Open in Colab

Walkthrough

The API for a transparent science on black-box AI

In this era of large-scale deep learning, the most interesting AI models are massive black boxes that are hard to run. Ordinary commercial inference service APIs let us interact with huge models, but they do not let us access model internals. The `nnsight` library is different: it provides full access to all the neural network internals. When used together with a remote service like the National Deep Inference Fabric (NDIF), it makes possible to run complex experiments on huge open models easily, with fully transparent access. Our team wants to enable entire labs and independent researchers alike, as we believe a large, passionate, and collaborative community will produce the next big insights on a profoundly important field.

1 First, let’s start small

Setup

Install nnsight:

```
pip install nnsight
```

Tracing Context

To demonstrate the core functionality and syntax of nnsight, we’ll define and use a tiny two layer neural network. Our little model here is composed of two submodules – linear layers ‘layer1’ and ‘layer2’. We specify the sizes of each of these modules and create some complementary example input.

[1]:

```
from collections import OrderedDict
import torch

input_size = 5
hidden_dims = 10
output_size = 2

net = torch.nn.Sequential(
    OrderedDict(
        [
            ("layer1", torch.nn.Linear(input_size, hidden_dims)),
            ("layer2", torch.nn.Linear(hidden_dims, output_size)),
        ]
    )
).requires_grad_(False)
```

The core object of the nnsight package is `NNsight`. This wraps around a given PyTorch model to enable investigation of its internal parameters.

[2]:
```
import nnsight
from nnsight import NNsight

tiny_model = NNsight(net)
```

Printing a PyTorch model shows a named hierarchy of modules which is very useful when accessing sub-components directly. NNsight reflect the same hierarchy and can be similarly printed.

[3]:
```
print(tiny_model)
```

```
Sequential(
  (layer1): Linear(in_features=5, out_features=10, bias=True)
  (layer2): Linear(in_features=10, out_features=2, bias=True)
)
```

Before we actually get to using the model we just created, let's talk about Python contexts. Python contexts define a scope using the `with` statement and are often used to create some object, or initiate some logic, that you later want to destroy or conclude. The most common application is opening files as in the following example:

```
with open('myfile.txt', 'r') as file:
  text = file.read()
```

Python uses the `with` keyword to enter a context-like object. This object defines logic to be run at the start of the `with` block, as well as logic to be run when exiting. When using `with` for a file, entering the context opens the file and exiting the context closes it. Being within the context means we can read from the file. Simple enough! Now we can discuss how `nnsight` uses contexts to enable intuitive access into the internals of a neural network. The main tool with `nnsight` is a context for tracing. We enter the tracing context by calling `model.trace(<input>)` on an `NNsight` model, which defines how we want to run the model. Inside the context, we will be able to customize how the neural network runs. The model is actually run upon exiting the tracing context.

[4]:

```
# random input
input = torch.rand((1, input_size))

with tiny_model.trace(input) as tracer:
    pass
```

But where's the output? To get that, we'll have to learn how to request it from within the tracing context.

Getting

Earlier, when we wrapped our little neural net with the `NNsight` class. This added a couple properties to each module in the model (including the root model itself). The two most important ones are `.input` and `.output`.

```
model.input
model.output
```

The names are self explanatory. They correspond to the inputs and outputs of their respective modules during a forward pass of the model. We can use these attributes inside the `with` block. However, it is important to understand that the model is not executed until the end of the tracing context. How can we access inputs and outputs before the model is run? The trick is deferred execution. `.input` and `.output` are Proxies for the eventual inputs and outputs of a module. In other words, when we access `model.output` what we are communicating to `nnsight` is, "When you compute the output of `model`, please grab it for me and put the value into its corresponding Proxy object. Let's try it:

[5]:
```
with tiny_model.trace(input) as tracer:

    output = tiny_model.output

print(output)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[5], line 5
      1 with tiny_model.trace(input) as tracer:
      3     output = tiny_model.output
----> 5 print(output)

File /opt/anaconda3/envs/nnsight/lib/python3.10/site-packages/nnsight/tracing/graph/
     66 def __str__(self) -> str:
     68     if not self.node.attached:
---> 70         return str(self.value)
     72     return f"{type(self).__name__} ({self.node.target.__name__})"

File /opt/anaconda3/envs/nnsight/lib/python3.10/site-packages/nnsight/tracing/graph/
     56 @property
     57 def value(self) -> Any:
     58     """Property to return the value of this proxy's node.
     59 
     60     Returns:
     61         Any: The stored value of the proxy, populated during execution of th
     62     """
---> 64     return self.node.value

File /opt/anaconda3/envs/nnsight/lib/python3.10/site-packages/nnsight/tracing/graph/
    133 """Property to return the value of this node.
    134 
    135 Returns:
   (...)
    139     ValueError: If the underlying ._value is inspect._empty (therefore never
    140 """
    142 if not self.done:
--> 143     raise ValueError("Accessing value before it's been set.")
    145 return self._value

ValueError: Accessing value before it's been set.
```

Oh no an error! "Accessing value before it's been set." Why doesn't our `output` have a `value`? Proxy objects will only have their value at the end of a context if we call `.save()` on them. This helps to reduce memory costs. Adding `.save()` fixes the error:

[67]:
```
with tiny_model.trace(input) as tracer:

    output = tiny_model.output.save()

print(output)
```

```
tensor([[-0.1301, -0.4906]])
```

Success! We now have the model output. We just completed out first intervention using `nnsight`. Each time we access a module's input or output, we create an *intervention* in the neural network's forward pass. Collectively these requests form the *intervention graph*. We call the process of executing it alongside the model's normal computation graph, *interleaving*.

▶ On Model output

Just like we saved the output of the model as a whole, we can save the output of any of its submodules. We use normal Python attribute syntax. We can discover how to access them by name by printing out the model:

[68]:

```
print(tiny_model)
```

```
Sequential(
  (layer1): Linear(in_features=5, out_features=10, bias=True)
  (layer2): Linear(in_features=10, out_features=2, bias=True)
)
```

Let's access the output of the first layer (which we've named 'layer1'):

[69]:
```
with tiny_model.trace(input) as tracer:

    l1_output = tiny_model.layer1.output.save()

print(l1_output)
```

```
tensor([[ 0.2732,  0.2355, -0.6433,  0.0475,  0.0904,  0.4407, -0.3099,  1.4903,
         -0.0748,  0.0088]])
```

Let's do the same for the input of layer2. While we're at it, let's also drop the `as tracer`, as we won't be needing the tracer object itself for a few sections:

[70]:
```
with tiny_model.trace(input):

    l2_input = tiny_model.layer2.input.save()

print(l2_input)
```

```
tensor([[ 0.2732,  0.2355, -0.6433,  0.0475,  0.0904,  0.4407, -0.3099,  1.4903,
         -0.0748,  0.0088]])
```

▶ On module inputs

Until now we were saving the output of the model and its submodules within the `Trace` context to then print it after exiting the context. We will continuing doing this in the rest of the tutorial since it's a good practice to save the computation results for later analysis. However, we can also log the outputs of the model and its submodules within the `Trace` context. This is useful for debugging and understanding the model's behavior while saving memory. Let's see how to do this:

[71]:
```
with tiny_model.trace(input) as tracer:
  tracer.log("Layer 1 - out: ", tiny_model.layer1.output)
```

```
Layer 1 - out:  tensor([[ 0.2732,  0.2355, -0.6433,  0.0475,  0.0904,  0.4407, -0.30
         -0.0748,  0.0088]])
```

Functions, Methods, and Operations

Now that we can access activations, we also want to do some post-processing on it. Let's find out which dimension of layer1's output has the highest value. We could do this by calling `torch.argmax(...)` after the tracing context or we can just leverage the fact that `nnsight` handles Pytorch functions and methods within the tracing context, by creating a Proxy request for it:

[72]:
```
with tiny_model.trace(input):

    # Note we don't need to call .save() on the output,
    # as we're only using its value within the tracing context.
    l1_output = tiny_model.layer1.output

    # We do need to save the argmax tensor however,
    # as we're using it outside the tracing context.
    l1_amax = torch.argmax(l1_output, dim=1).save()

print(l1_amax[0])
```

```
tensor(7)
```

Nice! That worked seamlessly, but hold on, how come we didn't need to call `.value[0]` on the result? In previous sections, we were just being explicit to get an understanding of Proxies and their value. In practice, however, `nnsight` knows that when outside of the tracing context we only care about the actual value, and so printing, indexing, and applying functions all immediately return and reflect the data in `.value`. So for the rest of the tutorial we won't use it. The same principles work for Pytorch methods and all operators as well:

[73]:
```
with tiny_model.trace(input):

    value = (tiny_model.layer1.output.sum() + tiny_model.layer2.output.sum()).save()

print(value)
```

```
tensor(0.9377)
```

The code block above is saying to `nnsight`, "Run the model with the given `input`. When the output of `tiny_model.layer1` is computed, take its sum. Then do the same for `tiny_model.layer2`. Now that both of those are computed, add them and make sure not to delete this value as I wish to use it outside of the tracing context."

Custom Functions

Everything within the tracing context operates on the intervention graph. Therefore, for `nnsight` to trace a function it must also be a part of the intervention graph. Out-of-the-box `nnsight` supports PyTorch functions and methods, all operators, as well the `einops` library. We don't need to do anything special to use them. But what do we do if we want to use custom functions? How do we add them to the intervention graph? Enter `nnsight.apply()`. It allows us to add new functions to the intervention graph. Let's see how it works:

[74]:

```
# Take a tensor and return the sum of its elements
def tensor_sum(tensor):
    flat = tensor.flatten()
    total = 0
    for element in flat:
        total += element.item()

    return torch.tensor(total)

with tiny_model.trace(input) as tracer:

    # Specify the function name and its arguments (in a comma-separated form) to add
    custom_sum = nnsight.apply(tensor_sum, tiny_model.layer1.output).save()
    sum = tiny_model.layer1.output.sum()
    sum.save()


print(custom_sum, sum)
```

```
tensor(1.5584) tensor(1.5584)
```

`nnsight.apply()` executes the function it wraps and returns its output as a Proxy object. We can then use this Proxy object as we would any other. The applications of `nnsight.apply` are wide: it can be used to wrap any custom function or functions from libraries that `nnsight` does not support out-of-the-box.

Setting

Getting and analyzing the activations from various points in a model can be really insightful, and a number of ML techniques do exactly that. However, often we not only want to view the computation of a model, but also to influence it. To demonstrate the effect of editing the flow of information through the model, let's set the first dimension of the first layer's output to 0. `NNsight` makes this really easy using the '=' operator:

[75]:
```
with tiny_model.trace(input):

    # Save the output before the edit to compare.
    # Notice we apply .clone() before saving as the setting operation is in-place.
    l1_output_before = tiny_model.layer1.output.clone().save()

    # Access the 0th index of the hidden state dimension and set it to 0.
    tiny_model.layer1.output[:, 0] = 0

    # Save the output after to see our edit.
    l1_output_after = tiny_model.layer1.output.save()

print("Before:", l1_output_before)
print("After:", l1_output_after)
```

```
Before: tensor([[ 0.2732,  0.2355, -0.6433,  0.0475,  0.0904,  0.4407, -0.3099,  1.4
         -0.0748,  0.0088]])
After: tensor([[ 0.0000,  0.2355, -0.6433,  0.0475,  0.0904,  0.4407, -0.3099,  1.49
         -0.0748,  0.0088]])
```

Seems our change was reflected. Now let's do the same for the last dimension:

[76]:

```
with tiny_model.trace(input):

    # Save the output before the edit to compare.
    # Notice we apply .clone() before saving as the setting operation is in-place.
    l1_output_before = tiny_model.layer1.output.clone().save()

    # Access the last index of the hidden state dimension and set it to 0.
    tiny_model.layer1.output[:, hidden_dims] = 0

    # Save the output after to see our edit.
    l1_output_after = tiny_model.layer1.output.save()

print("Before:", l1_output_before)
print("After:", l1_output_after)
```

```
Traceback (most recent call last):
  File "/Users/emmabortz/Documents/Projects/nnsight/src/nnsight/tracing/graph/node.p
    output = self.target(*args, **kwargs)
IndexError: index 10 is out of bounds for dimension 1 with size 10

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/var/folders/rx/0nl_h2cd54q90chs9hf0n5qr0000gn/T/ipykernel_80599/3404137504.
    tiny_model.layer1.output[:, hidden_dims] = 0

NNsightError: index 10 is out of bounds for dimension 1 with size 10
```

Oh no, we are getting an error! Ah of course, we needed to index at `hidden_dims - 1` not `hidden_dims`. If you've been using `nnsight`, you are probably familiar with error messages that can be quite difficult to troubleshoot. In `nnsight 0.4` we've now improved error messaging to be descriptive and line-specific, as you should see in the above example!

▶ Old NNsight error messaging

The error messaging feature can be toggled using `nnsight.CONFIG.APP.DEBUG` which defaults to true.

▶ Toggle Error Messaging

Now that we know more about NNsight's error messaging, let's try our setting operation again with the correct indexing and view the shape of the output before leaving the tracing context:

[77]:
```
with tiny_model.trace(input):

    # Save the output before the edit to compare.
    # Notice we apply .clone() before saving as the setting operation is in-place.
    l1_output_before = tiny_model.layer1.output.clone().save()

    print(f"Layer 1 output shape: {tiny_model.layer1.output.shape}")

    # Access the last index of the hidden state dimension and set it to 0.
    tiny_model.layer1.output[:, hidden_dims - 1] = 0

    # Save the output after to see our edit.
    l1_output_after = tiny_model.layer1.output.save()

print("Before:", l1_output_before)
print("After:", l1_output_after)
```

```
Layer 1 output shape: InterventionProxy (fetch_attr)
Before: tensor([[ 0.2732,  0.2355, -0.6433,  0.0475,  0.0904,  0.4407, -0.3099,  1.4
         -0.0748,  0.0088]])
After: tensor([[ 0.2732,  0.2355, -0.6433,  0.0475,  0.0904,  0.4407, -0.3099,  1.49
         -0.0748,  0.0000]])
```

Scan and Validate

Error codes are helpful, but sometimes you may want to quickly troubleshoot your code without actually running it. Enter “Scanning” and “Validating”! We can enable this features by setting the `scan=True` and `validate=True` flag in the `trace` method. “Scanning” runs “fake” inputs throught the model to collect information like shapes and types (i.e., scanning will populate all called `.inputs` and `.outputs`). “Validating” attempts to execute the intervention proxies with “fake” inputs to check if they work (i.e., executes all interventions in your code with fake tensors). “Validating” is dependent on “Scanning” to work correctly, so we need to run the scan of the model at least once to debug with validate. Let’s try it out on our example above.

[78]:
```python
# turn on scan and validate
with tiny_model.trace(input, scan=True, validate=True):

    l1_output_before = tiny_model.layer1.output.clone().save()

    # the error is happening here
    tiny_model.layer1.output[:, hidden_dims] = 0

    l1_output_after = tiny_model.layer1.output.save()

print("Before:", l1_output_before)
print("After:", l1_output_after)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[78], line 2
      1 # turn on scan and validate
----> 2 with tiny_model.trace(input, scan=True, validate=True):
      4     l1_output_before = tiny_model.layer1.output.clone().save()
      6     # the error is happening here

File ~/Documents/Projects/nnsight/src/nnsight/intervention/contexts/interleaving.py:
     92     self.invoker.__exit__(None, None, None)
     94 self._model._envoy._reset()
---> 96 super().__exit__(exc_type, exc_val, exc_tb)

File ~/Documents/Projects/nnsight/src/nnsight/tracing/contexts/tracer.py:25, in Trac
     21 from .globals import GlobalTracingContext
     23 GlobalTracingContext.try_deregister(self)
---> 25 return super().__exit__(exc_type, exc_val, exc_tb)

File ~/Documents/Projects/nnsight/src/nnsight/tracing/contexts/base.py:72, in Contex
     69 graph = self.graph.stack.pop()
     71 if isinstance(exc_val, BaseException):
---> 72     raise exc_val
     74 self.add(graph.stack[-1], graph, *self.args, **self.kwargs)
     76 if self.backend is not None:

Cell In[78], line 7
      4     l1_output_before = tiny_model.layer1.output.clone().save()
      6     # the error is happening here
----> 7     tiny_model.layer1.output[:, hidden_dims] = 0
      9     l1_output_after = tiny_model.layer1.output.save()
     11 print("Before:", l1_output_before)

File ~/Documents/Projects/nnsight/src/nnsight/tracing/graph/proxy.py:126, in Proxy._
    125 def __setitem__(self, key: Union[Self, Any], value: Union[Self, Any]) -> Non
--> 126     self.node.create(
    127         operator.setitem,
    128         self.node,
    129         key,
    130         value,
    131     )

File ~/Documents/Projects/nnsight/src/nnsight/tracing/graph/node.py:250, in Node.cre
    247     return value
    249 # Otherwise just create the Node on the Graph like normal.
--> 250 return self.graph.create(
    251     *args,
    252     **kwargs,
    253 )

File ~/Documents/Projects/nnsight/src/nnsight/tracing/graph/graph.py:131, in Graph.c
    128 # Redirection.
    129 graph = self.stack[-1] if redirect and self.stack else self
--> 131 return self.proxy_class(self.node_class(target, *args, graph=graph, **kwargs

File ~/Documents/Projects/nnsight/src/nnsight/intervention/graph/node.py:118, in Val
    111 super().__init__(*args, **kwargs)
    113 if (
    114     self.attached
    115     and self.fake_value is inspect._empty
    116     and not Protocol.is_protocol(self.target)
    117 ):
--> 118     self.fake_value = validate(self.target, *self.args, **self.kwargs)

File ~/Documents/Projects/nnsight/src/nnsight/intervention/graph/node.py:147, in val
    141 with FakeTensorMode(
    142     allow_non_fake_inputs=True,
    143     shape_env=ShapeEnv(assume_static_by_default=True),
    144 ) as fake_mode:
    145     with FakeCopyMode(fake_mode):
```

```
--> 147         with GlobalTracingContext.exit_global_tracing_context():
    149             if backwards_check(target, *args):
    150                 return None

File ~/Documents/Projects/nnsight/src/nnsight/tracing/contexts/globals.py:100, in Gl
     96 GlobalTracingContext.PATCHER.__enter__()
     98 if isinstance(exc_val, BaseException):
--> 100     raise exc_val

File ~/Documents/Projects/nnsight/src/nnsight/intervention/graph/node.py:156, in val
    150     return None
    152 args, kwargs = InterventionNode.prepare_inputs(
    153     (args, kwargs), fake=True
    154 )
--> 156 return target(
    157     *args,
    158     **kwargs,
    159 )

File /opt/anaconda3/envs/nnsight/lib/python3.10/site-packages/torch/_subclasses/fake
   2348 else:
   2349     with torch._C.DisableTorchFunctionSubclass():
-> 2350         return func(*args, **kwargs)

IndexError: index 10 is out of bounds for dimension 1 with size 10
```

The operations are never executed using tensors with real values so it doesn't incur any memory costs. Then, when creating proxy requests like the setting one above, `nnsight` also attempts to execute the request on the "fake" values we recorded. Hence, it lets us know if our request is feasible before even running the model. Here is a more detailed example of scan and validate in action!

▶ A word of caution

We can also use the `.scan()` method to get the shape of a module without having to fully run the model. If scan is enabled, our input is run though the model under its own "fake" context. This means the input makes its way through all of the model operations, allowing `nnsight` to record the shapes and data types of module inputs and outputs!

[79]:
```
with tiny_model.scan(input):

    dim = tiny_model.layer1.output.shape[-1]

print(dim)
```

```
10
```

We can also just replace proxy inputs and outputs with tensors of the same shape and type. Let's use the shape information we have at our disposal to add noise to the output, and replace it with this new noised tensor:

Gradients

`NNsight` also lets us apply backpropagation and access gradients with respect to a loss. Like `.input` and `.output` on modules, `nnsight` exposes `.grad` on Proxies themselves (assuming they are proxies of tensors):

[80]:
```
with tiny_model.trace(input):

    # We need to explicitly have the tensor require grad
    # as the model we defined earlier turned off requiring grad.
    tiny_model.layer1.output.requires_grad = True

    # We call .grad on a tensor Proxy to communicate we want to store its gradient.
    # We need to call .save() since .grad is its own Proxy.
    layer1_output_grad = tiny_model.layer1.output.grad.save()
    layer2_output_grad = tiny_model.layer2.output.grad.save()

    # Need a loss to propagate through the later modules in order to have a grad.
    loss = tiny_model.output.sum()
    loss.backward()

print("Layer 1 output gradient:", layer1_output_grad)
print("Layer 2 output gradient:", layer2_output_grad)
```

```
Layer 1 output gradient: tensor([[-0.1296,  0.4562, -0.1182, -0.3536,  0.0703,  0.14
          0.0196, -0.0452]])
Layer 2 output gradient: tensor([[1., 1.]])
```

All of the features we learned previously, also apply to `.grad`. In other words, we can apply operations to and edit the gradients. Let's zero the grad of `layer1` and double the grad of `layer2`.

[81]:
```
with tiny_model.trace(input):

    # We need to explicitly have the tensor require grad
    # as the model we defined earlier turned off requiring grad.
    tiny_model.layer1.output.requires_grad = True

    tiny_model.layer1.output.grad[:] = 0
    tiny_model.layer2.output.grad = tiny_model.layer2.output.grad * 2

    layer1_output_grad = tiny_model.layer1.output.grad.save()
    layer2_output_grad = tiny_model.layer2.output.grad.save()

    # Need a loss to propagate through the later modules in order to have a grad.
    loss = tiny_model.output.sum()
    loss.backward()

print("Layer 1 output gradient:", layer1_output_grad)
print("Layer 2 output gradient:", layer2_output_grad)
```

```
Layer 1 output gradient: tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
Layer 2 output gradient: tensor([[2., 2.]])
```

Early Stopping

If we are only interested in a model's intermediate computations, we can halt a forward pass run at any module level, reducing runtime and conserving compute resources. One examples where this could be particularly useful would if we are working with SAEs - we can train an SAE on one layer and then stop the execution.

[82]:
```
with tiny_model.trace(input):
   l1_out = tiny_model.layer1.output.save()
   tiny_model.layer1.output.stop()

# get the output of the first layer and stop tracing
print("L1 - Output: ", l1_out)
```

```
 L1 - Output:  tensor([[ 0.2732,  0.2355, -0.6433,  0.0475,  0.0904,  0.4407, -0.3099
         -0.0748,  0.0088]])
```

Interventions within the tracing context do not necessarily execute in the order they are defined. Instead, their execution is tied to the module they are associated with. As a result, if the forward pass is terminated early any interventions linked to modules beyond that point will be skipped, even if they were defined earlier in the context. In the example below, the output of layer 2 **cannot** be accessed since the model's execution was stopped at layer 1.

[83]:
```
with tiny_model.trace(input):
   l2_out = tiny_model.layer2.output.save()
   tiny_model.layer1.output.stop()

print("L2 - Output: ", l2_out)
```

```
 L2 - Output:


 ---------------------------------------------------------------------------
 ValueError                                Traceback (most recent call last)
 Cell In[83], line 5
       2    l2_out = tiny_model.layer2.output.save()
       3    tiny_model.layer1.output.stop()
 ----> 5 print("L2 - Output: ", l2_out)

 File ~/Documents/Projects/nnsight/src/nnsight/tracing/graph/proxy.py:70, in Proxy.__
      66 def __str__(self) -> str:
      68     if not self.node.attached:
 ---> 70         return str(self.value)
      72     return f"{type(self).__name__} ({self.node.target.__name__})"

 File ~/Documents/Projects/nnsight/src/nnsight/tracing/graph/proxy.py:64, in Proxy.va
      56 @property
      57 def value(self) -> Any:
      58     """Property to return the value of this proxy's node.
      59
      60     Returns:
      61         Any: The stored value of the proxy, populated during execution of th
      62     """
 ---> 64     return self.node.value

 File ~/Documents/Projects/nnsight/src/nnsight/tracing/graph/node.py:143, in Node.val
     133 """Property to return the value of this node.
     134
     135 Returns:
    (...)
     139     ValueError: If the underlying ._value is inspect._empty (therefore never
     140 """
     142 if not self.done:
 --> 143     raise ValueError("Accessing value before it's been set.")
     145 return self._value

 ValueError: Accessing value before it's been set.
```

Conditional Interventions

Interventions can also be made conditional. Inside the tracing context we can specify a new - conditional - context. This context will only execute the interventions within it if the condition is met.

[84]:
```
with tiny_model.trace(input) as tracer:

  rand_int = torch.randint(low=-10, high=10, size=(1,))

  with tracer.cond(rand_int % 2 == 0):
    tracer.log("Random Integer ", rand_int, " is Even")

  with tracer.cond(rand_int % 2 == 1):
    tracer.log("Random Integer ", rand_int, " is Odd")
```

```
Random Integer  tensor([-5])  is Odd
```

Conditional contexts can also be nested, if we want our interventions to depend on more than one condition at a time.

[21]:
```
with tiny_model.trace(input) as tracer:

  non_rand_int = 8

  with tracer.cond(non_rand_int > 0):
    with tracer.cond(non_rand_int % 2 == 0):
      tracer.log("Rand Int ", non_rand_int, " is Positive and Even")
```

```
Rand Int  8  is Positive and Even
```

With `nnsight 0.4` we can now also use Python `if` statements within the tracing context to create a conditional context! *Note: Colab behaves a little strangely with this feature the first time you run it - expect some lagging and warnings*

[85]:
```
with tiny_model.trace(input) as tracer:

  rand_int = torch.randint(low=-10, high=10, size=(1,))

  # Since this if statement is inside the tracing context the if will
  # create a conditional context and will only execute the intervention
  # if this condition is met
  if rand_int % 2 == 0:
    tracer.log("Random Integer ", rand_int, " is Even")

  if rand_int % 2 == 1:
    tracer.log("Random Integer ", rand_int, " is Odd")
```

```
Random Integer  tensor([2])  is Even
```

`elif` statements should also work as `if` statements within the tracing context:

[86]:

```
with tiny_model.trace(input) as tracer:

  rand_int = torch.randint(low=-10, high=10, size=(1,))

  # Since this if statement is inside the tracing context the if will
  # create a conditional context and will only execute the intervention
  # if this condition is met
  if rand_int % 2 == 0:
    tracer.log("Random Integer ", rand_int, " is Even")
  elif rand_int % 2 == 1:
    tracer.log("Random Integer ", rand_int, " is Odd")
```

```
Random Integer  tensor([-3])  is Odd
```

Iterative Interventions

With the iterator context, you can now run an intervention loop at scale. It iteratively executes and updates a single intervention graph. Use a `.session()` to define the Iterator context and pass in a sequence of items that you want to loop over at each iteration

[87]:

```
with tiny_model.session() as session:

  li = nnsight.list() # an NNsight built-in list object
  [li.append([num]) for num in range(0, 3)] # adding [0], [1], [2] to the list
  li2 = nnsight.list().save()

  # You can create nested Iterator contexts
  with session.iter(li) as item:
    with session.iter(item) as item_2:
      li2.append(item_2)

print("\nList: ", li2)
```

```
List:  [0, 1, 2]
```

With `nnsight 0.4` we can now also use Python `for` loops within a tracer context at scale. *NOTE: inline for loops (i.e., ``[x for x in <Proxy object>``]) are not currently supported.*

[88]:

```
# New: Using Python for loops for iterative interventions
with tiny_model.session() as session:

    li = nnsight.list()
    [li.append([num]) for num in range(0, 3)]
    li2 = nnsight.list().save()

    # Using regular for loops
    for item in li:
        for item_2 in item: # for loops can be nested!
            li2.append(item_2)

print("\nList: ", li2)
```

```
List:  [0, 1, 2]
```

2 Bigger

Now that we have the basics of `nnsight` under our belt, we can scale our model up and combine the techniques we've learned into more interesting experiments. The `NNsight` class is very bare bones. It wraps a pre-defined model and does no pre-processing on the inputs we enter. It's designed to be extended with more complex and powerful types of models, and we're excited to see what can be done to leverage its features! However, if you'd like to load a Language Model from HuggingFace with its tokenizer, the `LanguageModel` subclass greatly simplifies this process.

LanguageModel

`LanguageModel` is a subclass of `NNsight`. While we could define and create a model to pass in directly, `LanguageModel` includes special support for Huggingface language models, including automatically loading models from a Huggingface ID, and loading the model together with the appropriate tokenizer. Here is how we can use `LanguageModel` to load `GPT-2`:

[6]:
```python
from nnsight import LanguageModel

llm = LanguageModel("openai-community/gpt2", device_map="auto")

print(llm)
```

```
GPT2LMHeadModel(
  (transformer): GPT2Model(
    (wte): Embedding(50257, 768)
    (wpe): Embedding(1024, 768)
    (drop): Dropout(p=0.1, inplace=False)
    (h): ModuleList(
      (0-11): 12 x GPT2Block(
        (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (attn): GPT2SdpaAttention(
          (c_attn): Conv1D()
          (c_proj): Conv1D()
          (attn_dropout): Dropout(p=0.1, inplace=False)
          (resid_dropout): Dropout(p=0.1, inplace=False)
        )
        (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (mlp): GPT2MLP(
          (c_fc): Conv1D()
          (c_proj): Conv1D()
          (act): NewGELUActivation()
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
    )
    (ln_f): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  )
  (lm_head): Linear(in_features=768, out_features=50257, bias=False)
  (generator): Generator(
    (streamer): Streamer()
  )
)
```

▶ On Model Initialization

Let's now apply some of the features that we used on the small model to `GPT-2`. Unlike `NNsight`, `LanguageModel` does define logic to pre-process inputs upon entering the tracing context. This makes interacting with the model simpler (i.e., you can send prompts to the model without having to directly access the tokenizer). In the following example, we ablate the value coming from the last layer's MLP module and decode the logits to see what token the model predicts without influence from that particular module:

[90]:
```python
with llm.trace("The Eiffel Tower is in the city of"):

    # Access the last layer using h[-1] as it's a ModuleList
    # Access the first index of .output as that's where the hidden states are.
    llm.transformer.h[-1].mlp.output[0][:] = 0

    # Logits come out of model.lm_head and we apply argmax to get the predicted token
    token_ids = llm.lm_head.output.argmax(dim=-1).save()

print("\nToken IDs:", token_ids)

# Apply the tokenizer to decode the ids into words after the tracing context.
print("Prediction:", llm.tokenizer.decode(token_ids[0][-1]))
```

```
Token IDs: tensor([[ 262,   12,  417, 8765,   11,  257,  262, 3504,  338, 3576]],
       device='mps:0')
Prediction:  London
```

We just ran a little intervention on a much more complex model with many more parameters! However, we're missing an important piece of information: what the prediction would have looked like without our ablation. We could just run two tracing contexts and compare the outputs. However, this would require two forward passes through the model. `NNsight` can do better than that with batching.

Batching

Batching is a way to process multiple inputs in one forward pass. To better understand how batching works, we're going to bring back the `Tracer` object that we dropped before. When we call `.trace(...)`, it's actually creating two different contexts behind the scenes. The first one is the tracing context that we've discussed previously, and the second one is the invoker context. The invoker context defines the values of the `.input` and `.output` Proxies. If we call `.trace(...)` with some input, the input is passed on to the invoker. As there is only one input, only one invoker context is created. If we call `.trace()` without an input, then we can call `tracer.invoke(input1)` to manually create the invoker context with an input, `input1`. We can also repeatedly call `tracer.invoke(...)` to create the invoker context for additional inputs. Every subsequent time we call `.invoke(...)`, interventions within its context will only refer to the input in that particular invoke statement. When exiting the tracing context, the inputs from all of the invokers will be batched together, and they will be executed in one forward pass! To test this out, let's do the same ablation experiment, but also add a 'control' output for comparison:

▶ More on the invoker context

[91]:
```python
with llm.trace() as tracer:

    with tracer.invoke("The Eiffel Tower is in the city of"):

        # Ablate the last MLP for only this batch.
        llm.transformer.h[-1].mlp.output[0][:] = 0

        # Get the output for only the intervened on batch.
        token_ids_intervention = llm.lm_head.output.argmax(dim=-1).save()

    with tracer.invoke("The Eiffel Tower is in the city of"):

        # Get the output for only the original batch.
        token_ids_original = llm.lm_head.output.argmax(dim=-1).save()


print("Original token IDs:", token_ids_original)
print("Modified token IDs:", token_ids_intervention)

print("Original prediction:", llm.tokenizer.decode(token_ids_original[0][-1]))
print("Modified prediction:", llm.tokenizer.decode(token_ids_intervention[0][-1]))
```

```
Original token IDs: tensor([[ 198,   12,  417, 8765,  318,  257,  262, 3504, 7372, 6
       device='mps:0')
Modified token IDs: tensor([[ 262,   12,  417, 8765,   11,  257,  262, 3504,  338, 3
       device='mps:0')
Original prediction:  Paris
Modified prediction:  London
```

Based on our control results, our ablation did end up affecting what the model predicted. That's pretty neat! Another cool thing with multiple invokes is that Proxies can interact between them. Here, we transfer the token embeddings from a real prompt into another placeholder prompt. Therefore the latter prompt produces the output of the former prompt:

[92]:
```python
with llm.trace() as tracer:

    with tracer.invoke("The Eiffel Tower is in the city of"):
        embeddings = llm.transformer.wte.output

    with tracer.invoke("_ _ _ _ _ _ _ _ _ _"):
        llm.transformer.wte.output = embeddings
        token_ids_intervention = llm.lm_head.output.argmax(dim=-1).save()

    with tracer.invoke("_ _ _ _ _ _ _ _ _ _"):
      token_ids_original = llm.lm_head.output.argmax(dim=-1).save()

print("original prediction shape", token_ids_original[0][-1].shape)
print("Original prediction:", llm.tokenizer.decode(token_ids_original[0][-1]))

print("modified prediction shape", token_ids_intervention[0][-1].shape)
print("Modified prediction:", llm.tokenizer.decode(token_ids_intervention[0][-1]))
```

```
original prediction shape torch.Size([])
Original prediction:  _
modified prediction shape torch.Size([])
Modified prediction:  Paris
```

Multiple Token Generation

.next()

Some HuggingFace models define methods to generate multiple outputs at a time. `LanguageModel` wraps that functionality to provide the same tracing features by using `.generate(...)` instead of `.trace(...)`. This calls the underlying model's `.generate` method. It passes the output through a `.generator` module that we've added onto the model, allowing us to get the generate output at `.generator.output`. In a case like this, the underlying model is called more than once; the modules of said model produce more than one output. Which iteration should a given `module.output` refer to? That's where `Module.next()` comes in! Each module has a call index associated with it and `.next()` simply increments that attribute. At the time of execution, data is injected into the intervention graph only at the iteration that matches the call index.

[93]:
```
with llm.generate('The Eiffel Tower is in the city of', max_new_tokens=3) as tracer:

    hidden_states1 = llm.transformer.h[-1].output[0].save()

    # use module.next() to access the next intervention
    hidden_states2 = llm.transformer.h[-1].next().output[0].save()

    # saving the output allows you to save the hidden state across the initial prompt
    out = llm.generator.output.save()

print(hidden_states1.shape)
print(hidden_states2.shape)
print(out)
```

```
torch.Size([1, 10, 768])
torch.Size([1, 1, 768])
tensor([[ 464,   412,   733,   417, 8765,   318,   287,   262, 1748,   286, 6342,    11,
           290]], device='mps:0')
```

using .all()

With `nnsight 0.4` you can now use `.all()` to recursively apply interventions to a model. Calling `.all()` on a module within a model will recursively apply its `.input` and `.output` across all iterations. Previously, we'd need to loop across each new generated token, saving the intervention for every generated token and calling `.next()` to move forward.

[94]:

```
# Old approach:
prompt = 'The Eiffel Tower is in the city of'
layers = llm.transformer.h
n_new_tokens = 3
hidden_states = []
with llm.generate(prompt, max_new_tokens=n_new_tokens) as tracer:
    for i in range(n_new_tokens):
        # Apply intervention - set first layer output to zero
        layers[0].output[0][:] = 0

        # Append desired hidden state post-intervention
        hidden_states.append(layers[-1].output.save())

        # Move to next generated token
        layers[0].next()

print("Hidden state length: ",len(hidden_states))
```

```
 Hidden state length:  3
```

We can use also `.all()` to streamline the multiple token generation process. We simply call `.all` on the module where we are applying the intervention (in this case GPT-2's layers), apply our intervention, and append our hidden states (stored in an `nnsight.list()` object). Let's test this out for the multiple token generation case:

[95]:
```
# using .all():
prompt = 'The Eiffel Tower is in the city of'
layers = llm.transformer.h
n_new_tokens = 3
with llm.generate(prompt, max_new_tokens=n_new_tokens) as tracer:
    hidden_states = nnsight.list().save() # Initialize & .save() nnsight list

    # Call .all() to apply intervention to each new token
    layers.all()

    # Apply intervention - set first layer output to zero
    layers[0].output[0][:] = 0

    # Append desired hidden state post-intervention
    hidden_states.append(layers[-1].output) # no need to call .save
    # Don't need to loop or call .next()!

print("Hidden state length: ",len(hidden_states))
```

```
 Hidden state length:  3
```

Easy! Note that because `.all()` is recursive, it will only work to append outputs called on children of the module that `.all()` was called on. See example below for more information. TL;DR: apply `.all()` on the highest-level accessed module if interventions and outputs have different hierarchies within model structure.

▶ Recursive properties of .all()

Model Editing

NNsight's model editing feature allows you to create persistently modified versions of a model with a use of `.edit()`. Unlike interventions in a tracing context, which are temporary, the **Editor** context enables you to make lasting changes to a model instance. This feature is useful for: * Creating modified model variants without altering the original * Applying changes that persist across multiple forward passes * Comparing interventions between original and edited models Let's explore how to use the **Editor** context to make a simple persistent change to a model:

[96]:

```python
# we take the hidden states with the expected output "Paris"
with llm.trace("The Eiffel Tower is located in the city of") as tracer:
    hs11 = llm.transformer.h[11].output[0][:, -1, :].save()

# the edited model will now always predict "Paris" as the next token
with llm.edit() as llm_edited:
    llm.transformer.h[11].output[0][:, -1, :] = hs11

# we demonstrate this by comparing the output of an unmodified model...
with llm.trace("Vatican is located in the city of") as tracer:
    original_tokens = llm.lm_head.output.argmax(dim=-1).save()

# ...with the output of the edited model
with llm_edited.trace("Vatican is located in the city of") as tracer:
    modified_tokens = llm.lm_head.output.argmax(dim=-1).save()


print("\nOriginal Prediction: ", llm.tokenizer.decode(original_tokens[0][-1]))
print("Modified Prediction: ", llm.tokenizer.decode(modified_tokens[0][-1]))
```

```
Original Prediction:   Rome
Modified Prediction:   Paris
```

Edits defined within an **Editor** context create a new, modified version of the model by default, preserving the original. This allows for safe experimentation with model changes. If you wish to modify the original model directly, you can set `inplace=True` when calling `.edit()`. Use this option cautiously, as in-place edits alter the base model for all the consequent model calls.

[97]:

```python
# we use the hidden state we saved above (hs11)
with llm.edit(inplace=True) as llm_edited:
    llm.transformer.h[11].output[0][:, -1, :] = hs11

# we demonstrate this by comparing the output of an unmodified model...
with llm.trace("Vatican is located in the city of") as tracer:
    modified_tokens = llm.lm_head.output.argmax(dim=-1).save()

print("Modified In-place: ", llm.tokenizer.decode(modified_tokens[0][-1]))
```

```
Modified In-place:   Paris
```

If you've made in-place edits to your model and need to revert these changes, you can apply `.clear_edits()`. This method removes all edits applied to the model, effectively restoring it to its original state.

[98]:

```
llm.clear_edits()

with llm.trace("Vatican is located in the city of"):
    modified_tokens = llm.lm_head.output.argmax(dim=-1).save()

print("Edits cleared: ", llm.tokenizer.decode(modified_tokens[0][-1]))
```

```
 Edits cleared:    Rome
```

3 I thought you said huge models?

`NNsight` is only one part of our project to democratize access to AI internals. The other half is the National Deep Inference Fabric, or `NDIF`. `NDIF` hosts large models for shared access using `NNsight`, so you don't have to worry about any of the headaches of hosting large models yourself! The interaction between `NDIF` and `NNsight` is fairly straightforward. The **intervention graph** we create via the tracing context can be encoded into a custom json format and sent via an http request to the `NDIF` servers. `NDIF` then decodes the **intervention graph** and **interleaves** it alongside the specified model. To see which models are currently being hosted, check out the following status page: https://nnsight.net/status/

Remote execution

In its current state, `NDIF` requires you to receive an API key. Therefore, to run the rest of this walkthrough, you need one of your own. To get one, simply register at https://login.ndif.us. With a valid API key, you then can configure `nnsight` as follows:

[100]:
```
from nnsight import CONFIG

CONFIG.set_default_api_key("YOUR_API_KEY")
```

If you're running in a local IDE, this only needs to be run once as it will save the API key as the default in a .config file along with your `nnsight` installation. You can also add your API key to Google Colab secrets. To amp things up a few levels, let's demonstrate using `nnsight`'s tracing context with `Llama-3.1-8b`!

[101]:
```
import os

# Llama 3.1 8b is a gated model, so you need to apply for access on HuggingFace and i
os.environ['HF_TOKEN'] = "YOUR_HUGGING_FACE_TOKEN"
```

[8]:

```
from nnsight import LanguageModel

# We'll never actually load the parameters locally, so no need to specify a device_ma
llama = LanguageModel("meta-llama/Meta-Llama-3.1-8B")
# All we need to specify using NDIF vs executing locally is remote=True.
with llama.trace("The Eiffel Tower is in the city of", remote=True) as runner:

    hidden_states = llama.model.layers[-1].output.save()

    output = llama.output.save()

print(hidden_states)

print(output["logits"])
```

```
(tensor([[[ 1.7734,  2.6875,  0.8047,  ..., -1.8594,  2.2344,  3.2656],
          [ 0.0312, -0.0352, -2.8750,  ..., -0.8906, -0.0547,  1.6172],
          [ 1.3594, -2.0156,  1.7031,  ..., -1.7031, -0.7422,  1.4375],
          ...,
          [ 1.0000,  0.3203, -0.2656,  ..., -0.0723, -0.2559,  0.2090],
          [ 0.4707, -0.3496,  0.2422,  ...,  0.7344, -0.0078,  0.1133],
          [-0.0566, -0.3496,  0.4746,  ...,  0.9844,  0.6797, -0.8750]]],
       dtype=torch.bfloat16),)
tensor([[[ 6.3438,  8.3750, 12.8125,  ..., -4.3750, -4.3750, -4.3750],
          [-2.4375, -1.7266, -2.0156,  ..., -9.1250, -9.1250, -9.1250],
          [ 9.6875,  4.5625,  5.8750,  ..., -3.3906, -3.3906, -3.3906],
          ...,
          [ 2.3281,  1.0703, -0.3203,  ..., -7.1562, -7.1562, -7.1562],
          [11.1875,  6.0312,  4.9062,  ..., -3.5156, -3.5156, -3.5156],
          [ 8.0000,  5.2500,  4.3750,  ..., -3.9844, -3.9844, -3.9844]]],
       dtype=torch.bfloat16)
```

It really is as simple as `remote=True`. All of the techniques we went through in earlier sections work just the same when running locally or remotely.

Sessions

NDIF uses a queue to handle concurrent requests from multiple users. To optimize the execution of our experiments we can use the `session` context to efficiently package multiple interventions together as one single request to the server. This offers the following benefits: 1. All interventions within a session will be executed one after another without additional wait in the NDIF queue 2. All intermediate outputs for each intervention are stored on the server and can be accessed by other interventions in the same session without moving the data back and forth between NDIF and the local machine Let's take a look:

[9]:
```
with llama.session(remote=True) as session:

  with llama.trace("The Eiffel Tower is in the city of") as t1:
    # capture the hidden state from layer 32 at the last token
    hs_31 = llama.model.layers[31].output[0][:, -1, :] # no .save()
    t1_tokens_out = llama.lm_head.output.argmax(dim=-1).save()

  with llama.trace("Buckingham Palace is in the city of") as t2:
    llama.model.layers[1].output[0][:, -1, :] = hs_31[:]
    t2_tokens_out = llama.lm_head.output.argmax(dim=-1).save()

print("\nT1 - Original Prediction: ", llama.tokenizer.decode(t1_tokens_out[0][-1]))
print("T2 - Modified Prediction: ", llama.tokenizer.decode(t2_tokens_out[0][-1]))
```

```
T1 - Original Prediction:   Paris
T2 - Modified Prediction:  ://
```

In the example above, we are interested in replacing the hidden state of a later layer with an earlier one. Since we are using a `session`, we don't have to save the hidden state from Tracer 1 to reference it in Tracer 2. It is important to note that all the traces defined within the `session` context are executed sequentially, strictly following the order of definition (i.e. `t2` being executed after `t1` and `t3` after `t2` etc.). The `session` context object has its own methods to log values and be terminated early.

[104]:
```
with llama.session(remote=True) as session:
  session.log("-- Early Stop --")
  nnsight.stop
```

In addition to the benefits mentioned above, the `session` context also enables interesting experiments not possible with other `nnsight` tools — since every trace is run on its own model, it means that within one session we can run interventions between different models — for example, we could swap activations between base and instruct versions of the Llama model and compare their outputs. And `session` can also be used to run similar experiments entirely locally!

Streaming

Streaming enables users apply functions and datasets locally during remote model execution. This allows users to stream results for immediate consumption (i.e., seeing tokens as they are generated) or applying non-whitelisted functions such as model tokenizers, large local datasets, and more!

- `nnsight.local()` context sends values immediately to user's local machine from server
- Intervention graph is executed locally on downstream nodes
- Exiting local context uploads data back to server
- `@nnsight.trace` function decorator enables custom functions to be added to intervention graph when using `nnsight.local()`

`nnsight.local()`

You may sometimes want to locally access and manipulate values during remote execution. Using `.local()` on a proxy, you can send remote content to your local machine and apply local functions. The intervention graph is then executed locally on downstream nodes (until you send execution back to the remote server by exiting the `.local()` context). There are a few use cases for streaming with `.local()`, including live chat generation and applying large datasets or non-whitelisted local functions to the intervention graph. Now let's explore how streaming works. We'll start by grabbing some hidden states of the model and printing their value using `tracer.log()`. Without calling `nnsight.local()`, these operations will all occur remotely.

[120]:
```python
# This will give you a remote LOG response because it's coming from the remote server
with llama.trace("hello", remote=True) as tracer:

    hs = llama.model.layers[-1].output[0]

    tracer.log(hs[0,0,0])

    out =  llama.lm_head.output.save()

print(out)
```

```
tensor([[[ 6.3438,  8.3750, 12.8125,  ..., -4.3750, -4.3750, -4.3750],
         [10.2500,  2.1094,  2.8281,  ..., -8.2500, -8.2500, -8.2500]]],
       dtype=torch.bfloat16)
```

Now, let's try the same operation using the `nnsight.local()` context. This will send the operations to get and print the hidden states to your local machine, changing how the logging message is formatted (local formatting instead of remote).

[121]:
```python
# This will print locally because it's already local
with llama.trace("hello", remote=True) as tracer:

    with nnsight.local():
        hs = llama.model.layers[-1].output[0]
        tracer.log(hs[0,0,0])

    out =  llama.lm_head.output.save()

print(out)
```

```
tensor(1.7656, dtype=torch.bfloat16)


tensor([[[ 6.3438,  8.3750, 12.8125,  ..., -4.3750, -4.3750, -4.3750],
         [10.2500,  2.1094,  2.8281,  ..., -8.2500, -8.2500, -8.2500]]],
       dtype=torch.bfloat16)
```

`@nnsight.trace` function decorator

We can also use function decorators to create custom functions to be used during `.local` calls. This is a handy way to enable live streaming of a chat or to train probing classifiers on model hidden states. Let's try out `@nnsight.trace` and `nnsight.local()` to access a custom function during remote execution.

```
[122]: # first, let's define our function
       @nnsight.trace # decorator that enables this function to be added to the intervention
       def my_local_fn(value):
           return value * 0

       # We use a local function to ablate some hidden states
       # This downloads the data for the .local context, and then uploads it back to set the
       with llama.generate("hello", remote=True) as tracer:

           hs = llama.model.layers[-1].output[0]

           with nnsight.local():

               hs = my_local_fn(hs)

           llama.model.layers[-1].output[0][:] = hs

           out =  llama.lm_head.output.save()
```

Note that without calling `.local`, the remote API does not know about `my_local_fn` and will throw a whitelist error. A whitelist error occurs because you are being allowed access to the function.

```
[123]: with llama.trace("hello", remote=True) as tracer:

           hs = llama.model.layers[-1].output[0]

           hs = my_local_fn(hs) # no .local - will cause an error

           llama.model.layers[-1].output[0][:] = hs * 2

           out =  llama.lm_head.output.save()

       print(out)
```

```
---------------------------------------------------------------------------
FunctionWhitelistError                    Traceback (most recent call last)
Cell In[123], line 1
----> 1 with llama.trace("hello", remote=True) as tracer:
      3     hs = llama.model.layers[-1].output[0]
      5     hs = my_local_fn(hs) # no .local - will cause an error

File ~/Documents/Projects/nnsight/src/nnsight/intervention/contexts/interleaving.py:
     92     self.invoker.__exit__(None, None, None)
     94 self._model._envoy._reset()
---> 96 super().__exit__(exc_type, exc_val, exc_tb)

File ~/Documents/Projects/nnsight/src/nnsight/tracing/contexts/tracer.py:25, in Trac
     21 from .globals import GlobalTracingContext
     23 GlobalTracingContext.try_deregister(self)
---> 25 return super().__exit__(exc_type, exc_val, exc_tb)

File ~/Documents/Projects/nnsight/src/nnsight/tracing/contexts/base.py:82, in Contex
     78 graph = graph.stack.pop()
     80 graph.alive = False
---> 82 self.backend(graph)

File ~/Documents/Projects/nnsight/src/nnsight/intervention/backends/remote.py:77, in
     72 def __call__(self, graph: Graph):
     74     if self.blocking:
     75 
     76         # Do blocking request.
---> 77         result = self.blocking_request(graph)
     79     else:
     80 
     81         # Otherwise we are getting the status / result of the existing job.
     82         result = self.non_blocking_request(graph)

File ~/Documents/Projects/nnsight/src/nnsight/intervention/backends/remote.py:289, i
    280 sio.connect(
    281     self.ws_address,
    282     socketio_path="/ws/socket.io",
    283     transports=["websocket"],
    284     wait_timeout=10,
    285 )
    287 remote_graph = preprocess(graph)
--> 289 data, headers = self.request(remote_graph)
    291 headers["session_id"] = sio.sid
    293 # Submit request via

File ~/Documents/Projects/nnsight/src/nnsight/intervention/backends/remote.py:60, in
     58 def request(self, graph: Graph) -> Tuple[bytes, Dict[str, str]]:
---> 60     data = RequestModel.serialize(graph, self.format, self.zlib)
     62     headers = {
     63         "model_key": self.model_key,
     64         "format": self.format,
   (...)
     67         "sent-timestamp": str(time.time()),
     68     }
     70     return data, headers

File ~/Documents/Projects/nnsight/src/nnsight/schema/request.py:43, in RequestModel.
     38 @staticmethod
     39 def serialize(graph: Graph, format:str, _zlib:bool) -> bytes:
     41     if format == "json":
---> 43         data = RequestModel(graph=graph)
     45         json = data.model_dump(mode="json")
     47         data = msgspec.json.encode(json)

File ~/Documents/Projects/nnsight/src/nnsight/schema/request.py:30, in RequestModel._
     28 def __init__(self, *args, memo: Dict = None, **kwargs):
---> 30     super().__init__(*args, memo=memo or dict(), **kwargs)
     32     if memo is None:
     34         self.memo = {**MEMO}
    [... skipping hidden 1 frame]

File ~/Documents/Projects/nnsight/src/nnsight/schema/format/types.py:276, in GraphMo
    273 @staticmethod
    274 def to_model(value: Graph) -> Self:
--> 276     return GraphModel(graph=value, nodes=value.nodes)

    [... skipping hidden 1 frame]

File ~/Documents/Projects/nnsight/src/nnsight/schema/format/types.py:77, in memoized
     75 def inner(value):
---> 77     model = fn(value)
     79     _id = id(value)
     81     MEMO[_id] = model

File ~/Documents/Projects/nnsight/src/nnsight/schema/format/types.py:101, in NodeMod
     97 @staticmethod
     98 @memoized
     99 def to_model(value: Node) -> Self:
--> 101     return NodeModel(target=value.target, args=value.args, kwargs=value.kwar

    [... skipping hidden 1 frame]

File ~/Documents/Projects/nnsight/src/nnsight/schema/format/types.py:244, in Functio
    239 @staticmethod
    240 def to_model(value:FUNCTION):
    242     model = FunctionModel(function_name=get_function_name(value))
--> 244     FunctionModel.check_function_whitelist(model.function_name)
    246     return model

File ~/Documents/Projects/nnsight/src/nnsight/schema/format/types.py:251, in Functio
    248 @classmethod
    249 def check_function_whitelist(cls, qualname: str) -> str:
    250     if qualname not in FUNCTIONS_WHITELIST:
--> 251         raise FunctionWhitelistError(
    252             f"Function with name `{qualname}` not in function whitelist."
    253         )
    255     return qualname

FunctionWhitelistError: Function with name `__main__.my_local_fn` not in function wh
```

Example: Live-streaming remote chat

Now that we can access data within the tracing context on our local computer, we can apply non-whitelisted functions, such as the model's tokenizer, within our tracing context. Let's build a decoding function that will decode tokens into words and print the result.

[124]:

```
@nnsight.trace
def my_decoding_function(tokens, model, max_length=80, state=None):
    # Initialize state if not provided
    if state is None:
        state = {'current_line': '', 'current_line_length': 0}

    token = tokens[-1] # only use last token

    # Decode the token
    decoded_token = llama.tokenizer.decode(token).encode("unicode_escape").decode()

    if decoded_token == '\\n':  # Handle explicit newline tokens
        # Print the current line and reset state
        print('',flush=True)
        state['current_line'] = ''
        state['current_line_length'] = 0
    else:
        # Check if adding the token would exceed the max length
        if state['current_line_length'] + len(decoded_token) > max_length:
            print('',flush=True)
            state['current_line'] = decoded_token  # Start a new line with the curren
            state['current_line_length'] = len(decoded_token)
            print(state['current_line'], flush=True, end="")  # Print the current lin
        else:
            # Add a space if the line isn't empty and append the token
            if state['current_line']:
                state['current_line'] += decoded_token
            else:
                state['current_line'] = decoded_token
            state['current_line_length'] += len(decoded_token)
            print(state['current_line'], flush=True, end="")  # Print the current lin

    return state
```

Now we can decode and print our model outputs throughout token generation by accessing our decoding function through `nnsight.local()`.

[125]:

```
import torch

nnsight.CONFIG.APP.REMOTE_LOGGING = False

prompt = "A press release is an official statement delivered to members of the news me
# prompt = "Your favorite board game is"

print("Prompt: ",prompt,'\n', end ="")

# Initialize the state for decoding
state = {'current_line': '', 'current_line_length': 0}

with llama.generate(prompt, remote=True, max_new_tokens = 50) as generator:
    # Call .all() to apply to each new token
    llama.all()

    all_tokens = nnsight.list().save()

    # Access model output
    out = llama.lm_head.output.save()

    # Apply softmax to obtain probabilities and save the result
    probs = torch.nn.functional.softmax(out, dim=-1)
    max_probs = torch.max(probs, dim=-1)
    tokens = max_probs.indices.cpu().tolist()
    all_tokens.append(tokens[0]).save()

    with nnsight.local():
        state = my_decoding_function(tokens[0], llama, max_length=20, state=state)
```

```
Prompt:  A press release is an official statement delivered to members of the news m


 providing information, an official statement, or making an announcement.A press rel
```

Looping across sessions

We mention earlier that the `session` context enables multi-tracing execution. But how can we optimize a process that would require running an intervention graph in a loop? If we create a simple `for` loop with a **Tracer context** inside, this will result in creating a new intervention graph at each iteration, which is not scalable. We solve this problem the `nnsight` way via the **Iterator context**: an intervention loop that iteratively executes and updates a single intervention graph. Use a `session` to define the **Iterator context** and pass in a sequence of items that you want to loop over at each iteration:

[126]:

```
with llama.session(remote=True) as session:

  with session.iter([0, 1, 2]) as item:
    # define intervention body here ...

    with llama.trace("_"):
      # define interventions here ...
      pass

    with llama.trace("_"):
      # define interventions here ...
      pass
```

The `Iterator` context extends all the `nnsight` graph-based functionalities, but also closely mimics the conventional `for` loop statement in Python, which allows it to support all kind of iterative operations with a use of `as item` syntax:

[127]:
```
with llama.session(remote=True) as session:

  li = nnsight.list()
  [li.append([num]) for num in range(0, 3)] # adding [0], [1], [2] to the list
  li2 = nnsight.list().save()

  # You can create nested Iterator contexts
  with session.iter(li) as item:
    with session.iter(item) as item_2:
      li2.append(item_2)

print("\nList: ", li2)
```