PYTORCH

PYTORCH

decisions and challenges of writing an imperative deep learning framework

Adam Paszke, Sam Gross, Soumith Chintala & the team

Facebook AI Research

Overview of the talk

Computation Graph Toolkits → Deep Learning Frameworks → Imperative Frameworks → JIT Compilation

Computation Graph Toolkits

- Declarative
- Imperative

Implementation

Advantages & Disadvantages

Deep Learning Frameworks

Added features on top of computation graphs

Imperative Frameworks

Design

Challenges

Overheads

JIT Compilation

Lazy Execution

JIT Fusion

Subgraph caching

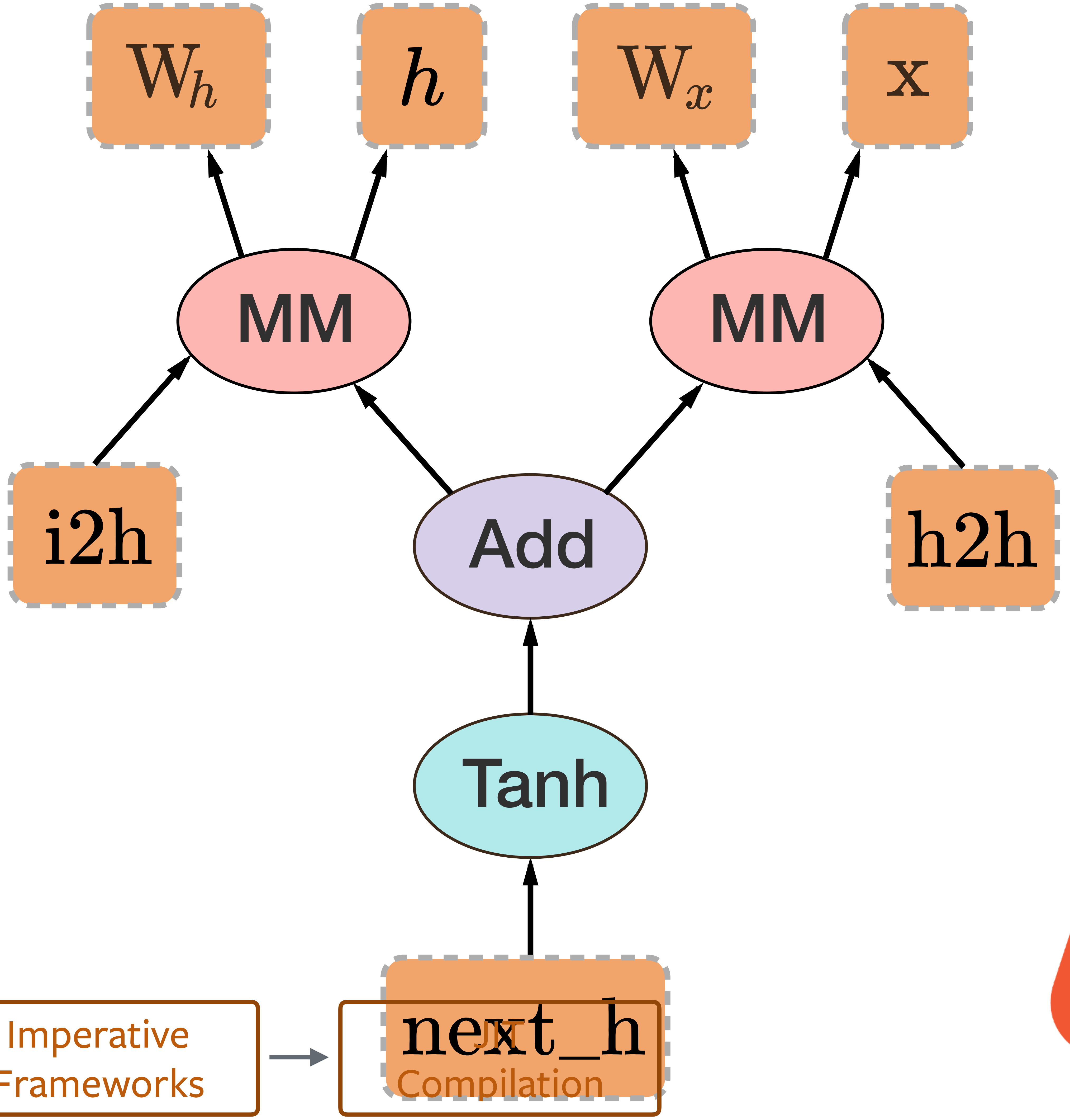
differences from AOT

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

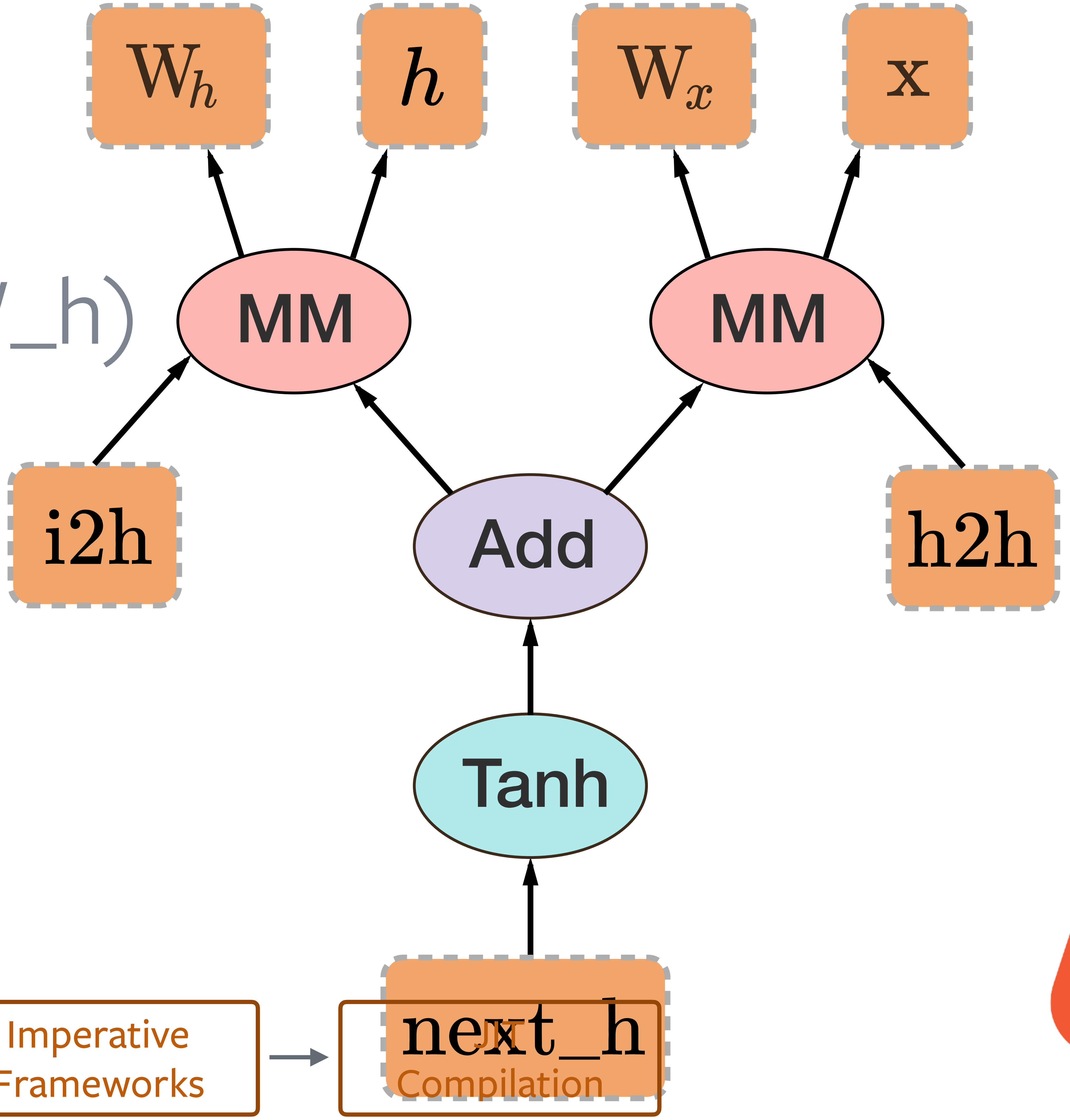


Deep Learning Frameworks

In addition to Computation Graph Toolkits

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

d(i2h)/d(W_h)

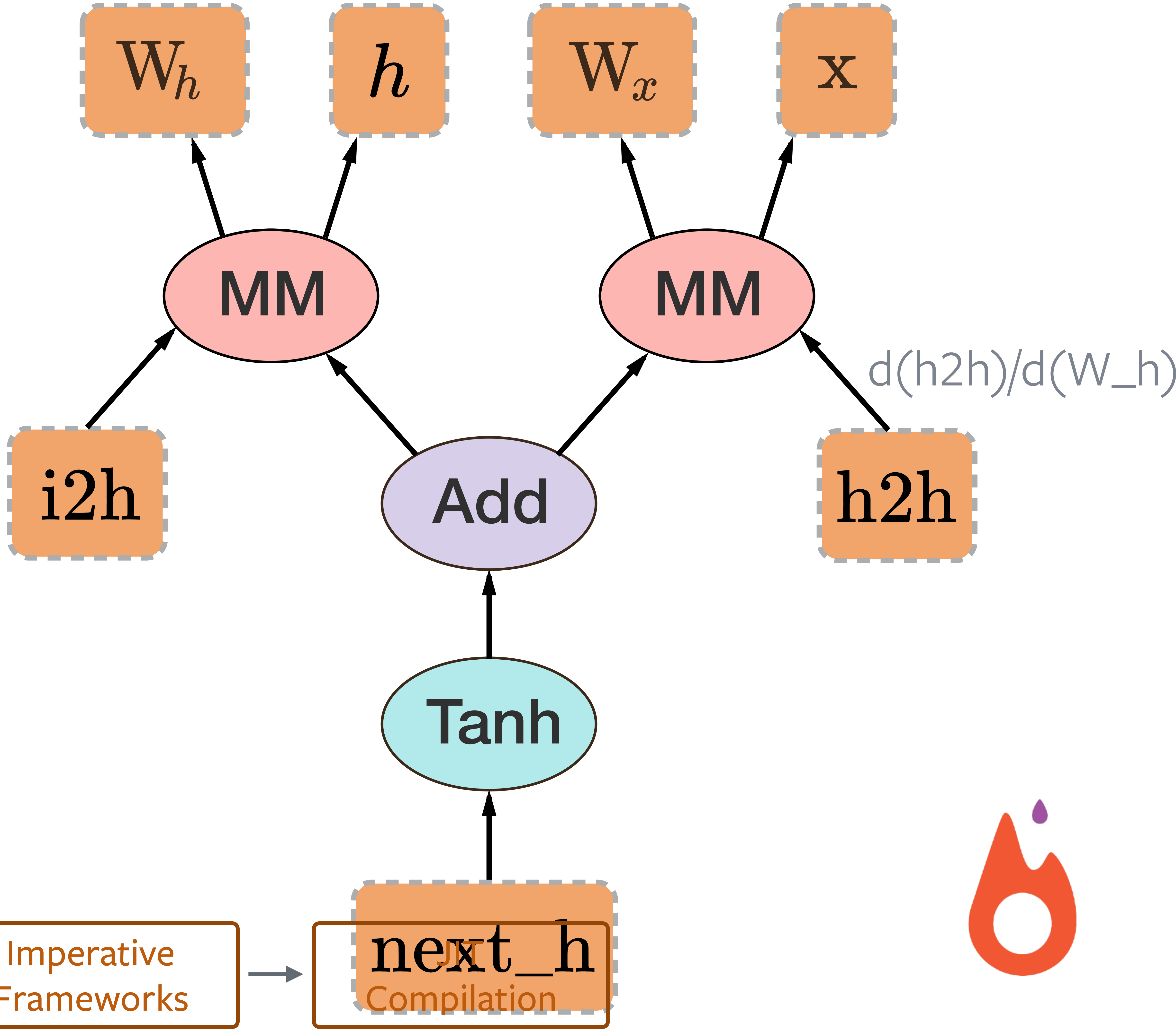


Computation Graphs → Deep Learning Frameworks → Imperative Frameworks → JIT Compilation

Deep Learning Frameworks

In addition to Computation Graph Toolkits

- Provide gradient computation
  - Gradient of one variable w.r.t. any variable in graph
- Provide integration with high performance DL libraries like CuDNN



Computation Graph Toolkits

Computation Graph Toolkits

Declarative Toolkits

theano

Caffe

Computation Graphs → Deep Learning Frameworks → Imperative Frameworks → JIT Compilation

Computation Graph Toolkits

Declarative Toolkits

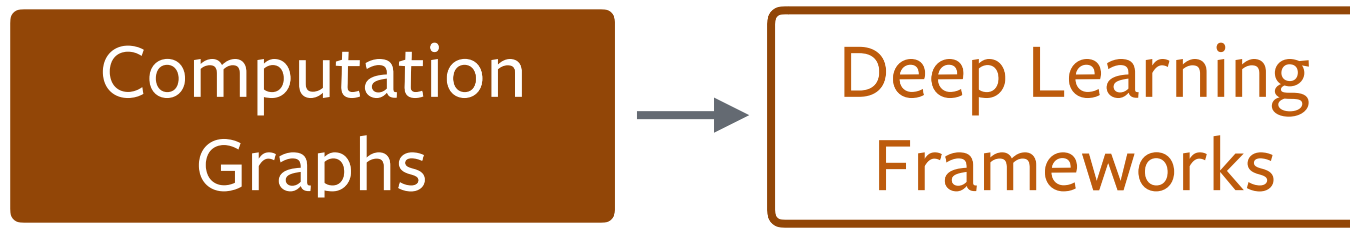
- Declare a computation
  - with placeholder variables
- Compile it
- Run it in a Session

Computation Graph

Declarative Toolkits

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

```python
import tensorflow as tf
import numpy as np

trX = np.linspace(-1, 1, 101)
trY = 2 * trX + np.random.randn(*trX.shape) * 0.33

X = tf.placeholder("float")
Y = tf.placeholder("float")

def model(X, w):
    return tf.multiply(X, w)

w = tf.Variable(0.0, name="weights")
y_model = model(X, w)

cost = tf.square(Y - y_model)

train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

with tf.Session() as sess:
    tf.global_variables_initializer().run()

    for i in range(100):
        for (x, y) in zip(trX, trY):
            sess.run(train_op, feed_dict={X: x, Y: y})

    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

```python
import tensorflow as tf
import numpy as np

trX = np.linspace(-1, 1, 101)
trY = 2 * trX + np.random.randn(*trX.shape) * 0.33

X = tf.placeholder("float")
Y = tf.placeholder("float")

def model(X, w):
    return tf.multiply(X, w)

w = tf.Variable(0.0, name="weights")
y_model = model(X, w)

cost = tf.square(Y - y_model)

train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

with tf.Session() as sess:
    tf.global_variables_initializer().run()

    for i in range(100):
        for (x, y) in zip(trX, trY):
            sess.run(train_op, feed_dict={X: x, Y: y})

    print(sess.run(w))
```

Computation Graphs → Deep Learning Frameworks

Computation Graph Toolkits

Declarative Toolkits

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

```
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))
```

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

Computation Graphs → Deep Learning Frameworks → Imperative Frameworks → JIT Compilation

Computation Graph

Declarative Toolkits

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

Has it's own execution engine

```python
import tensorflow as tf
import numpy as np

trX = np.linspace(-1, 1, 101)
trY = 2 * trX + np.random.randn(*trX.shape) * 0.33

X = tf.placeholder("float")
Y = tf.placeholder("float")

def model(X, w):
    return tf.multiply(X, w)

w = tf.Variable(0.0, name="weights")
y_model = model(X, w)

cost = tf.square(Y - y_model)

train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

with tf.Session() as sess:
    tf.global_variables_initializer().run()

    for i in range(100):
        for (x, y) in zip(trX, trY):
            sess.run(train_op, feed_dict={X: x, Y: y})

    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**

```python
import tensorflow as tf
import numpy as np

trX = np.linspace(-1, 1, 101)
trY = 2 * trX + np.random.randn(*trX.shape) * 0.33

X = tf.placeholder("float")
Y = tf.placeholder("float")

def model(X, w):
    return tf.multiply(X, w)

w = tf.Variable(0.0, name="weights")
y_model = model(X, w)

cost = tf.square(Y - y_model)

train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

with tf.Session() as sess:
    tf.global_variables_initializer().run()

    for i in range(100):
        for (x, y) in zip(trX, trY):
            sess.run(train_op, feed_dict={X: x, Y: y})

    print(sess.run(w))
```

Computation Graphs → Deep Learning Frameworks

Computation Graph

Declarative Toolkits

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

```python
import tensorflow as tf
import numpy as np

trX = np.linspace(-1, 1, 101)
trY = 2 * trX + np.random.randn(*trX.shape) * 0.33

X = tf.placeholder("float")
Y = tf.placeholder("float")

def model(X, w):
    return tf.multiply(X, w)

w = tf.Variable(0.0, name="weights")
y_model = model(X, w)

cost = tf.square(Y - y_model)

train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

with tf.Session() as sess:
    tf.global_variables_initializer().run()

    for i in range(100):
        for (x, y) in zip(trX, trY):
            sess.run(train_op, feed_dict={X: x, Y: y})

    print(sess.run(w))
```

Graph can be serialized easily

Computation Graphs → Deep Learning Frameworks

Computation Graph

Declarative Toolkits

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

```python
import tensorflow as tf
import numpy as np

trX = np.linspace(-1, 1, 101)
trY = 2 * trX + np.random.randn(*trX.shape) * 0.33

X = tf.placeholder("float")
Y = tf.placeholder("float")

def model(X, w):
    return tf.multiply(X, w)

w = tf.Variable(0.0, name="weights")
y_model = model(X, w)

cost = tf.square(Y - y_model)

train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

with tf.Session() as sess:
    tf.global_variables_initializer().run()

    for i in range(100):
        for (x, y) in zip(trX, trY):
            sess.run(train_op, feed_dict={X: x, Y: y})

    print(sess.run(w))
```

Own Virtual Machine

Computation Graphs → Deep Learning Frameworks

Computation Graph

Declarative Toolkits

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

Own Virtual Machine

- separate debugging tools

```python
import tensorflow as tf
import numpy as np

trX = np.linspace(-1, 1, 101)
trY = 2 * trX + np.random.randn(*trX.shape) * 0.33

X = tf.placeholder("float")
Y = tf.placeholder("float")

def model(X, w):
    return tf.multiply(X, w)

w = tf.Variable(0.0, name="weights")
y_model = model(X, w)

cost = tf.square(Y - y_model)

train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

with tf.Session() as sess:
    tf.global_variables_initializer().run()

    for i in range(100):
        for (x, y) in zip(trX, trY):
            sess.run(train_op, feed_dict={X: x, Y: y})

    print(sess.run(w))
```

Computation Graphs → Deep Learning Frameworks

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**

```python
import tensorflow as tf
import numpy as np

trX = np.linspace(-1, 1, 101)
trY = 2 * trX + np.random.randn(*trX.shape) * 0.33

X = tf.placeholder("float")
Y = tf.placeholder("float")

def model(X, w):
    return tf.multiply(X, w)

w = tf.Variable(0.0, name="weights")
y_model = model(X, w)

cost = tf.square(Y - y_model)

train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

with tf.Session() as sess:
    tf.global_variables_initializer().run()

    for i in range(100):
        for (x, y) in zip(trX, trY):
            sess.run(train_op, feed_dict={X: x, Y: y})

    print(sess.run(w))
```

Imperative Toolkits

Computation Graph Toolkits

Imperative Toolkits

- Run a computation
- computation is run

PYTORCH

HIPS Autograd

Dynet

Chainer

Computation Graphs → Deep Learning Frameworks → Imperative Frameworks → JIT Compilation

Computation Graph

Imperative Toolkits

- Run a computation
- computation is run!

```
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!

```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

```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

```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

Oldest debugging method of all time

```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

Oldest debugging method of all time

```
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 Graphs → Deep Learning Frameworks

Frameworks

Compilation

Computation Graph

Imperative Toolkits

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

Oldest debugging method of all time

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

```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 T

Imperative Toolkits

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

Oldest debugging method of all time

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

```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)
    print(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
- Linear program flow
  - Linear thinking for user

```
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)
    print(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 Graphs → Deep Learning Frameworks → Imperative Frameworks → JIT Compilation

Computation Graph

Imperative Toolkits

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

```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)
    print(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

Cannot compile program

- Line

Cannot optimize

  - Linear thinking for user

```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)
    print(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 Graphs → Deep Learning Frameworks → Imperative Frameworks → JIT Compilation

Computation Graph

Imperative Toolkits

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

Cannot compile program

Line

Cannot optimize

Cannot do static analysis

```
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)
    print(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 Graphs → Deep Learning Frameworks → Imperative Frameworks → JIT Compilation

Computation Graph

Imperative Toolkits

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

Cannot compile program

- Line

Cannot optimize

Cannot do static analysis (more on this later)

```
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)
    print(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)
```

Imperative Frameworks

Imperative Frameworks: PyTorch

Graph is built on the fly

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

Imperative Frameworks: PyTorch

Graph is built on the fly

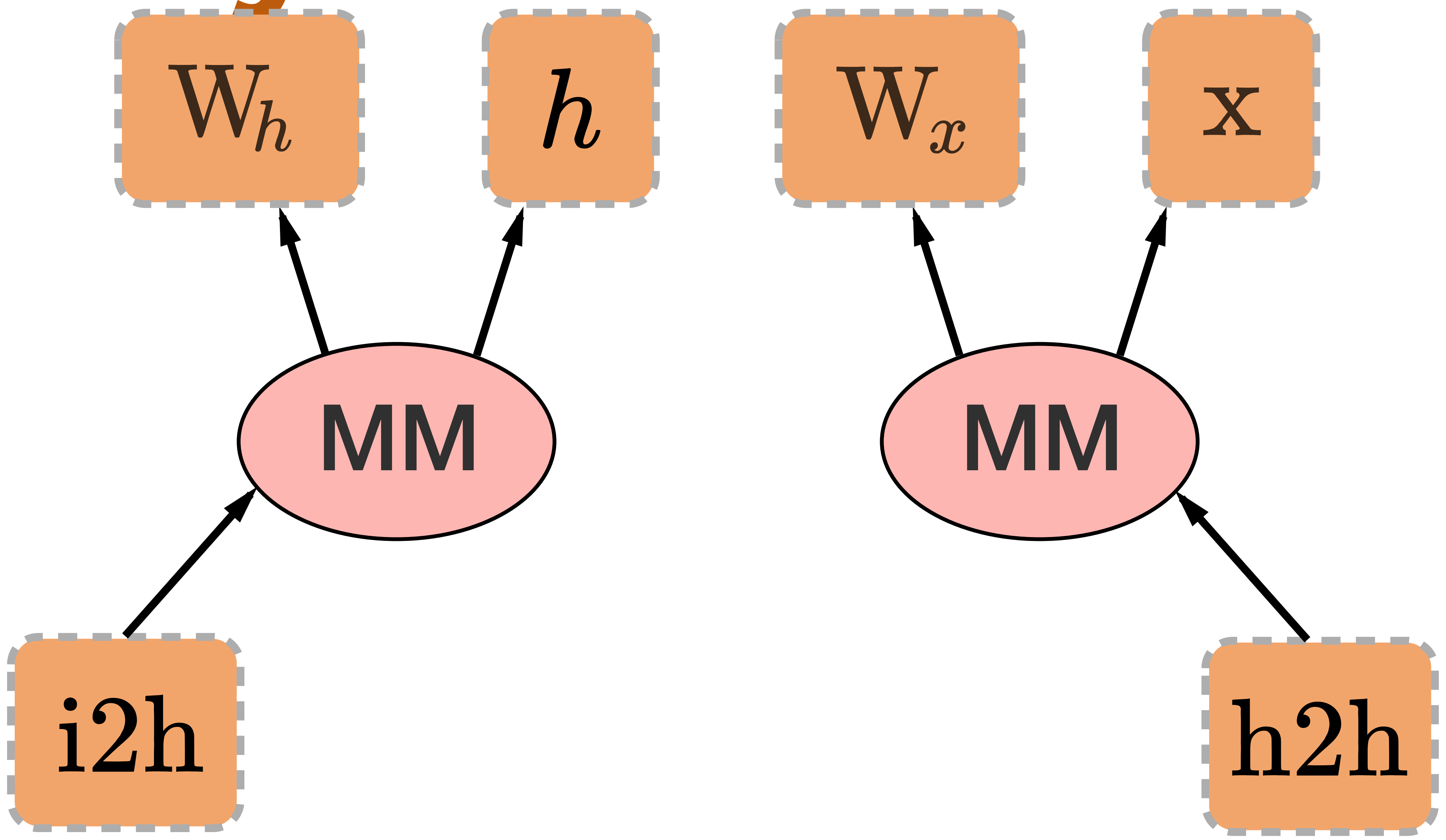
W_h h W_x x

```python
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))
```

Imperative Frameworks: PyTorch

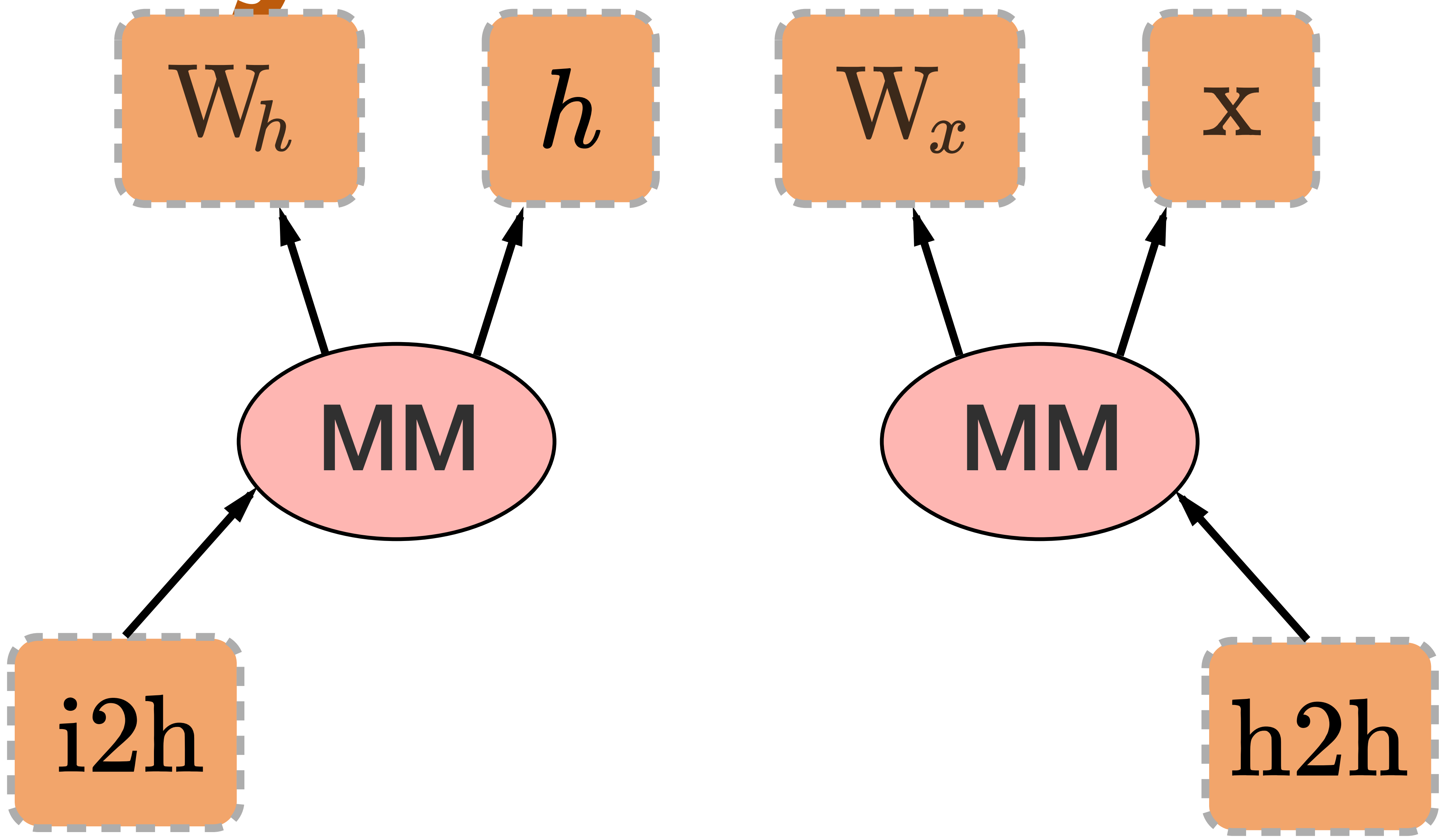
Graph is built on the fly

```
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())
```



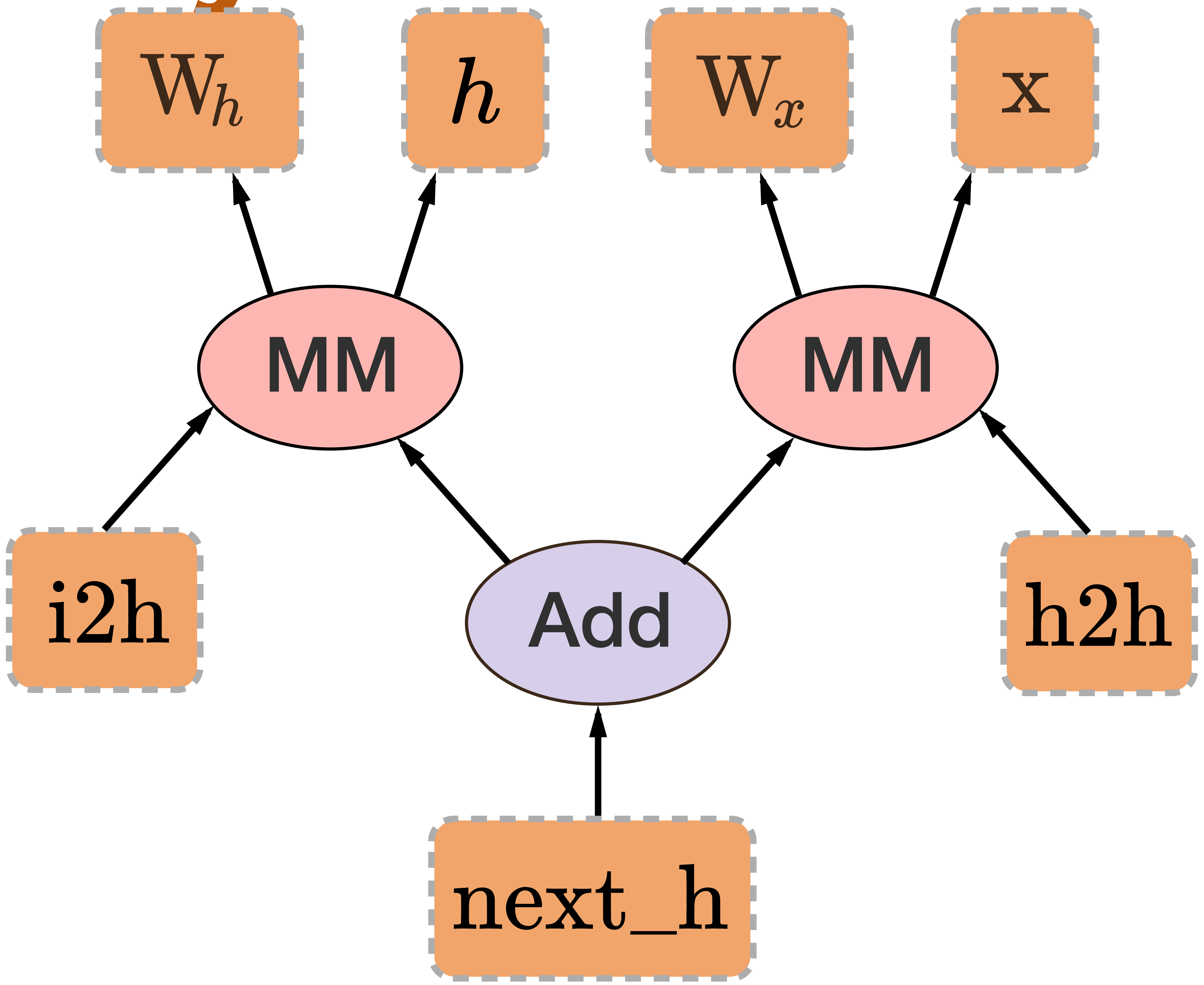
Imperative Frameworks: PyTorch

```
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
```



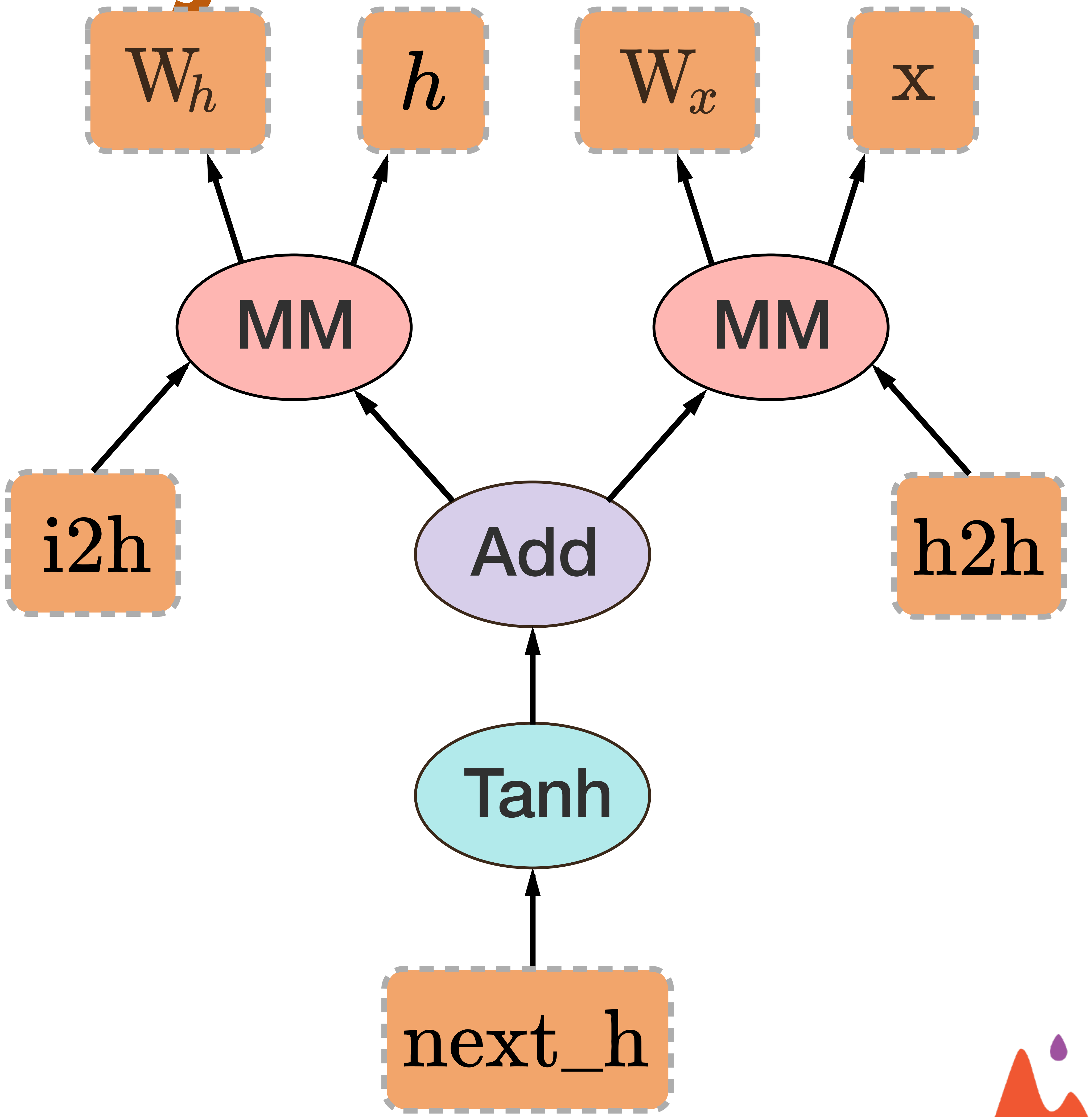
Imperative Frameworks: PyTorch

```
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
```



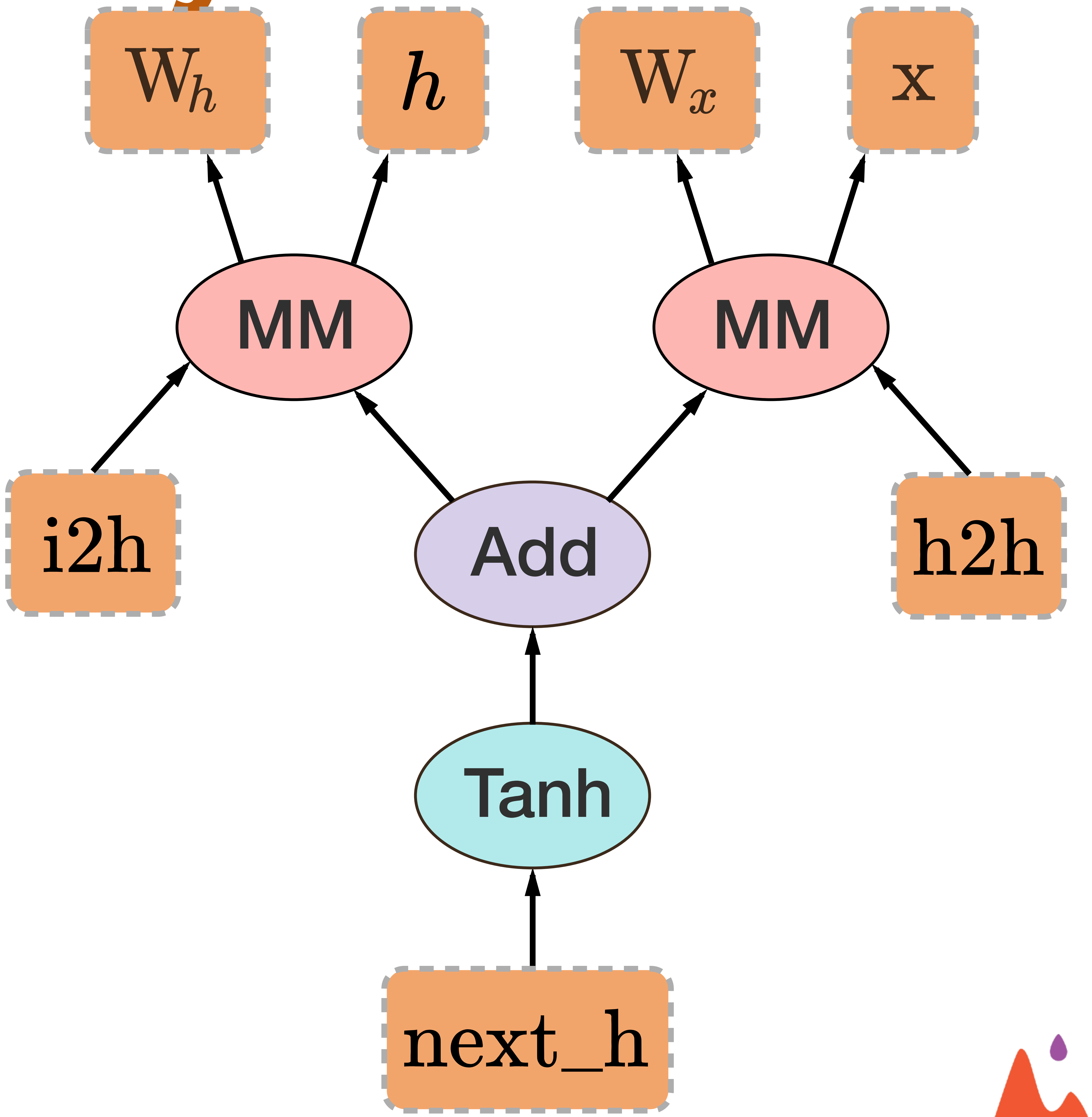
Imperative Frameworks: PyTorch

```python
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()
```



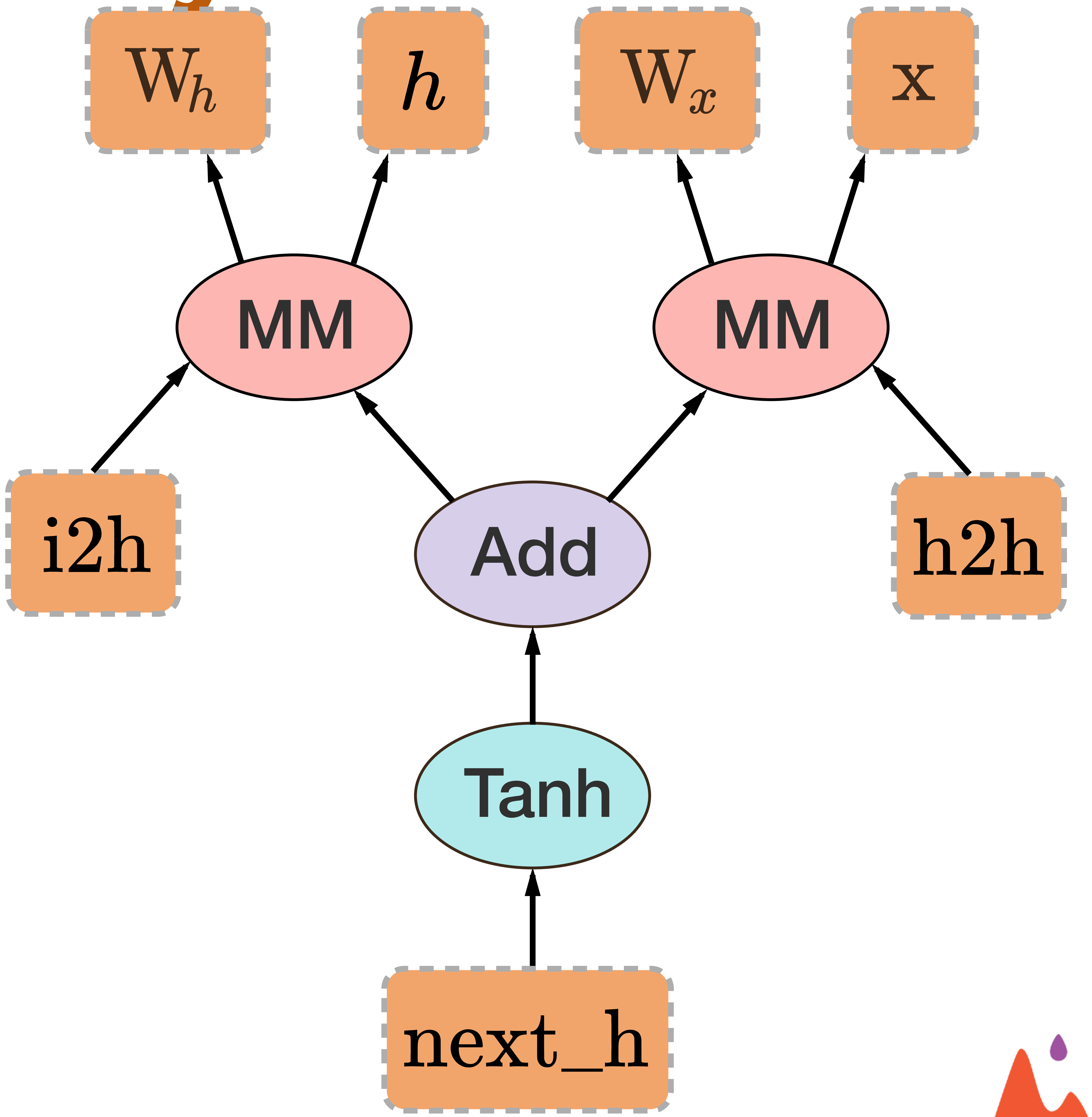
Imperative Frameworks: PyTorch

```
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))
```



Imperative Frameworks: PyTorch

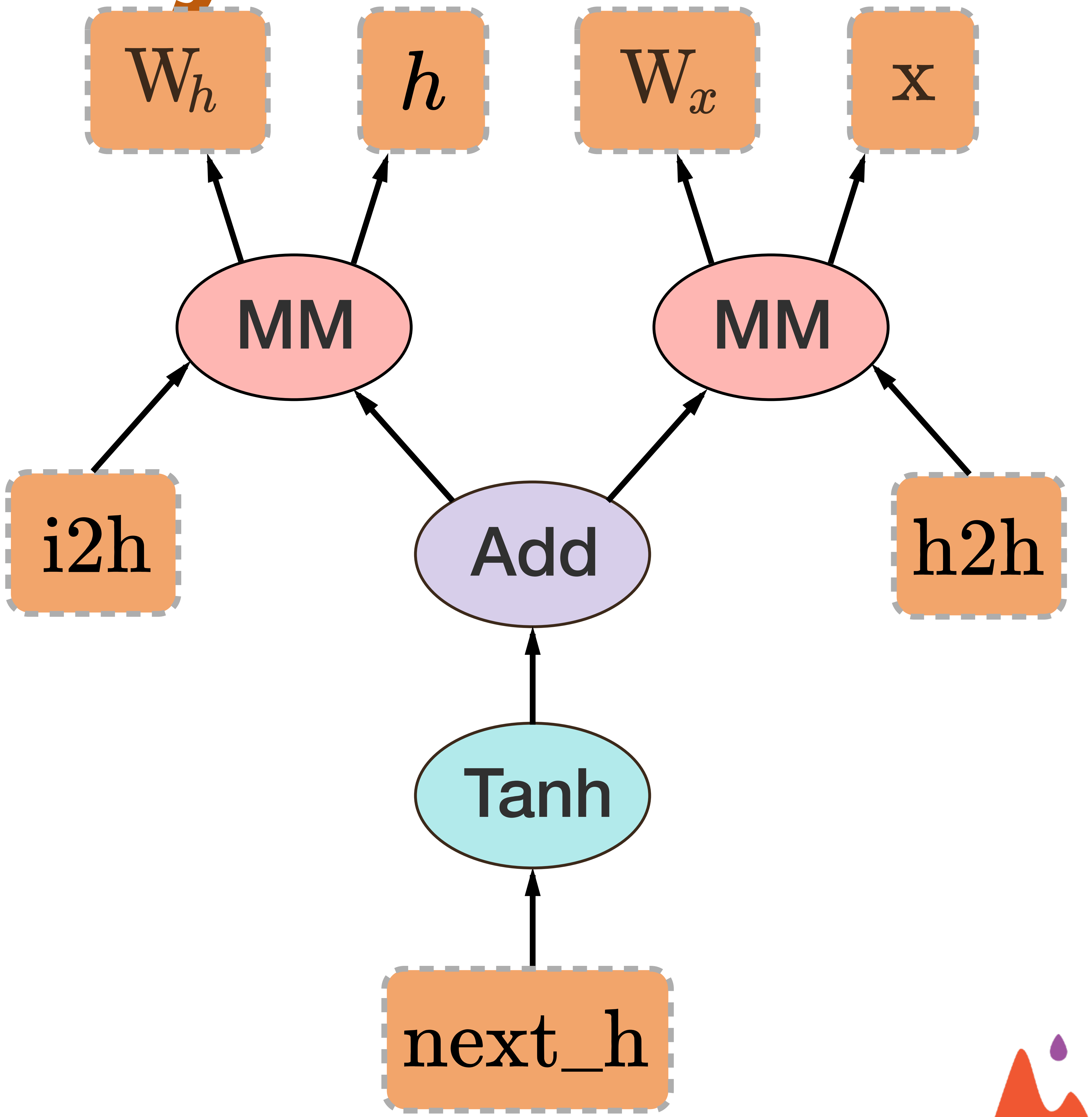
```python
from torch.autograd import Variable

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))
```



Hence, graph construction has to be FAST

Imperative Frameworks: PyTorch

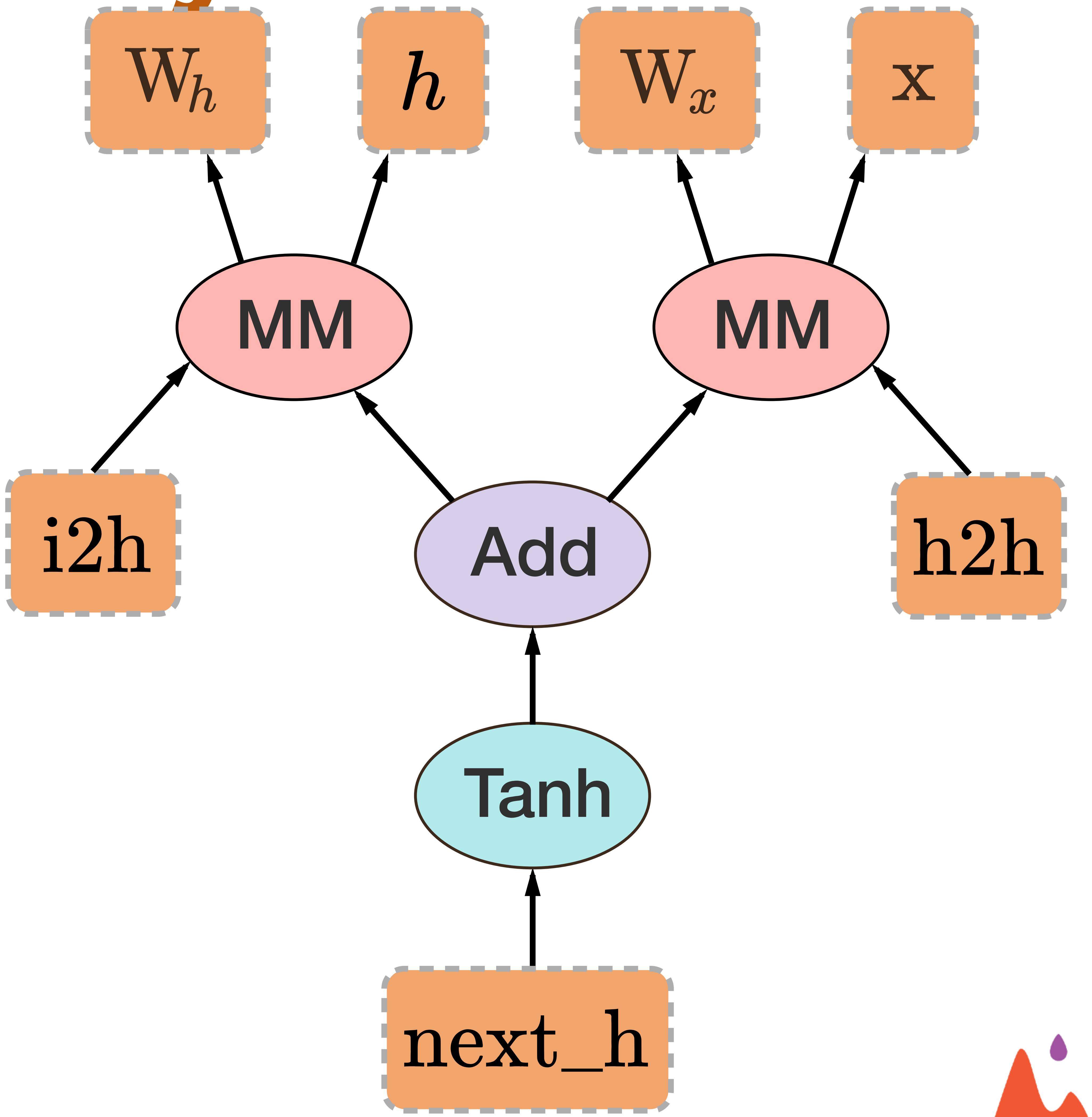
```
from torch.autograd import Variable

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))
```



Initially written in Python

Imperative Frameworks: PyTorch

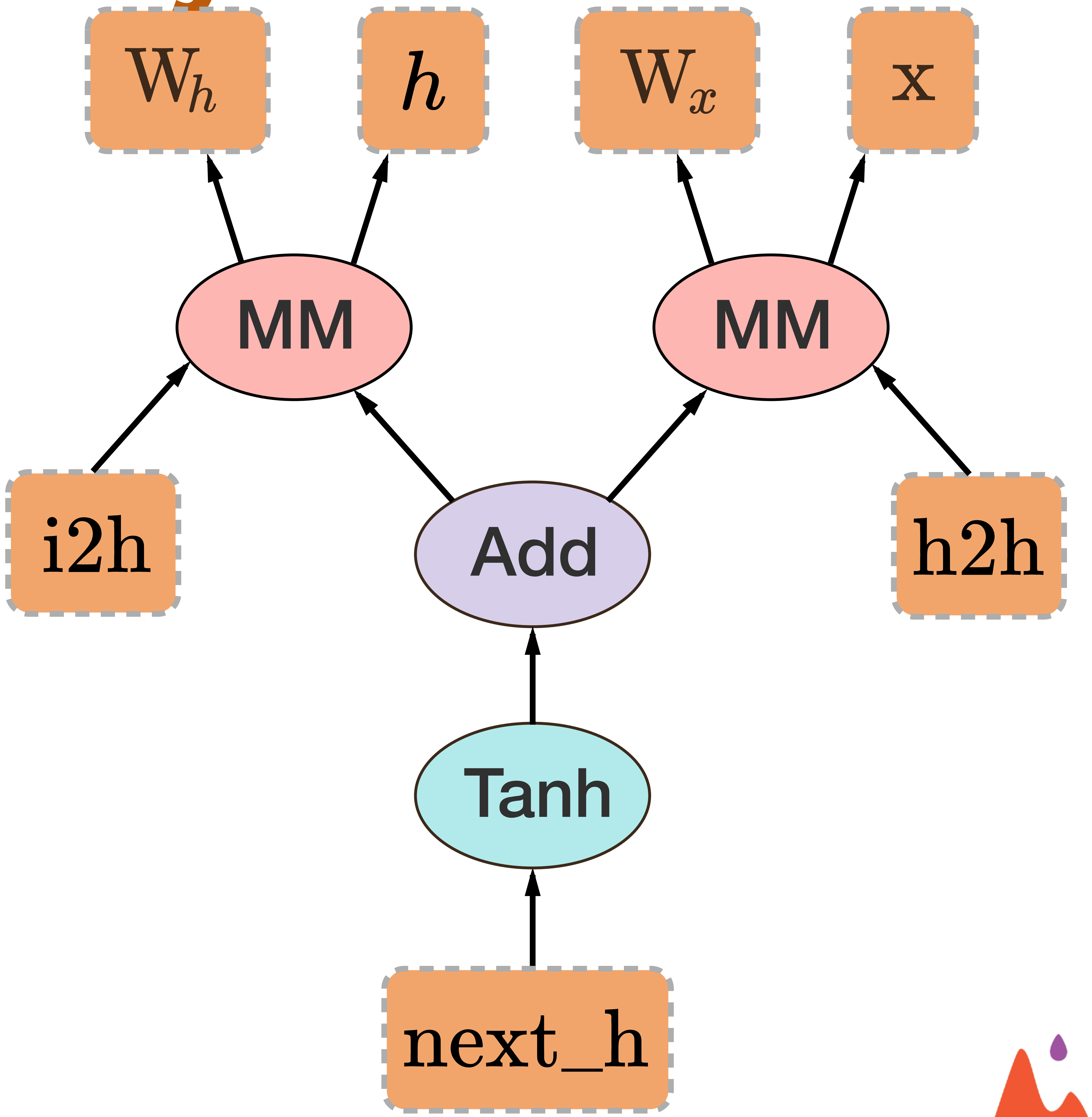
```python
from torch.autograd import Variable

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))
```



Overhead too high

Imperative Frameworks: PyTorch

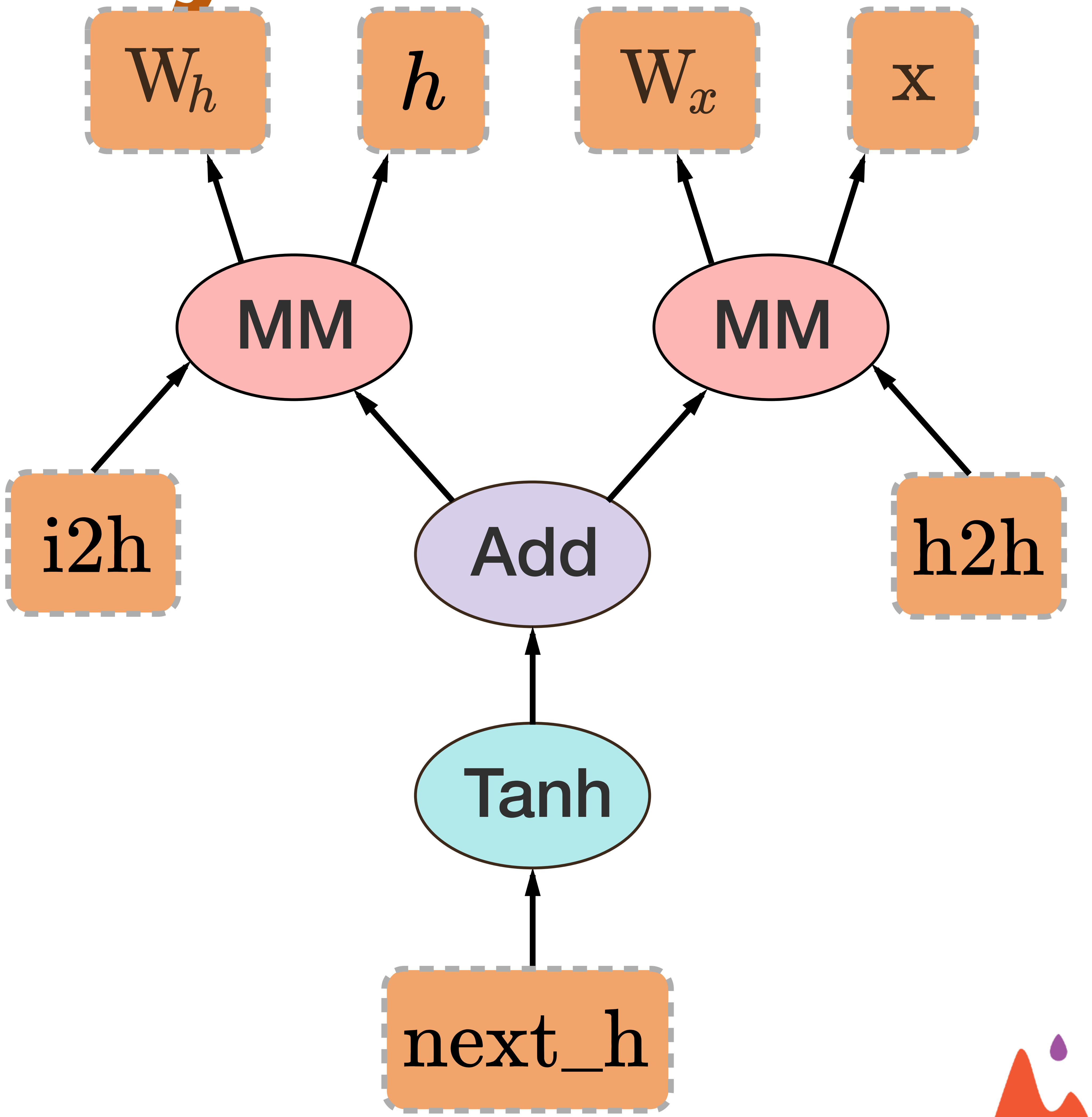
```
from torch.autograd import Variable

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))
```



Moved to CPython

Imperative Frameworks: PyTorch

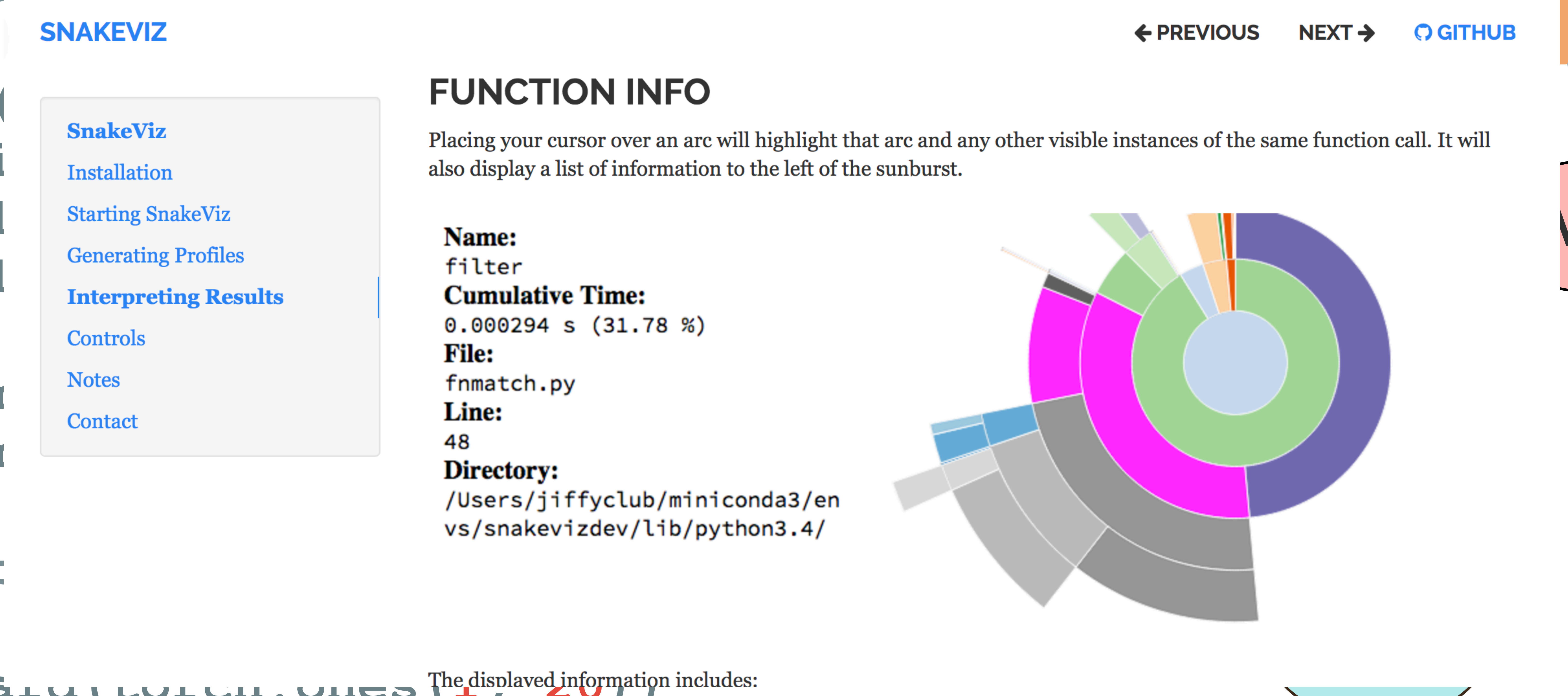
```
from torch.autograd import Variable

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))
```

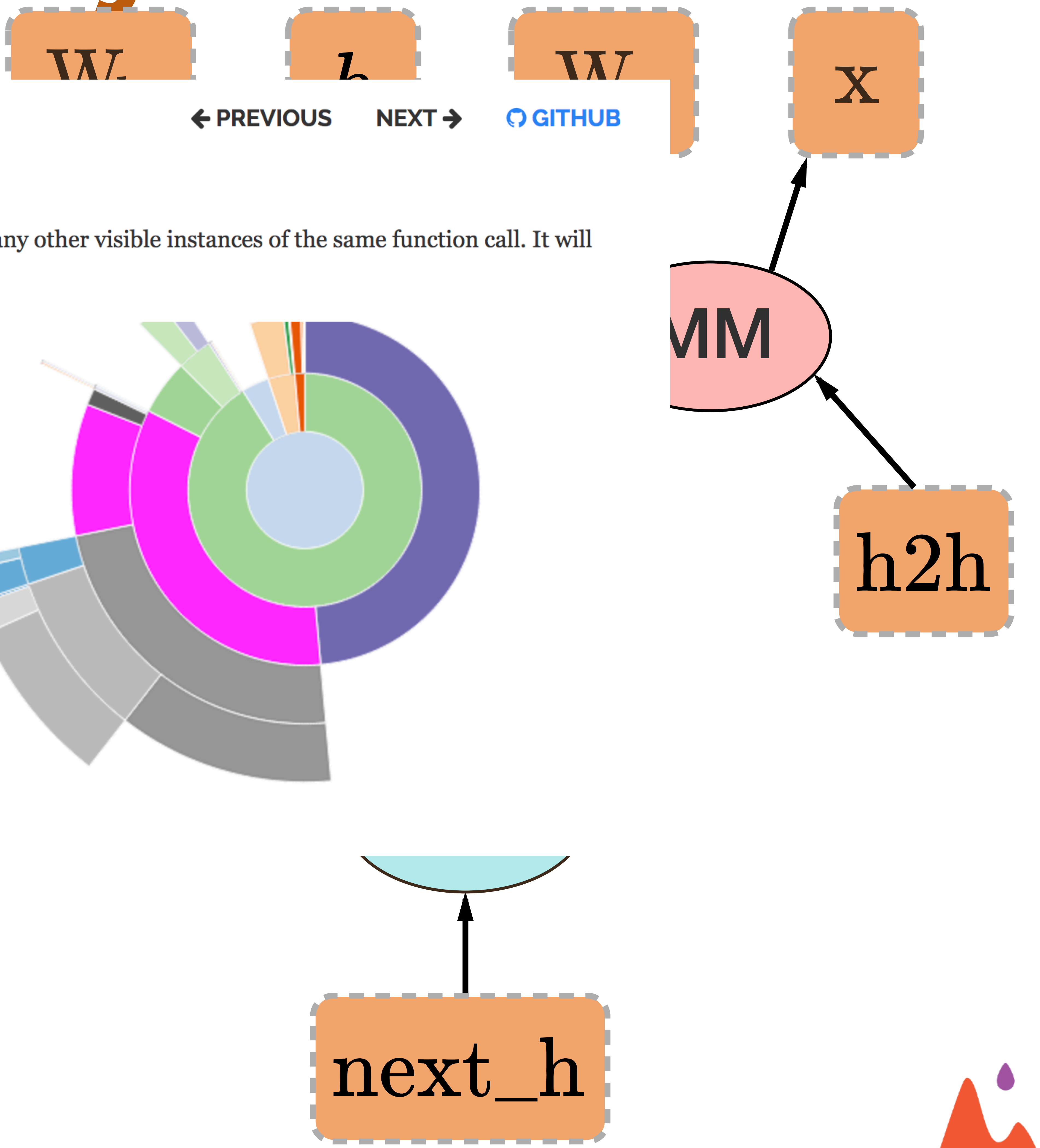


Moved to CPython used Flame graphs to find and optimize hotspots

Imperative Frameworks: PyTorch

```
from torch.autograd import Variable

x = Variable(
prev_h = Vari
W_h = Variabl
W_x = Variabl

i2h = torch.m
h2h = torch.m
next_h = i2h
next_h = next

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



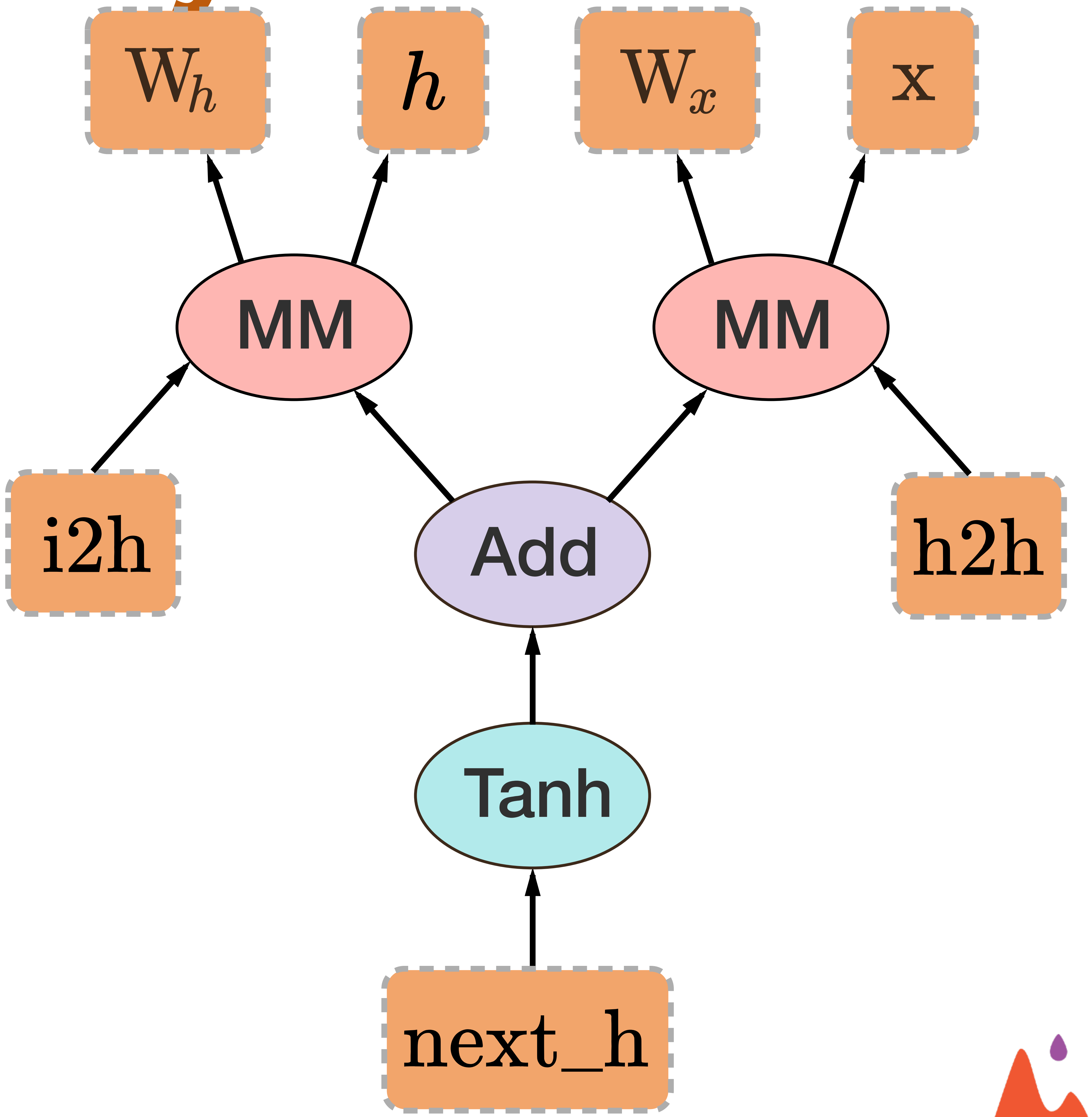


Moved to CPython used Flame graphs to find and optimize hotspots

Computation Graphs → Deep Learning Frameworks → **Imperative Frameworks** → JIT Compilation

Imperative Frameworks: PyTorch

```
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))
```



Overall speed as fast as declarative frameworks

Performance stats from Nov 2016

Task	Torch		PyTorch	
ResNet-101	544ms	10GB (5,4GB)	516ms	4,9GB
ResNet-101 2GPU	580ms	10GB (5,6GB)	640ms	4,9GB
Penn Treebank 2-layer LSTM	**57ms**	865MB	**62ms**	370MB

Performance stats from Nov 2016

Task	Torch		PyTorch	
ResNet-101	544ms	10GB (5,4GB)	516ms	4,9GB
ResNet-101 2GPU	580ms	10GB (5,6GB)	640ms	4,9GB
Penn Treebank 2-layer LSTM	**57ms**	865MB	**62ms**	370MB

Pure Python internals = **140ms**

Graph Construction Speed

- PyTorch: nanoseconds to microseconds
- TensorFlow: milliseconds to several seconds
- Theano: seconds to minutes (hours?)
- MXNet: i dont know -> ask your instructor :)

Additional valuable features in PyTorch

- Low memory usage even without a static optimizer

Computation Graphs → Deep Learning Frameworks → **Imperative Frameworks** → JIT Compilation

Performance stats from Nov 2016

Task	Torch		PyTorch	
ResNet-101	544ms	**10GB** (5,4GB)	516ms	**4,9GB**
ResNet-101 2GPU	580ms	**10GB** (5,6GB)	640ms	**4,9GB**
Penn Treebank 2-layer LSTM	57ms	865MB	62ms	370MB

Additional valuable features in PyTorch

- Low memory usage even without a static optimizer
- Intermediate buffers are always freed
- Developers given constructs to allocate temporary buffers
  - save_for_backward
  - requires_grad

Additional valuable features in PyTorch

- Low memory usage even without a static optimizer
- Intermediate buffers are always freed
- Developers given constructs to allocate temporary buffers
  - save_for_backward
  - requires_grad
- in-place operations

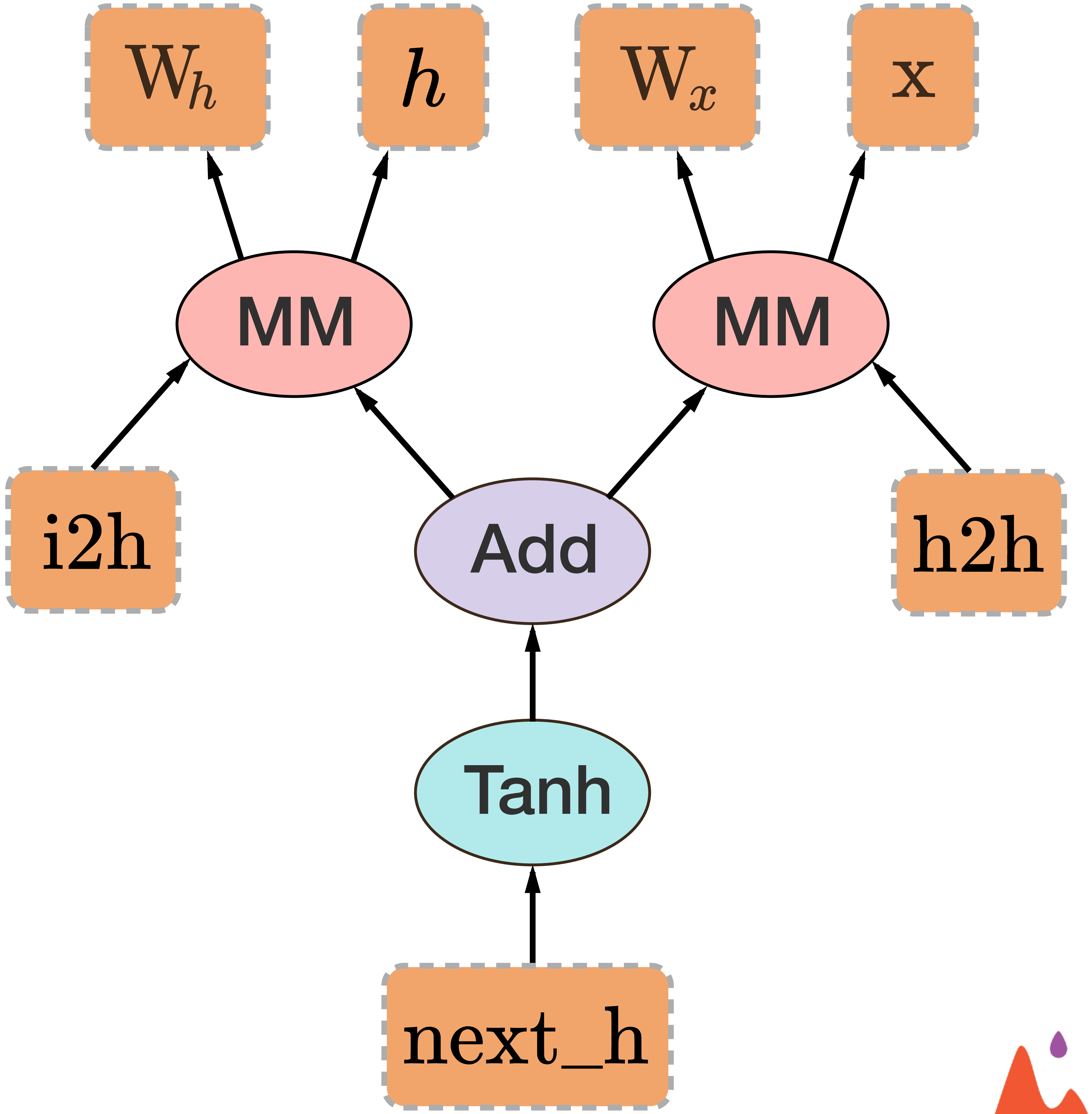
JIT Compilation

JIT Compilation

- Possible in Imperative Frameworks
- The key idea is deferred or lazy evaluation
  - y = x + 2
  - z = y * y
  - # nothing is executed yet, but the graph is being constructed
  - print(z) # now the entire graph is executed: z = (x+2) * (x+2)
- We can do just in time compilation on the graph before execution

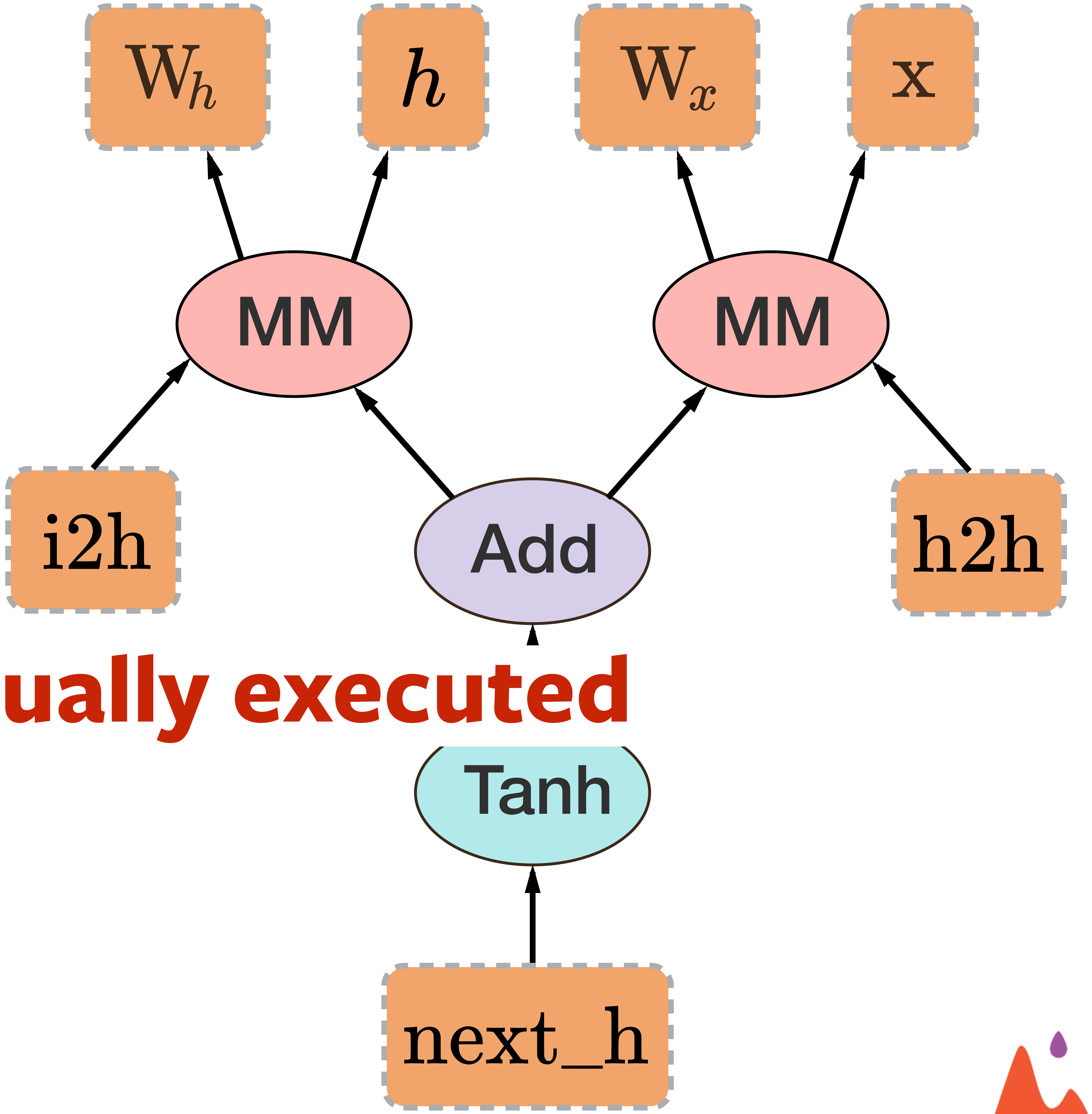
Lazy Evaluation

```
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))
```



Lazy Evaluation

```
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 = nex
```

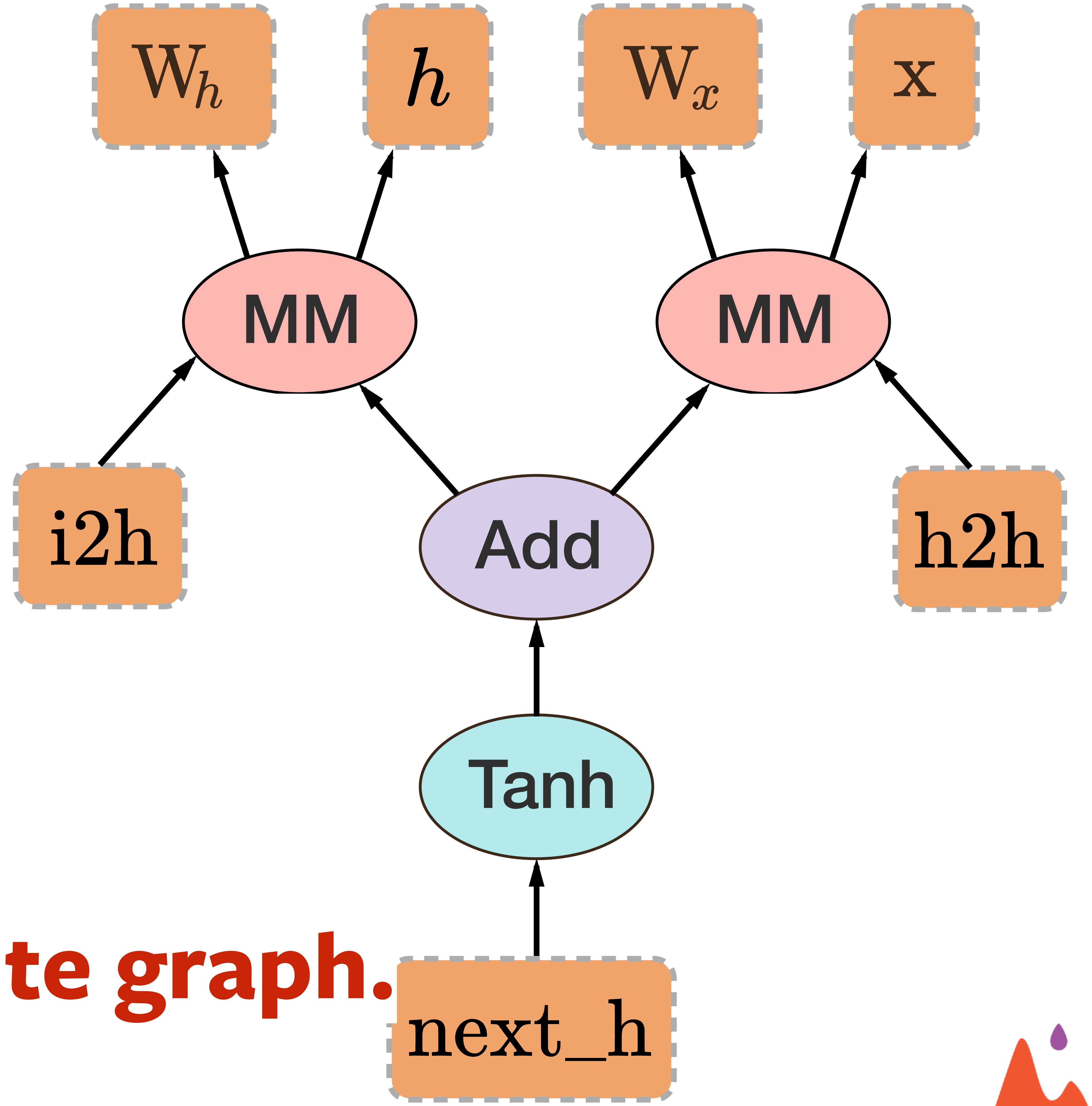


Graph built but not actually executed

Lazy Evaluation

```
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()

print(next_h)
```

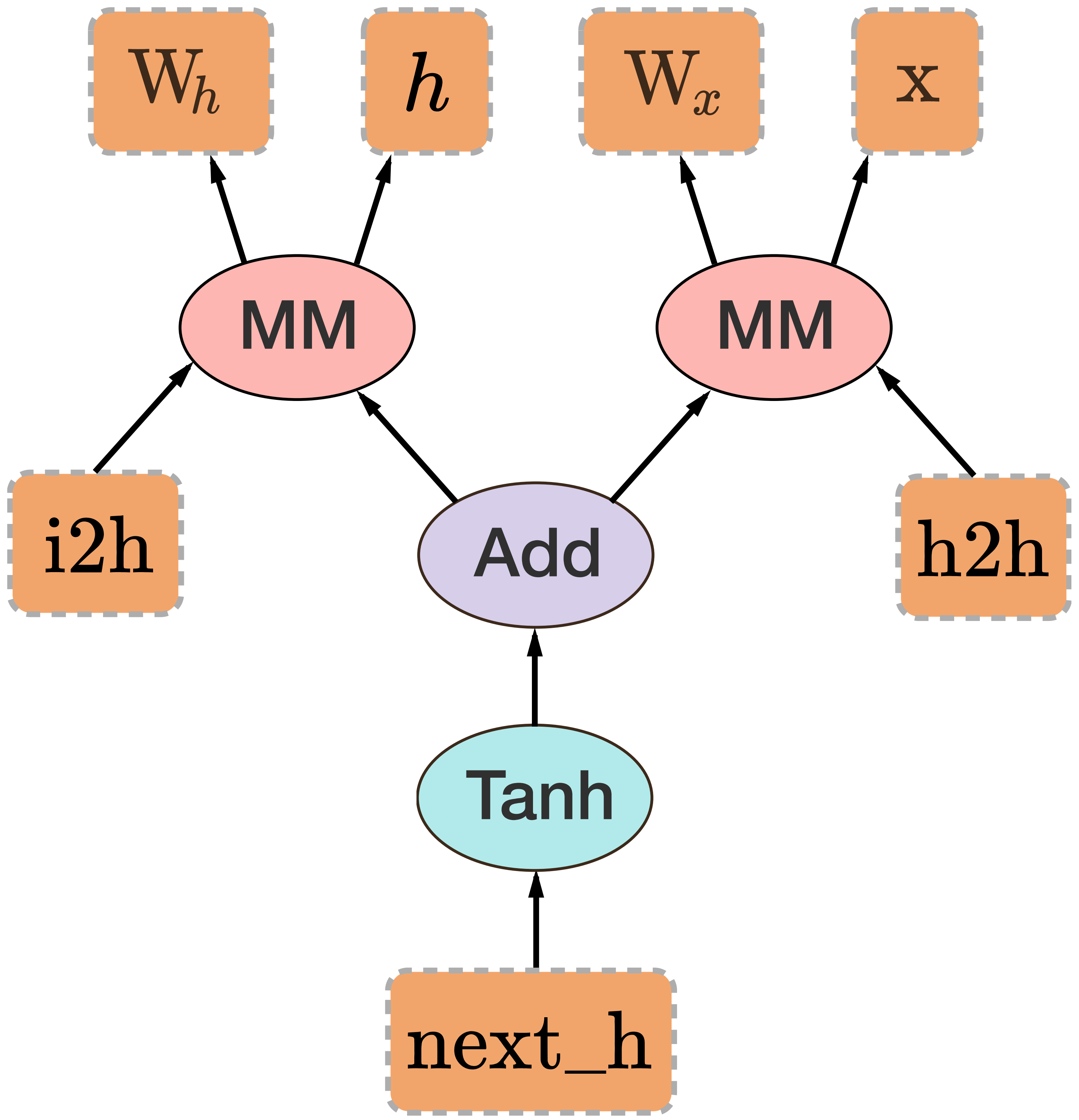
Data accessed. Execute graph.



Computation Graphs → Deep Learning Frameworks → Imperative Frameworks → **JIT Compilation**

Lazy Evaluation

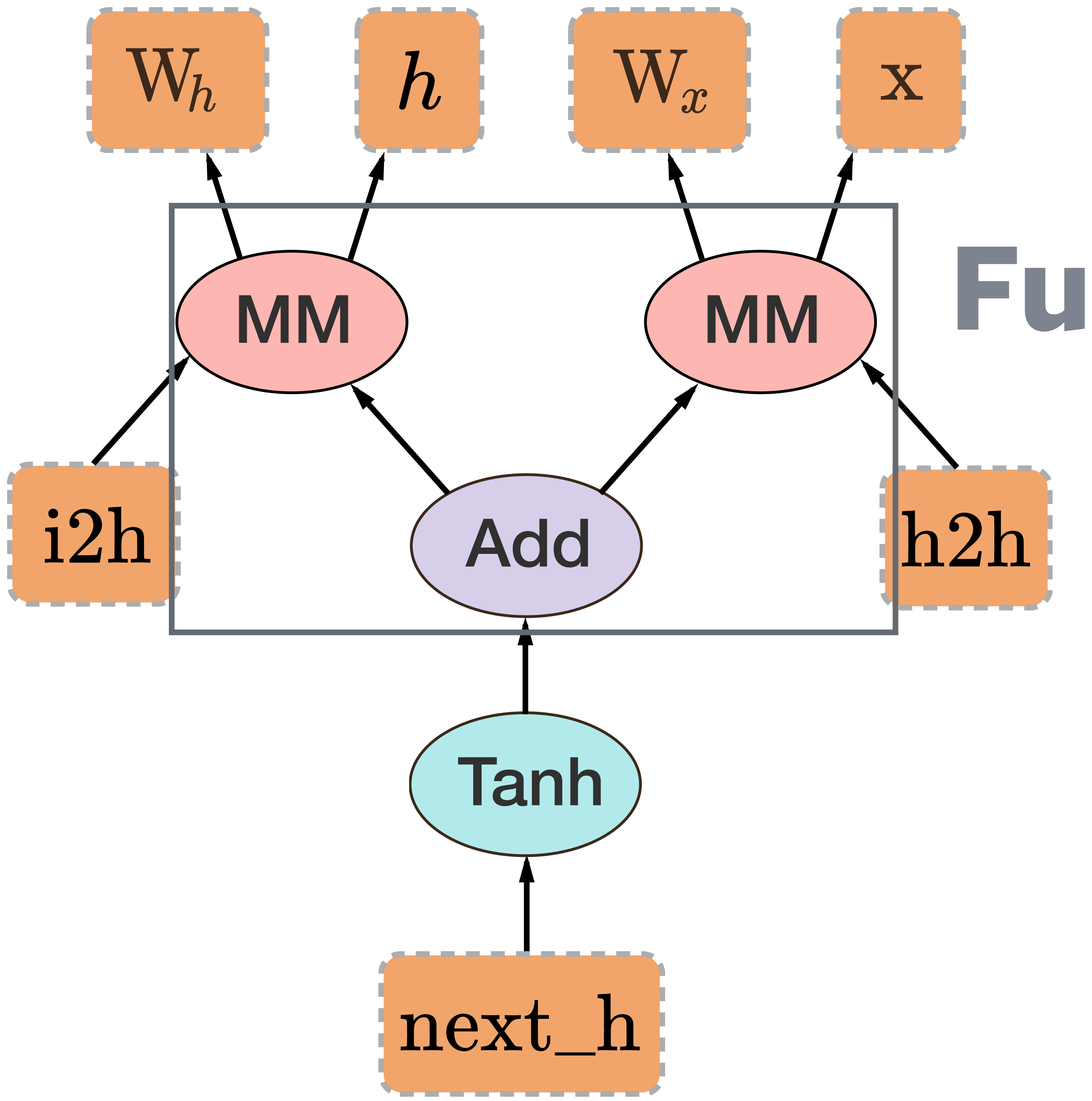
- A little bit of time between building and executing graph
  - Use it to compile the graph just-in-time



Computation Graphs → Deep Learning Frameworks → Imperative Frameworks → JIT Compilation

JIT Compilation

- Fuse and optimize operations

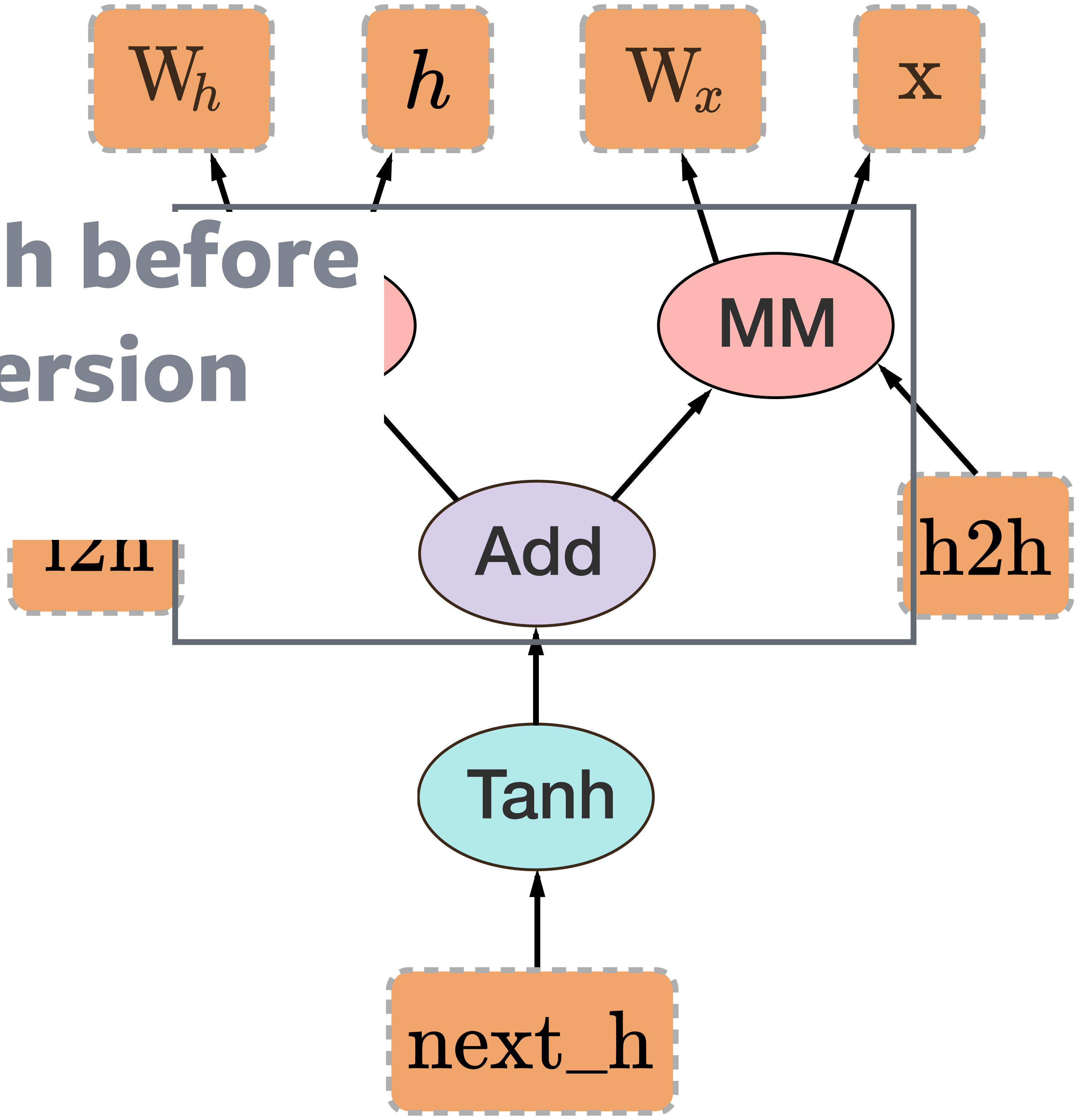


Fuse operations. Ex:

```
x = [0, 1, 2, 3, 4]
for i in range(len(x)):
  x[i] = x[i] + 1

for i in range(len(x)):
  x[i] = x[i] * 2


# Fused
for i in range(len(x)):
  x[i] = (x[i] + 1) * 2
```

JIT Compilation

- Cache subgraphs

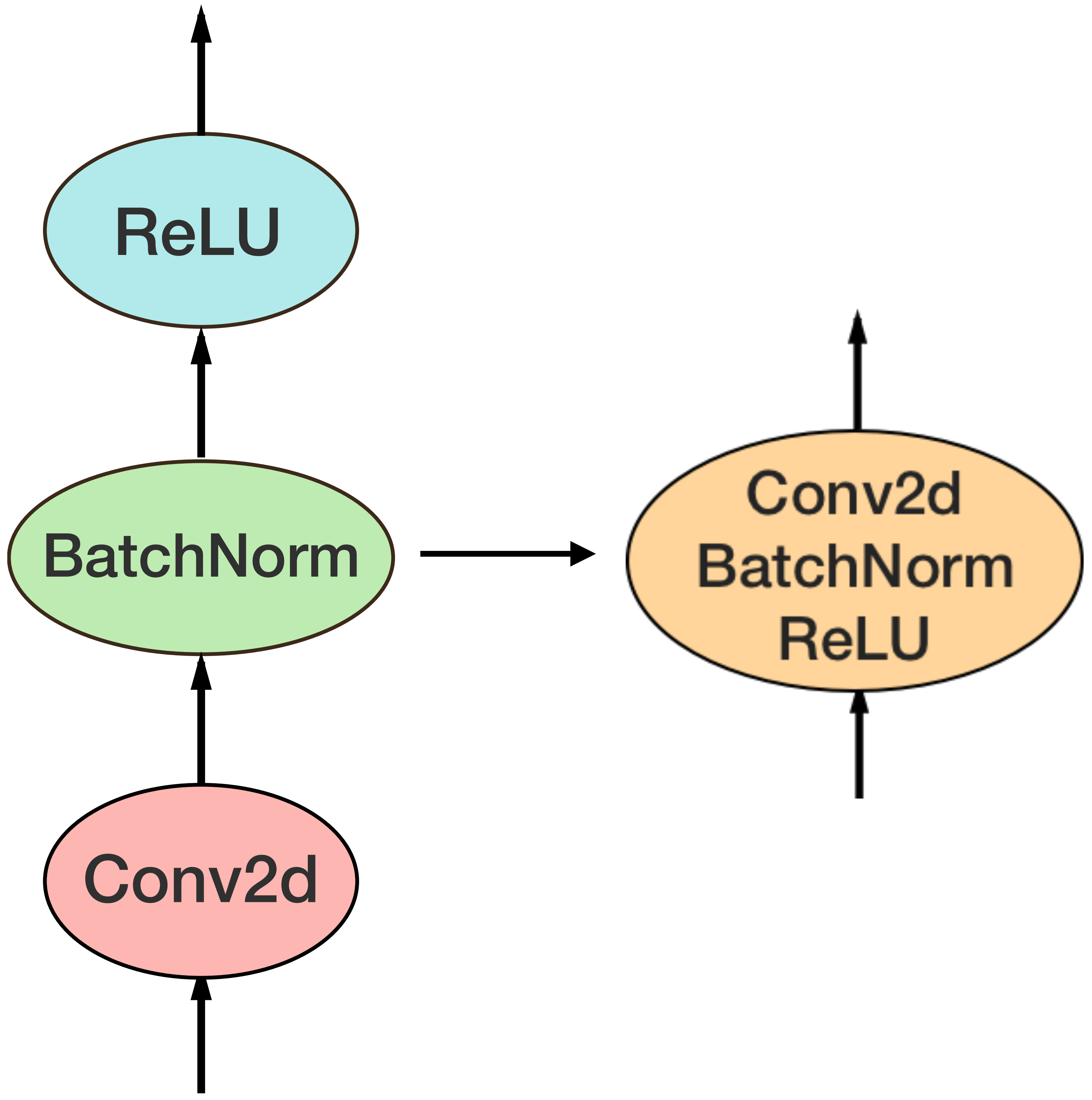


I've seen this part of the graph before let me pull up the compiled version from cache

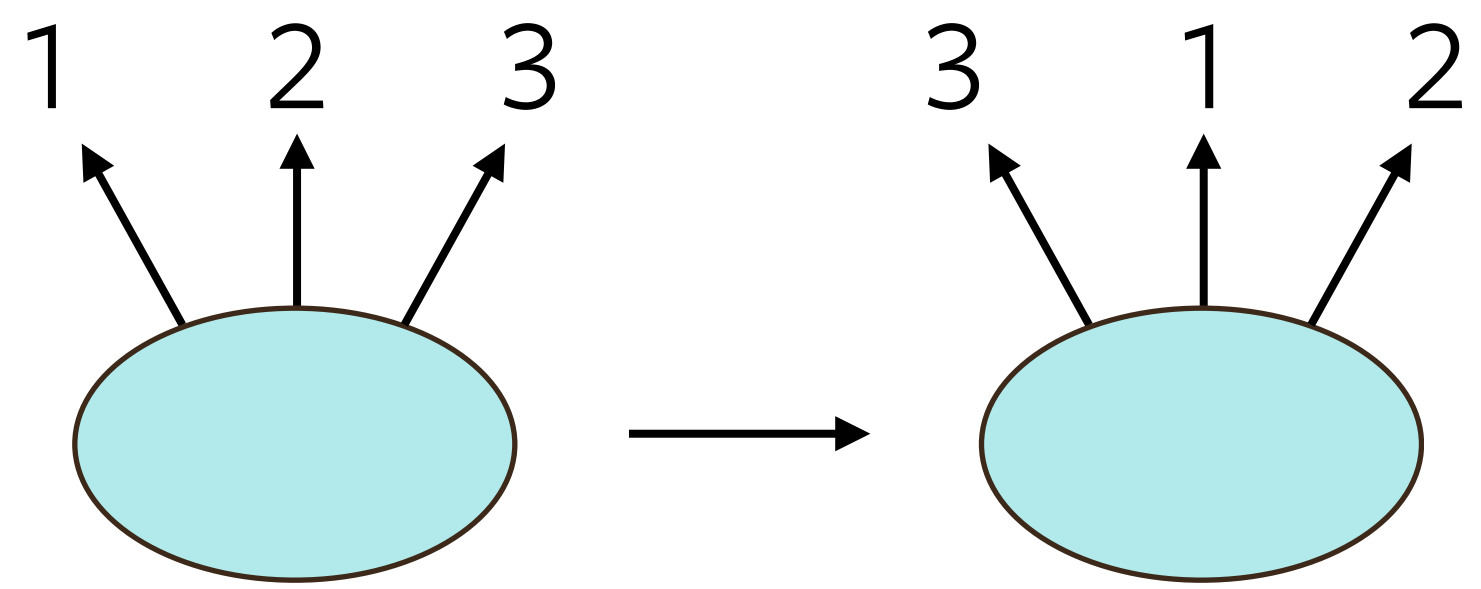
Computation Graphs → Deep Learning Frameworks → Imperative Frameworks → **JIT Compilation**

Compilation benefits

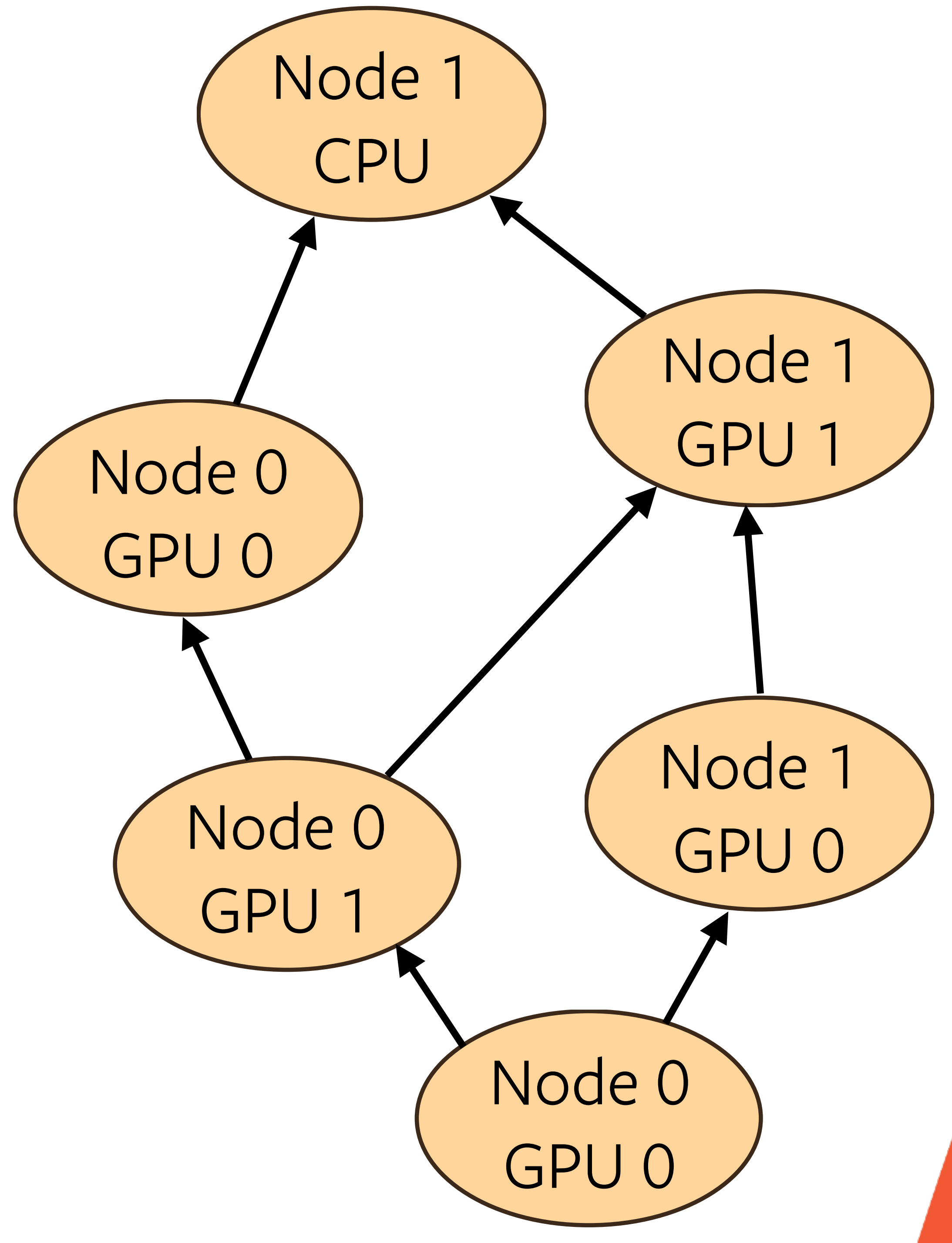
Kernel fusion



Out-of-order execution



Automatic work placement



JIT Compilation

- Possible in Dynamic Frameworks
- The key idea is deferred or lazy evaluation
  - y = x + 2
  - z = y * y
  - # nothing is executed yet, but the graph is being constructed
  - print(z) # now the entire graph is executed: z = (x+2) * (x+2)
- We can do just in time compilation on the graph before execution
- We can cache repeating patterns in subsets of the graph
  - to avoid recompilation
- Compiler is very different from Ahead-of-time compiler
  - fast compilation
  - compile traces rather than full graph

Computation Graphs → Deep Learning Frameworks → Imperative Frameworks → JIT Compilation

Review

Computation Graph Toolkits → Deep Learning Frameworks → Imperative Frameworks → JIT Compilation

- Computation Graph Toolkits
  - Declarative
  - Imperative
  - Implementation
  - Advantages & Disadvantages
- Deep Learning Frameworks
  - Added features on top of computation graphs
- Imperative Frameworks
  - Design Challenges
  - Overheads
- JIT Compilation
  - Lazy Execution
  - JIT Fusion
  - Subgraph caching
  - differences from AOT

PYTORCH

http://pytorch.org
Released Jan 18th
40,000+ downloads
250+ community repos
4200+ user posts
330k+ forum views

With ❤ from

facebook
NVIDIA
salesforce
ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIE
PARIS INSTITUTE OF TECHNOLOGY
Carnegie Mellon University
UNIVERSITE PIERRE & MARIE CURIE
Digital Reasoning
Stanford University
UNIVERSITY OF OXFORD
NYU
Inria
ENS
ÉCOLE NORMALE SUPÉRIEURE

You?