Torchlens automatically logs and visualizes the whole forward pass for an arbitrary model:

Pass in a PyTorch model and input:

```python
import torch
from torch import nn
import torchlens as tl

class DemoNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Linear(in_features=5, out_features=5)
        self.relu_module = nn.ReLU()
        self.register_buffer('example_buffer', torch.ones(5))

    def forward(self, x):
        x = x + 1
        x = self.relu_module(x)
        x = self.fc(x)
        x = nn.functional.relu(x)
        x = x + torch.randn(x.shape)
        y = x ** 2
        x = x / y
        x = x + self.example_buffer
        return x


demo_network = DemoNetwork()
x = torch.rand(5, 5)
model_history = tl.get_model_activations(demo_network, x, vis_opt='unrolled')
```

get_model_activations returns a data structure logging the full forward pass:

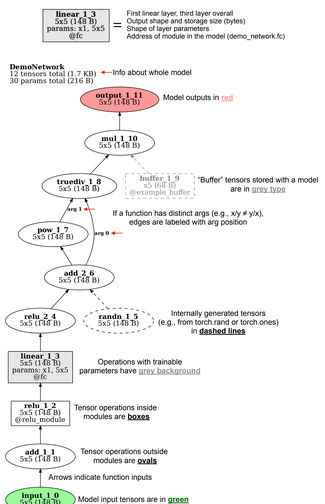
```
print(model_history
Log of DemoNetwork forward pass:
    Model structure: purely feedforward, with branching; 2 total modules.
    12 tensors (1.7 KB) computed in forward pass; 12 tensors (1.7 KB) saved.
    2 parameter operations (30 params total; 216 B).
    Random seed: 58234853@
    Time elapsed: 0.011s
    Module Hierarchy:
        relu_module
        fc
    Layers:
        0: input_1_0
        1: add_1_1
        2: relu_1_2
        3: linear_1_3
        4: relu_2_4
        5: add_2_6
        6: randn_1_5
        7: linear_1_3:2
        8: relu_2_4:2
        9: add_2_6:2
        10: randn_1_5:2
```

Can index it to fetch layer information:

```
print(model_history['linear_1_3'])
Layer linear_1_3, operation 4/12:
    Output tensor: shape=(5, 5), dype=torch.float32, size=148 B
        tensor([[-0.7488, -0.8825,  0.2709,  1.3066, -1.7198],
                [-0.9291, -0.9717,  0.1531,  1.2980, -1.4515],
                [-0.6025, -0.8066, -0.1089,  1.2378, -1.7563],
                [-0.6795, -0.6625, -0.0675,  0.9598, -1.6292],
                [-0.5661, -0.6767, -0.1114,  0.9888, -1.7476]])
    Params: Computed from params with shape (5,), (5, 5); 30 params total (216 B)
    Parent Layers: relu_1_2
    Child Layers: relu_2_4
    Function: linear (gradfunc=AddmmBackward0)
    Computed inside module: fc
    Time elapsed:  1.118E-04s
    Output of modules: fc
    Output of bottom-level module: fc
    Lookup keys: -9, 3, fc, fc:1, linear_1_3, linear_1_3:1
```

Can extract saved tensor via tensor_contents field:

```
print(model_history['linear_1_3'].tensor_contents)
tensor([[-0.7488, -0.8825,  0.2709,  1.3066, -1.7198],
        [-0.9291, -0.9717,  0.1531,  1.2980, -1.4515],
        [-0.6025, -0.8066, -0.1089,  1.2378, -1.7563],
        [-0.6795, -0.6625, -0.0675,  0.9598, -1.6292],
        [-0.5661, -0.6767, -0.1114,  0.9888, -1.7476]])
```



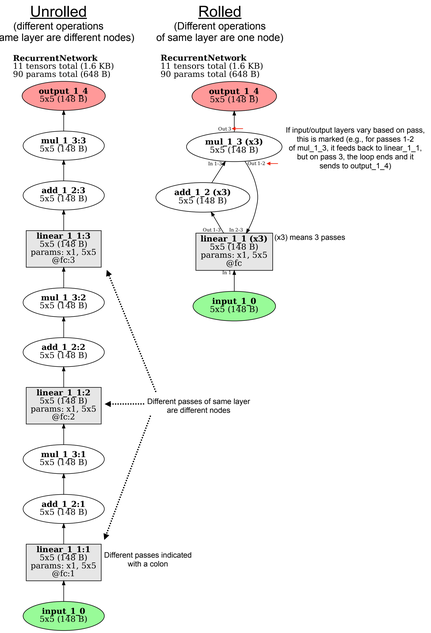
Torchlens logs all passes of recurrent networks and can visualize in unrolled or rolled format:

Set up recurrent network:

```python
class RecurrentNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Linear(in_features=5, out_features=5)

    def forward(self, x):
        for p in range(3):
            x = self.fc(x)
            x = x + 1
            x = x * 2
        return x


demo_network = RecurrentNetwork()
x = torch.rand(5, 5)
model_history = tl.get_model_activations(demo_network, x, vis_opt='unrolled')
```

Pull out passes of a layer with a colon (e.g., linear_1_1:2 for the second pass):

```
print(model_history)
print(model_history['linear_1_1:2'])

Log of RecurrentNetwork forward pass:
    Model structure: recurrent (at most 3 loops), without branching; 1 total modules.
    11 tensors (1.6 KB) computed in forward pass; 11 tensors (1.6 KB) saved,
    6 parameter operations (90 params total; 648 B).
    Random seed: 2703757312
    Time elapsed: 0.017s
    Module Hierarchy:
        fc
    Layers:
        0: input_1_0
        1: linear_1_1:1  (1/3 passes)
        2: add_1_2:1  (1/3 passes)
        3: mul_1_3:1  (1/3 passes)
        4: linear_1_1:2  (2/3 passes)
        5: add_1_2:2  (2/3 passes)
        6: mul_1_3:2  (2/3 passes)
        7: linear_1_1:3  (3/3 passes)
        8: add_1_2:3  (3/3 passes)
        9: mul_1_3:3  (3/3 passes)
        10: output_1_4


Layer linear_1_1 (pass 2/3), operation 5/11:
    Output tensor: shape=(5, 5), dype=torch.float32, size=148 B
        tensor([[-0.5046, -0.5935,  1.6663,  1.3821,  1.0772],
                [-0.3811, -0.4549,  1.3965,  0.8859,  1.3137],
                [-0.4680, -0.5890,  1.4675,  1.0654,  1.0059],
                [-0.5087, -0.5647,  1.8539,  1.3671,  1.4766],
                [-0.5659, -0.5722,  1.8987,  1.6058,  1.2814]])
    Params: Computed from params with shape (5,), (5, 5); 30 params total (216 B)
    Parent Layers: mul_1_3:1
    Child Layers: add_1_2:2
    Function: linear (gradfunc=AddmmBackward0)
    Computed inside module: fc
    Time elapsed:  8.297E-05s
    Output of modules: fc
    Output of bottom-level module: fc
    Lookup keys: -7, 4, fc:2, linear_1_1:2
```



Torchlens also indicates different aspects of network organization, like nested submodules, branching, or conditional (if-then) branches:

Set up model with nested submodules:

```python
class Submodule2(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        x = x + 2
        w = torch.cos(x)
        y = torch.sin(x)
        z = torch.tan(x)
        x = w + y + z
        return x


class Submodule1(nn.Module):
    def __init__(self):
        super().__init__()
        self.submodule2 = Submodule2()

    def forward(self, x):
        x = x + 3
        x = self.submodule2(x)
        x = x * 4
        return x


class NetworkOrganizationDemo(nn.Module):
    def __init__(self):
        super().__init__()
        self.submodule1 = Submodule1()

    def forward(self, x):
        if torch.sum(x) > 0:
            x = x * 2
        else:
            x = x + 1
        w = 1 / x
        y = torch.log(x)
        z = x ** 2
        x = w + y + z
        x = self.submodule1(x)
        x = x + 1
        x = x * 2
        return x


demo_network = NetworkOrganizationDemo()
x = torch.rand(5, 5)
model_history = tl.get_model_activations(demo_network, x, vis_opt='unrolled')
```

History log indicates module nesting:

```
print(model_history)

Log of NetworkOrganizationDemo forward pass:
    Model structure: purely feedforward, with branching; 2 total modules.
    21 tensors (2.8 KB) computed in forward pass; 21 tensors (2.8 KB) saved.
    0 parameter operations (0 params total; 0 B).
    Random seed: 425335997
    Time elapsed: 0.025s
    Module Hierarchy:
        submodule1:
            submodule1.submodule2
    Layers:
        0: input_1_0
        1: sum_1_1
        2: mul_1_2
        3: gt_1_3
        4: reciprocal_1_4
        5: log_1_5
        6: pow_1_6
        7: mul_2_7
        8: add_1_8
```

Can also index layers by the module for which they are the output:

```
print(model_history['submodule1.submodule2'])

Layer add_6_16, operation 17/21:
    Output tensor: shape=(5, 5), dype=torch.float32, size=148 B
        tensor([[  2.7773,   2.6319,   2.5537,  -0.8818,  -1.5334],
                [  1.6304,  -0.9774,  23.2859,  -1.5232,  -0.6535],
                [  1.6395,  -0.9999,   1.9072,   4.9053,   1.4397],
                [ -0.9523,  -1.1453,  -1.3403,  12.9639,  -0.9456],
                [  2.3906,  -0.9434,  -2.5448,   2.3512, -31.2896]])
    Params: no params used
    Parent Layers: tan_1_15, add_5_14
    Child Layers: mul_3_17
    Function: __add__ (gradfunc=AddBackward0)
    Computed inside module: submodule1.submodule2
    Time elapsed:  6.938E-05s
    Output of modules: submodule1.submodule2
    Lookup keys: -5, 16, add_6_16, add_6_16:1, submodule1.submodule2, submodule1.submodule2:1
```