Neural Network Layers in Pytorch

2018.12.25

For basic network model implementation

Why Pytorch

- Combined Caffe and Torch
- Dynamic computational graph
- Looks exactly like numpy, and write in natural Python
- Support ONNX
- **More important, recently Facebook opensource the NLP toolkit base on Pytorch——Pytext**

A zoo of frameworks!

Caffe
(UC Berkeley)

Caffe2
(Facebook)

Torch
(NYU / Facebook)

PyTorch
(Facebook)

Theano
(U Montreal)

TensorFlow
(Google)

PaddlePaddle
(Baidu)

Chainer

MXNet
(Amazon)
Developed by U Washington, CMU, MIT, Hong Kong U, etc but main framework of choice at AWS

CNTK
(Microsoft)

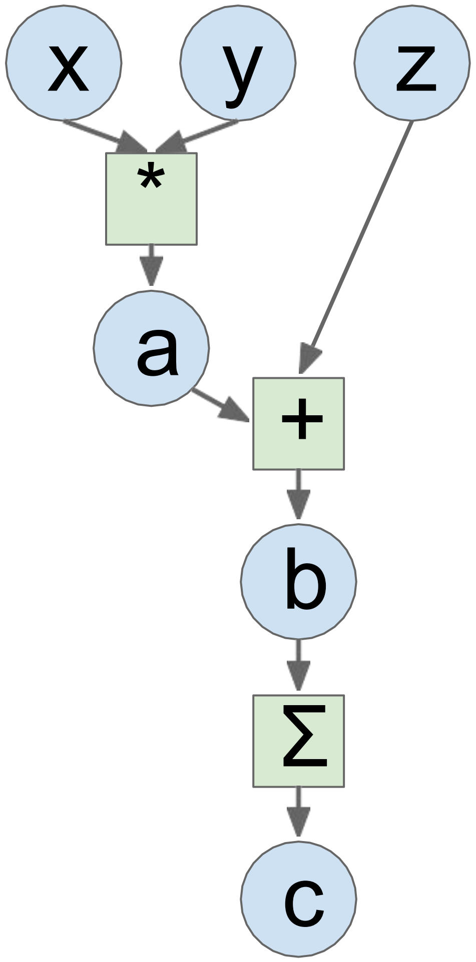
Deeplearning4j

And others...

Computational Graphs

Numpy

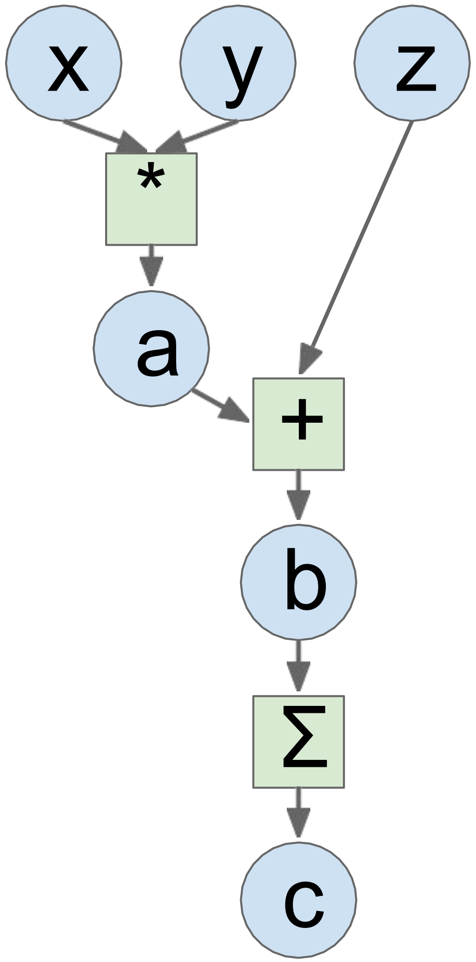
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)
```



Computational Graphs

Numpy

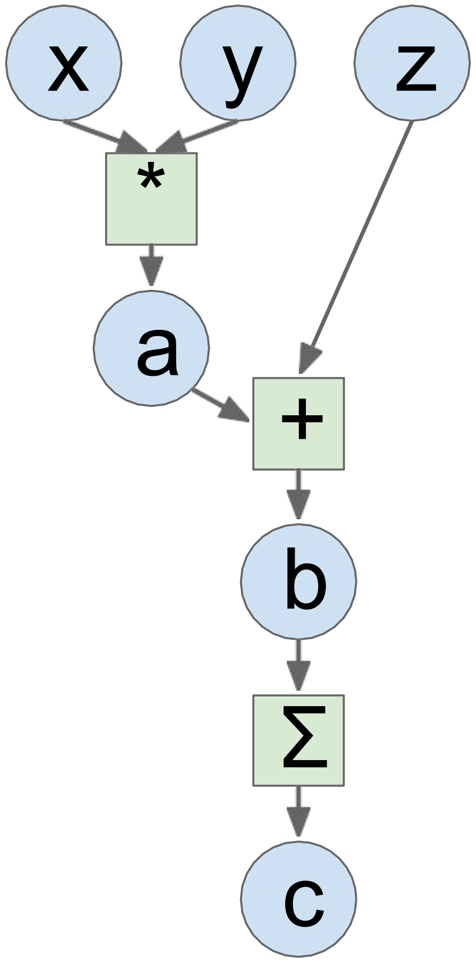
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



Computational Graphs

Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



Good:
- Clean API, easy to write numeric code

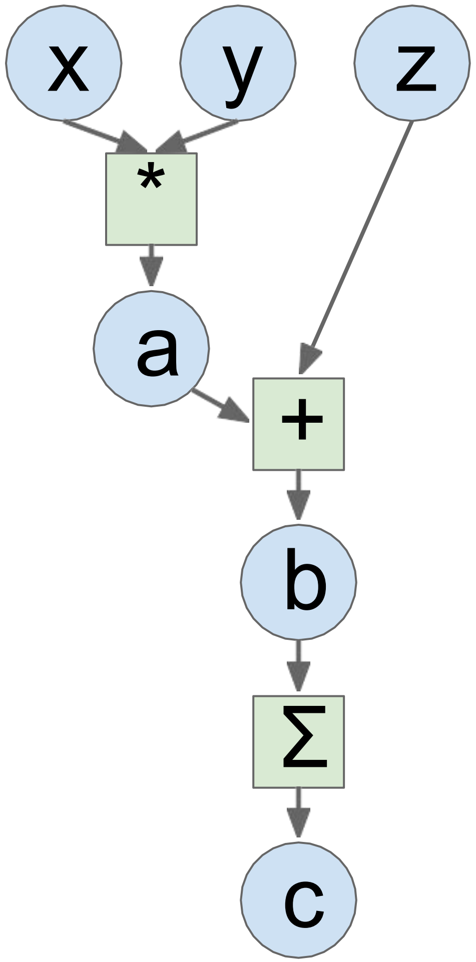
Bad:
- Have to compute our own gradients
- Can't run on GPU

Computational Graphs

Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)
```

```
grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

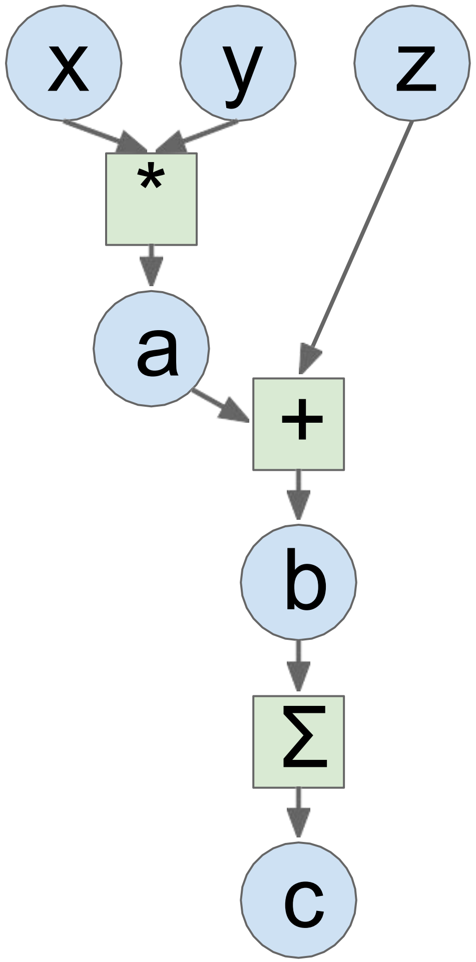
```
import torch

N, D = 3, 4
x = torch.randn(N, D)
y = torch.randn(N, D)
z = torch.randn(N, D)

a = x * y
b = a + z
c = torch.sum(b)
```

Looks exactly like numpy!

Computational Graphs

Numpy

```python
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

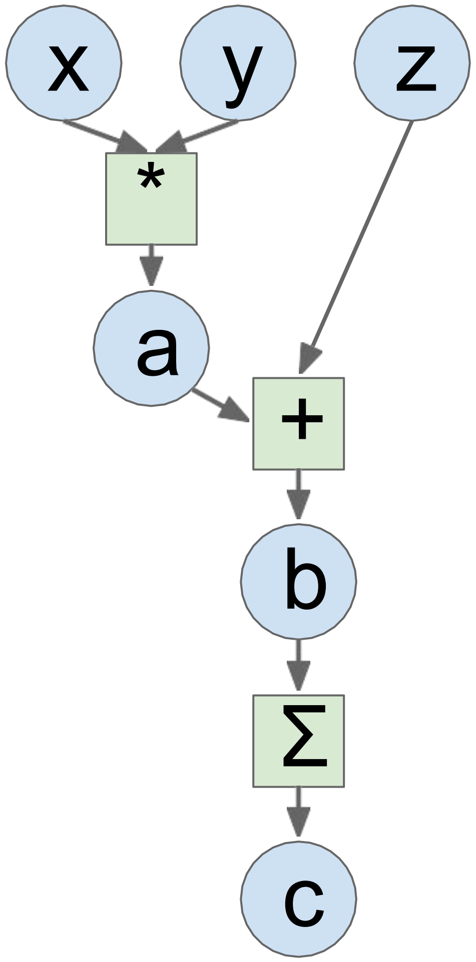
```python
import torch

N, D = 3, 4
x = torch.randn(N, D, requires_grad=True)
y = torch.randn(N, D)
z = torch.randn(N, D)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
print(x.grad)
```

PyTorch handles gradients for us!

Computational Graphs

Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

```
import torch

device = 'cuda:0'
N, D = 3, 4
x = torch.randn(N, D, requires_grad=True,
                device=device)
y = torch.randn(N, D, device=device)
z = torch.randn(N, D, device=device)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
print(x.grad)
```

Trivial to run on GPU - just construct arrays on a different device!

PyTorch: **Dynamic** Computation Graphs

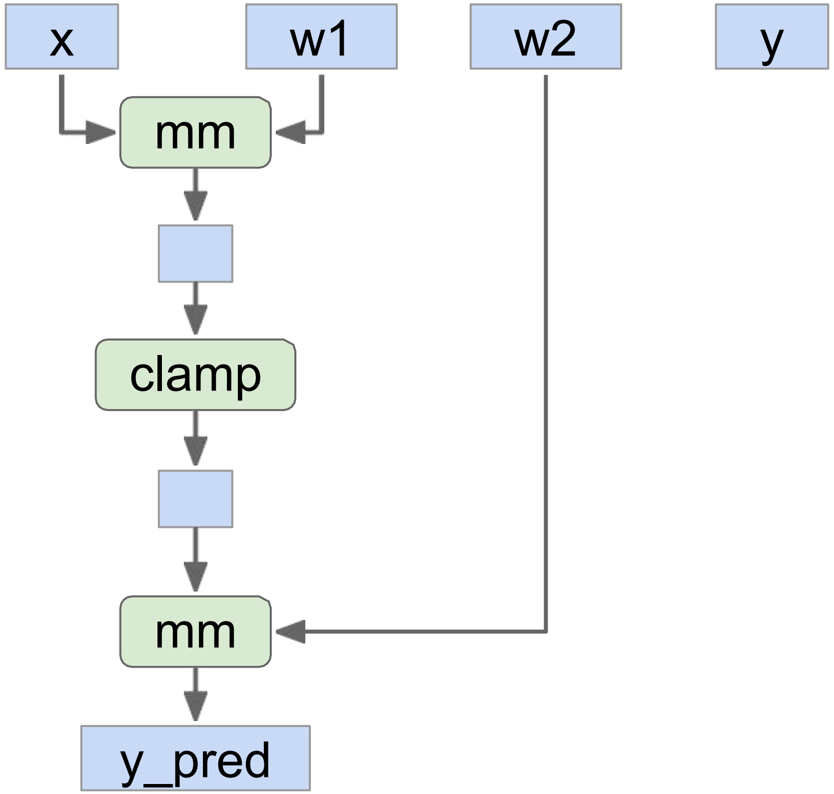
```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

PyTorch: **Dynamic** Computation Graphs


```python
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Create Tensor objects

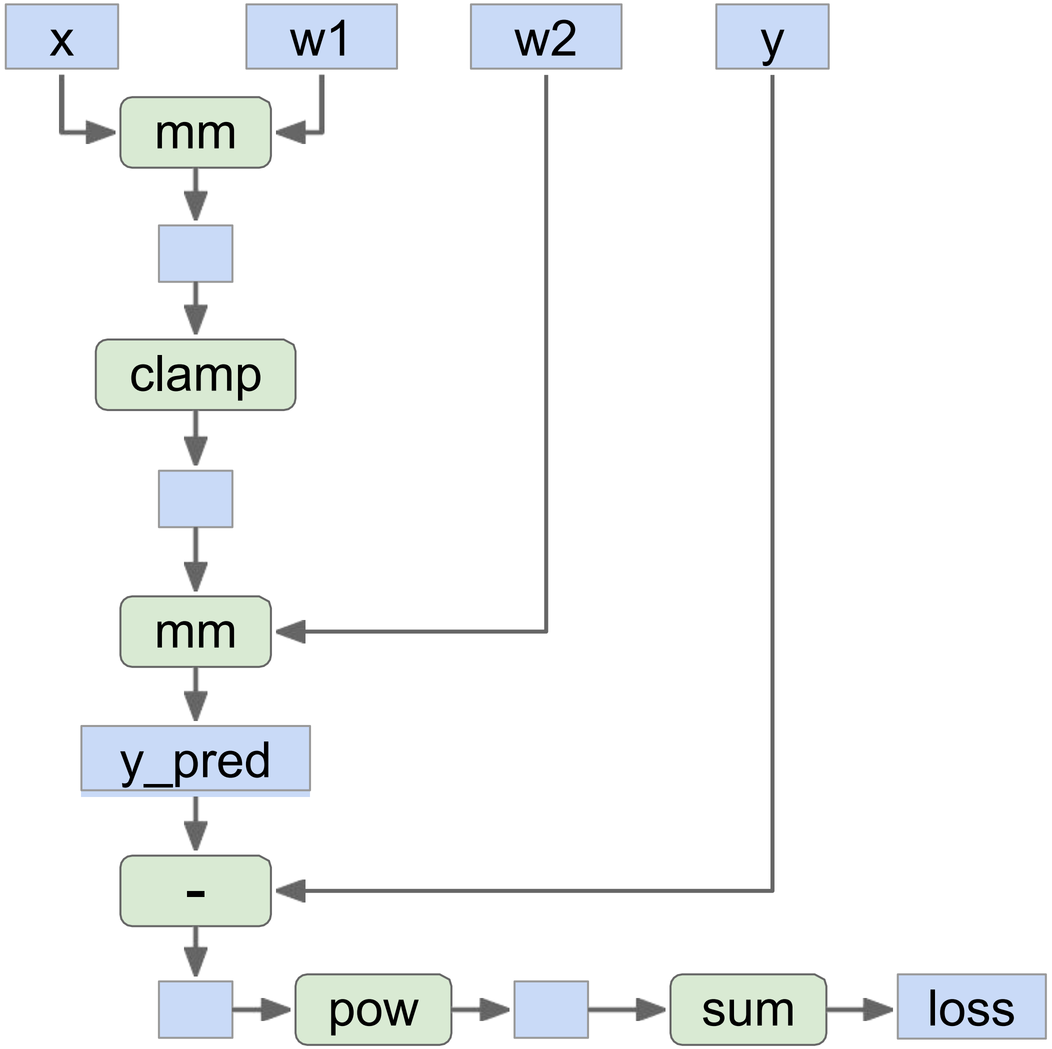
PyTorch: **Dynamic** Computation Graphs




```python
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Build graph data structure AND perform computation

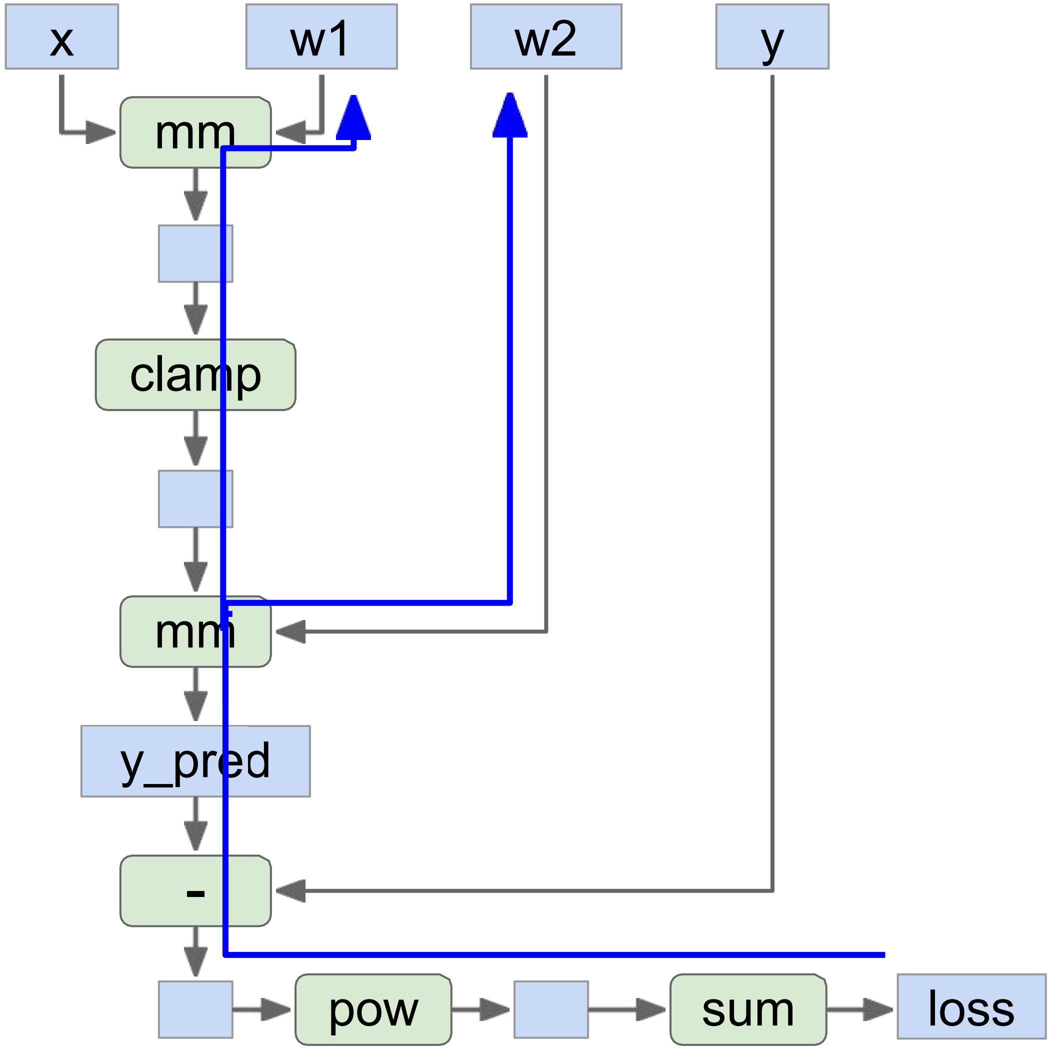
PyTorch: **Dynamic** Computation Graphs




```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Build graph data structure AND perform computation

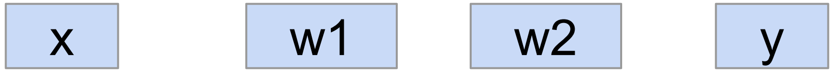
PyTorch: **Dynamic** Computation Graphs




```python
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Search for path between loss and w1, w2 (for backprop) AND perform computation

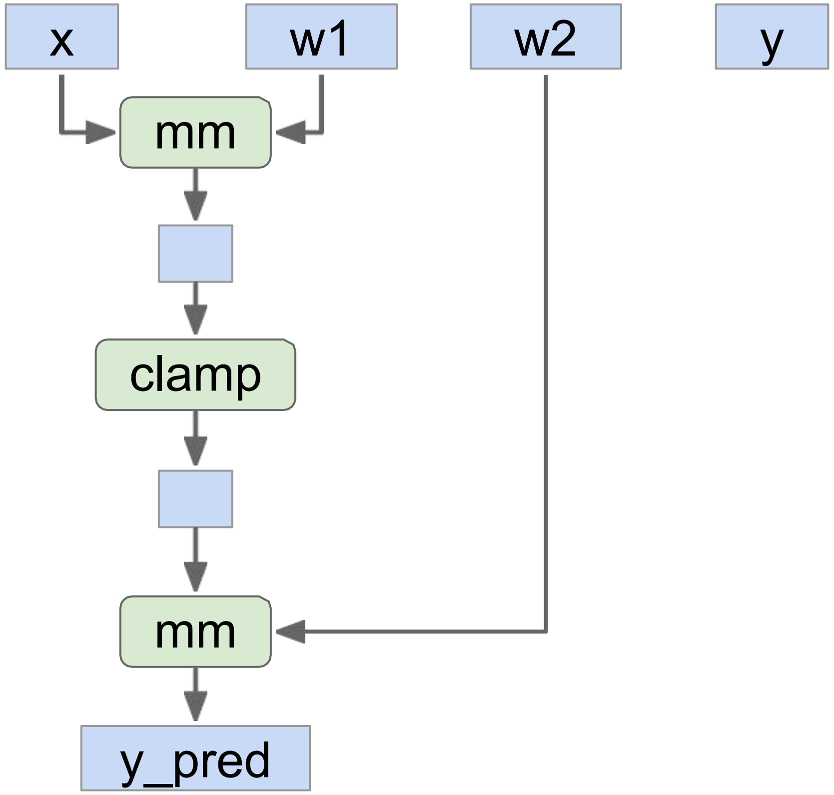
PyTorch: **Dynamic** Computation Graphs




```python
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Throw away the graph, backprop path, and rebuild it from scratch on every iteration

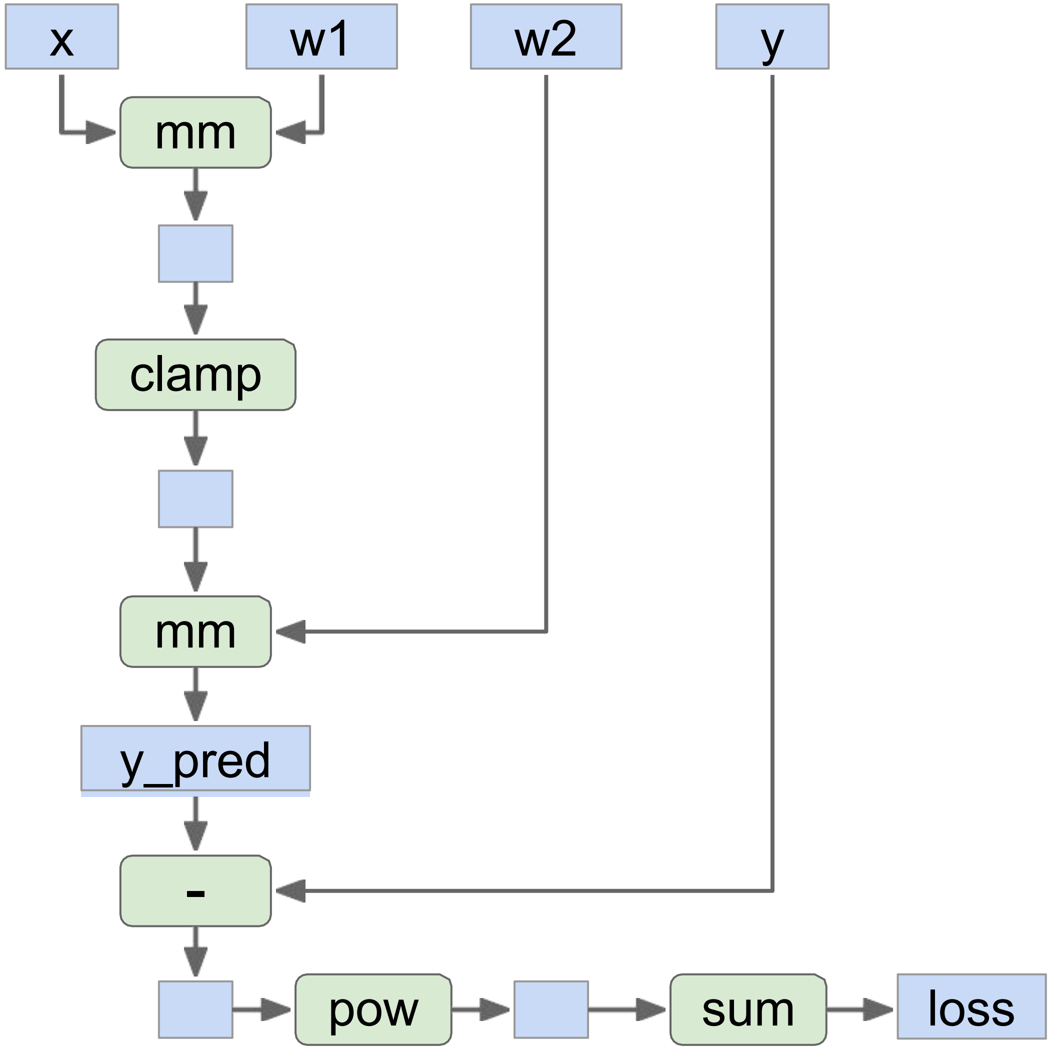
PyTorch: **Dynamic** Computation Graphs




```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Build graph data structure AND perform computation

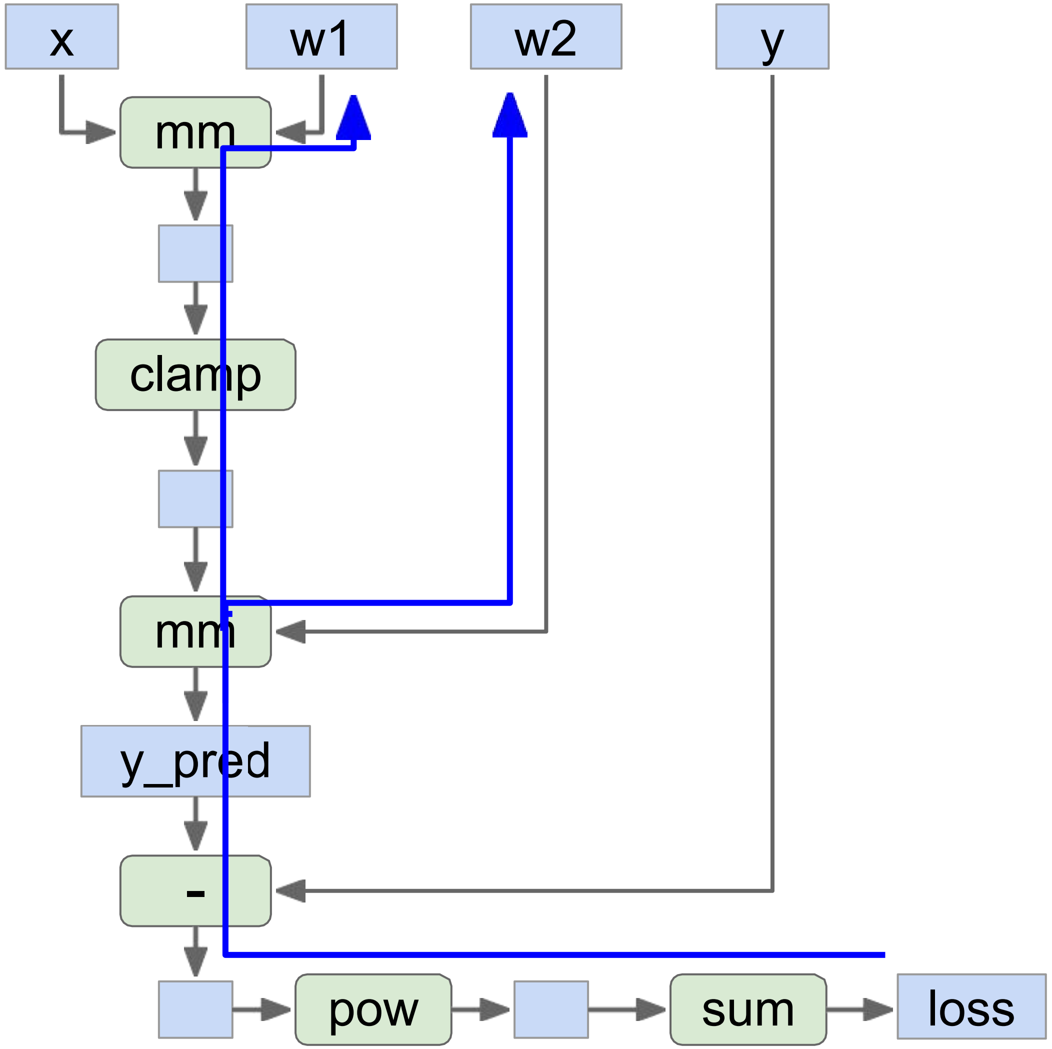
PyTorch: **Dynamic** Computation Graphs




```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Build graph data structure AND perform computation

PyTorch: **Dynamic** Computation Graphs




```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Search for path between loss and w1, w2
(for backprop) AND perform computation

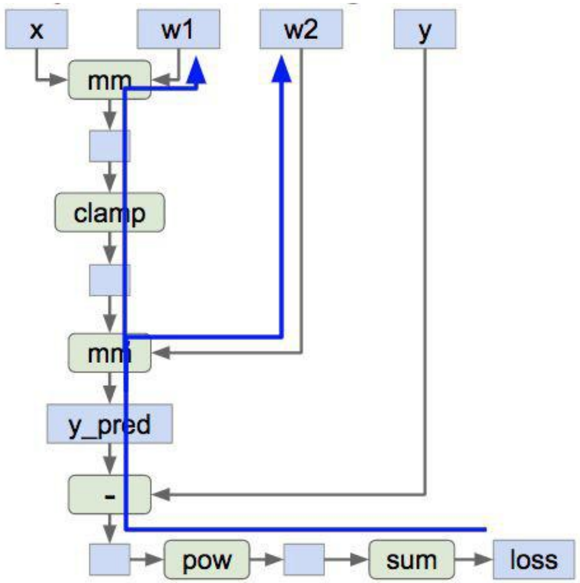
PyTorch: **Dynamic** Computation Graphs

Building the graph and **computing** the graph happen at the same time.

Seems inefficient, especially if we are building the same graph over and over again...

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Static Computation Graphs



Alternative: **Static** graphs

Step 1: Build computational graph describing our computation (including finding paths for backprop)

Step 2: Reuse the same graph on every iteration

```
graph = build_graph()

for x_batch, y_batch in loader:
    run_graph(graph, x=x_batch, y=y_batch)
```

TensorFlow: Neural Net

First **define** computational graph

```python
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])
```

Then **run** the graph many times

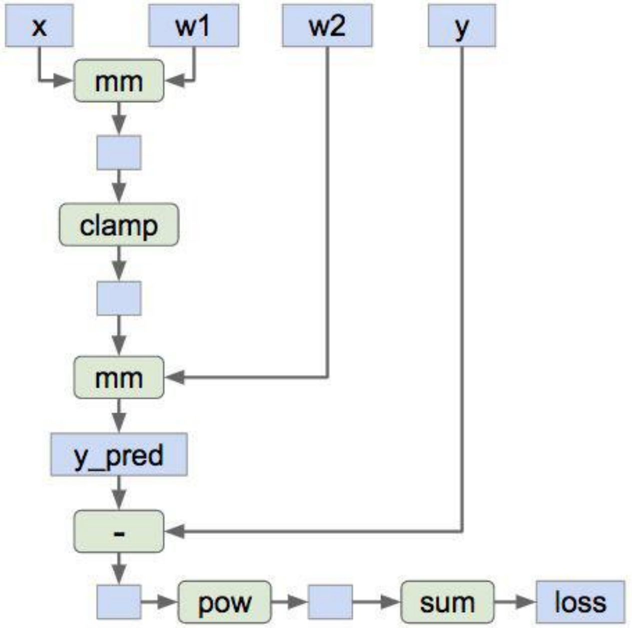
```python
with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net

Create **placeholders** for input x, weights w1 and w2, and targets y

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

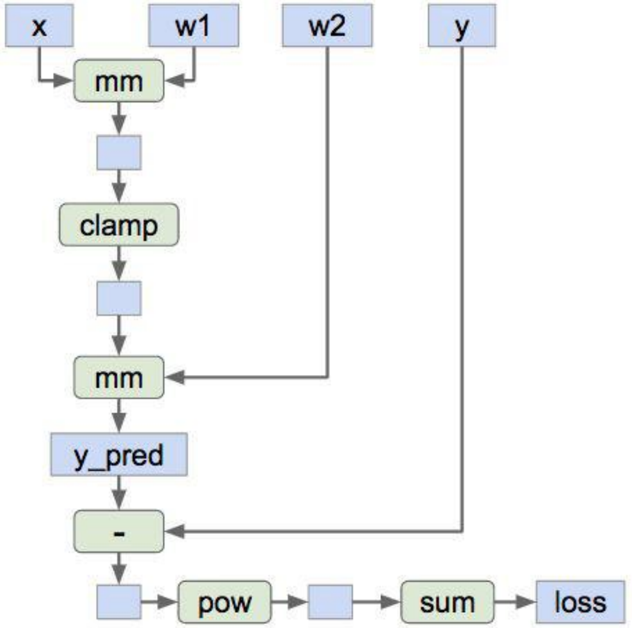
TensorFlow: Neural Net



Forward pass: compute prediction for y and loss. No computation - just building graph

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

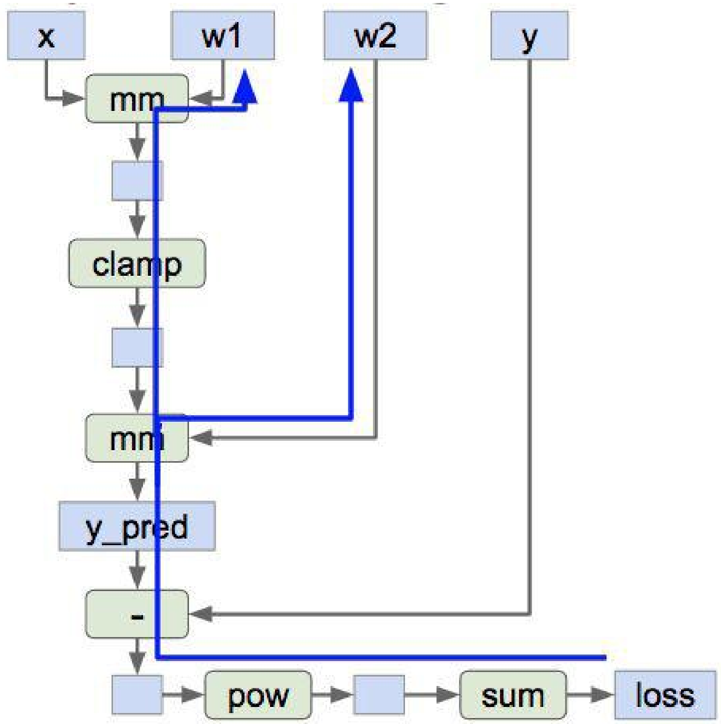
TensorFlow: Neural Net



Tell TensorFlow to compute loss of gradient with respect to w1 and w2.
No compute - just building the graph

```python
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

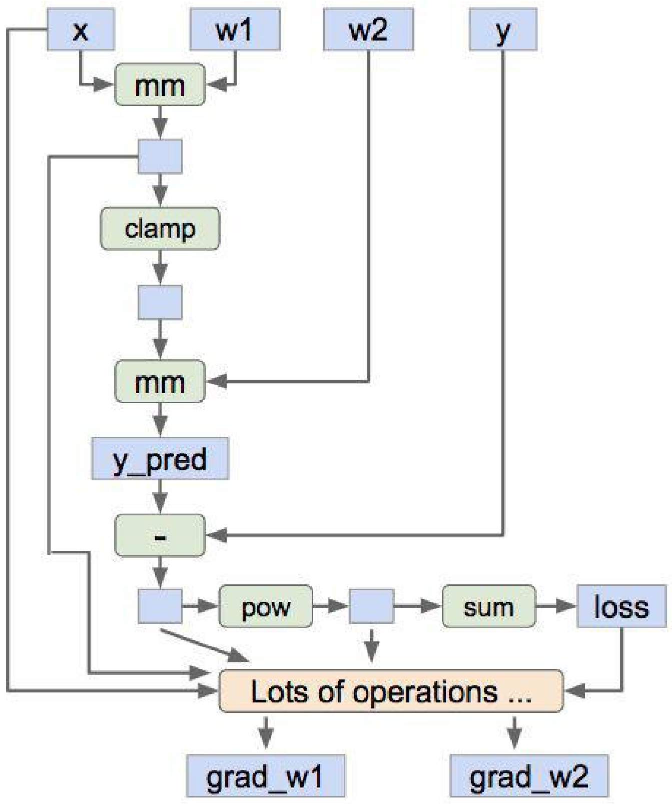
TensorFlow: Neural Net



Find paths between loss and w1, w2

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

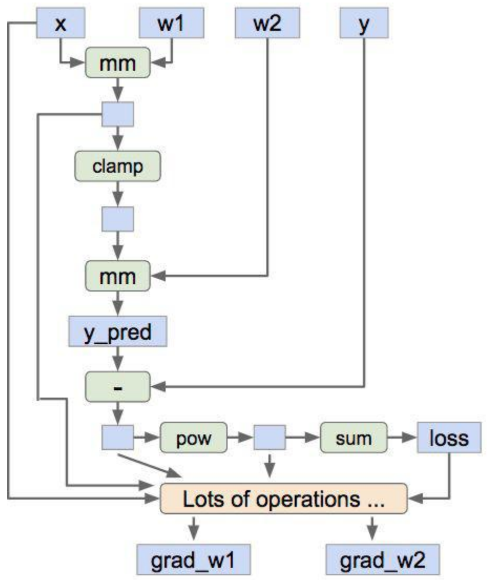
TensorFlow: Neural Net



Add new operators to the graph which compute grad_w1 and grad_w2

```python
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

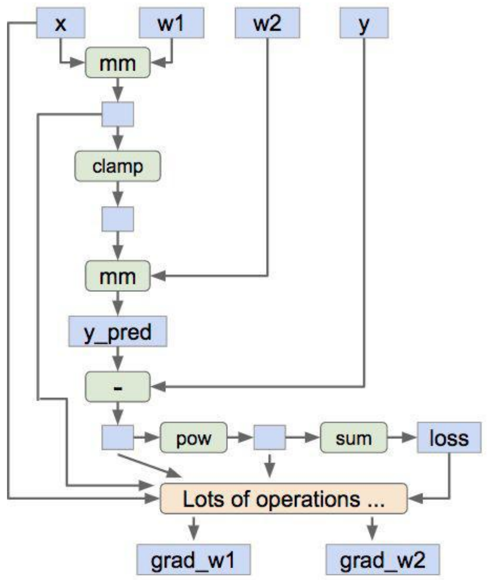
TensorFlow: Neural Net



Now done building our graph, so we enter a **session** so we can actually run the graph

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

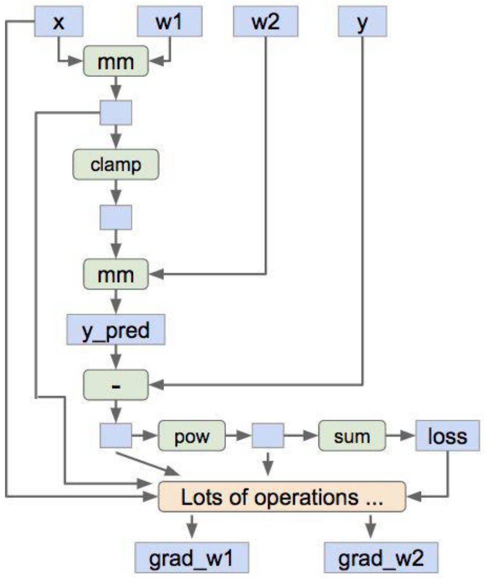
TensorFlow: Neural Net



Create numpy arrays that will fill in the placeholders above

```python
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net



Run the graph: feed in the numpy arrays for x, y, w1, and w2; get numpy arrays for loss, grad_w1, and grad_w2

```python
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

Static vs Dynamic Graphs

TensorFlow: Build graph once, then run many times (**static**)

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
updates = tf.group(new_w1, new_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                               feed_dict=values)
```

Build graph

Run each iteration

PyTorch: Each forward pass defines a new graph (**dynamic**)

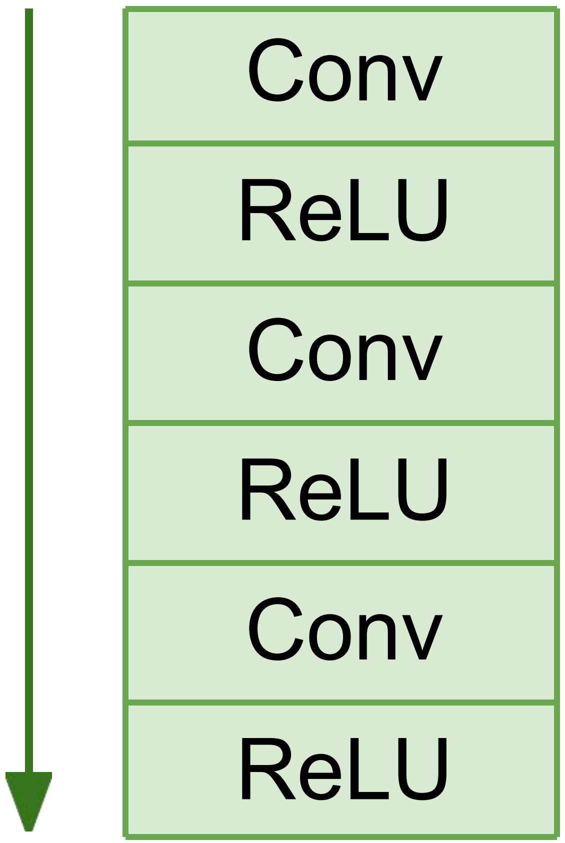
```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

New graph each iteration

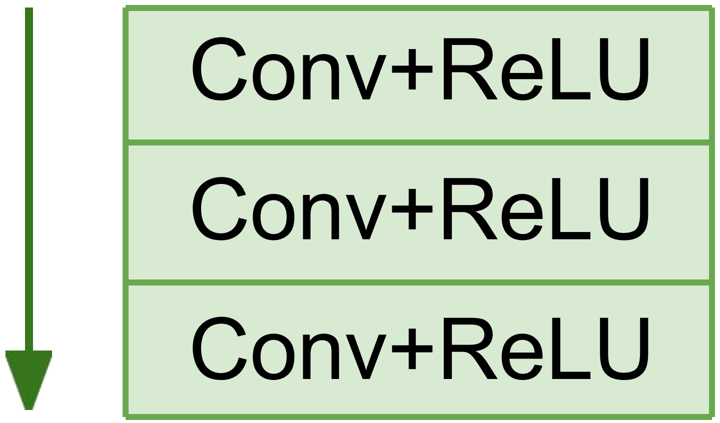
Static vs Dynamic: Optimization

With static graphs, framework can **optimize** the graph for you before it runs!

The graph you wrote



Equivalent graph with **fused operations**



Static vs Dynamic: Serialization

Static

Once graph is built, can **serialize** it and run it without the code that built the graph!

Dynamic

Graph building and execution are intertwined, so always need to keep code around

Static vs Dynamic: Conditional

$$y = \begin{cases} w1 * x & \text{if } z > 0 \\ w2 * x & \text{otherwise} \end{cases}$$

Static vs Dynamic: Conditional

$$y = \begin{cases} w1 * x & \text{if } z > 0 \\ w2 * x & \text{otherwise} \end{cases}$$

PyTorch: Normal Python

```
N, D, H = 3, 4, 5

x = torch.randn(N, D, requires_grad=True)
w1 = torch.randn(D, H)
w2 = torch.randn(D, H)

z = 10
if z > 0:
    y = x.mm(w1)
else:
    y = x.mm(w2)
```

Static vs Dynamic: Conditional

$$y = \begin{cases} w1 * x & \text{if } z > 0 \\ w2 * x & \text{otherwise} \end{cases}$$

PyTorch: Normal Python

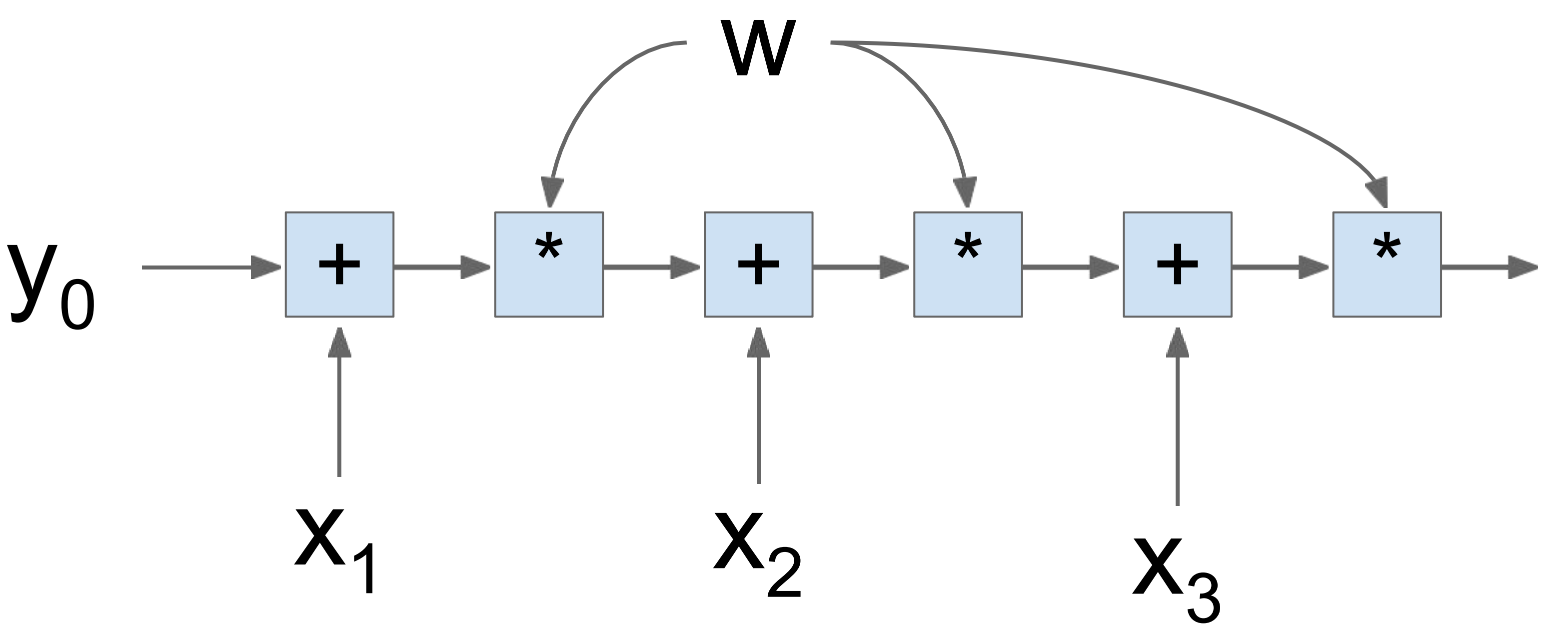
```
N, D, H = 3, 4, 5

x = torch.randn(N, D, requires_grad=True)
w1 = torch.randn(D, H)
w2 = torch.randn(D, H)

z = 10
if z > 0:
    y = x.mm(w1)
else:
    y = x.mm(w2)
```

TensorFlow: Special TF control flow operator!

```
N, D, H = 3, 4, 5
x = tf.placeholder(tf.float32, shape=(N, D))
z = tf.placeholder(tf.float32, shape=None)
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(D, H))

def f1(): return tf.matmul(x, w1)
def f2(): return tf.matmul(x, w2)
y = tf.cond(tf.less(z, 0), f1, f2)

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        z: 10,
        w1: np.random.randn(D, H),
        w2: np.random.randn(D, H),
    }
    y_val = sess.run(y, feed_dict=values)
```

Static vs Dynamic: Loops

$$y_t = (y_{t-1} + x_t) * w$$

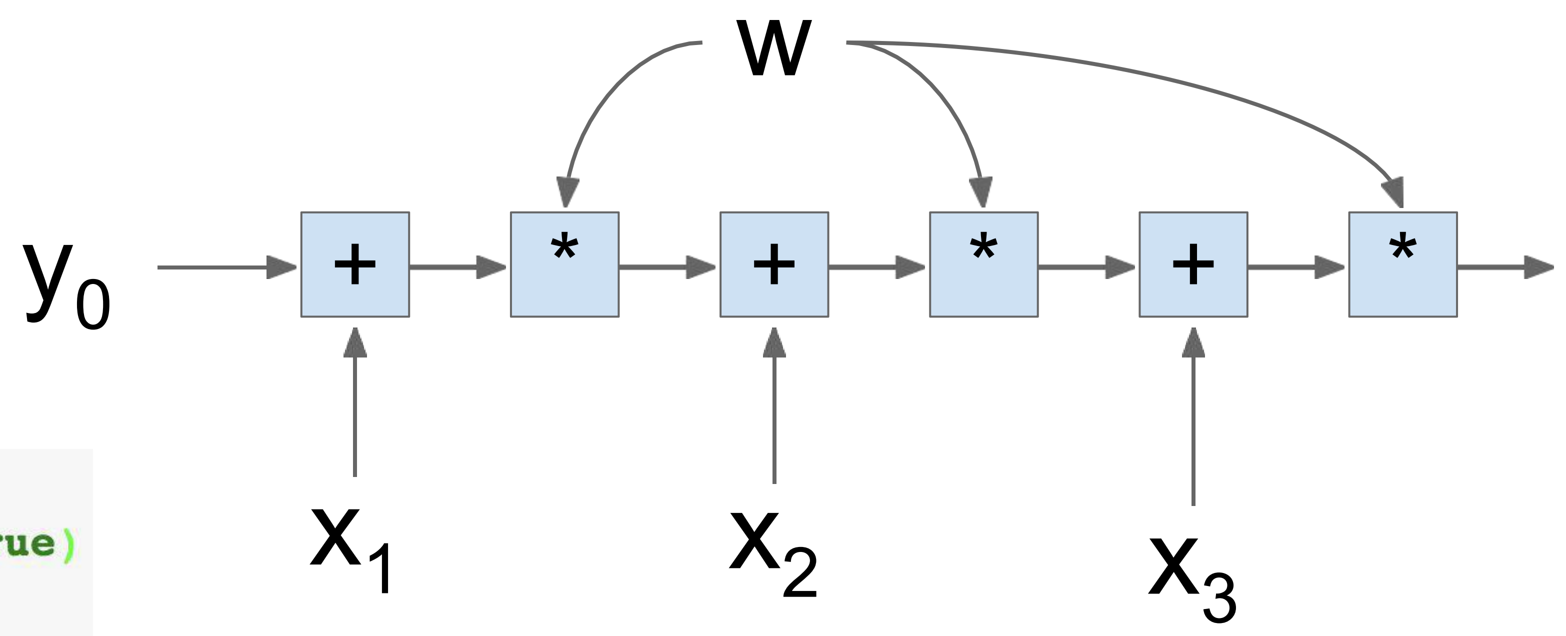


Static vs Dynamic: Loops

$$y_t = (y_{t-1} + x_t) * w$$

PyTorch: Normal Python

```
T, D = 3, 4
y0 = torch.randn(D, requires_grad=True)
x = torch.randn(T, D)
w = torch.randn(D)

y = [y0]
for t in range(T):
    prev_y = y[-1]
    next_y = (prev_y + x[t]) * w
    y.append(next_y)
```



Static vs Dynamic: Loops

$$y_t = (y_{t-1} + x_t) * w$$

PyTorch: Normal Python

```
T, D = 3, 4
y0 = torch.randn(D, requires_grad=True)
x = torch.randn(T, D)
w = torch.randn(D)

y = [y0]
for t in range(T):
    prev_y = y[-1]
    next_y = (prev_y + x[t]) * w
    y.append(next_y)
```

TensorFlow: Special TF control flow

```
T, N, D = 3, 4, 5
x = tf.placeholder(tf.float32, shape=(T, D))
y0 = tf.placeholder(tf.float32, shape=(D,))
w = tf.placeholder(tf.float32, shape=(D,))

def f(prev_y, cur_x):
    return (prev_y + cur_x) * w

y = tf.foldl(f, x, y0)

with tf.Session() as sess:
    values = {
        x: np.random.randn(T, D),
        y0: np.random.randn(D),
        w: np.random.randn(D),
    }
    y_val = sess.run(y, feed_dict=values)
```

ONNX Support

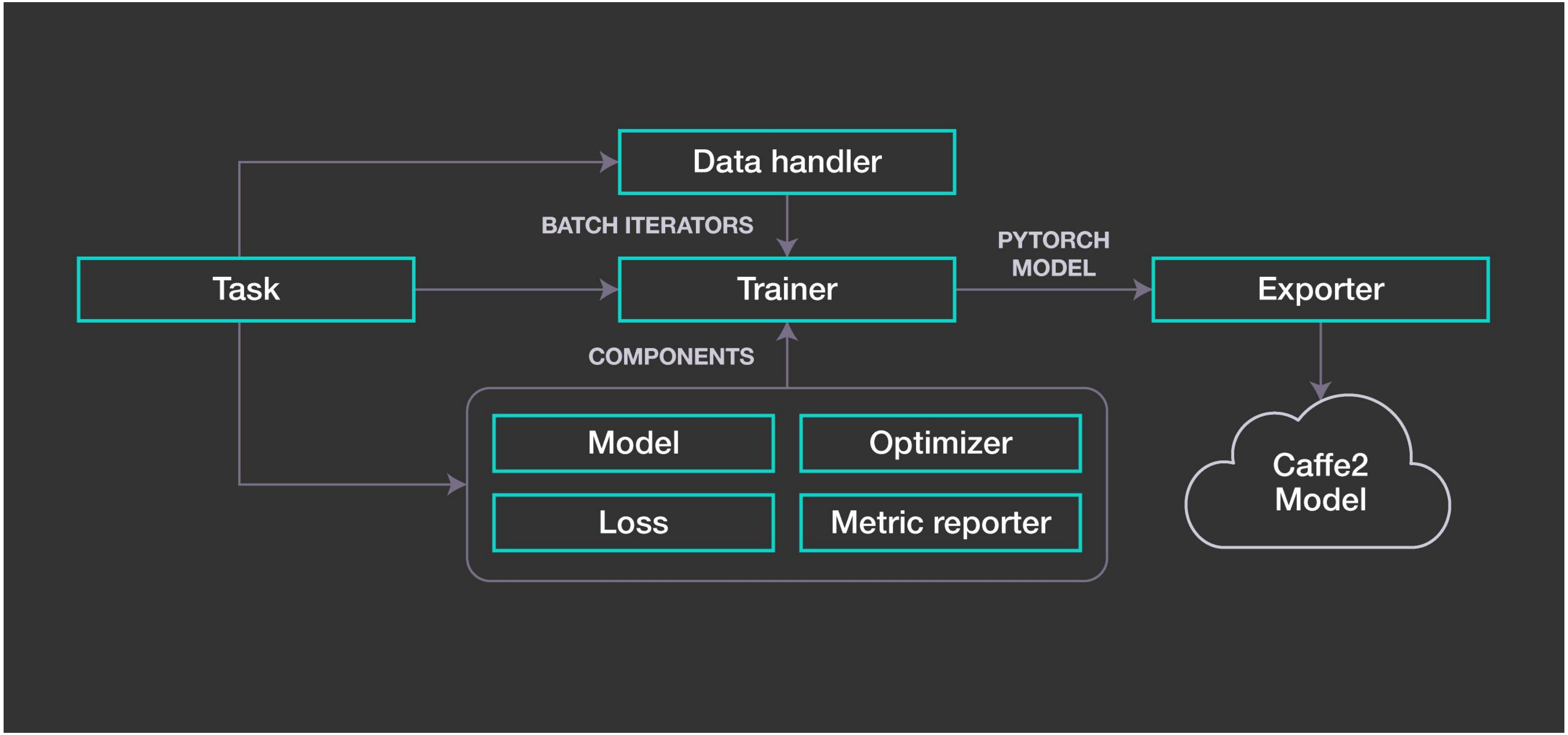
ONNX is an open-source standard for neural network models

Goal: Make it easy to train a network in one framework, then run it in another framework

Supported by PyTorch, Caffe2, Microsoft CNTK, Apache MXNet

https://github.com/onnx/onnx

PyText workflow



```
{
  "config": {
    "task": {
      "SemanticParsingTask": {
        "model": {
          "lstm": {
            "dropout": 0.34,
            "lstm_dim": 164,
            "num_layers": 2,
            "bidirectional": true
          },
          "ablation": {
            "use_buffer": true,
            "use_stack": true,
            "use_action": true,
            "use_last_open_NT_feature": false
          },
          "constraints": {
            "intent_slot_nesting": true,
            "ignore_loss_for_unsupported": false,
            "no_slots_inside_unsupported": true
          },
          "max_open_NT": 10,
          "dropout": 0.34,
          "compositional_type": "sum"
        },
```



```python
#!/usr/bin/env python3
# Copyright (c) Facebook, Inc. and its affiliates. All Rights Reserved

import json

import atis
from flask import Flask, request


app = Flask(__name__)


@app.route("/")
def predict():
    return json.dumps(atis.predict(request.args.get("text", "")))


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=3000)
```

Torch.nn

The nn modules in PyTorch provides us a higher level API to build and train deep network.

torch.nn

- Containers
- Linear layers
- Convolution layers
- Pooling layers
- Recurrent layers
- Dropout layers
- Normalization layers
- Non-linear activations
- Loss functions
- Padding layers
- Sparse layers
- Distance functions
- Vision layers
- Utilities

Containers

- Module
- Sequential
- ModuleList
- ParameterList

Module

- Base class for all neural network modules.
- Your models should also subclass this class.
- Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

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

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

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

Sequential

- A sequential container. Modules will be added to it in the order they are passed in the constructor.
- Alternatively, an ordered dict of modules can also be passed in.

```
# Example of using Sequential
model = nn.Sequential(
          nn.Conv2d(1,20,5),
          nn.ReLU(),
          nn.Conv2d(20,64,5),
          nn.ReLU()
        )

# Example of using Sequential with OrderedDict
model = nn.Sequential(OrderedDict([
          ('conv1', nn.Conv2d(1,20,5)),
          ('relu1', nn.ReLU()),
          ('conv2', nn.Conv2d(20,64,5)),
          ('relu2', nn.ReLU())
        ]))
```

ModuleList

- ModuleList can be indexed like a regular Python list, but modules it contains are properly registered, and will be visible by all Module methods.

```
class MyModule(nn.Module):
    def __init__(self):
        super(MyModule, self).__init__()
        self.linears = nn.ModuleList([nn.Linear(10, 10) for i in range(10)])

    def forward(self, x):
        # ModuleList can act as an iterable, or be indexed using ints
        for i, l in enumerate(self.linears):
            x = self.linears[i // 2](x) + l(x)
        return x
```

ParameterList

- ParameterList can be indexed like a regular Python list, but parameters it contains are properly registered, and will be visible by all Module methods.

```
class MyModule(nn.Module):
    def __init__(self):
        super(MyModule, self).__init__()
        self.params = nn.ParameterList([nn.Parameter(torch.randn(10, 10)) for i in range(10)])

    def forward(self, x):
        # ParameterList can act as an iterable, or be indexed using ints
        for i, p in enumerate(self.params):
            x = self.params[i // 2].mm(x) + p.mm(x)
        return x
```

Linear layers

- Linear
- Bilinear

Linear

Applies a linear transformation to the incoming data: $y = xA^T + b$

```
>>> m = nn.Linear(20, 30)
>>> input = torch.randn(128, 20)
>>> output = m(input)
>>> print(output.size())
torch.Size([128, 30])
```

- torch.nn.Linear(*in_features*, *out_features*, *bias=True*)
- Parameters:
  - **in_features** – size of each input sample
  - **out_features** – size of each output sample
  - **bias** – If set to False, the layer will not learn an additive bias. Default: True

Bilinear

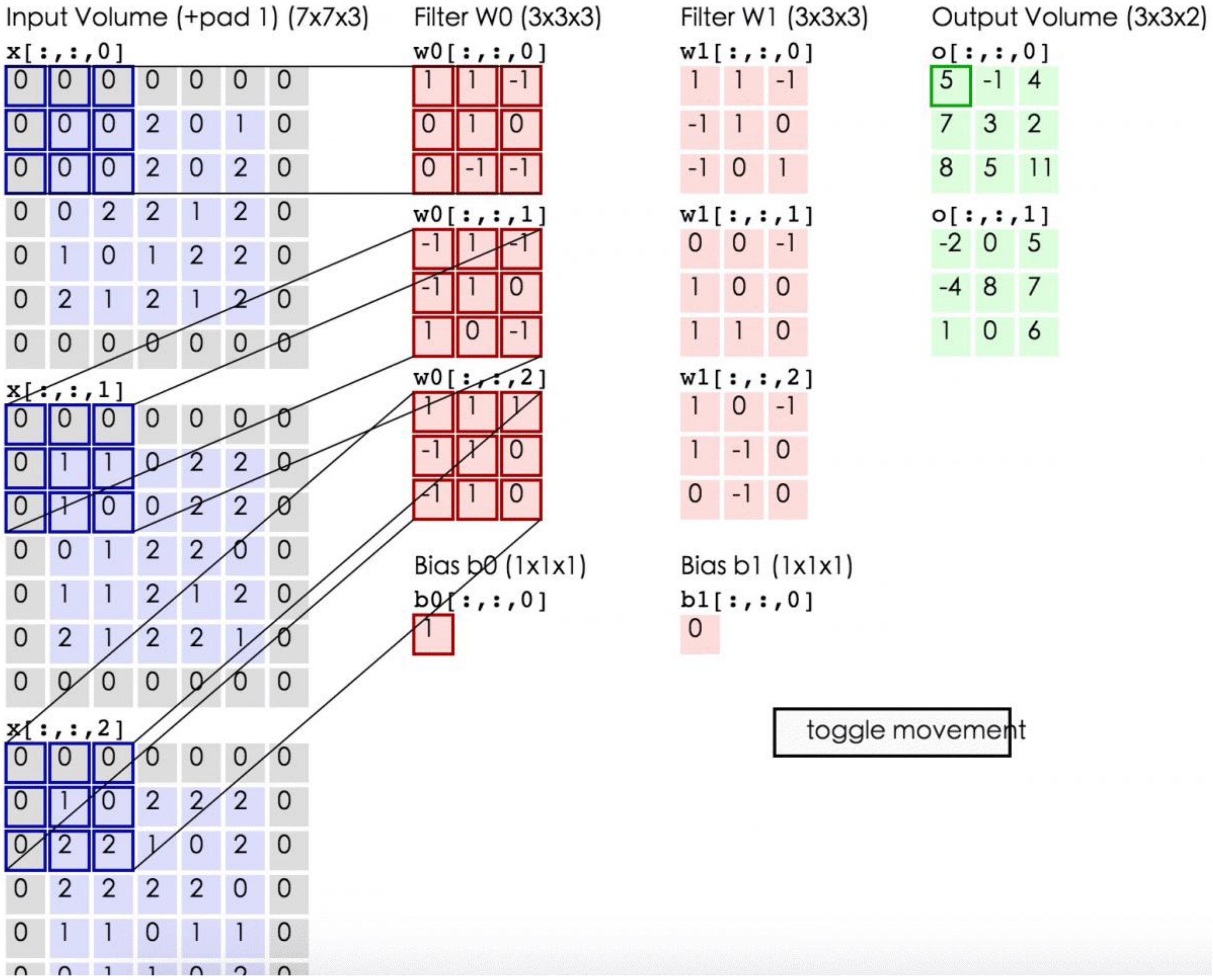
Applies a linear transformation to the incoming data: $y = x_1 A x_2 + b$

```
>>> m = nn.Bilinear(20, 30, 40)
>>> input1 = torch.randn(128, 20)
>>> input2 = torch.randn(128, 30)
>>> output = m(input1, input2)
>>> print(output.size())
torch.Size([128, 40])
```

- torch.nn.Bilinear(*in1_features*, *in2_features*, *out_features*, *bias=True*)
- Parameters:
  - **in1_features** – size of each first input sample
  - **in2_features** – size of each second input sample
  - **out_features** – size of each output sample
  - **bias** – If set to False, the layer will not learn an additive bias. Default: True

Convolution Layers

- Conv1d
- Conv2d
- Conv3d
- ConvTranspose1d
- ConvTranspose2d
- ConvTranspose3d



Input Volume (+pad 1) (7x7x3)
x[:,:,0]
x[:,:,1]
x[:,:,2]
Filter W0 (3x3x3)
w0[:,:,0]
w0[:,:,1]
w0[:,:,2]
Bias b0 (1x1x1)
b0[:,:,0]
Filter W1 (3x3x3)
w1[:,:,0]
w1[:,:,1]
w1[:,:,2]
Bias b1 (1x1x1)
b1[:,:,0]
Output Volume (3x3x2)
o[:,:,0]
o[:,:,1]
toggle movement

Conv2d

- torch.nn.Conv2d(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*)
- In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

Conv2d

- **Parameters:**
  - **in_channels** (*int*) – Number of channels in the input image
  - **out_channels** (*int*) – Number of channels produced by the convolution
  - **kernel_size** (*int or tuple*) – Size of the convolving kernel
  - **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
  - **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input. Default: 0
  - **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1
  - **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
  - **bias** (*bool, optional*) – If True, adds a learnable bias to the output. Default: True

Conv2d

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

```
>>> # With square kernels and equal stride
>>> m = nn.Conv2d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
>>> # non-square kernels and unequal stride and with padding and dilation
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2), dilation=(3, 1))
>>> input = torch.randn(20, 16, 50, 100)
>>> output = m(input)
```

Conv1d,2d,3d

- The parameters kernel_size, stride, padding, dilation can either be:
  - a single int – in which case the same value is used for the depth(, height and width) dimension
  - a tuple of three ints – in which case, the first *int* is used for the depth dimension, the second *int* for the height dimension and the third *int* for the width dimension

ConvTranspose

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where

$$H_{out} = (H_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{kernel\_size}[0] + \text{output\_padding}[0]$$

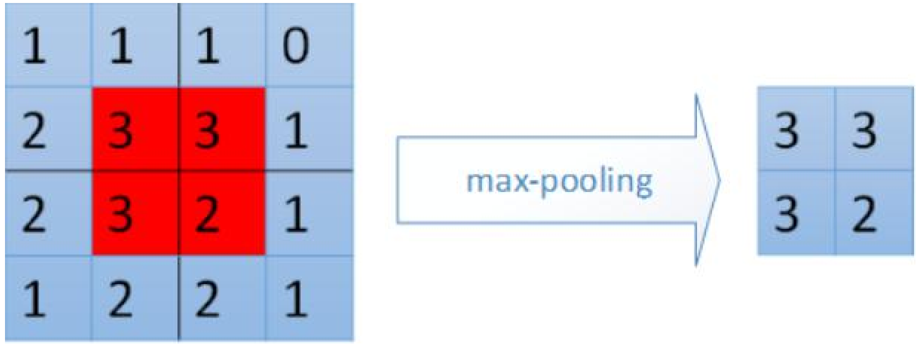
$$W_{out} = (W_{in} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{kernel\_size}[1] + \text{output\_padding}[1]$$

- exchange forward and backward propagation
- detail: https://github.com/vdumoulin/conv_arithmetic

Pooling Layers

- MaxPool1d
- MaxPool2d
- MaxPool3d
- MaxUnpool1d
- MaxUnpool2d
- MaxUnpool3d
- AvgPool1d
- AvgPool2d
- AvgPool3d
- AdaptiveMaxPool1d
- AdaptiveMaxPool2d
- AdaptiveMaxPool3d
- AdaptiveAvgPool1d
- AdaptiveAvgPool2d
- AdaptiveAvgPool3d

MaxPool



torch.nn.MaxPool2d(*kernel_size*, *stride=None*, *padding=0*, *dilation=1*, *return_indices=False*, *ceil_mode=False*)

- In the simplest case, the output value of the layer with input size N, C, H, W),output ($N, C, Hout, Wout$) and kernel_size (kH, kW) can be precisely described as:

$$out(N_i, C_j, h, w) = \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)$$

MaxPool

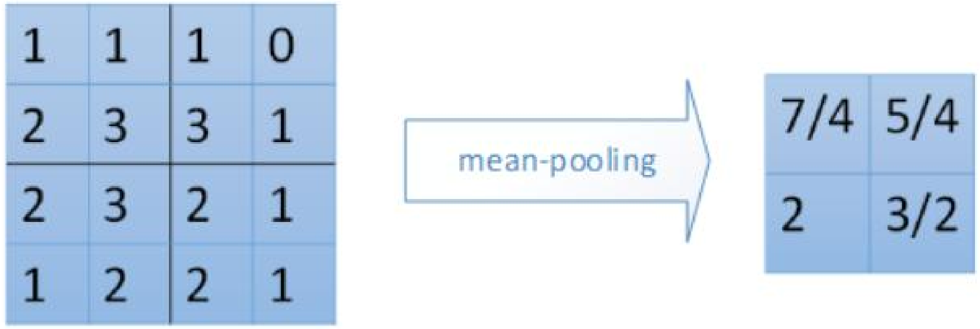
- **kernel_size** – the size of the window to take a max over
- **stride** – the stride of the window. Default value is kernel_size
- **padding** – implicit zero padding to be added on both sides
- **dilation** – a parameter that controls the stride of elements in the window
- **return_indices** – if True, will return the max indices along with the outputs. Useful for torch.nn.MaxUnpool2d later
- **ceil_mode** – when True, will use *ceil* instead of *floor* to compute the output shape

MaxUnpool

- Computes a partial inverse of MaxPool1d.
- MaxPool1d is not fully invertible, since the non-maximal values are lost.
- MaxUnpool1d takes in as input the output of MaxPool1d including the indices of the maximal values and computes a partial inverse in which all non-maximal values are set to zero.

```
>>> pool = nn.MaxPool1d(2, stride=2, return_indices=True)
>>> unpool = nn.MaxUnpool1d(2, stride=2)
>>> input = torch.tensor([[[1., 2, 3, 4, 5, 6, 7, 8]]])
>>> output, indices = pool(input)
>>> unpool(output, indices)
tensor([[[ 0.,  2.,  0.,  4.,  0.,  6.,  0., 8.]]])
```

AvgPool



- Applies a 1D average pooling over an input signal composed of several input planes.
- In the simplest case, the output value of the layer with input size (N,C,L), output (N,C,$Lout$) and kernel_size k can be precisely described as:

$$\text{out}(N_i, C_j, l) = \frac{1}{k} \sum_{m=0}^{k} \text{input}(N_i, C_j, \text{stride} \times l + m)$$

AvgPool

- **kernel_size** – the size of the window
- **stride** – the stride of the window. Default value is kernel_size
- **padding** – implicit zero padding to be added on both sides
- **ceil_mode** – when True, will use *ceil* instead of *floor* to compute the output shape
- **count_include_pad** – when True, will include the zero-padding in the averaging calculation

AdaptiveMaxPool

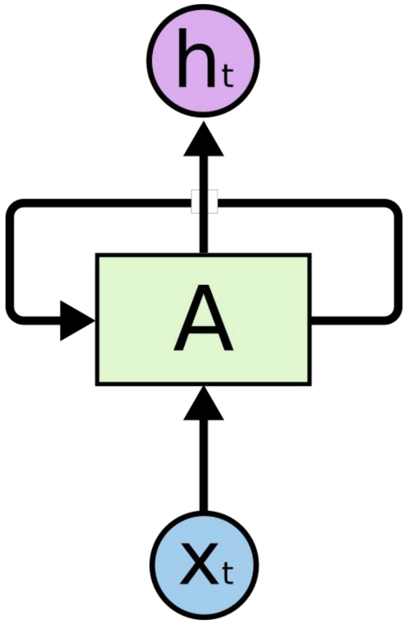
- torch.nn.AdaptiveMaxPool2d(*output_size*, *return_indices=False*)

```
>>> # target output size of 5x7
>>> m = nn.AdaptiveMaxPool2d((5,7))
>>> input = torch.randn(1, 64, 8, 9)
>>> output = m(input)
>>> # target output size of 7x7 (square)
>>> m = nn.AdaptiveMaxPool2d(7)
>>> input = torch.randn(1, 64, 10, 9)
>>> output = m(input)
>>> # target output size of 10x7
>>> m = nn.AdaptiveMaxPool2d((None, 7))
>>> input = torch.randn(1, 64, 10, 9)
>>> output = m(input)
```

Recurrent layers

- RNN
- LSTM
- GRU
- RNNCell
- LSTMCell
- GRUCell

RNN



- Applies a multi-layer Elman RNN with tanh*tanh* or ReLU*ReLU* non-linearity to an input sequence.
- For each element in the input sequence, each layer computes the following function:

$$h_t = \tanh(w_{ih} x_t + b_{ih} + w_{hh} h_{(t-1)} + b_{hh})$$

where h_t is the hidden state at time *t*, x_t is the input at time *t*, and $h_{(t-1)}$ is the hidden state of the previous layer at time *t-1* or the initial hidden state at time *0*. If nonlinearity is *'relu'*, then *ReLU* is used instead of *tanh*.

RNN

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting num_layers=2 would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- **nonlinearity** – The non-linearity to use. Can be either ‘tanh’ or ‘relu’. Default: ‘tanh’
- **bias** – If False, then the layer does not use bias weights *b_ih* and *b_hh*. Default: True
- **batch_first** – If True, then the input and output tensors are provided as *(batch, seq, feature)*. Default: False
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last layer, with dropout probability equal to dropout. Default: 0
- **bidirectional** – If True, becomes a bidirectional RNN. Default: False

RNN

- **input** of shape *(seq_len, batch, input_size)*: tensor containing the features of the input sequence.
- **h_0** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided. If the RNN is bidirectional, num_directions should be 2, else it should be 1.

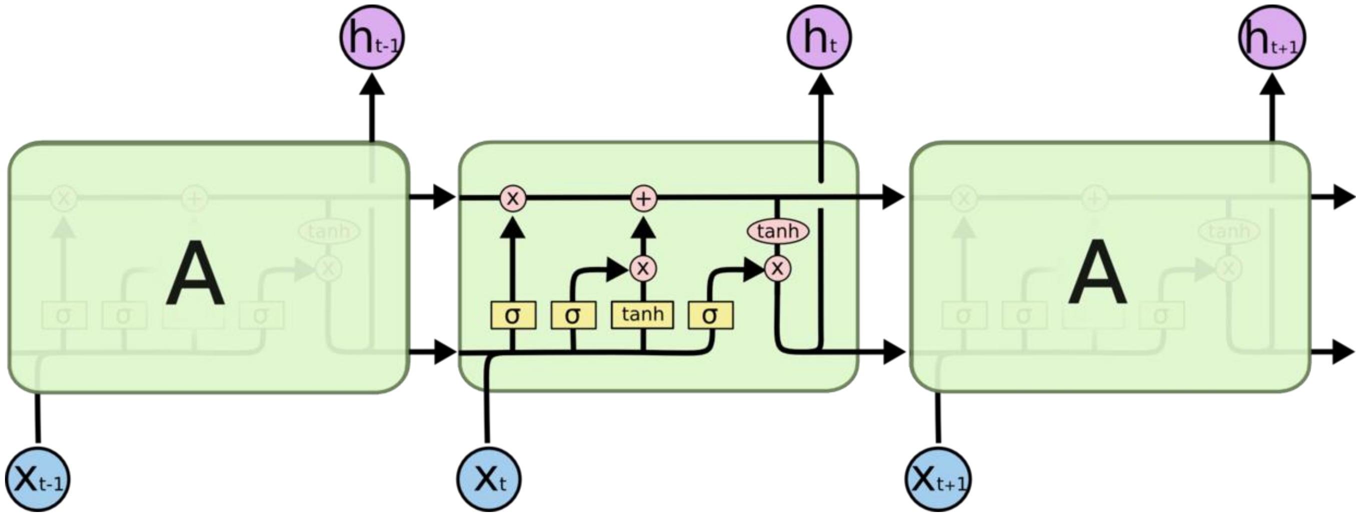
RNN

- **output** of shape *(seq_len, batch, num_directions * hidden_size)*: tensor containing the output features (*h_k*) from the last layer of the RNN, for each *k*.
- **h_n** (num_layers * num_directions, batch, hidden_size): tensor containing the hidden state for *k* = *seq_len*.

RNN

```
>>> rnn = nn.RNN(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> output, hn = rnn(input, h0)
```

LSTM



$$\begin{aligned}
i_t &= \sigma(W_{ii} x_t + b_{ii} + W_{hi} h_{(t-1)} + b_{hi}) \\
f_t &= \sigma(W_{if} x_t + b_{if} + W_{hf} h_{(t-1)} + b_{hf}) \\
g_t &= \tanh(W_{ig} x_t + b_{ig} + W_{hg} h_{(t-1)} + b_{hg}) \\
o_t &= \sigma(W_{io} x_t + b_{io} + W_{ho} h_{(t-1)} + b_{ho}) \\
c_t &= f_t c_{(t-1)} + i_t g_t \\
h_t &= o_t \tanh(c_t)
\end{aligned}$$

- where h_t is the hidden state at time t, *ct* is the cell state at time t, x_t is the input at time t, $h_{(t-1)}$ is the hidden state of the layer at time *t-1* or the initial hidden state at time *0*, and i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively.

LSTM

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting num_layers=2 would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **bias** – If False, then the layer does not use bias weights *b_ih* and *b_hh*. Default: True
- **batch_first** – If True, then the input and output tensors are provided as (batch, seq, feature). Default: False
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to dropout. Default: 0
- **bidirectional** – If True, becomes a bidirectional LSTM. Default: False

LSTM

- **input** of shape *(seq_len, batch, input_size)*: tensor containing the features of the input sequence. The input can also be a packed variable length sequence.
- **h_0** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the initial hidden state for each element in the batch. If the RNN is bidirectional, num_directions should be 2, else it should be 1.
- **c_0** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the initial cell state for each element in the batch.
- If *(h_0, c_0)* is not provided, both **h_0** and **c_0** default to zero.

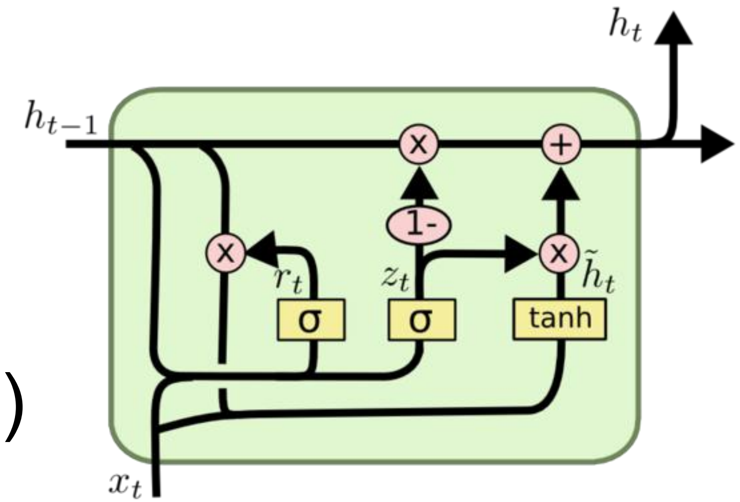
LSTM

- **output** of shape *(seq_len, batch, num_directions * hidden_size)*: tensor containing the output features (h_t) from the last layer of the LSTM, for each t.
- **h_n** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the hidden state for *t* = *seq_len*.
- **c_n** (num_layers * num_directions, batch, hidden_size): tensor containing the cell state for *t* = *seq_len*

LSTM

```
>>> rnn = nn.LSTM(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> c0 = torch.randn(2, 3, 20)
>>> output, (hn, cn) = rnn(input, (h0, c0))
```

GRU



- Applies a multi-layer gated recurrent unit (GRU)

RNN to an input sequence.

- For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
r_t &= \sigma(W_{ir} x_t + b_{ir} + W_{hr} h_{(t-1)} + b_{hr}) \\
z_t &= \sigma(W_{iz} x_t + b_{iz} + W_{hz} h_{(t-1)} + b_{hz}) \\
n_t &= \tanh(W_{in} x_t + b_{in} + r_t (W_{hn} h_{(t-1)} + b_{hn})) \\
h_t &= (1 - z_t) n_t + z_t h_{(t-1)}
\end{aligned}
$$

where h_t is the hidden state at time t, x_t is the input at time t, $h_{(t-1)}$ is the hidden state of the layer at time *t-1* or the initial hidden state at time *0*, and r_t, z_t, n_t are the reset, update, and new gates, respectively.

GRU

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting num_layers=2 would mean stacking two GRUs together to form a *stacked GRU*, with the second GRU taking in outputs of the first GRU and computing the final results. Default: 1
- **bias** – If False, then the layer does not use bias weights *b_ih* and *b_hh*. Default: True
- **batch_first** – If True, then the input and output tensors are provided as (batch, seq, feature). Default: False
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each GRU layer except the last layer, with dropout probability equal to dropout. Default: 0
- **bidirectional** – If True, becomes a bidirectional GRU. Default: False

GRU

- **input** of shape *(seq_len, batch, input_size)*: tensor containing the features of the input sequence.
- **h_0** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided. If the RNN is bidirectional, num_directions should be 2, else it should be 1.

GRU

- **output** of shape *(seq_len, batch, num_directions * hidden_size)*: tensor containing the output features h_t from the last layer of the GRU, for each t.
- **h_n** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the hidden state for *t* = *seq_len*

GRU

```
>>> rnn = nn.GRU(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> output, hn = rnn(input, h0)
```

RNN&RNNCell

- RNN
- Full Sequence
- The former is more complete and easier to package
- RNN layer is implemented by calling RNNCell

- RNN Cell
- One time step
- the latter is more flexible

Dropout layers

- Dropout
- Dropout2d
- Dropout3d
- AlphaDropout

Normalization layers

- BatchNorm1d
- BatchNorm2d
- BatchNorm3d
- InstanceNorm1d
- InstanceNorm2d
- InstanceNorm3d
- LocalResponseNorm

Non-linear Activations

- ReLU
- ReLU6
- ELU
- SELU
- PReLU
- LeakyReLU
- Threshold
- Hardtanh
- Sigmoid
- Tanh
- LogSigmoid
- Softplus
- Softshrink
- Softsign
- Tanhshrink
- Softmin
- Softmax
- Softmax2d
- LogSoftmax

Loss functions

- L1Loss
- MSELoss
- CrossEntropyLoss
- NLLLoss
- PoissonNLLLoss
- KLDivLoss
- BCELoss
- BCEWithLogitsLoss
- MarginRankingLoss
- HingeEmbeddingLoss
- MultiLabelMarginLoss
- SmoothL1Loss
- SoftMarginLoss
- MultiLabelSoftMarginLoss
- CosineEmbeddingLoss
- MultiMarginLoss
- TripletMarginLoss

Padding Layers

- ReflectionPad2d
- ReplicationPad2d
- ReplicationPad3d
- ZeroPad2d
- ConstantPad2d

Sparse layers

- Embedding
- EmbeddingBag

Distance functions

- CosineSimilarity
- PairwiseDistance

Vision layers

- PixelShuffle
- Upsample
- UpsamplingNearest2d
- UpsamplingBilinear2d

Utilities

- clip_grad_norm
- weight_norm
- remove_weight_norm
- PackedSequence
- pack_padded_sequence
- pad_packed_sequence
- pad_sequence
- pack_sequence

Model work choices

- Choice of last layer
- For a regression, a linear layer generating a scalar value; for a vector regression, a linear layer generating more than one scalar value;
- Choice of loss function
- Optimization

Baseline model

Problem type	Activation function	Loss function
Binary classification	Sigmoid	CrossEntropyLoss
Multi-class classification	Softmax	CrossEntropyLoss
Multi-label classification	Sigmoid	CrossEntropyLoss
Regression	None	MSE
Vector regression	None	MSE