imperative Toolkits

- Run a computation
- computation is run!
- no separate execution engine

```
1   import torch
2   from torch.autograd import Variable
3
4   trX = torch.linspace(-1, 1, 101)
5   trY = 2 * trX + torch.randn(*trX.size()) * 0.33
6
7   w = Variable(trX.new([0.0]), requires_grad=True)
8
9   for i in range(100):
10    for (x, y) in zip(trX, trY):
11      X = Variable(x)
12      Y = Variable(y)
13
14      y_model = X * w.expand_as(X)
15      cost = (Y - Y_model) ** 2
16      cost.backward(torch.ones(*cost.size()))
17
18      w.data = w.data + 0.01 * w.grad.data
19
20    print(w)
```

# Computation Graph

## Imperative Toolkits

- Run a computation
- computation is run!
- no separate execution engine
- debugging is easy

```
1   import torch
2   from torch.autograd import Variable
3
4   trX = torch.linspace(-1, 1, 101)
5   trY = 2 * trX + torch.randn(*trX.size()) * 0.33
6
7   w = Variable(trX.new([0.0]), requires_grad=True)
8
9   for i in range(100):
10    for (x, y) in zip(trX, trY):
11      X = Variable(x)
12      Y = Variable(y)
13
14      y_model = X * w.expand_as(X)
15      cost = (Y - Y_model) ** 2
16      cost.backward(torch.ones(*cost.size()))
17
18      w.data = w.data + 0.01 * w.grad.data
19
20    print(w)
```

# Computation Graph

## Imperative Toolkits

- Run a computation
- computation is run!
- no separate execution engine
- debugging is easy

```
1   import torch
2   from torch.autograd import Variable
3
4   trX = torch.linspace(-1, 1, 101)
5   trY = 2 * trX + torch.randn(*trX.size()) * 0.33
6
7   w = Variable(trX.new([0.0]), requires_grad=True)
8
9   for i in range(100):
10    for (x, y) in zip(trX, trY):
11      X = Variable(x)
12      Y = Variable(y)
13
14      y_model = X * w.expand_as(X)
15      cost = (Y - Y_model) ** 2
16      cost.backward(torch.ones(*cost.size()))
17
18      w.data = w.data + 0.01 * w.grad.data
19
20    print(w)
```

# Computation Graph

## Imperative Toolkits

- Run a computation
- computation is run!
- no separate execution engine
- debugging is easy

```
print(x)
y = foo(x)
print(y)
```

```python
import torch
from torch.autograd import Variable

trX = torch.linspace(-1, 1, 101)
trY = 2 * trX + torch.randn(*trX.size()) * 0.33

w = Variable(trX.new([0.0]), requires_grad=True)

for i in range(100):
  for (x, y) in zip(trX, trY):
    X = Variable(x)
    Y = Variable(y)

    y_model = X * w.expand_as(X)
    cost = (Y - Y_model) ** 2
    cost.backward(torch.ones(*cost.size()))

    w.data = w.data + 0.01 * w.grad.data

  print(w)
```

# Computation Graph

## Imperative Toolkits

- Run a computation
- computation is run!
- no separate execution engine
- debugging is easy

print(”hello”)
y = foo(x)
print(”hello2”)

```
1   import torch
2   from torch.autograd import Variable
3
4   trX = torch.linspace(-1, 1, 101)
5   trY = 2 * trX + torch.randn(*trX.size()) * 0.33
6
7   w = Variable(trX.new([0.0]), requires_grad=True)
8
9   for i in range(100):
10    for (x, y) in zip(trX, trY):
11      X = Variable(x)
12      Y = Variable(y)
13
14      y_model = X * w.expand_as(X)
15      cost = (Y - Y_model) ** 2
16      cost.backward(torch.ones(*cost.size()))
17
18      w.data = w.data + 0.01 * w.grad.data
19
20    print(w)
```

# Computation Graph T...

## Imperative Toolkits

- Run a computation
- computation is run!
- no separate execution engine
- debugging is easy

```
print("hello")
y = foo(x)
print("hello2")
```

```
1  import torch
2  from torch.autograd import Variable
3
4  trX = torch.linspace(-1, 1, 101)
5  trY = 2 * trX + torch.randn(*trX.size()) * 0.33
6
7  w = Variable(trX.new([0.0]), requires_grad=True)
8
9  for i in range(100):
10   for (x, y) in zip(trX, trY):
11     X = Variable(x)
12     Y = Variable(y)
13     print(Y)
14
15     y_model = X * w.expand_as(X)
16     cost = (Y - Y_model) ** 2
17     cost.backward(torch.ones(*cost.size()))
18
19     w.data = w.data + 0.01 * w.grad.data
20
21   print(w)
```

# Computation Graph

## Imperative Toolkits

- Run a computation
- computation is run!
- no separate execution engine
- debugging is easy
- Linear program flow
  - Linear thinking for user

```
1   import torch
2   from torch.autograd import Variable
3
4   trX = torch.linspace(-1, 1, 101)
5   trY = 2 * trX + torch.randn(*trX.size()) * 0.33
6
7   w = Variable(trX.new([0.0]), requires_grad=True)
8
9   for i in range(100):
10    for (x, y) in zip(trX, trY):
11      X = Variable(x)
12      Y = Variable(y)
13      print(Y)
14
15      y_model = X * w.expand_as(X)
16      cost = (Y - Y_model) ** 2
17      cost.backward(torch.ones(*cost.size()))
18
19      w.data = w.data + 0.01 * w.grad.data
20
21    print(w)
```

# Computation Graph

## Imperative Toolkits

- Run a computation
- computation is run!
- no separate execution engine
- Cannot compile program
- Linear program flow
  - Linear thinking for user

```
1   import torch
2   from torch.autograd import Variable
3
4   trX = torch.linspace(-1, 1, 101)
5   trY = 2 * trX + torch.randn(*trX.size()) * 0.33
6
7   w = Variable(trX.new([0.0]), requires_grad=True)
8
9   for i in range(100):
10    for (x, y) in zip(trX, trY):
11      X = Variable(x)
12      Y = Variable(y)
13      print(Y)
14
15      y_model = X * w.expand_as(X)
16      cost = (Y - Y_model) ** 2
17      cost.backward(torch.ones(*cost.size()))
18
19      w.data = w.data + 0.01 * w.grad.data
20
21    print(w)
```

# Computation Graph

## Imperative Toolkits

- Run a computation
- computation is run!
- no separate execution engine
- Cannot compile program
- Cannot optimize
- Linear thinking for user

```
1   import torch
2   from torch.autograd import Variable
3
4   trX = torch.linspace(-1, 1, 101)
5   trY = 2 * trX + torch.randn(*trX.size()) * 0.33
6
7   w = Variable(trX.new([0.0]), requires_grad=True)
8
9   for i in range(100):
10    for (x, y) in zip(trX, trY):
11      X = Variable(x)
12      Y = Variable(y)
13      print(Y)
14
15      y_model = X * w.expand_as(X)
16      cost = (Y - Y_model) ** 2
17      cost.backward(torch.ones(*cost.size()))
18
19      w.data = w.data + 0.01 * w.grad.data
20
21    print(w)
```

# Computation Graph

## Imperative Toolkits

- Run a computation
- computation is run!
- no separate execution engine
- Cannot compile program
- Cannot optimize
- Cannot do static analysis

```
1   import torch
2   from torch.autograd import Variable
3
4   trX = torch.linspace(-1, 1, 101)
5   trY = 2 * trX + torch.randn(*trX.size()) * 0.33
6
7   w = Variable(trX.new([0.0]), requires_grad=True)
8
9   for i in range(100):
10    for (x, y) in zip(trX, trY):
11      X = Variable(x)
12      Y = Variable(y)
13      print(Y)
14
15      y_model = X * w.expand_as(X)
16      cost = (Y - Y_model) ** 2
17      cost.backward(torch.ones(*cost.size()))
18
19      w.data = w.data + 0.01 * w.grad.data
20
21    print(w)
```

# Computation Graph

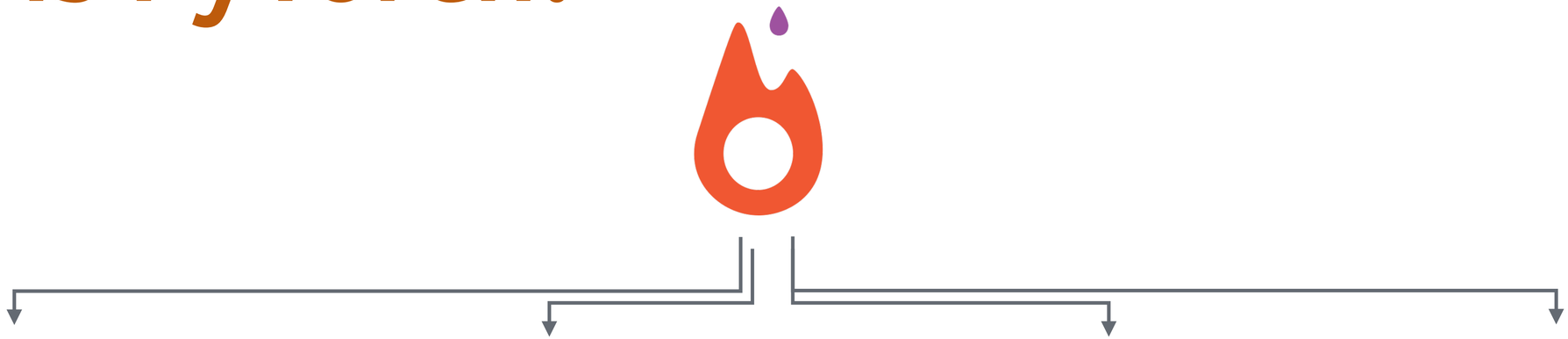
## Imperative Toolkits

- Run a computation
- computation is run!
- no separate execution engine
- Cannot compile program
- Cannot optimize
- Cannot do static analysis (more on this later)

```
1   import torch
2   from torch.autograd import Variable
3
4   trX = torch.linspace(-1, 1, 101)
5   trY = 2 * trX + torch.randn(*trX.size()) * 0.33
6
7   w = Variable(trX.new([0.0]), requires_grad=True)
8
9   for i in range(100):
10    for (x, y) in zip(trX, trY):
11      X = Variable(x)
12      Y = Variable(y)
13      print(Y)
14
15      y_model = X * w.expand_as(X)
16      cost = (Y - Y_model) ** 2
17      cost.backward(torch.ones(*cost.size()))
18
19      w.data = w.data + 0.01 * w.grad.data
20
21    print(w)
```

# PyTorch

# What is PyTorch?



automatic differentiation engine

**Ndarray library with GPU support**

gradient based optimization package

Utilities (data loading, etc.)

Deep Learning

Reinforcement Learning

Numpy-alternative

# ndarray library

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

```numpy
# -*- coding: utf-8 -*-
import numpy as np

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

# Randomly initialize weights
w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.dot(w1)
    h_relu = np.maximum(h, 0)
    y_pred = h_relu.dot(w2)

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h < 0] = 0
    grad_w1 = x.T.dot(grad_h)

    # Update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Numpy

```pytorch
import torch


dtype = torch.FloatTensor
# dtype = torch.cuda.FloatTensor # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

# Randomly initialize weights
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    # Update weights using gradient descent
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch

# ndarray / Tensor library

Tensors are similar to numpy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.

```
from __future__ import print_function
import torch
```

Construct a 5x3 matrix, uninitialized:

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

Out:

```
1.00000e-25 *
  0.4136  0.0000  0.0000
  0.0000  1.6519  0.0000
  1.6518  0.0000  1.6519
  0.0000  1.6518  0.0000
  1.6520  0.0000  1.6519
[torch.FloatTensor of size 5x3]
```

# ndarray / Tensor library

Construct a randomly initialized matrix

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

Out:

```
0.2598  0.7231  0.8534
 0.3928  0.1244  0.5110
 0.5476  0.2700  0.5856
 0.7288  0.9455  0.8749
 0.6663  0.8230  0.2713
[torch.FloatTensor of size 5x3]
```

Get its size

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

Out:

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

# ndarray / Tensor library

You can use standard numpy-like indexing with all bells and whistles!

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

Out:

```
0.7231
 0.1244
 0.2700
 0.9455
 0.8230
[torch.FloatTensor of size 5]
```

# ndarray / Tensor library

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

Out:

```
0.7931  1.1872  1.6143
 1.1946  0.4669  0.9639
 0.7576  0.8136  1.1897
 0.7431  1.8579  1.3400
 0.8188  1.1041  0.8914
[torch.FloatTensor of size 5x3]
```

# NumPy bridge

## Converting torch Tensor to numpy Array

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

Out:

```
1
 1
 1
 1
 1
[torch.FloatTensor of size 5]
```

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

Out:

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

# NumPy bridge

### Converting torch Tensor to numpy Array

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

Out:

```
1
 1
 1
 1
 1
[torch.FloatTensor of size 5]
```

**Zero memory-copy**
**very efficient**

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

Out:

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

# NumPy bridge

See how the numpy array changed in value.

```
a.add_(1)
print(a)
print(b)
```

Out:

```
2
 2
 2
 2
 2
[torch.FloatTensor of size 5]

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

# NumPy bridge

### Converting numpy Array to torch Tensor

See how changing the np array changed the torch Tensor automatically

```
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

Out:

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

 2
 2
 2
 2
 2
[torch.DoubleTensor of size 5]
```

All the Tensors on the CPU except a CharTensor support converting to NumPy and back.

# Seamless GPU Tensors

## CUDA Tensors

Tensors can be moved onto GPU using the `.cuda` function.

```
# let us run this cell only if CUDA is available
if torch.cuda.is_available():
    x = x.cuda()
    y = y.cuda()
    x + y
```

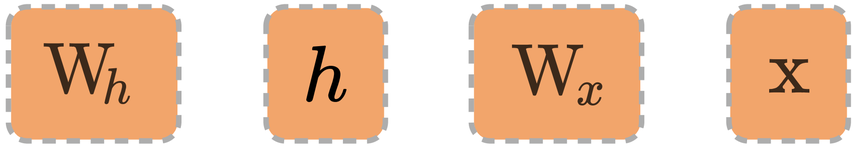
# automatic differentiation engine

for deep learning and reinforcement learning

# PyTorch Autograd

```
from torch.autograd import Variable
```

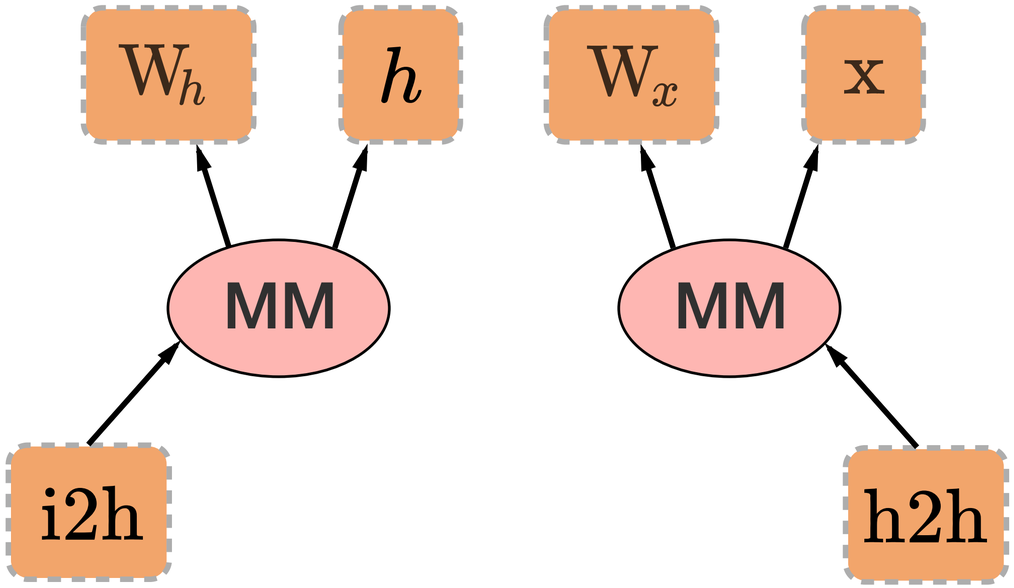
# PyTorch Autograd




```
from torch.autograd import Variable

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

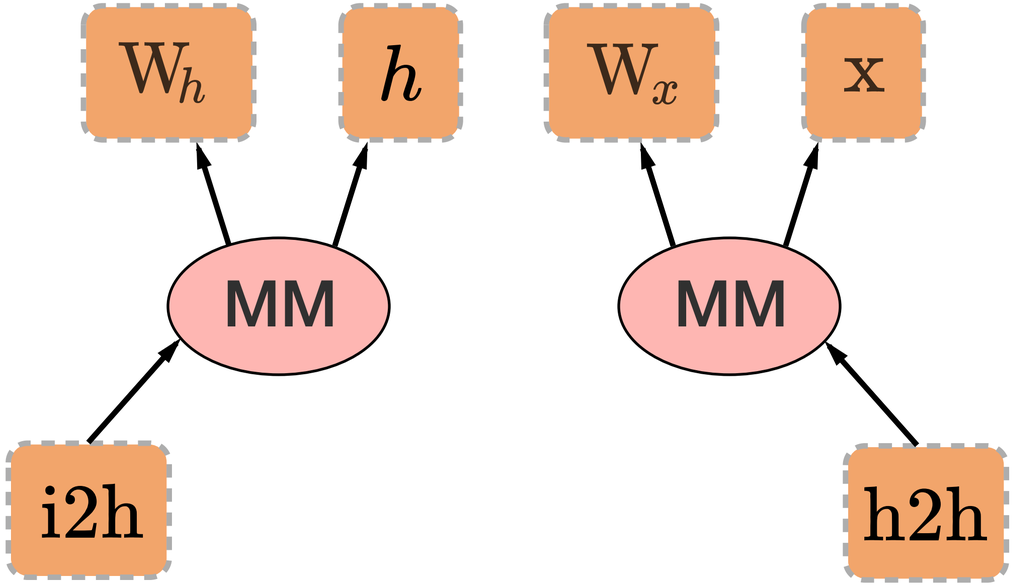
# PyTorch Autograd

```
from torch.autograd import Variable

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

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
```



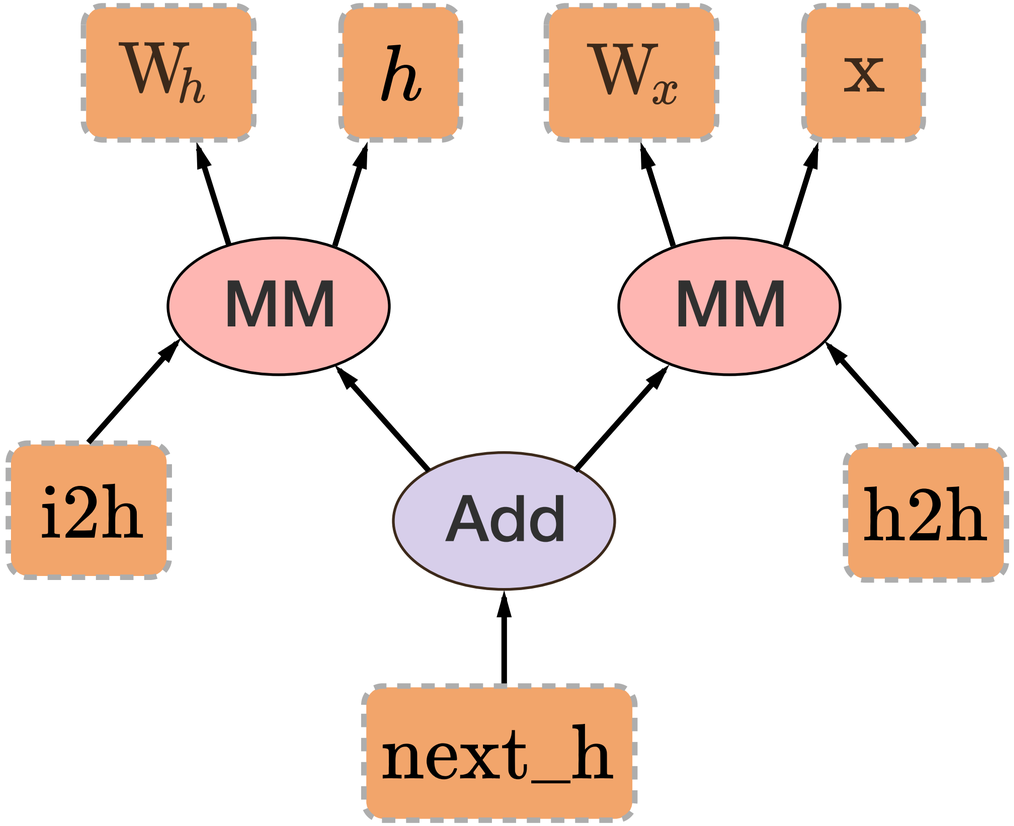
# PyTorch Autograd

```
from torch.autograd import Variable

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

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
```



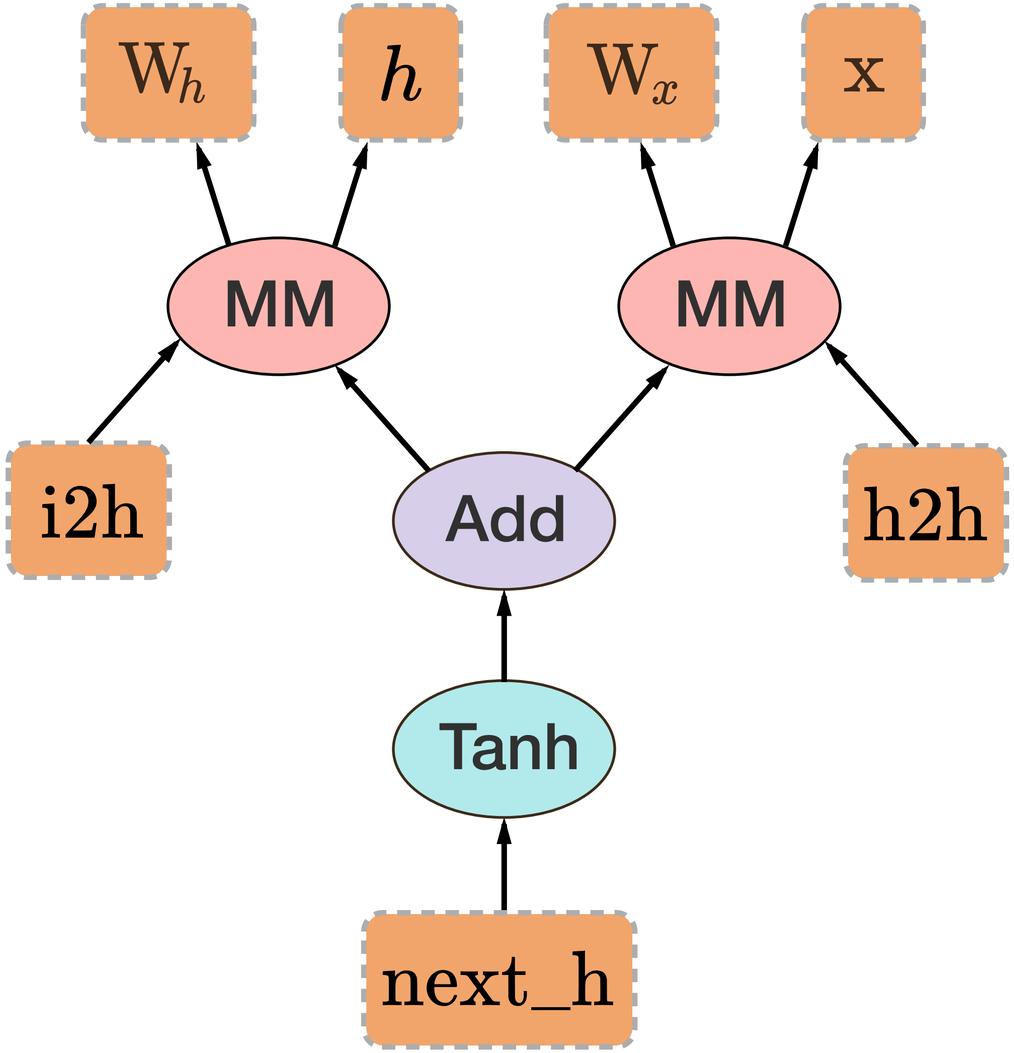
# PyTorch Autograd

```
from torch.autograd import Variable

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

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
```



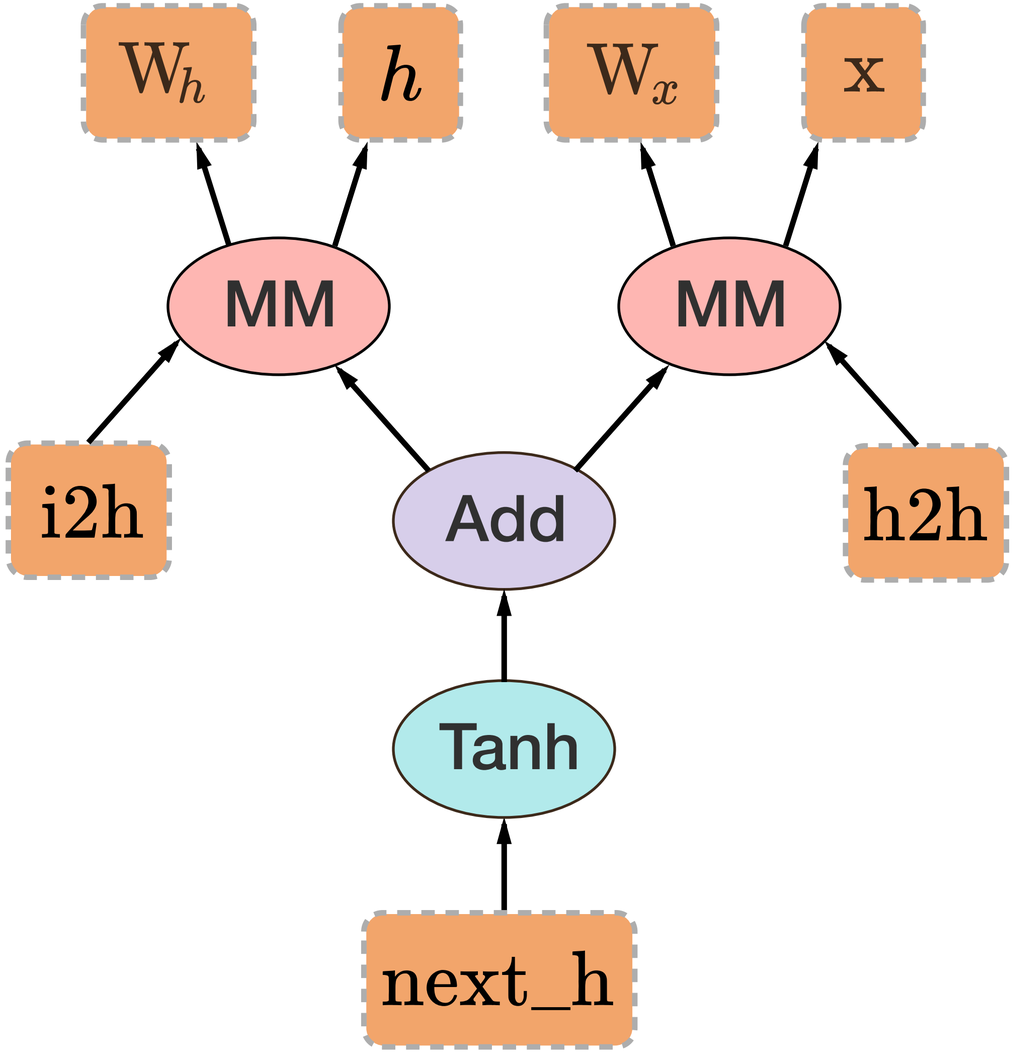
# PyTorch Autograd

```
from torch.autograd import Variable

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

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()
```



# PyTorch Autograd

```
from torch.autograd import Variable

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

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
```



# Neural Networks

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
```

# Neural Networks

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
```

# Neural Networks

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
```

# Optimization package

SGD, Adagrad, RMSProp, LBFGS, etc.

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

ML Workflows → Pain Points → Core Philosophy (of PyTorch) → Upcoming features

what do they look like in the deep learning space

how does PyTorch deal with them?

# ML Workflows

# ML Workflows


# Work items in practice

Writing Dataset loaders

Building models

Implementing Training loop

Checkpointing models

Interfacing with environments

Building optimizers

Dealing with GPUs

Building Baselines

# Work items in practice

Writing Dataset loaders

Building models

Implementing Training loop

Checkpointing models

Python + PyTorch - an environment to do all of this

Interfacing with environments

Building optimizers

Dealing with GPUs

Building Baselines

# Writing Data Loaders

- every dataset is slightly differently formatted

# Writing Data Loaders

- every dataset is slightly differently formatted
- have to be preprocessed and normalized differently

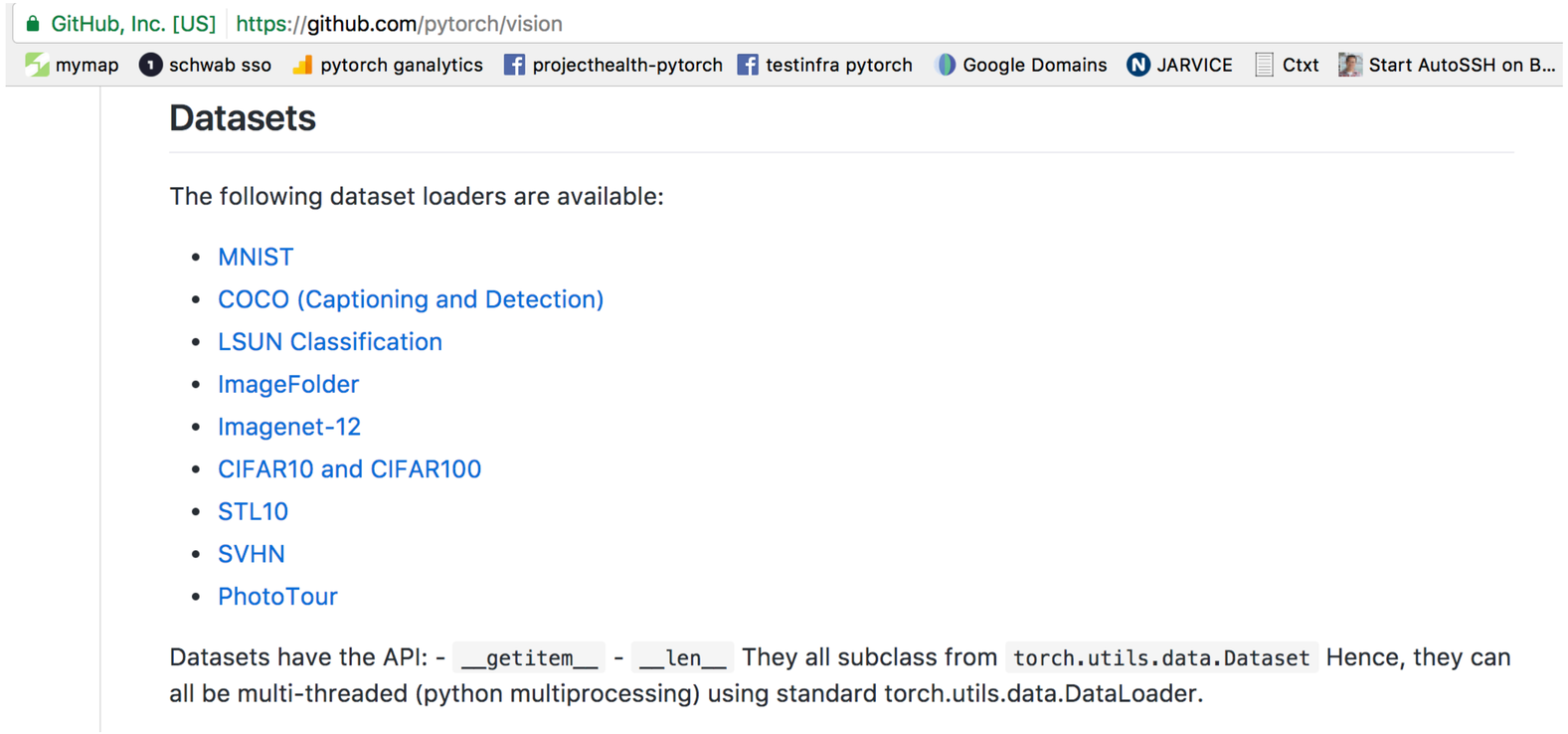
# Writing Data Loaders

- every dataset is slightly differently formatted
- have to be preprocessed and normalized differently
- need a multithreaded Data loader to feed GPUs fast enough

# Writing Data Loaders

## PyTorch solution:

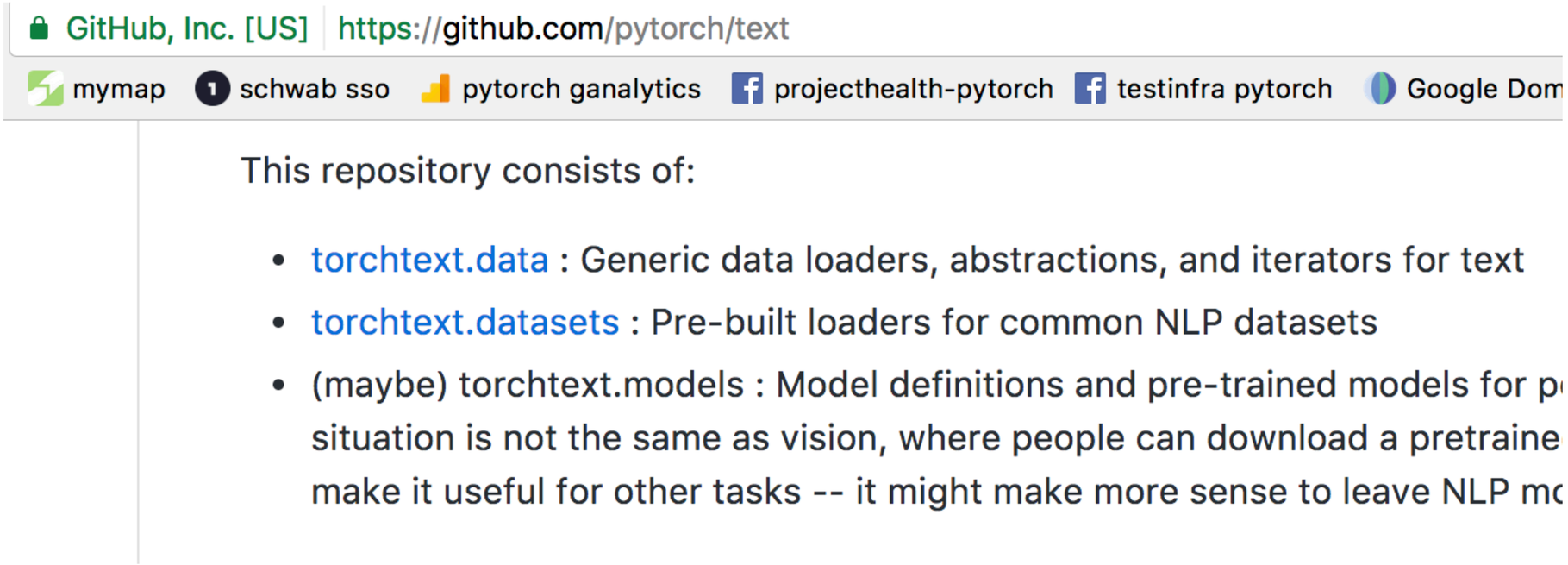
- share data loaders across the community!




# Writing Data Loaders

## PyTorch solution:

- share data loaders across the community!




# Writing Data Loaders

## PyTorch solution:

- use regular Python to write Datasets:

  leverage existing Python code

# Writing Data Loaders

## PyTorch solution:

- use regular Python to write Datasets:

  leverage existing Python code

  Example: ParlAI

ParlAI

---

ParlAI (pronounced “par-lay”) is a framework for dialog AI research, implemented in Python.

Its goal is to provide researchers:

- a unified framework for training and testing dialog models
- multi-task training over many datasets at once
- seamless integration of Amazon Mechanical Turk for data collection and human evaluation

Over 20 tasks are supported in the first release, including popular datasets such as SQuAD, bAbI tasks, MCTest, WikiQA, WebQuestions, SimpleQuestions, WikiMovies, QACNN & QADailyMail, CBT, BookTest, bAbI Dialog tasks, Ubuntu Dialog, OpenSubtitles, Cornell Movie and VQA-COCO2014.

# Writing Data Loaders

## PyTorch solution:

- Code in practice

# Writin

## PyTorch

- Code in p

```
if opt.dataset in ['imagenet', 'folder', 'lfw']:
    # folder dataset
    dataset = dset.ImageFolder(root=opt.dataroot,
                               transform=transforms.Compose([
                                   transforms.Scale(opt.imageSize),
                                   transforms.CenterCrop(opt.imageSize),
                                   transforms.ToTensor(),
                                   transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                               ]))
elif opt.dataset == 'lsun':
    dataset = dset.LSUN(db_path=opt.dataroot, classes=['bedroom_train'],
                        transform=transforms.Compose([
                            transforms.Scale(opt.imageSize),
                            transforms.CenterCrop(opt.imageSize),
                            transforms.ToTensor(),
                            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                        ]))
elif opt.dataset == 'cifar10':
    dataset = dset.CIFAR10(root=opt.dataroot, download=True,
                           transform=transforms.Compose([
                               transforms.Scale(opt.imageSize),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ])
    )
assert dataset
dataloader = torch.utils.data.DataLoader(dataset, batch_size=opt.batchSize,
                                         shuffle=True, num_workers=int(opt.workers))
```

# Writing Da

## PyTorch soluti

- Code in practic

Code for writing
a new
pytorch dataset

```
def __init__(self, root, annFile, transform=None, target_transform=None):
    from pycocotools.coco import COCO
    self.root = os.path.expanduser(root)
    self.coco = COCO(annFile)
    self.ids = list(self.coco.imgs.keys())
    self.transform = transform
    self.target_transform = target_transform

def __getitem__(self, index):
    """
    Args:
        index (int): Index

    Returns:
        tuple: Tuple (image, target). target is a list of captions for the image.
    """
    coco = self.coco
    img_id = self.ids[index]
    ann_ids = coco.getAnnIds(imgIds=img_id)
    anns = coco.loadAnns(ann_ids)
    target = [ann['caption'] for ann in anns]

    path = coco.loadImgs(img_id)[0]['file_name']

    img = Image.open(os.path.join(self.root, path)).convert('RGB')
    if self.transform is not None:
        img = self.transform(img)

    if self.target_transform is not None:
        target = self.target_transform(target)

    return img, target

def __len__(self):
    return len(self.ids)
```

# Writing Da

## PyTorch soluti

- Code in practic

Code for writing a new pytorch dataset

```
def __init__(self, root, annFile, transform=None, target_transform=None):
    from pycocotools.coco import COCO
    self.root = os.path.expanduser(root)
    self.coco = COCO(annFile)
    self.ids = list(self.coco.imgs.keys())
    self.transform = transform
    self.target_transform = target_transform
```

```
def __getitem__(self, index):
    """
    Args:
        index (int): Index

    Returns:
        tuple: Tuple (image, target). target is a list of captions for the image.
    """
    coco = self.coco
    img_id = self.ids[index]
    ann_ids = coco.getAnnIds(imgIds=img_id)
    anns = coco.loadAnns(ann_ids)
    target = [ann['caption'] for ann in anns]

    path = coco.loadImgs(img_id)[0]['file_name']

    img = Image.open(os.path.join(self.root, path)).convert('RGB')
    if self.transform is not None:
        img = self.transform(img)

    if self.target_transform is not None:
        target = self.target_transform(target)

    return img, target
```

```
def __len__(self):
    return len(self.ids)
```

# Writing Da

## PyTorch soluti

- Code in practic

Code for writing a new pytorch dataset

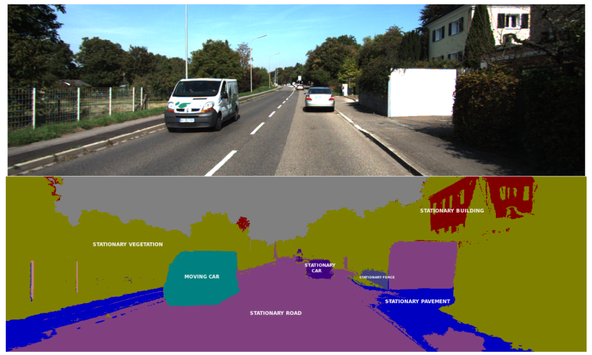
```
def __init__(self, root, annFile, transform=None, target_transform=None):
    from pycocotools.coco import COCO
    self.root = os.path.expanduser(root)
    self.coco = COCO(annFile)
    self.ids = list(self.coco.imgs.keys())
    self.transform = transform
    self.target_transform = target_transform

def __getitem__(self, index):
    """
    Args:
        index (int): Index

    Returns:
        tuple: Tuple (image, target). target is a list of captions for the image.
    """
    coco = self.coco
    img_id = self.ids[index]
    ann_ids = coco.getAnnIds(imgIds=img_id)
    anns = coco.loadAnns(ann_ids)
    target = [ann['caption'] for ann in anns]

    path = coco.loadImgs(img_id)[0]['file_name']

    img = Image.open(os.path.join(self.root, path)).convert('RGB')
    if self.transform is not None:
        img = self.transform(img)

    if self.target_transform is not None:
        target = self.target_transform(target)

    return img, target

def __len__(self):
    return len(self.ids)
```

# Under the hood of Data Loaders

**PyTorch solution:**

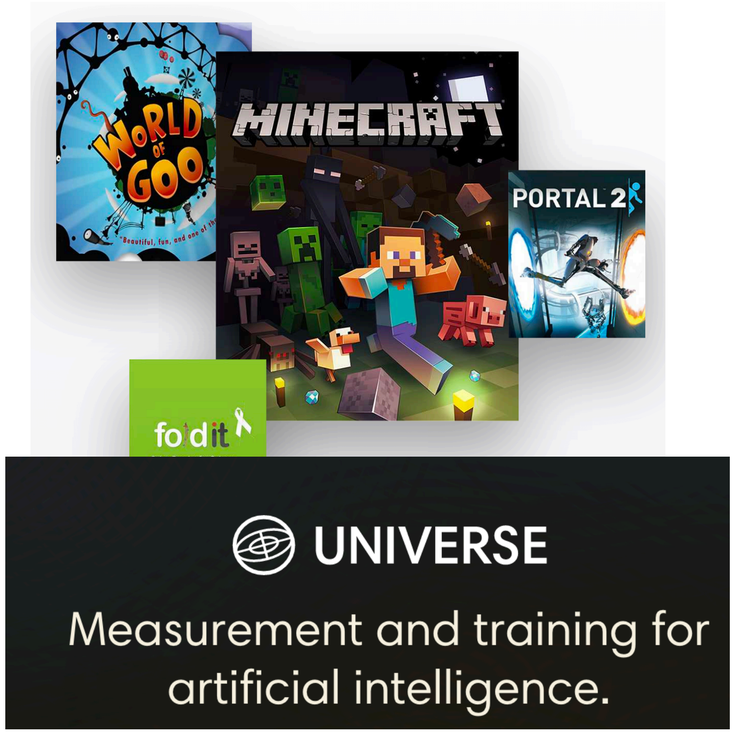
- torch.multiprocessing
  - a minor fork of python multiprocessing
  - custom pickler
  - Tensors are automatically shared across processes

# Interfacing with environments



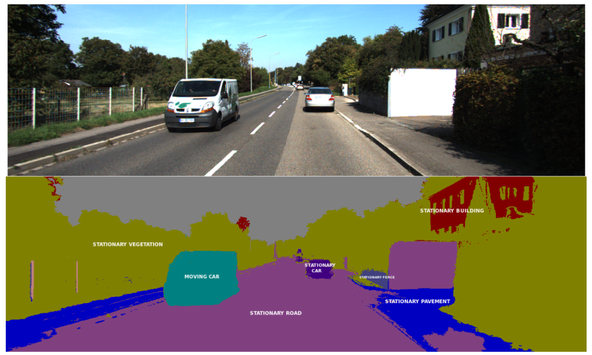

Cars


Video games



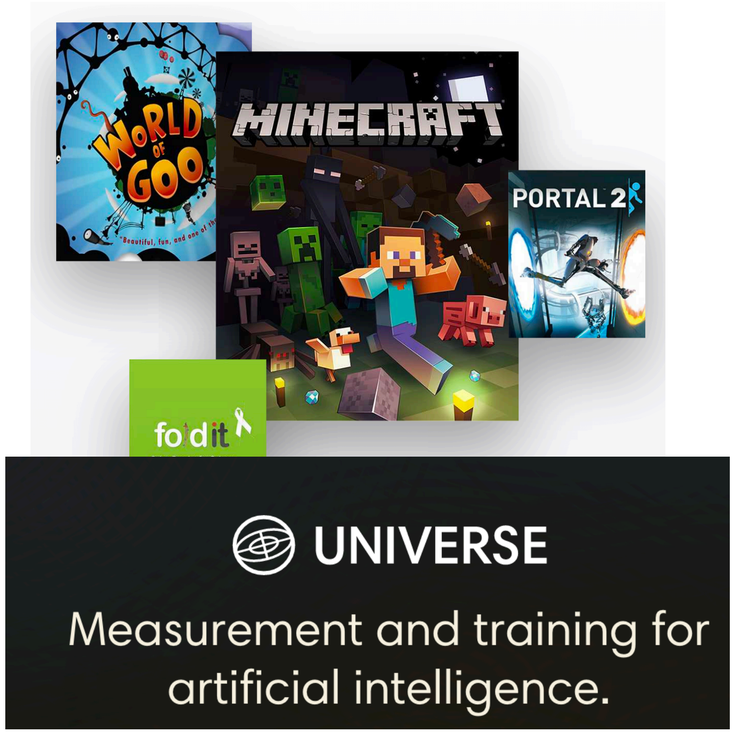

Internet

# Interfacing with environments



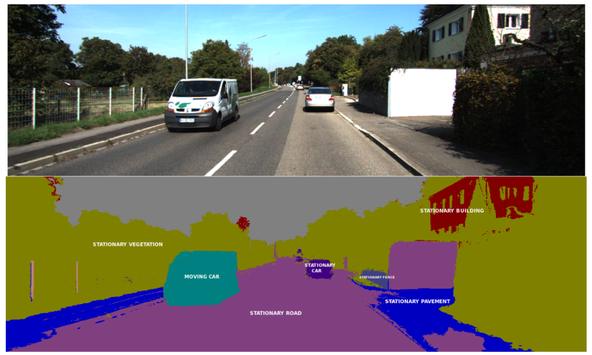

Cars


Video games



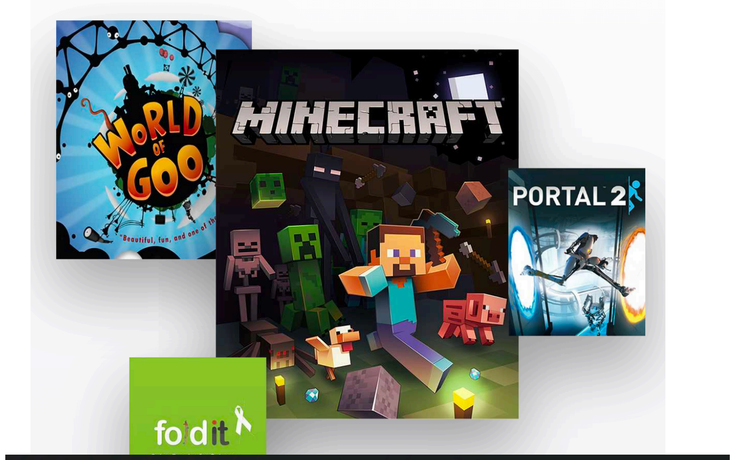

**Pretty much every environment provides a Python API**

# Interfacing with environments




Cars


Video games




**Natively interact with the environment directly**

# Debugging

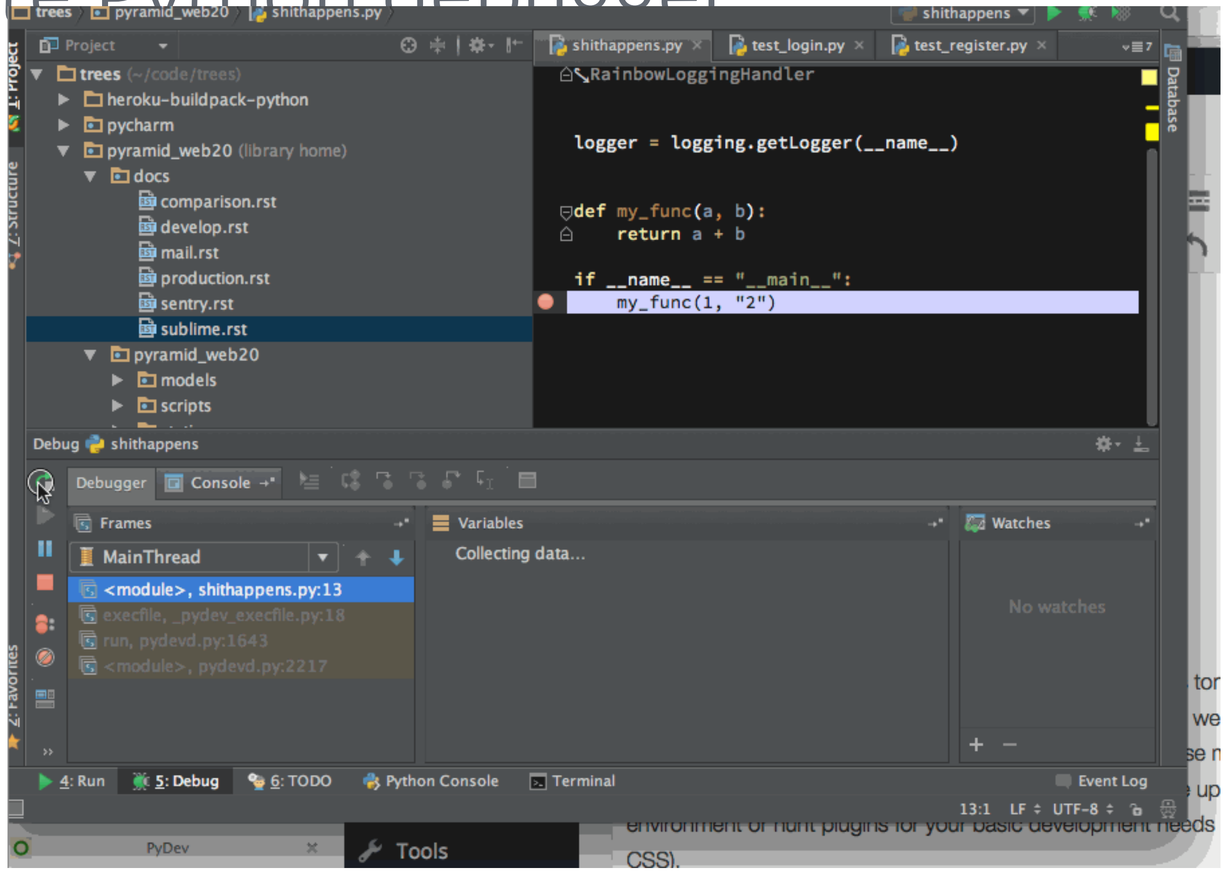
- PyTorch is a Python extension

# Debugging

- PyTorch is a Python extension
- Use your favorite Python debugger

# Debugging

- PyTorch is a Python extension
- Use your favorite Python debugger



# Debugging

- PyTorch is a Python extension
- Use your favorite Python debugger
- Use the most popular debugger:

# Debugging

- PyTorch is a Python extension
- Use your favorite Python debugger
- Use the most popular debugger:

```
print(foo)
```

# Identifying bottlenecks

- PyTorch is a Python extension
- Use your favorite Python profiler

SNAKEVIZ

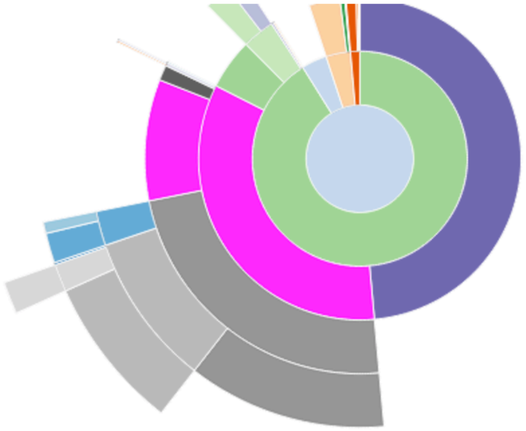
← PREVIOUS NEXT → G

**SnakeViz**
Installation
Starting SnakeViz
Generating Profiles
**Interpreting Results**
Controls
Notes
Contact

## FUNCTION INFO

Placing your cursor over an arc will highlight that arc and any other visible instances of the same function call. It also display a list of information to the left of the sunburst.

**Name:**
filter
**Cumulative Time:**
0.000294 s (31.78 %)
**File:**
fnmatch.py
**Line:**
48
**Directory:**
/Users/jiffyclub/miniconda3/envs/snakevizdev/lib/python3.4/



# Identifying bottlenecks

- PyTorch is a Python extension
- Use your favorite Python profiler: Line_Profiler

```
File: pystone.py
Function: Proc2 at line 149
Total time: 0.606656 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   149                                           @profile
   150                                           def Proc2(IntParIO):
   151     50000        82003      1.6     13.5      IntLoc = IntParIO + 10
   152     50000        63162      1.3     10.4      while 1:
   153     50000        69065      1.4     11.4          if Char1Glob == 'A':
   154     50000        66354      1.3     10.9              IntLoc = IntLoc - 1
   155     50000        67263      1.3     11.1              IntParIO = IntLoc - IntGlob
   156     50000        65494      1.3     10.8              EnumLoc = Ident1
   157     50000        68001      1.4     11.2          if EnumLoc == Ident1:
   158     50000        63739      1.3     10.5              break
   159     50000        61575      1.2     10.1      return IntParIO
```

# Compilation Time

- PyTorch is written for the impatient

# Compilation Time

- PyTorch is written for the impatient
- Absolutely no compilation time when writing your scripts whatsoever

# Compilation Time

- PyTorch is written for the impatient
- Absolutely no compilation time when writing your scripts whatsoever
- All core kernels are pre-compiled

# Ecosystem

- Use the entire Python ecosystem at your will

# Ecosystem

- Use the entire Python ecosystem at your will
- Including SciPy, Scikit-Learn, etc.

# Ecosyster

- Use the entire
- Including SciF

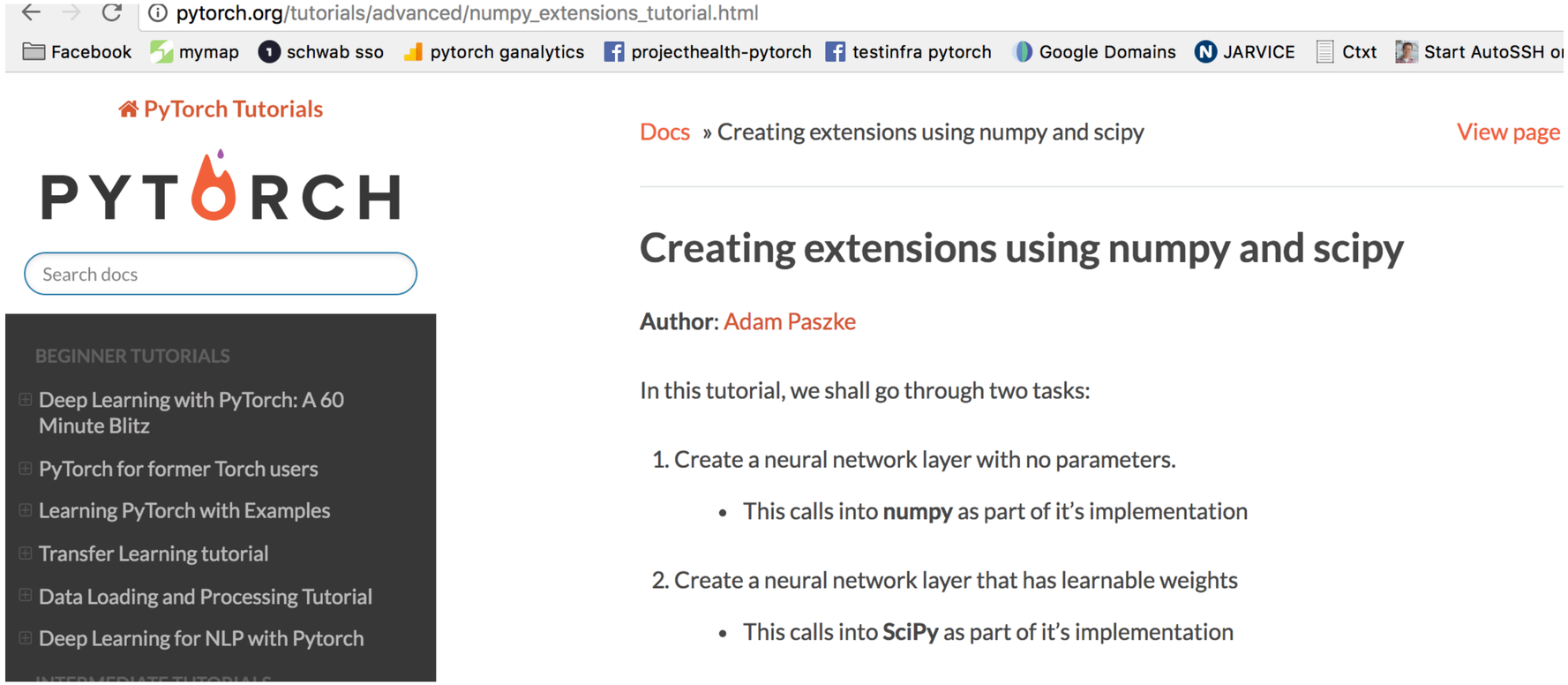
**Brandon Amos** @brandondamos · Jan 18

Why PyTorch's layer creation is powerful: Here's my layer that solves an optimization problem with a primal-dual interior point method.

```
def forward_single(input_i, Q, G, A, b, h, U_Q, U_S, R):
    nineq, nz = G.size()
    neq = A.size(0) if A.ndimension() > 0 else 0

    d = torch.ones(nineq).type_as(Q)
    nb = -b if b is not None else None
    factor_kkt(U_S, R, d, neq)
    x, s, z, y = solve_kkt(U_Q, d, G, A, U_S, input_i, torch.zeros(nineq).type_as(Q),
                           -h, nb, neq, nz)


    if torch.min(s) < 0:
        s -= torch.min(s) - 1
    if torch.min(z) < 0:
        z -= torch.min(z) - 1

    for i in range(20):
        rx = (torch.mv(A.t(),y) if neq > 0 else 0.) + \
             torch.mv(G.t(), z) + torch.mv(Q,x) + input_i
        rs = z
        ry = torch.mv(G,x) + s - h
        rz = torch.mv(A,x) - b if neq > 0 else torch.Tensor([0.])
        mu = torch.dot(s,z)/nineq
        pri_resid = torch.norm(ry) + torch.norm(rz)
        dual_resid = torch.norm(rx)
        d = z/s

        factor_kkt(U_S, R, d, neq)
        dx_aff, ds_aff, dz_aff, dy_aff = solve_kkt(U_Q, d, G, A, U_S,
                                                   rx, rs, ry, rz, neq, nz)

        # compute centering directions
        alpha = min(min(get_step(z,dz_aff), get_step(s, ds_aff)), 1.0)
        sig = (torch.dot(s + alpha*ds_aff, z + alpha*dz_aff)/(torch.dot(s,z)))**3
        dx_cor, ds_cor, dz_cor, dy_cor = solve_kkt(
            U_Q, d, G, A, U_S, torch.zeros(nz).type_as(Q),
            (-mu*sig*torch.ones(nineq).type_as(Q) + ds_aff*dz_aff)/s,
            torch.zeros(nineq).type_as(Q), torch.zeros(neq).type_as(Q), neq, nz)

        dx = dx_aff + dx_cor
        ds = ds_aff + ds_cor
        dz = dz_aff + dz_cor
        dy = dy_aff + dy_cor if neq > 0 else None
```

# Ecosystem

- Use the entire Python ecosystem at your will
- Including SciPy, Scikit-Learn, etc.




# Ecosystem

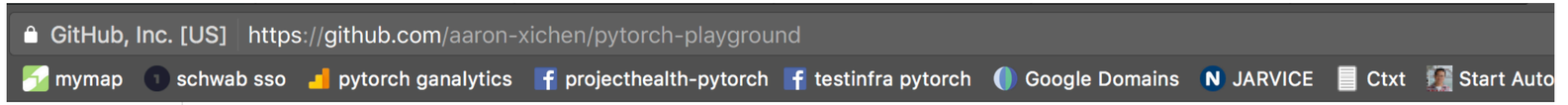
- A shared model-zoo:

We provide pre-trained models for the ResNet variants and AlexNet, using the PyTorch `torch.utils.model_zoo`. These can constructed by passing `pretrained=True`:

```
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
alexnet = models.alexnet(pretrained=True)
```

# Ecosystem

- A shared




We evaluate the performance of popular dataset and models with linear quantized method. The bit-width of running mean and running variance in BN are 10 bits for all results. (except for 32-float)

| Model | 32-float | 12-bit | 10-bit | 8-bit | 6-bit |
|---|---|---|---|---|---|
| MNIST | 98.42 | 98.43 | 98.44 | 98.44 | 98.32 |
| SVHN | 96.03 | 96.03 | 96.04 | 96.02 | 95.46 |
| CIFAR10 | 93.78 | 93.79 | 93.80 | 93.58 | 90.86 |
| CIFAR100 | 74.27 | 74.21 | 74.19 | 73.70 | 66.32 |
| STL10 | 77.59 | 77.65 | 77.70 | 77.59 | 73.40 |
| AlexNet | 55.70/78.42 | 55.66/78.41 | 55.54/78.39 | 54.17/77.29 | 18.19/36.25 |
| VGG16 | 70.44/89.43 | 70.45/89.43 | 70.44/89.33 | 69.99/89.17 | 53.33/76.32 |
| VGG19 | 71.36/89.94 | 71.35/89.93 | 71.34/89.88 | 70.88/89.62 | 56.00/78.62 |
| ResNet18 | 68.63/88.31 | 68.62/88.33 | 68.49/88.25 | 66.80/87.20 | 19.14/36.49 |
| ResNet34 | 72.50/90.86 | 72.46/90.82 | 72.45/90.85 | 71.47/90.00 | 32.25/55.71 |
| ResNet50 | 74.98/92.17 | 74.94/92.12 | 74.91/92.09 | 72.54/90.44 | 2.43/5.36 |
| ResNet101 | 76.69/93.30 | 76.66/93.25 | 76.22/92.90 | 65.69/79.54 | 1.41/1.18 |
| ResNet152 | 77.55/93.59 | 77.51/93.62 | 77.40/93.54 | 74.95/92.46 | 9.29/16.75 |
| SqueezeNetV0 | 56.73/79.39 | 56.75/79.40 | 56.70/79.27 | 53.93/77.04 | 14.21/29.74 |
| SqueezeNetV1 | 56.52/79.13 | 56.52/79.15 | 56.24/79.03 | 54.56/77.33 | 17.10/32.46 |
| InceptionV3 | 76.41/92.78 | 76.43/92.71 | 76.44/92.73 | 73.67/91.34 | 1.50/4.82 |

# Linear style of programming

- PyTorch is an imperative / eager computational toolkit

# Linear style of progra

- PyTorch is an imperative / eager con

```
1   import torch
2   from torch.autograd import Variable
3
4   trX = torch.linspace(-1, 1, 101)
5   trY = 2 * trX + torch.randn(*trX.size()) * 0.33
6
7   w = Variable(trX.new([0.0]), requires_grad=True)
8
9   for i in range(100):
10    for (x, y) in zip(trX, trY):
11      X = Variable(x)
12      Y = Variable(y)
13      print(Y)
14
15      y_model = X * w.expand_as(X)
16      cost = (Y - Y_model) ** 2
17      cost.backward(torch.ones(*cost.size()))
18
19      w.data = w.data + 0.01 * w.grad.data
20
21    print(w)
```

# Linear style of programming

- PyTorch is an imperative / eager computational toolkit
  - Not unique to PyTorch

# Linear style of programming

- PyTorch is an imperative / eager computational toolkit
  - Not unique to PyTorch
    - Chainer, Dynet, MXNet-Imperative, TensorFlow-imperative, TensorFlow-eager, etc.

# Linear style of programming

- PyTorch is an imperative / eager computational toolkit
  - Not unique to PyTorch
    - Chainer, Dynet, MXNet-gluon, TensorFlow-imperative, TensorFlow-eager, etc.
  - Least overhead, designed with this in mind
    - max of 20 to 30 microseconds overhead per node creation
      - vs several milliseconds / seconds in other options

# The Philosophy of PyTorch

# The Philosophy of PyTorch

- Stay out of the way
- Cater to the impatient
- Promote linear code-flow
- Full interop with the Python ecosystem
- Be as fast as anything else

# Recent and Upcoming features

# Distributed PyTorch

- MPI style distributed communication
- Broadcast Tensors to other nodes
- Reduce Tensors among nodes
  - for example: sum gradients among all nodes

# Higher order derivatives

- grad(grad(grad(grad(grad(grad(grad(torch.norm(x))))))))

# Higher order derivatives

- grad(grad(grad(grad(grad(grad(grad(torch.norm(x)))))))
- Useful to implement crazy ideas

# Broadcasting and Advanced Indexing

- Numpy-style broadcasting
- Numpy-style indexing that covers advanced cases

# JIT Compilation (upcoming)

- A full tracing JIT used to cache and compile subgraphs

# Tracing JIT

```
def foo(x):
    sum = x.sum().data[0]
    if sum < 5:
     return x + 1
    else:
     return x + 2

x = torch.Tensor([-1, 0, 1, 2])
x2 = torch.Tensor([1, 2, 3, 4])

y  = foo(x)  # 0, 1, 2, 3
y2 = foo(x2) # 3, 4, 5, 6
```

# Tracing JIT

```
def foo(x):
    sum = x.sum().data[0]
    if sum < 5:
     return x + 1
    else:
     return x + 2
```

```
x = torch.Tensor([-1, 0, 1, 2])
x2 = torch.Tensor([1, 2, 3, 4])

y  = foo(x)  # 0, 1, 2, 3
y2 = foo(x2) # 3, 4, 5, 6
```

# Tracing JIT

```
def foo(x):
    sum = x.sum().data[0]
    if sum < 5:
     return x + 1
    else:
     return x + 2
```

```
x = torch.Tensor([-1, 0, 1, 2])
x2 = torch.Tensor([1, 2, 3, 4])
```

```
y  = foo(x)  # 0, 1, 2, 3
y2 = foo(x2) # 3, 4, 5, 6
```

# Tracing JIT

```
def foo(x):
    sum = x.sum().data[0]
    if sum < 5:
     return x + 1
    else:
     return x + 2

x = torch.Tensor([-1, 0, 1, 2])
x2 = torch.Tensor([1, 2, 3, 4])

y  = foo(x)  # 0, 1, 2, 3
y2 = foo(x2) # 3, 4, 5, 6
```

# Tracing JIT

```
def foo(x):
    sum = x.sum().data[0]
    if sum < 5:
     return x + 1
    else:
     return x + 2

x = torch.Tensor([-1, 0, 1, 2])
x2 = torch.Tensor([1, 2, 3, 4])

y  = foo(x)  # 0, 1, 2, 3
y2 = foo(x2) # 3, 4, 5, 6

jitfoo = torch.jit.traced(foo)
```

# Tracing JIT

```
def foo(x):
    sum = x.sum().data[0]
    if sum < 5:
     return x + 1
    else:
     return x + 2

x = torch.Tensor([-1, 0, 1, 2])
x2 = torch.Tensor([1, 2, 3, 4])

y  = foo(x)  # 0, 1, 2, 3
y2 = foo(x2) # 3, 4, 5, 6

jitfoo = torch.jit.traced(foo)

y  = jitfoo(x)
```

# Tracing JIT

```
def foo(x):
    sum = x.sum().data[0]
    if sum < 5:
     return x + 1
    else:
     return x + 2

x = torch.Tensor([-1, 0, 1, 2])
x2 = torch.Tensor([1, 2, 3, 4])

y  = foo(x)  # 0, 1, 2, 3
y2 = foo(x2) # 3, 4, 5, 6

jitfoo = torch.jit.traced(foo)

y  = jitfoo(x)
```

## Trace

Sum(x) -> t1

# Tracing JIT

```
def foo(x):
    sum = x.sum().data[0]
    if sum < 5:
      return x + 1
    else:
      return x + 2

x = torch.Tensor([-1, 0, 1, 2])
x2 = torch.Tensor([1, 2, 3, 4])

y  = foo(x)  # 0, 1, 2, 3
y2 = foo(x2) # 3, 4, 5, 6

jitfoo = torch.jit.traced(foo)

y  = jitfoo(x)
```

**Trace**

Sum(x) -> t1

# Tracing JIT

```
def foo(x):
    sum = x.sum().data[0]
    if sum < 5:
     return x + 1
    else:
     return x + 2

x = torch.Tensor([-1, 0, 1, 2])
x2 = torch.Tensor([1, 2, 3, 4])

y  = foo(x)  # 0, 1, 2, 3
y2 = foo(x2) # 3, 4, 5, 6

jitfoo = torch.jit.traced(foo)

y  = jitfoo(x)
```

**Trace**

Sum(x) -> t1

Add(x, 1) -> retval

# Tracing JIT

```
def foo(x):
    sum = x.sum().data[0]
    if sum < 5:
     return x + 1
    else:
     return x + 2

x = torch.Tensor([-1, 0, 1, 2])
x2 = torch.Tensor([1, 2, 3, 4])

y  = foo(x)  # 0, 1, 2, 3
y2 = foo(x2) # 3, 4, 5, 6

jitfoo = torch.jit.traced(foo)

y  = jitfoo(x)  # 0, 1, 2, 3
```

**Trace**

Sum(x) -> t1

Add(x, 1) -> retval

# Tracing JIT

```
def foo(x):
    sum = x.sum().data[0]
    if sum < 5:
     return x + 1
    else:
     return x + 2

x = torch.Tensor([-1, 0, 1, 2])
x2 = torch.Tensor([1, 2, 3, 4])

y  = foo(x)  # 0, 1, 2, 3
y2 = foo(x2) # 3, 4, 5, 6

jitfoo = torch.jit.traced(foo)

y  = jitfoo(x)  # 0, 1, 2, 3
y2 = jitfoo(x2)
```

**Trace**

Sum(x) -> t1

Add(x, 1) -> retval

# Tracing JIT

```
def foo(x):
    sum = x.sum().data[0]
    if sum < 5:
     return x + 1
    else:
     return x + 2

x = torch.Tensor([-1, 0, 1, 2])
x2 = torch.Tensor([1, 2, 3, 4])

y  = foo(x)  # 0, 1, 2, 3
y2 = foo(x2) # 3, 4, 5, 6

jitfoo = torch.jit.traced(foo)

y  = jitfoo(x)  # 0, 1, 2, 3
y2 = jitfoo(x2)
```

**Trace**

```
Sum(x) -> t1
Add(x, 1) -> retval
```

# Tracing JIT

```
def foo(x):
    sum = x.sum().data[0]
    if sum < 5:
     return x + 1
    else:
     return x + 2

x = torch.Tensor([-1, 0, 1, 2])
x2 = torch.Tensor([1, 2, 3, 4])

y  = foo(x)  # 0, 1, 2, 3
y2 = foo(x2) # 3, 4, 5, 6

jitfoo = torch.jit.traced(foo)

y  = jitfoo(x)  # 0, 1, 2, 3
y2 = jitfoo(x2) # 2, 3, 4, 5
```

**Trace**

Sum(x) -> t1

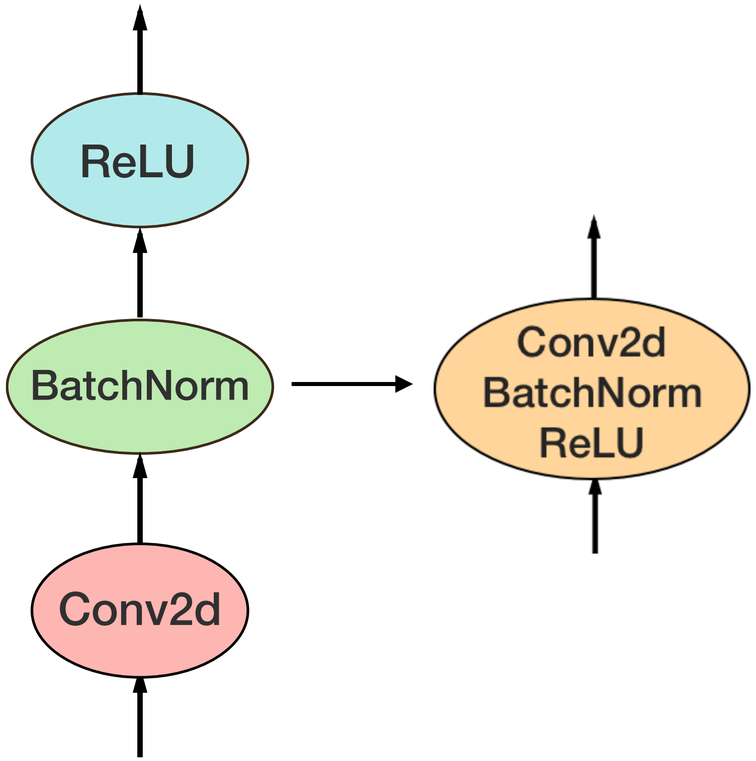
Add(x, 1) -> retval

# Tracing JIT

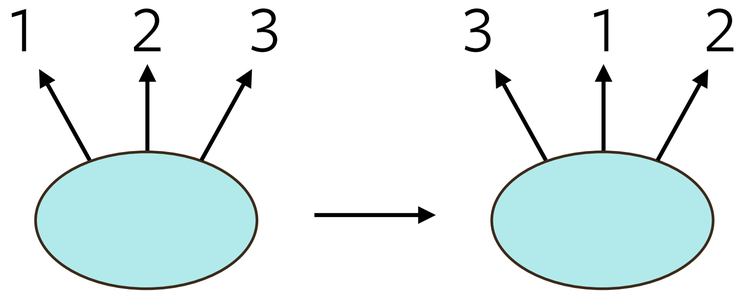
```
@torch.jit.trace
def foo(x):
    sum = x.sum().data[0]
    if sum < 5:
     return x + 1
    else:
     return x + 2

x = torch.Tensor([-1, 0, 1, 2])
x2 = torch.Tensor([1, 2, 3, 4])

y  = foo(x)  # 0, 1, 2, 3
y2 = foo(x2) # 2, 3, 4, 5
```

# Compilation benefits

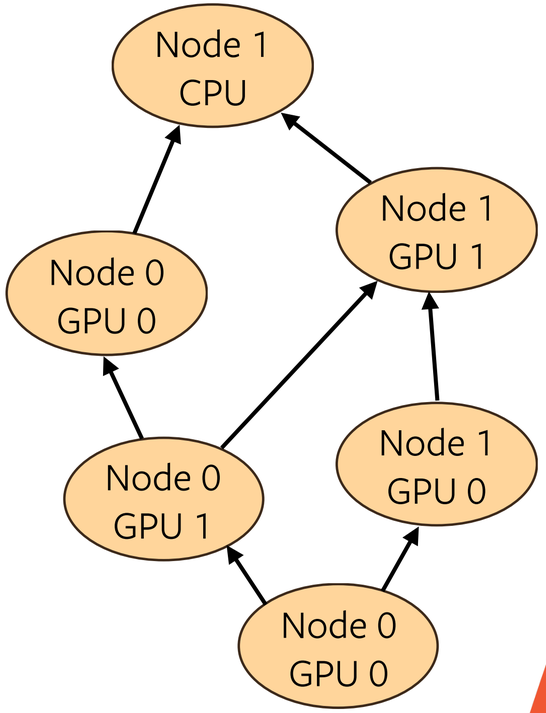
## Kernel fusion




## Out-of-order execution




## Automatic work placement



# JIT Compilation (upcoming)

- A full tracing JIT used to cache and compile subgraphs
- fuse operations on the fly and generate optimized code code

# JIT Compilation (upcoming)

- A full tracing JIT used to cache and compile subgraphs
- fuse operations on the fly and generate optimized code code
- use tracer to ship models to other frameworks such as Caffe2, TensorFlow and pure C++ runtimes

# PYTORCH

With ❤ from

http://pytorch.org

Released Jan 2017

200,000+ downloads

1500+ community repos

11,500+ user posts

290 contributors