Pytorch Tutorial: Autograd: The Killer Feature!

Nicholas Lubbers and Arvind Mohan, Los Alamos National Laboratory

(Caveat emptor: We are not affiliated directly with PyTorch. But we find it really useful, and hope you will too.)

> *An open source deep learning platform that provides a seamless path from research prototyping to production deployment.*

LA-UR-24-23810 https://pytorch.org/

Talk Outline

1. What is PyTorch? When, who and why.
2. Comparison with Tensorflow, Keras, and other deep learning frameworks.
3. Tensors and Data types
4. Similarities and Differences with NumPy
5. Understanding documentation & source code- GPU Computing in python with PyTorch

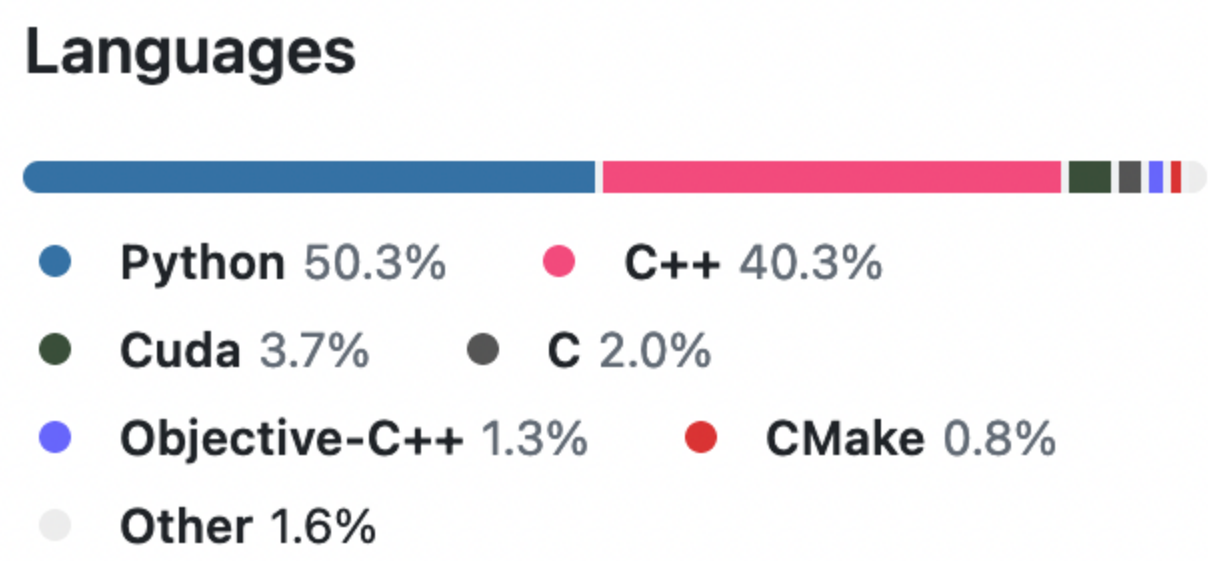
What is PyTorch? When, who and why.

1. From the webpage:

> *An open source deep learning platform that provides a seamless path from research prototyping to production deployment.*

2. PyTorch is a "second-generation" framework, an evolution of the original "Torch" Library. Torch is written in C++, and the original interface was built for the LUA programming language.

3. Much of Pytorch is still written in C++/Cuda:



Let's get started.

What is PyTorch? When, who and why.

4. Pytorch = "Numpy" + "GPU" + "Automatic Differentiation"

    - from the perspective scientific computing, pytorch has a lot of useful tools for generic operations on multiple-dimensional arrays (a.k.a. Tensors) -- it's not just for neural networks!

5. It is primarily developed by Facebook's artificial-intelligence research group along with Universities & other Corporations. It is completely **Open Source Software**.

Let’s get started.

Let’s get started.

Let's get started.

In [1]:

```
import torch as th
print("Pytorch Version:",th.__version__)
```

```
Pytorch Version: 2.2.2
```

Let's get started.

In [1]:
```
import torch as th
print("Pytorch Version:",th.__version__)
```

```
Pytorch Version: 2.2.2
```

In [2]:
```
# Let's construct a 2-d array, or as Pytorch calls it, a Tensor:
x = th.Tensor([[1,2,3],[4,5,6]])
x
```

Out[2]:
```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

Let's get started.

In [1]:
```
import torch as th
print("Pytorch Version:",th.__version__)
```

```
Pytorch Version: 2.2.2
```

In [2]:
```
# Let's construct a 2-d array, or as Pytorch calls it, a Tensor:
x = th.Tensor([[1,2,3],[4,5,6]])
x
```

Out[2]:
```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

In [3]:
```
# You can readily convert it to a numpy array:
x_np = x.numpy()
x_np
```

Out[3]:
```
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
```

In [4]:
```python
# The `shape` attribute gives the dimensions of a Tensor, very similar t
x.shape
```

Out[4]:
```
torch.Size([2, 3])
```

In [4]:
```
# The `shape` attribute gives the dimensions of a Tensor, very similar t
x.shape
```

Out[4]:
```
torch.Size([2, 3])
```

In [5]:
```
x_np.shape
```

Out[5]:
```
(2, 3)
```

In [4]:
```
# The `shape` attribute gives the dimensions of a Tensor, very similar t
x.shape
```

Out[4]:
```
torch.Size([2, 3])
```

In [5]:
```
x_np.shape
```

Out[5]:
```
(2, 3)
```

In [6]:
```
# The `dtype` attribute is helpful:
x.dtype
```

Out[6]:
```
torch.float32
```

In [7]:
```python
# We can convert using a `to` method:
# Dtypes:
# float16/32/64
# int/uint 8/16/32/64

xp = x.to(th.int64)
xp,xp.dtype
```

Out[7]:
```
(tensor([[1, 2, 3],
         [4, 5, 6]]),
 torch.int64)
```

In [7]:
```python
# We can convert using a `to` method:
# Dtypes:
# float16/32/64
# int/uint 8/16/32/64

xp = x.to(th.int64)
xp,xp.dtype
```

Out[7]:
```
(tensor([[1, 2, 3],
         [4, 5, 6]]),
 torch.int64)
```

In [8]:
```python
#There are many convenient mathematical operations,
#such as the transpose:
x.t()
```

Out[8]:
```
tensor([[1., 4.],
        [2., 5.],
        [3., 6.]])
```

Numpy compatibility

There are several mathematical functions in pytorch which have their (usually similarly named) equivalents in Pytorch.

Numpy compatibility

There are several mathematical functions in pytorch which have their (usually similarly named) equivalents in Pytorch.

In [10]:
```
y = th.sin(x)
print('Pytorch sine function:\n', y)
x_np = x.numpy()

y_np = np.sin(x_np)
print('Numpy sine function:\n', y_np)
```

```
Pytorch sine function:
 tensor([[ 0.8415,  0.9093,  0.1411],
        [-0.7568, -0.9589, -0.2794]])
Numpy sine function:
 [[ 0.841471   0.9092974  0.14112  ]
 [-0.7568025 -0.9589243 -0.2794155]]
```

Numpy functions can even be applied to pytorch tensors (with important caveats -- more on that later!)

Numpy functions can even be applied to pytorch tensors (with important caveats -- more on that later!)

In [11]:
```
np.sin(x)
```

Out[11]:
```
tensor([[ 0.8415,  0.9093,  0.1411],
        [-0.7568, -0.9589, -0.2794]])
```

Numpy functions can even be applied to pytorch tensors (with important caveats -- more on that later!)

In [11]:
```
np.sin(x)
```

Out[11]:
```
tensor([[ 0.8415,  0.9093,  0.1411],
        [-0.7568, -0.9589, -0.2794]])
```

In [12]:
```
# If you have a numpy tensor, use `th.as_tensor`
th.as_tensor(x_np)
```

Out[12]:
```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

Numpy functions can even be applied to pytorch tensors (with important caveats -- more on that later!)

In [11]:
```
np.sin(x)
```

Out[11]:
```
tensor([[ 0.8415,  0.9093,  0.1411],
        [-0.7568, -0.9589, -0.2794]])
```

In [12]:
```
# If you have a numpy tensor, use `th.as_tensor`
th.as_tensor(x_np)
```

Out[12]:
```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

In [13]:
```
# Or `th.from_numpy`:
th.from_numpy(x_np)
```

Out[13]:
```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

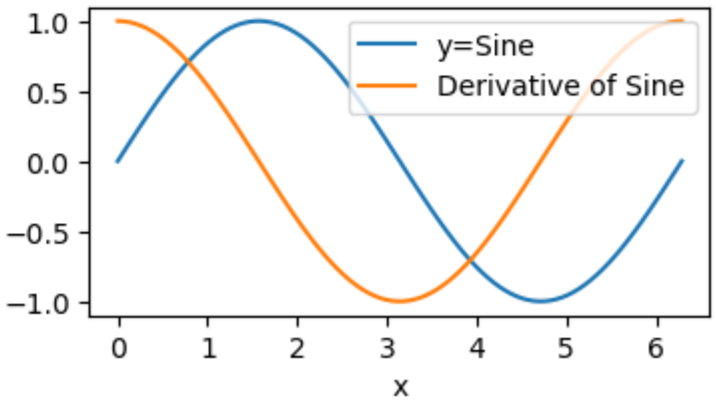
And of course, automatic differentiation

And of course, automatic differentiation

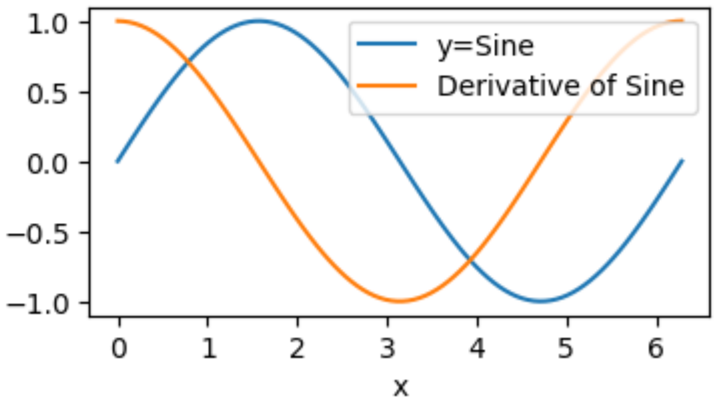
In [15]:

```
x = th.linspace(0,2*np.pi,100)
x.requires_grad_(True) # <- ??? We will explain this in the automatic di
y = th.sin(x)
from torch.autograd import grad
y_prime = grad(y.sum(),x)[0] # Here's the magic function `grad`
```

In [16]:
```
fig,ax = plt.subplots(1,1,figsize=(4,2))
plt.plot(x.detach().numpy(),y.detach().numpy(),label="y=Sine") # Usually
plt.xlabel("x")
plt.plot(x.detach().numpy(),y_prime.numpy(),label="Derivative of Sine")
plt.legend()
plt.show()
```



In [16]:
```
fig,ax = plt.subplots(1,1,figsize=(4,2))
plt.plot(x.detach().numpy(),y.detach().numpy(),label="y=Sine") # Usually
plt.xlabel("x")
plt.plot(x.detach().numpy(),y_prime.numpy(),label="Derivative of Sine")
plt.legend()
plt.show()
```



Pytorch has **automatically** constructed the derivative.

These *Automatic Differentiation* features are key to training Neural Networks, but also very useful for physical codes

GPU features

You can send a tensor to be stored on a GPU by using the `Tensor.cuda` method:

GPU features

You can send a tensor to be stored on a GPU by using the `Tensor.cuda` method:

In [17]:

```
x.cuda()
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[17], line 1
----> 1 x.cuda()

File ~/opt/miniconda3/envs/torch_tutorial/lib/python3.12/site-packages/torch/cuda/__init__.py:293, in _lazy_init()
    288     raise RuntimeError(
    289         "Cannot re-initialize CUDA in forked subprocess. To use CUDA with "
    290         "multiprocessing, you must use the 'spawn' start method"
    291     )
    292 if not hasattr(torch._C, "_cuda_getDeviceCount"):
--> 293     raise AssertionError("Torch not compiled with CUDA enabled")
    294 if _cudart is None:
    295     raise AssertionError(
    296         "libcudart functions unavailable. It looks like you have a broken build?"
    297     )

AssertionError: Torch not compiled with CUDA enabled
```

... *if* you have a GPU on your machine!

GPU Support:

1. Nvidia/CUDA: Primary intended use of pytorch, very good support.
2. AMD/ROCm: According to forums, this is reasonably good now.
3. Apple Metal (mps): Reasonably good, but not all operations are available yet

As of 2024, in the end, you are very likely to run computationlly intensive code on linux with an nvidia GPU. This is the ideal scenario for pytorch.

In [18]:

```
mps_tensor = th.ones(5,device='mps')
mps_tensor
```

```
/Users/nlubbers/opt/miniconda3/envs/torch_tutorial/lib/python3.1
2/site-packages/torch/_tensor_str.py:137: UserWarning: MPS: nonz
ero op is supported natively starting from macOS 13.0. Falling b
ack on CPU. This may have performance implications. (Triggered i
nternally at /Users/runner/work/_temp/anaconda/conda-bld/pytorch
_1711403226120/work/aten/src/ATen/native/mps/operations/Indexin
g.mm:283.)
  nonzero_finite_vals = torch.masked_select(
```

Out[18]:

```
tensor([1., 1., 1., 1., 1.], device='mps:0')
```

In [18]:
```
mps_tensor = th.ones(5,device='mps')
mps_tensor
```

```
/Users/nlubbers/opt/miniconda3/envs/torch_tutorial/lib/python3.1
2/site-packages/torch/_tensor_str.py:137: UserWarning: MPS: nonz
ero op is supported natively starting from macOS 13.0. Falling b
ack on CPU. This may have performance implications. (Triggered i
nternally at /Users/runner/work/_temp/anaconda/conda-bld/pytorch
_1711403226120/work/aten/src/ATen/native/mps/operations/Indexin
g.mm:283.)
  nonzero_finite_vals = torch.masked_select(
```

Out[18]:
```
tensor([1., 1., 1., 1., 1.], device='mps:0')
```

Note how even just displaying an MPS tensor throws a warning...

Expect that the more recent features in pytorch have rough edges!

Some python DL frameworks:

1. Pytorch
2. Jax ← Can be trickier to use, but well-respected and more suited for certain problems
3. Keras ← Aims at "standard" ML problems more than research
4. Tensorflow ← Popularized by google corporate but bogged down with confusion
5. Wikipedia: Comparison of deep-learning software

There are so many options: Why Pytorch?

- As we've seen, the interface is very close to numpy. If you know numpy, you can already code in pytorch.
- Documentation/Community Support:
    - Link to documentation
    - Link to forums
- Community Support
- PyTorch is Pythonic -- the programming paradigm follows python seamlessly.
- Rapid Development & Debugging

Still not convinced?

How to print in pytorch:

Still not convinced?

How to print in pytorch:

In [19]:
```
x=th.Tensor([[1,2,3],[4,5,6]])
print(x)
```

```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

Still not convinced?

How to print in pytorch:

In [19]:
```
x=th.Tensor([[1,2,3],[4,5,6]])
print(x)
```

```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

stackoverflow: How to print the value of a Tensor object in TensorFlow?

313 upvotes -- best answer has three paragraphs *plus* two footnotes

(In all seriousness, things have gotten better in TensorFlow since this was posted. But we still prefer PyTorch.)

JAX documentation: Runtime value debugging in JAX

> ***TL;DR*** *Use `jax.debug.print()` to print values to stdout in `jax.jit` -, `jax.pmap` -, and `pjit-decorated functions`, and `jax.debug.breakpoint()` to pause execution of your compiled function to inspect values in the call stack.*

Pytorch works seamlessly with python functions

Pytorch works seamlessly with python functions

In [20]:
```
def my_function(x):
    print("Input x shape and dtype:",x.shape,x.dtype)
    print("Input x values:",x)
    y = x**2 - 2*x
    return y
```

Pytorch works seamlessly with python functions

In [20]:
```
def my_function(x):
    print("Input x shape and dtype:",x.shape,x.dtype)
    print("Input x values:",x)
    y = x**2 - 2*x
    return y
```

In [21]:
```
x = th.arange(4)
y = my_function(x)
print("Y =",y)
```

```
Input x shape and dtype: torch.Size([4]) torch.int64
Input x values: tensor([0, 1, 2, 3])
Y = tensor([ 0, -1,  0,  3])
```

Let’s look more at devices and dtypes in pytorch

- implicit upcasting does happen in pytorch

Let's look more at devices and dtypes in pytorch

- implicit upcasting does happen in pytorch

In [22]:
```
x = th.arange(4,dtype=th.int)
y = th.arange(4,dtype=th.float)
x,y,x+y
```

Out[22]:
```
(tensor([0, 1, 2, 3], dtype=torch.int32),
 tensor([0., 1., 2., 3.]),
 tensor([0., 2., 4., 6.]))
```

In [23]:

```
x = th.arange(4,dtype=th.float32)
y = th.arange(4,dtype=th.float64)
x,y,x+y
```

Out[23]:

```
(tensor([0., 1., 2., 3.]),
 tensor([0., 1., 2., 3.], dtype=torch.float64),
 tensor([0., 2., 4., 6.], dtype=torch.float64))
```

In [24]:

```python
x = th.arange(4,dtype=th.int)
y = th.ones(4,dtype=th.bool)
x,y,x+y
```

Out[24]:

```
(tensor([0, 1, 2, 3], dtype=torch.int32),
 tensor([True, True, True, True]),
 tensor([1, 2, 3, 4], dtype=torch.int32))
```

Note that pytorch uses float32 by default:

Note that pytorch uses float32 by default:

In [25]:
```
th.float == th.float32
```

Out[25]:
```
True
```

Note that pytorch uses float32 by default:

In [25]:
```
th.float == th.float32
```

Out[25]:
```
True
```

In [26]:
```
th.float == th.float64
```

Out[26]:
```
False
```

Note that pytorch uses float32 by default:

In [25]:
```
th.float == th.float32
```

Out[25]:
```
True
```

In [26]:
```
th.float == th.float64
```

Out[26]:
```
False
```

In [27]:
```
th.ones(1).dtype
```

Out[27]:
```
torch.float32
```

- The default floating point type can be changed though:

- The default floating point type can be changed though:

In [28]:
```
th.set_default_dtype(th.float64)
print(th.ones(1).dtype)
th.set_default_dtype(th.float32)
```

```
torch.float64
```

There is likewise torch.cuda.set_device() for where new tensors are created (which GPU) when a specific GPU is not explicitly specified.

There is likewise torch.cuda.set_device() for where new tensors are created (which GPU) when a specific GPU is not explicitly specified.

In [29]:
```
th.cuda.set_device
```

Out[29]:
```
<function torch.cuda.set_device(device: Union[torch.device, st
r, int, NoneType]) -> None>
```

Creating new tensors

The easiest way to create compatible tensors is to use the property of an existing tensor.

Just grab the dtype and device!

Creating new tensors

The easiest way to create compatible tensors is to use the property of an existing tensor.

Just grab the dtype and device!

In [30]:

```
x = th.arange(4,dtype=int)

y = th.ones(4,dtype=x.dtype,device=x.device)

# Also:
y= th.ones_like(x)
# But note that this function gives the same shape as X, as well

# Can also be done like this:
y = x.new_ones(4)
```

Creating new tensors

The easiest way to create compatible tensors is to use the property of an existing tensor.

Just grab the dtype and device!

In [30]:
```
x = th.arange(4,dtype=int)

y = th.ones(4,dtype=x.dtype,device=x.device)

# Also:
y= th.ones_like(x)
# But note that this function gives the same shape as X, as well

# Can also be done like this:
y = x.new_ones(4)
```

Note that the operation of creating a completely new tensor from scratch is comparatively rare in most pytorch code; usually you create tensors by combining them with existing tensors.

In-place operations

In-place operations are supported, but not encouraged. This has to do with how pytorch's autograd works, more on that later.

In-place operations

In-place operations are supported, but not encouraged. This has to do with how pytorch's autograd works, more on that later.

In [31]:
```
x = th.ones(4)
x[2]=3
x
```

Out[31]:
```
tensor([1., 1., 3., 1.])
```

In-place operations can be 'dangerous' with autograd. But Pytorch will try to inform you when in-place operations will break autograd!

In-place operations can be 'dangerous' with autograd. But Pytorch will try to inform you when in-place operations will break autograd!

In [32]:
```
x = th.ones(4)
x.requires_grad_()
x[2]=3
x
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[32], line 3
      1 x = th.ones(4)
      2 x.requires_grad_()
----> 3 x[2]=3
      4 x

RuntimeError: a view of a leaf Variable that requires grad is being used in an in-place operation.
```

Note that there are cases where in-place operations work with autograd. But they are still not a good idea -- they will not match the usual semantics precisely. (Usually one performs in-place operations to save memory, but this will often not actually save memory when using autograd)

Note that there are cases where in-place operations work with autograd. But they are still not a good idea -- they will not match the usual semantics precisely. (Usually one performs in-place operations to save memory, but this will often not actually save memory when using autograd)

In [33]:
```
x = th.ones(4)
x.requires_grad_()

y = 2*x
print(y)
y[2]=3
print(y)
```

```
tensor([2., 2., 2., 2.], grad_fn=<MulBackward0>)
tensor([2., 2., 3., 2.], grad_fn=<CopySlices>)
```

Note that there are cases where in-place operations work with autograd. But they are still not a good idea -- they will not match the usual semantics precisely. (Usually one performs in-place operations to save memory, but this will often not actually save memory when using autograd)

In [33]:
```
x = th.ones(4)
x.requires_grad_()

y = 2*x
print(y)
y[2]=3
print(y)
```

```
tensor([2., 2., 2., 2.], grad_fn=<MulBackward0>)
tensor([2., 2., 3., 2.], grad_fn=<CopySlices>)
```

Note here that the 'grad_fn' changed.

Pytorch can also be used to write generic scientific code!

Recent support for linear algebra (`th.linalg`) libaries for FFT (`th.fft`), sparse matrices (`th.sparse`) and even solving system of equations. While Numpy also has these capabilities, Pytorch shines in enabling GPU-accelerated versions of popular matrix operations. More Details here: https://pytorch.org/blog/torch-linalg-autograd/.

We will now show a few examples of the capabilities.

Matrix operations and Decompositions

Matrix operations and Decompositions

In [34]:
```
N = 3
A = th.randn(N, N, dtype=th.complex128)
print("A is: ", A)
```

```
A is:  tensor([[-0.6456+0.4686j,  0.2998-0.1719j,  0.2149-0.4210
j],
        [ 1.0444-0.0777j, -1.0309-0.3838j, -0.1683+1.3113j],
        [ 0.2602+0.0985j, -0.4137-0.3570j, -0.1110-0.3575j]],
       dtype=torch.complex128)
```

Matrix operations and Decompositions

In [34]:
```
N = 3
A = th.randn(N, N, dtype=th.complex128)
print("A is: ", A)
```

```
A is:  tensor([[-0.6456+0.4686j,  0.2998-0.1719j,  0.2149-0.4210
j],
        [ 1.0444-0.0777j, -1.0309-0.3838j, -0.1683+1.3113j],
        [ 0.2602+0.0985j, -0.4137-0.3570j, -0.1110-0.3575j]],
       dtype=torch.complex128)
```

In [35]:
```
A.T.conj() # Hermitian conjugate
```

Out[35]:
```
tensor([[-0.6456-0.4686j,  1.0444+0.0777j,  0.2602-0.0985j],
        [ 0.2998+0.1719j, -1.0309+0.3838j, -0.4137+0.3570j],
        [ 0.2149+0.4210j, -0.1683-1.3113j, -0.1110+0.3575j]],
       dtype=torch.complex128)
```

Matrix operations and Decompositions

In [34]:
```
N = 3
A = th.randn(N, N, dtype=th.complex128)
print("A is: ", A)
```

```
A is:  tensor([[-0.6456+0.4686j,  0.2998-0.1719j,  0.2149-0.4210
j],
        [ 1.0444-0.0777j, -1.0309-0.3838j, -0.1683+1.3113j],
        [ 0.2602+0.0985j, -0.4137-0.3570j, -0.1110-0.3575j]],
       dtype=torch.complex128)
```

In [35]:
```
A.T.conj() # Hermitian conjugate
```

Out[35]:
```
tensor([[-0.6456-0.4686j,  1.0444+0.0777j,  0.2602-0.0985j],
        [ 0.2998+0.1719j, -1.0309+0.3838j, -0.4137+0.3570j],
        [ 0.2149+0.4210j, -0.1683-1.3113j, -0.1110+0.3575j]],
       dtype=torch.complex128)
```

In [36]:
```
D = th.eye(N) #diagonal matrix
print("Diagonal matrix is: ", D)
```

```
Diagonal matrix is:  tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
```

In [37]:
```
B = A @ A.T.conj() + D
print(B)
```

```
tensor([[ 1.9793+0.0000j, -1.5421+0.5207j, -0.0578+0.4873j],
        [-1.5421-0.5207j,  5.0547+0.0000j,  0.3775-0.5381j],
        [-0.0578-0.4873j,  0.3775+0.5381j,  1.5162+0.0000j]],
       dtype=torch.complex128)
```

In [37]:
```python
B = A @ A.T.conj() + D
print(B)
```

```
tensor([[ 1.9793+0.0000j, -1.5421+0.5207j, -0.0578+0.4873j],
        [-1.5421-0.5207j,  5.0547+0.0000j,  0.3775-0.5381j],
        [-0.0578-0.4873j,  0.3775+0.5381j,  1.5162+0.0000j]],
       dtype=torch.complex128)
```

The @ tells Pytorch that this is matrix multiplication, and it will use efficient routines to compute the output. This notation is convenient with Python PEP 465. (The "@" operator was invented for numpy)

Cholesky Decomposition

Cholesky Decomposition

In [38]:
```
L = th.linalg.cholesky(B)
L
```

Out[38]:
```
tensor([[ 1.4069+0.0000j,  0.0000+0.0000j,  0.0000+0.0000j],
        [-1.0961-0.3701j,  1.9278+0.0000j,  0.0000+0.0000j],
        [-0.0411-0.3464j,  0.1060+0.0901j,  1.1727+0.0000j]],
       dtype=torch.complex128)
```

Cholesky Decomposition

In [38]:
```
L = th.linalg.cholesky(B)
L
```

Out[38]:
```
tensor([[ 1.4069+0.0000j,  0.0000+0.0000j,  0.0000+0.0000j],
        [-1.0961-0.3701j,  1.9278+0.0000j,  0.0000+0.0000j],
        [-0.0411-0.3464j,  0.1060+0.0901j,  1.1727+0.0000j]],
       dtype=torch.complex128)
```

In [39]:
```
L.T.conj()
```

Out[39]:
```
tensor([[ 1.4069-0.0000j, -1.0961+0.3701j, -0.0411+0.3464j],
        [ 0.0000-0.0000j,  1.9278-0.0000j,  0.1060-0.0901j],
        [ 0.0000-0.0000j,  0.0000-0.0000j,  1.1727-0.0000j]],
       dtype=torch.complex128)
```

Cholesky Decomposition

In [38]:
```
L = th.linalg.cholesky(B)
L
```

Out[38]:
```
tensor([[ 1.4069+0.0000j,  0.0000+0.0000j,  0.0000+0.0000j],
        [-1.0961-0.3701j,  1.9278+0.0000j,  0.0000+0.0000j],
        [-0.0411-0.3464j,  0.1060+0.0901j,  1.1727+0.0000j]],
       dtype=torch.complex128)
```

In [39]:
```
L.T.conj()
```

Out[39]:
```
tensor([[ 1.4069-0.0000j, -1.0961+0.3701j, -0.0411+0.3464j],
        [ 0.0000-0.0000j,  1.9278-0.0000j,  0.1060-0.0901j],
        [ 0.0000-0.0000j,  0.0000-0.0000j,  1.1727-0.0000j]],
       dtype=torch.complex128)
```

In [40]:
```
L @ L.T.conj()
```

Out[40]:
```
tensor([[ 1.9793+0.0000j, -1.5421+0.5207j, -0.0578+0.4873j],
        [-1.5421-0.5207j,  5.0547+0.0000j,  0.3775-0.5381j],
        [-0.0578-0.4873j,  0.3775+0.5381j,  1.5162+0.0000j]],
       dtype=torch.complex128)
```

We know that Cholesky decomposition for a matrix B is . Lets check our computation with a nifty utility called `allclose` in Pytorch. It asks you to provide two tensors for comparison, and a relative tolerance within which they can be deemed to be the same. Default tolerance is .

We know that Cholesky decomposition for a matrix B is . Lets check our computation with a nifty utility called `allclose` in Pytorch. It asks you to provide two tensors for comparison, and a relative tolerance within which they can be deemed to be the same. Default tolerance is .

In [41]:

```
th.allclose(B, L @ L.T.conj(), rtol=1e-07)
```

Out[41]:

```
True
```

QR Decomposition

Quite common, where we decompose a square matrix into an orthogonal matrix and an upper triangular matrix

QR Decomposition

Quite common, where we decompose a square matrix into an orthogonal matrix and an upper triangular matrix

In [42]:
```
Q, R = th.linalg.qr(B)
```

QR Decomposition

Quite common, where we decompose a square matrix into an orthogonal matrix and an upper triangular matrix

In [42]:
```
Q, R = th.linalg.qr(B)
```

In [43]:
```
Q
```

Out[43]:
```
tensor([[-0.7586+0.0000e+00j, -0.5884+2.0440e-01j,  0.0013+1.91
22e-01j],
        [ 0.5910+1.9956e-01j, -0.7784-4.4779e-03j,  0.0538-4.57
51e-02j],
        [ 0.0222+1.8676e-01j, -0.0034+7.8565e-02j, -0.9790-2.57
26e-18j]],
       dtype=torch.complex128)
```

QR Decomposition

Quite common, where we decompose a square matrix into an orthogonal matrix and an upper triangular matrix

In [42]:
```
Q, R = th.linalg.qr(B)
```

In [43]:
```
Q
```

Out[43]:
```
tensor([[-0.7586+0.0000e+00j, -0.5884+2.0440e-01j,  0.0013+1.91
22e-01j],
        [ 0.5910+1.9956e-01j, -0.7784-4.4779e-03j,  0.0538-4.57
51e-02j],
        [ 0.0222+1.8676e-01j, -0.0034+7.8565e-02j, -0.9790-2.57
26e-18j]],
       dtype=torch.complex128)
```

In [44]:
```
R
```

Out[44]:
```
tensor([[-2.6091+0.0000j,  4.2662-1.4623j,  0.1932-1.0462j],
        [ 0.0000+0.0000j, -2.8796+0.0000j, -0.1630+0.0265j],
        [ 0.0000+0.0000j,  0.0000+0.0000j, -1.3463+0.0000j]],
       dtype=torch.complex128)
```

Again, lets double check our results

Again, lets double check our results

In [45]:
```
th.allclose(B, Q @ R)
```

Out[45]:
```
True
```

Eigenvalue decomposition

Pytorch even has various flavors of *Eigenvalue decomposition*!

Eigenvalue decomposition

Pytorch even has various flavors of *Eigenvalue decomposition*!

In [46]:
```
eigvals, eigvecs = th.linalg.eig(B)
```

Eigenvalue decomposition

Pytorch even has various flavors of *Eigenvalue decomposition*!

In [46]:
```
eigvals, eigvecs = th.linalg.eig(B)
```

In [47]:
```
eigvals
```

Out[47]:
```
tensor([5.9008-2.0839e-16j, 1.1233-9.1986e-18j, 1.5260-4.4529e-
18j],
       dtype=torch.complex128)
```

In [48]:

```
eigval_matrix = th.diag_embed(eigvals) # convert a 1D array to an equiva
```

In [48]:
```
eigval_matrix = th.diag_embed(eigvals) # convert a 1D array to an equiva
```

In [49]:
```
eigval_matrix
```

Out[49]:
```
tensor([[5.9008-2.0839e-16j, 0.0000+0.0000e+00j, 0.0000+0.0000
e+00j],
        [0.0000+0.0000e+00j, 1.1233-9.1986e-18j, 0.0000+0.0000
e+00j],
        [0.0000+0.0000e+00j, 0.0000+0.0000e+00j, 1.5260-4.4529e
-18j]],
       dtype=torch.complex128)
```

In [48]:
```
eigval_matrix = th.diag_embed(eigvals) # convert a 1D array to an equiva
```

In [49]:
```
eigval_matrix
```

Out[49]:
```
tensor([[5.9008-2.0839e-16j, 0.0000+0.0000e+00j, 0.0000+0.0000
e+00j],
        [0.0000+0.0000e+00j, 1.1233-9.1986e-18j, 0.0000+0.0000
e+00j],
        [0.0000+0.0000e+00j, 0.0000+0.0000e+00j, 1.5260-4.4529e
-18j]],
       dtype=torch.complex128)
```

In [50]:
```
eigvecs
```

Out[50]:
```
tensor([[-0.3743+0.1294j,  0.7604+0.0000j,  0.0720+0.5097j],
        [ 0.9006+0.0000j,  0.2200+0.0354j, -0.1263+0.3511j],
        [ 0.0969+0.1504j, -0.0510+0.6079j,  0.7719+0.0000j]],
       dtype=torch.complex128)
```

In [48]:
```
eigval_matrix = th.diag_embed(eigvals) # convert a 1D array to an equiva
```

In [49]:
```
eigval_matrix
```

Out[49]:
```
tensor([[5.9008-2.0839e-16j, 0.0000+0.0000e+00j, 0.0000+0.0000
e+00j],
        [0.0000+0.0000e+00j, 1.1233-9.1986e-18j, 0.0000+0.0000
e+00j],
        [0.0000+0.0000e+00j, 0.0000+0.0000e+00j, 1.5260-4.4529e
-18j]],
       dtype=torch.complex128)
```

In [50]:
```
eigvecs
```

Out[50]:
```
tensor([[-0.3743+0.1294j,  0.7604+0.0000j,  0.0720+0.5097j],
        [ 0.9006+0.0000j,  0.2200+0.0354j, -0.1263+0.3511j],
        [ 0.0969+0.1504j, -0.0510+0.6079j,  0.7719+0.0000j]],
       dtype=torch.complex128)
```

In [51]:
```
th.allclose(eigvecs @ eigval_matrix @ th.linalg.inv(eigvecs) , B) #Check
```

Out[51]:
```
True
```

Solve system of linear equations

Solve

Solve system of linear equations

Solve

In [52]:
```
A = th.randn(N, N, dtype=th.float32)
b = th.ones(N, dtype=th.float32)
print("A is: ", A)
print("b is: ", b)
```

```
A is:  tensor([[-0.5344,  0.2178,  0.8357],
        [-1.3800,  0.2744,  0.1503],
        [-0.9498, -0.3696,  1.1155]])
b is:  tensor([1., 1., 1.])
```

Solve system of linear equations

Solve

In [52]:
```
A = th.randn(N, N, dtype=th.float32)
b = th.ones(N, dtype=th.float32)
print("A is: ", A)
print("b is: ", b)
```

```
A is:  tensor([[-0.5344,  0.2178,  0.8357],
        [-1.3800,  0.2744,  0.1503],
        [-0.9498, -0.3696,  1.1155]])
b is:  tensor([1., 1., 1.])
```

In [53]:
```
x = th.linalg.solve(A, b)
print("Solution x is: ", x)
```

```
Solution x is:  tensor([-0.5123,  0.6906,  0.6890])
```

Solve system of linear equations

Solve

In [52]:
```
A = th.randn(N, N, dtype=th.float32)
b = th.ones(N, dtype=th.float32)
print("A is: ", A)
print("b is: ", b)
```

```
A is:  tensor([[-0.5344,  0.2178,  0.8357],
        [-1.3800,  0.2744,  0.1503],
        [-0.9498, -0.3696,  1.1155]])
b is:  tensor([1., 1., 1.])
```

In [53]:
```
x = th.linalg.solve(A, b)
print("Solution x is: ", x)
```

```
Solution x is:  tensor([-0.5123,  0.6906,  0.6890])
```

In [54]:
```
th.allclose(A @ x, b) # verify
```

Out[54]:
```
True
```

Fourier and inverse Fourier Transforms

Lets use from the previous example

Fourier and inverse Fourier Transforms

Lets use from the previous example

In [55]:
```
x_fft = th.fft.fft(x)
x_fft_inverse = th.fft.ifft(x_fft)
```

Fourier and inverse Fourier Transforms

Lets use from the previous example

In [55]:
```
x_fft = th.fft.fft(x)
x_fft_inverse = th.fft.ifft(x_fft)
```

In [56]:
```
print("real domain: ", x)
print("Fourier domain: ", x_fft)
print("Inverse of Fourier domain: ", x_fft_inverse)
print(th.allclose(th.real(x_fft_inverse),x))
```

```
real domain:  tensor([-0.5123,  0.6906,  0.6890])
Fourier domain:  tensor([ 0.8673+0.0000j, -1.2021-0.0013j, -1.20
21+0.0013j])
Inverse of Fourier domain:  tensor([-0.5123+0.j,  0.6906+0.j,
0.6890+0.j])
True
```

Sparse Matrix Operations

Pytorch has the ability to take advantage of performance benefits when performing sparse matrix operations with `th.sparse`

Lets define a dense matrix with very few non-zero elements and convert it into a sparse matrix

Sparse Matrix Operations

Pytorch has the ability to take advantage of performance benefits when performing sparse matrix operations with `th.sparse`

Lets define a dense matrix with very few non-zero elements and convert it into a sparse matrix

In [57]:
```
# Create arbitrary diagonal matrices so there are many "zero" elements
a = th.diag_embed(th.tensor([1.0,1.0,1.0,1.0,1.0,1.0,1.0]))
b  = th.diag_embed(th.randn(7))
print('a is: ', a)
print('b is: ', b)
```

```
a is:  tensor([[1., 0., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0., 0.],
        [0., 0., 0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 0., 0., 1.]])
b is:  tensor([[ 0.5963,  0.0000,  0.0000,  0.0000,  0.0000,  0.
0000,  0.0000],
        [ 0.0000,  0.9775,  0.0000,  0.0000,  0.0000,  0.0000,
0.0000],
        [ 0.0000,  0.0000, -0.1606,  0.0000,  0.0000,  0.0000,
0.0000],
        [ 0.0000,  0.0000,  0.0000, -0.7291,  0.0000,  0.0000,
0.0000],
        [ 0.0000,  0.0000,  0.0000,  0.0000,  1.0850,  0.0000,
0.0000],
        [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  2.0574,
0.0000],
        [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,
0.9218]])
```

Convert diagonal matrices to CSR sparse layout

https://en.wikipedia.org
/wiki/Sparse_matrix#Compressed_sparse_row_(CSR,_CRS_or_Yale_format)

Convert diagonal matrices to CSR sparse layout

https://en.wikipedia.org
/wiki/Sparse_matrix#Compressed_sparse_row_(CSR,_CRS_or_Yale_format)

In [58]:
```
sp_a = a.to_sparse_csr()
sp_b = b.to_sparse_csr()
```

```
/var/folders/01/5fs_12112d51__2md2crbk70000tpn/T/ipykernel_71682
/1102660417.py:1: UserWarning: Sparse CSR tensor support is in b
eta state. If you miss a functionality in the sparse tensor supp
ort, please submit a feature request to https://github.com/pytor
ch/pytorch/issues. (Triggered internally at /Users/runner/work/_
temp/anaconda/conda-bld/pytorch_1711403226120/work/aten/src/ATen
/SparseCsrTensorImpl.cpp:55.)
  sp_a = a.to_sparse_csr()
```

Convert diagonal matrices to CSR sparse layout

https://en.wikipedia.org
/wiki/Sparse_matrix#Compressed_sparse_row_(CSR,_CRS_or_Yale_format)

In [58]:
```
sp_a = a.to_sparse_csr()
sp_b = b.to_sparse_csr()
```

```
/var/folders/01/5fs_12112d51__2md2crbk70000tpn/T/ipykernel_71682
/1102660417.py:1: UserWarning: Sparse CSR tensor support is in b
eta state. If you miss a functionality in the sparse tensor supp
ort, please submit a feature request to https://github.com/pytor
ch/pytorch/issues. (Triggered internally at /Users/runner/work/_
temp/anaconda/conda-bld/pytorch_1711403226120/work/aten/src/ATen
/SparseCsrTensorImpl.cpp:55.)
  sp_a = a.to_sparse_csr()
```

In [59]:
```
sp_a,sp_b
```

Out[59]:

```
(tensor(crow_indices=tensor([0, 1, 2, 3, 4, 5, 6, 7]),
        col_indices=tensor([0, 1, 2, 3, 4, 5, 6]),
        values=tensor([1., 1., 1., 1., 1., 1., 1.]), size=(7,
7), nnz=7,
        layout=torch.sparse_csr),
 tensor(crow_indices=tensor([0, 1, 2, 3, 4, 5, 6, 7]),
        col_indices=tensor([0, 1, 2, 3, 4, 5, 6]),
        values=tensor([ 0.5963,  0.9775, -0.1606, -0.7291,  1.0
850,  2.0574,
                        0.9218]), size=(7, 7), nnz=7, layout=to
rch.sparse_csr))
```

Perform sparse matrix multiplication with `th.matmul` or the `@` operator:

Perform sparse matrix multiplication with `th.matmul` or the `@` operator:

In [60]:
```
sp_matmul = sp_a @ sp_b
print(sp_matmul)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[60], line 1
----> 1 sp_matmul = sp_a @ sp_b
      2 print(sp_matmul)

RuntimeError: addmm: computation on CPU is not implemented for SparseCsr + SparseCsr @ SparseCsr without MKL. PyTorch built with MKL has better support for addmm with sparse CPU tensors.
```

Perform sparse matrix multiplication with `th.matmul` or the `@` operator:

In [60]:
```
sp_matmul = sp_a @ sp_b
print(sp_matmul)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[60], line 1
----> 1 sp_matmul = sp_a @ sp_b
      2 print(sp_matmul)

RuntimeError: addmm: computation on CPU is not implemented for SparseCsr + SparseCsr @ SparseCsr without MKL. PyTorch built with MKL has better support for addmm with sparse CPU tensors.
```

Alas, this is an example of missing support for some platforms (MKL not available for ARM)

Convert back to dense to print out the full matrix with built-in converter

Convert back to dense to print out the full matrix with built-in converter

In [61]:
```
dense_a = sp_a.to_dense()
print('Matmul solution: ', dense_a)
```

```
Matmul solution:  tensor([[1., 0., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0., 0.],
        [0., 0., 0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 0., 0., 1.]])
```

These are just a few examples: Check Pytorch documentation for a full list of capabilities!

But why does a DL library like Pytorch have these capabilities, you ask? **That is because you can even embed these operations inside your neural networks and train them seamlessly!** This is more than just doing math (which you are welcome to do), but highly customizable machine learning models.

Automatic Differentiation in PyTorch

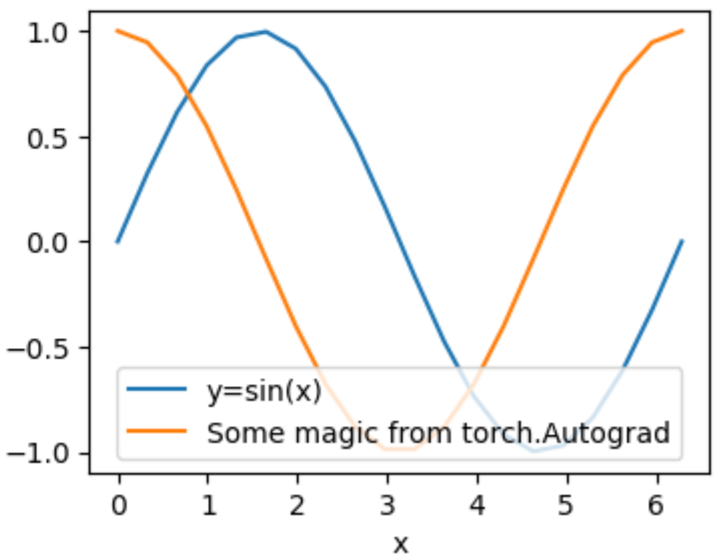
- Here we will explore how PyTorch is able to easily perform the gradient operations needed for training the network
- There are a few ways to actually implement backwards automatic differentiation - I will focus on how it is done in pytorch. The principles are always similar.

In [63]:
```
## This package was INCREDIBLY helpful for this tutorial.
## I want to emphasize that the plots shown here are made by
## live inspection of python objects in memory.
from torchviz import make_dot

######                          V This is what we want to learn about
from torch.autograd import grad
```

Let's revisit the example from the first part:

Let's revisit the example from the first part:

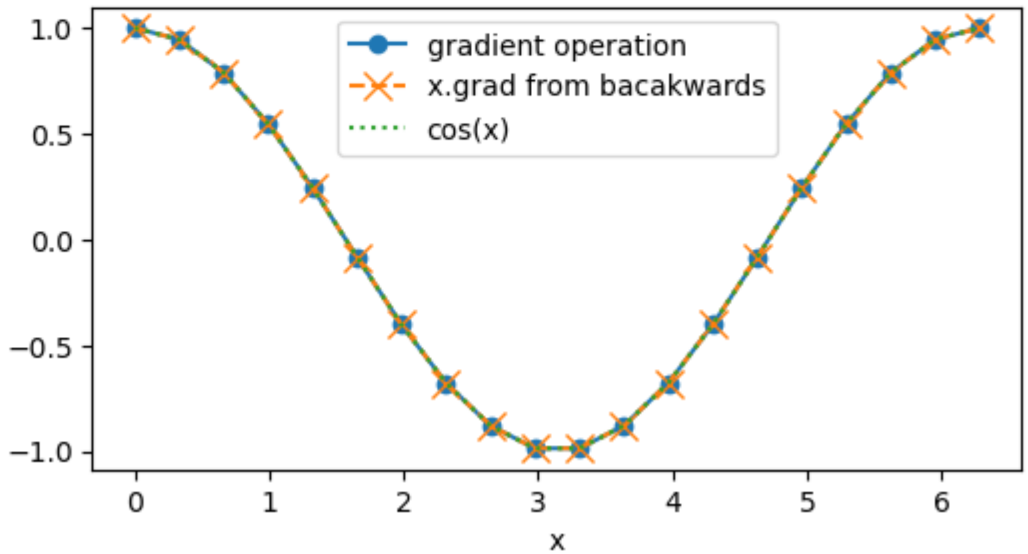
In [64]:
```
x = th.linspace(0,2*np.pi,20)
x.requires_grad_(True) # <- What is this?
y = th.sin(x)
y_prime = grad(y.sum(),x)[0]
fig, ax = plt.subplots(1,1,figsize=(4,3))
plt.plot(x.detach().numpy(),y.detach().numpy(),label="y=sin(x)")
plt.xlabel("x")
plt.plot(x.detach().numpy(),y_prime.numpy(),label="Some magic from torch
plt.legend(loc=8)
plt.show()
```



Taking the gradient on y.sum() gives us a tensor that looks like the derivative of x Let's compare this against a `.backward()` call.

Taking the gradient on y.sum() gives us a tensor that looks like the derivative of x Let's compare this against a `.backward()` call.

In [65]:
```
x = th.linspace(0,2*np.pi,20)
x.requires_grad_(True) # <- ???
y = th.sin(x)

y.sum().backward()

plt.figure(figsize=(6,3))
plt.plot(x.detach().numpy(),y_prime.detach().numpy(),label="gradient ope
plt.plot(x.detach().numpy(),x.grad.numpy(),label="x.grad from bacakwards
plt.plot(x.detach().numpy(),np.cos(x.detach().numpy()),label="cos(x)",ls
plt.xlabel("x")
plt.legend()
plt.show()
```



gradient operation
x.grad from bacakwards
cos(x)
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6
x

In [66]:
```python
print("x.grad:",x.grad)
print("y_prime:",y_prime)
print("x.grad is y_prime:",x.grad is y_prime)
```

```
x.grad: tensor([ 1.0000,  0.9458,  0.7891,  0.5469,  0.2455, -0.
0826, -0.4017, -0.6773,
        -0.8795, -0.9864, -0.9864, -0.8795, -0.6773, -0.4017, -
0.0826,  0.2455,
         0.5469,  0.7891,  0.9458,  1.0000])
y_prime: tensor([ 1.0000,  0.9458,  0.7891,  0.5469,  0.2455, -
0.0826, -0.4017, -0.6773,
        -0.8795, -0.9864, -0.9864, -0.8795, -0.6773, -0.4017, -
0.0826,  0.2455,
         0.5469,  0.7891,  0.9458,  1.0000])
x.grad is y_prime: False
```

In [66]:

```
print("x.grad:",x.grad)
print("y_prime:",y_prime)
print("x.grad is y_prime:",x.grad is y_prime)
```

```
x.grad: tensor([ 1.0000,  0.9458,  0.7891,  0.5469,  0.2455, -0.
0826, -0.4017, -0.6773,
        -0.8795, -0.9864, -0.9864, -0.8795, -0.6773, -0.4017, -
0.0826,  0.2455,
         0.5469,  0.7891,  0.9458,  1.0000])
y_prime: tensor([ 1.0000,  0.9458,  0.7891,  0.5469,  0.2455, -
0.0826, -0.4017, -0.6773,
        -0.8795, -0.9864, -0.9864, -0.8795, -0.6773, -0.4017, -
0.0826,  0.2455,
         0.5469,  0.7891,  0.9458,  1.0000])
x.grad is y_prime: False
```

So `backward` and `grad` are in this example doing something similar to each other, which in this example produces `cos(x)` , but as different results. We'll cover the differences later.

What's up with that `detach` stuff anyway?

Let's see what happens if we just skip it:

What's up with that `detach` stuff anyway?

Let's see what happens if we just skip it:

In [67]:
```
x.numpy()
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[67], line 1
----> 1 x.numpy()

RuntimeError: Can't call numpy() on Tensor that requires grad. Use tensor.detach().numpy() instead.
```

We got a relatively helpful error message that tells us how to "solve" the problem.

But why is it there?

To prevent accidentally breaking the automatic differentiation features.

Autograd requires *tracking* what operations are applied to differentiable variables. When you convert to numpy, without the extra metadata from pytorch, the operations on the array can't be tracked.

You can also use `x.data`. I recommend against getting in such a habit: x.data is *unsafe* with respect to autograd.

Views in autograd

Different copies of the same information can be created.

Views in autograd

Different copies of the same information can be created.

In [68]:
```
x = th.arange(3,dtype=th.get_default_dtype()).requires_grad_(True)
print(x) # x viewed from "inside of the autograd system"
print(x.clone()) # A fresh copy of x that is "inside the autograd systen
print(x.data) # x viewed from "outside of the autograd system"
print('pointers:',x.data_ptr(),x.clone().data_ptr(),x.data.data_ptr())
```

```
tensor([0., 1., 2.], requires_grad=True)
tensor([0., 1., 2.], grad_fn=<CloneBackward0>)
tensor([0., 1., 2.])
pointers: 5158864064 4620276416 5158864064
```

Views in autograd

Different copies of the same information can be created.

In [68]:
```
x = th.arange(3,dtype=th.get_default_dtype()).requires_grad_(True)
print(x) # x viewed from "inside of the autograd system"
print(x.clone()) # A fresh copy of x that is "inside the autograd systen
print(x.data) # x viewed from "outside of the autograd system"
print('pointers:',x.data_ptr(),x.clone().data_ptr(),x.data.data_ptr())
```

```
tensor([0., 1., 2.], requires_grad=True)
tensor([0., 1., 2.], grad_fn=<CloneBackward0>)
tensor([0., 1., 2.])
pointers: 5158864064 4620276416 5158864064
```

So ultimately, they all point at the same memory location. It's just the metadata that is different.

Unused inputs

Unused inputs

In [69]:

```
x = th.Tensor([1,2,3]).requires_grad_(True)
x2 = x.clone()
y = x.sum()
```

Unused inputs

In [69]:
```
x = th.Tensor([1,2,3]).requires_grad_(True)
x2 = x.clone()
y = x.sum()
```

In [70]:
```
grad(y,[x,x2])
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[70], line 1
----> 1 grad(y,[x,x2])

File ~/opt/miniconda3/envs/torch_tutorial/lib/python3.12/site-packages/torch/autograd/__init__.py:411, in grad(outputs, inputs, grad_outputs, retain_graph, create_graph, only_inputs, allow_unused, is_grads_batched, materialize_grads)
    407     result = _vmap_internals._vmap(vjp, 0, 0, allow_none_pass_through=True)(
    408         grad_outputs_
    409     )
    410 else:
--> 411     result = Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the backward pass
    412         t_outputs,
    413         grad_outputs_,
    414         retain_graph,
    415         create_graph,
    416         inputs,
    417         allow_unused,
    418         accumulate_grad=False,
    419     )  # Calls into the C++ engine to run the backward pass
    420 if materialize_grads:
    421     if any(
    422         result[i] is None and not is_tensor_like(inputs[i])
```

Okay, let's skip the traceback which is unnecessarily long.

Okay, let's skip the traceback which is unnecessarily long.

```
In [72]: with suppress_traceback():
             grad(y,[x,x2])
```

```
RuntimeError: One of the differentiated Tensors appears to not h
ave been used in the graph. Set allow_unused=True if this is the
desired behavior.
```

Okay, let's skip the traceback which is unnecessarily long.

In [72]:
```
with suppress_traceback():
    grad(y,[x,x2])
```

```
RuntimeError: One of the differentiated Tensors appears to not h
ave been used in the graph. Set allow_unused=True if this is the
desired behavior.
```

This returns an error because `x2` has been detached from autograd, so it doesn't appear to have anything to do with `y`.

Note: suppress_traceback is just a convenience function defined in the notebook version of these slides. It only skips the full traceback by printing the error message directly.

Allow_unused

The `allowed_unused` flag will tell pytorch to forgive you for taking the gradients that are ill-defined

Allow_unused

The `allowed_unused` flag will tell pytorch to forgive you for taking the gradients that are ill-defined

In [73]:

```
gradient_x, gradient_x2 = grad(y,[x,x2],allow_unused=True)
print("gradient_x:",gradient_x)
print("gradient_x2:",gradient_x2)
```

```
gradient_x: tensor([1., 1., 1.])
gradient_x2: None
```

Allow_unused

The `allowed_unused` flag will tell pytorch to forgive you for taking the gradients that are ill-defined

In [73]:
```
gradient_x, gradient_x2 = grad(y,[x,x2],allow_unused=True)
print("gradient_x:",gradient_x)
print("gradient_x2:",gradient_x2)
```

```
gradient_x: tensor([1., 1., 1.])
gradient_x2: None
```

If something is not part of the computation and we pass `allow_unused=True` , the gradient is simply `None`

The autograd graph

The autograd graph

It is possible to inspect how functions in autograd are linked together.

The autograd graph

It is possible to inspect how functions in autograd are linked together.

In [74]:
```
x = th.Tensor([1,2,3]).requires_grad_(True)
y = x.sum()
print('y:',y)
## `y.grad_fn` is a container for autograd
print('y.grad_fn :',y.grad_fn)
## Among other things, it contains links to other functions that have be
print('y.grad_fn.next_functions :',y.grad_fn.next_functions)
```

```
y: tensor(6., grad_fn=<SumBackward0>)
y.grad_fn : <SumBackward0 object at 0x137d77f70>
y.grad_fn.next_functions : ((<AccumulateGrad object at 0x137d62f
e0>, 0),)
```

The autograd graph

It is possible to inspect how functions in autograd are linked together.

In [74]:
```
x = th.Tensor([1,2,3]).requires_grad_(True)
y = x.sum()
print('y:',y)
## `y.grad_fn` is a container for autograd
print('y.grad_fn :',y.grad_fn)
## Among other things, it contains links to other functions that have be
print('y.grad_fn.next_functions :',y.grad_fn.next_functions)
```

```
y: tensor(6., grad_fn=<SumBackward0>)
y.grad_fn : <SumBackward0 object at 0x137d77f70>
y.grad_fn.next_functions : ((<AccumulateGrad object at 0x137d62f
e0>, 0),)
```

We could try to untangle this manually. However, this will be a cumbersome way to investigate. Part of the difficulty is that a lot of autograd is written in C++ rather than Python.

Visualizing the autograd graph

Luckily the `torchviz` package has come to the rescue with `torchviz.make_dot()`

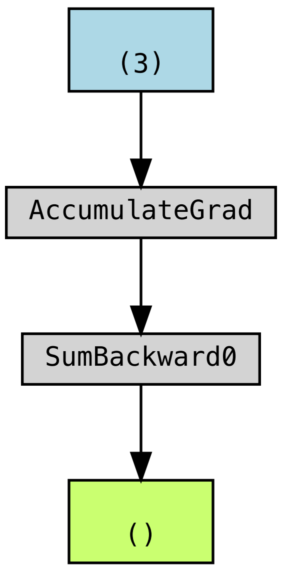
Visualizing the autograd graph

Luckily the `torchviz` package has come to the rescue with `torchviz.make_dot()`

In [75]:

```
x = th.Tensor([1,2,3]).requires_grad_(True)
y = x.sum()
make_dot(y)
```

Out[75]:



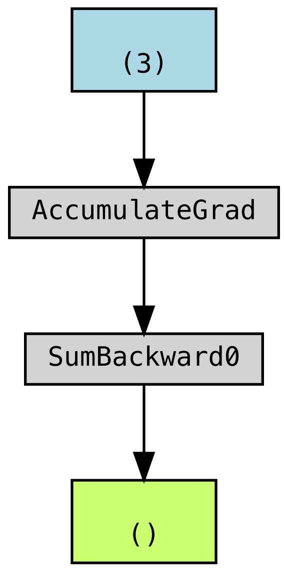
Visualizing the autograd graph

Luckily the `torchviz` package has come to the rescue with `torchviz.make_dot()`

In [75]:

```
x = th.Tensor([1,2,3]).requires_grad_(True)
y = x.sum()
make_dot(y)
```

Out[75]:



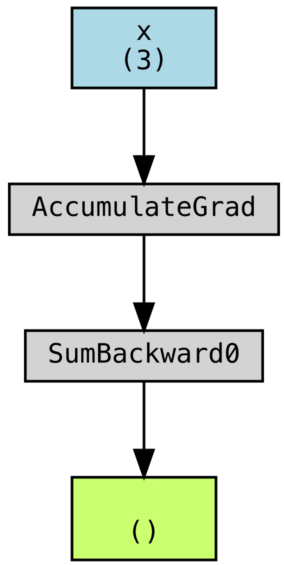
- Blue boxes are inputs.
- The green box is the thing we asked to visualize
    - In this case, y was the sum of x.

We can annotate tensors with names to understand this graph better.

We can annotate tensors with names to understand this graph better.

In [76]:
```
make_dot(y,params={"x":x})
```

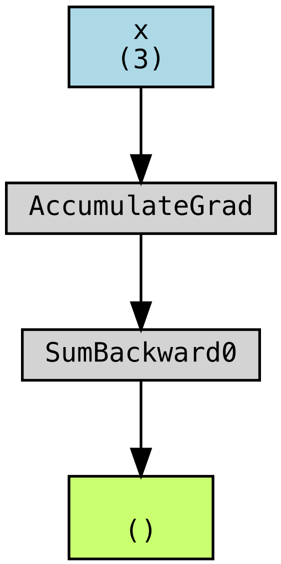
Out[76]:



We can annotate tensors with names to understand this graph better.

In [76]:
```
make_dot(y,params={"x":x})
```

Out[76]:

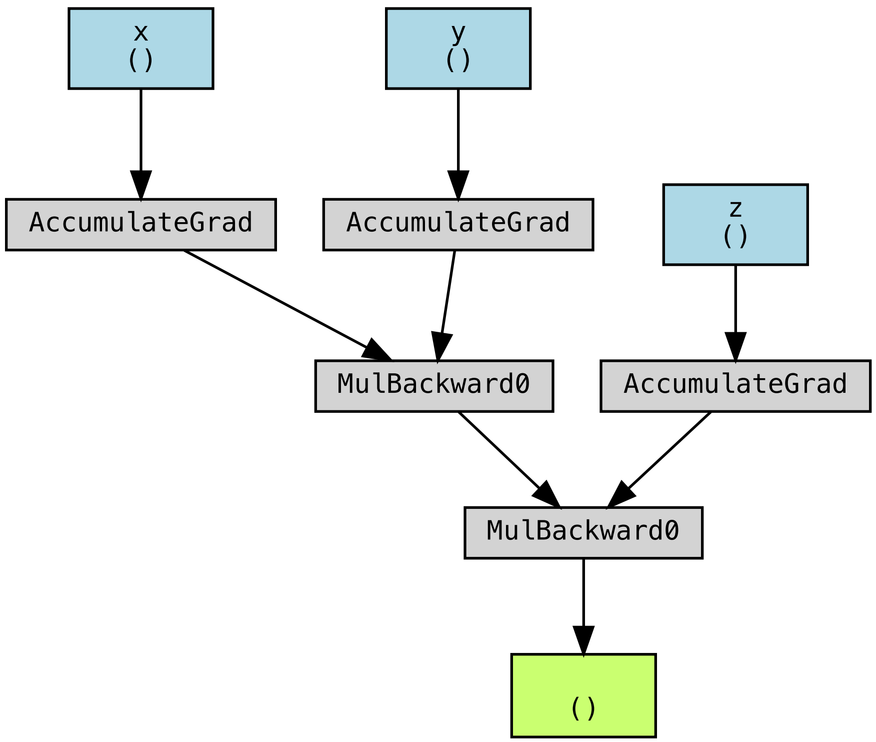


(Digression: why is it necessary to add a name dict directly?)

In [77]:

```
x,y,z = th.rand(3)
x.requires_grad_(True)
y.requires_grad_(True)
z.requires_grad_(True)
x_clone = x.clone()
w = (x*y)*z
make_dot(w,{'x':x,'y':y,'z':z,'x_clone':x_clone})
```

Out[77]:



- Blue boxes are inputs.
- *Grey boxes are intermediate operations*
    - We need to save their values in memory in order to calculate gradients later.
- The green box is the thing we asked to visualize.

Tip: Because of the interrmedate storage, sometimes you can cause memory leaks by performing too many options that require gradient. The solution is to use the `torch.autograd.no_grad` context manager

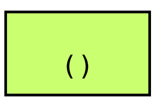
torch.autograd.no_grad

torch.autograd.no_grad

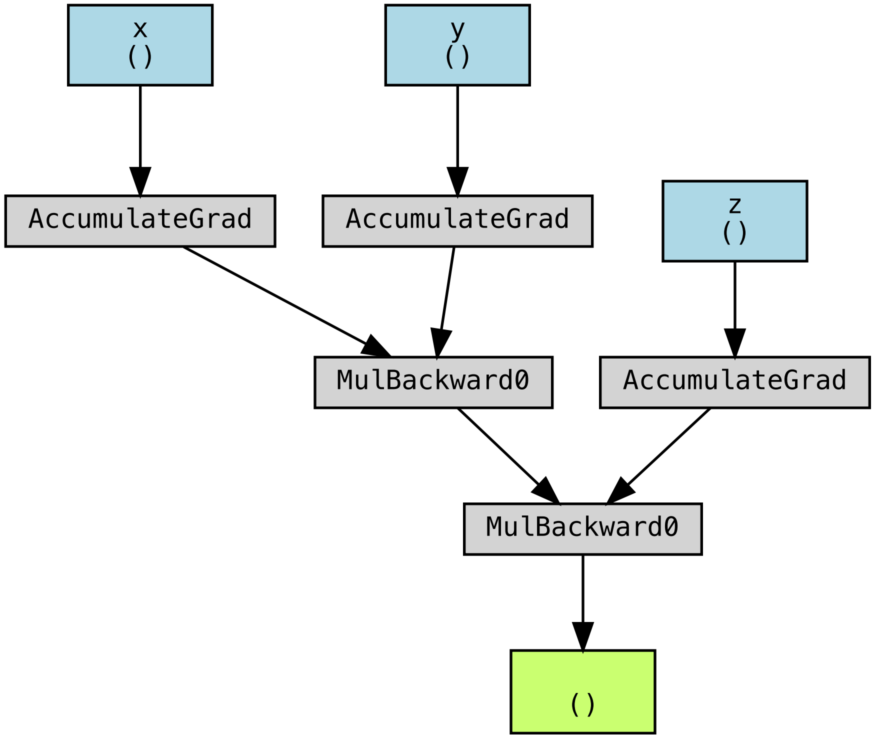
In [78]:

```
x,y,z = th.rand(3)

x.requires_grad_(True)
y.requires_grad_(True)
z.requires_grad_(True)

with th.autograd.no_grad():
    w = (x*y)*z
v = (x*y)*z



print("Graph for w:")
display(make_dot(w,{'x':x,'y':y,'z':z}))
print("Graph for v:")
display(make_dot(v,{'x':x,'y':y,'z':z}))
```

```
Graph for w:
```




```
Graph for v:
```

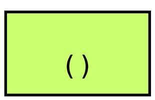

x
()
y
()
AccumulateGrad
AccumulateGrad
z
()
MulBackward0
AccumulateGrad
MulBackward0
()

torch.autograd.no_grad

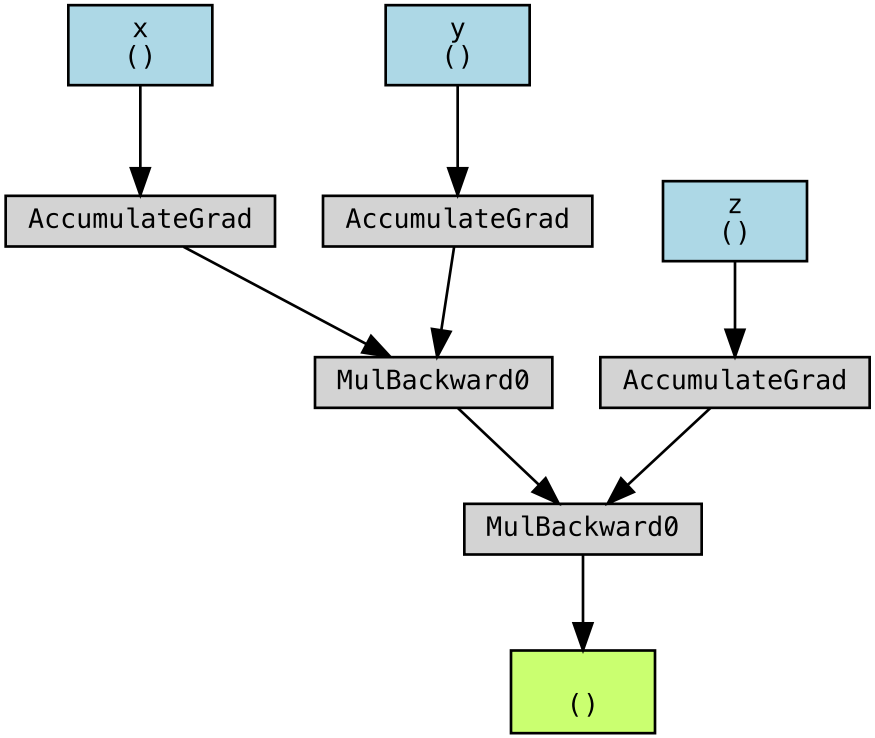
In [78]:

```
x,y,z = th.rand(3)

x.requires_grad_(True)
y.requires_grad_(True)
z.requires_grad_(True)

with th.autograd.no_grad():
    w = (x*y)*z
v = (x*y)*z


print("Graph for w:")
display(make_dot(w,{'x':x,'y':y,'z':z}))
print("Graph for v:")
display(make_dot(v,{'x':x,'y':y,'z':z}))
```

```
Graph for w:
```




```
Graph for v:
```



Within the scope of the `with` block, things that require gradient were not saved. When the with_block is complete, autograd is returned to its prior state.

```
torch.autograd.enable_grad()
```

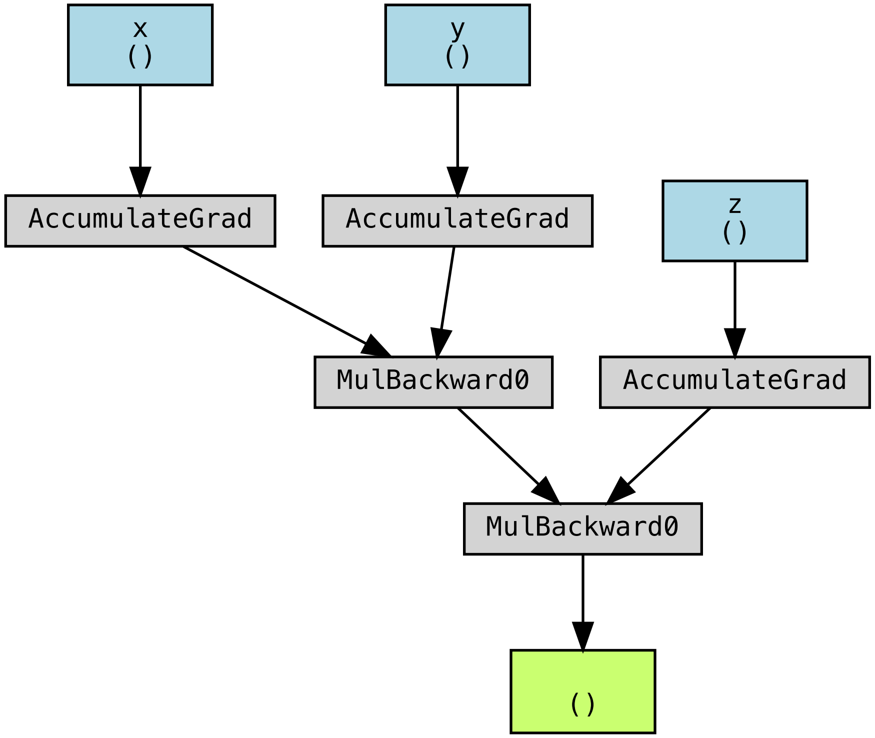
In [79]:

```
with th.autograd.no_grad():  # autograd is off in this block
    with th.autograd.enable_grad():  # autograd is on in this block
        w = (x*y)*z
    v = (x+y)  # Back to autograd off
```

Graph for W:

Graph for W:

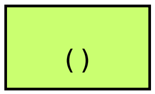
In [80]:
```
display(make_dot(w,{'x':x,'y':y,'z':z}))
```



Graph for v:

Graph for v:

In [81]:
```
display(make_dot(v,{'x':x,'y':y,'z':z}))
```



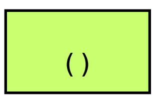
Notes:

- It is also possible to manage autograd state using `autograd.set_grad_enabled(state)` where state is a boolean. This behaves like `enable_grad` or `no_grad` respectively.
- You can use `set_grad_enabled` an imperative function call rather than as a decorator, but this will most likely be more complicated code as you have to remember to change it again if you want it to go back.
- The documentation for `no_grad` says that it has no effect when inside an `enable_grad` context. But that doesn't *seem* to be the case, as we can check:

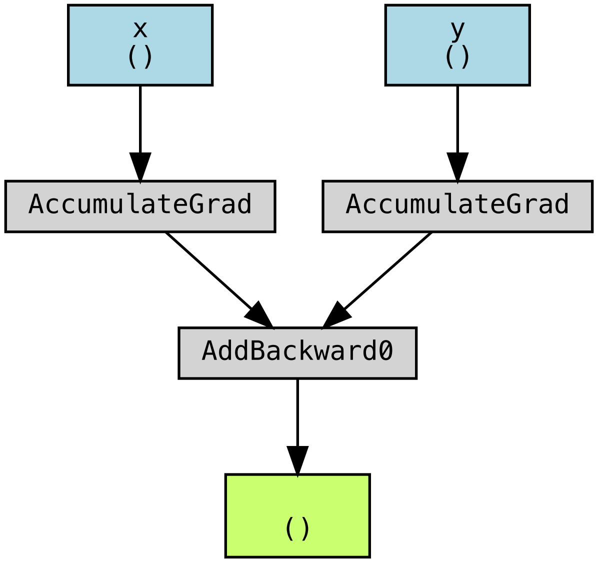
In [82]:

```
with th.autograd.enable_grad():
    ## autograd is off in this block
    with th.autograd.no_grad():
        # autograd is on in this block
        w = (x*y)*z
        print("w requires grad:",w.requires_grad)
    ## Back to autograd off
    v = (x+y)
    print("v requires grad:",w.requires_grad)

print("Graph for w:")
display(make_dot(w,{'x':x,'y':y,'z':z}))

print("Graph for v:")
display(make_dot(v,{'x':x,'y':y,'z':z}))
# autograd is on in this block
```

```
w requires grad: False
v requires grad: False
Graph for w:
```




```
Graph for v:
```


x
()
y
()
AccumulateGrad
AccumulateGrad
AddBackward0
()

Pytorch grad modes

Grad Modes

Apart from setting `requires_grad` there are also three grad modes that can be selected from Python that can affect how computations in PyTorch are processed by autograd internally: default mode (grad mode), no-grad mode, and inference mode, all of which can be togglable via context managers and decorators.

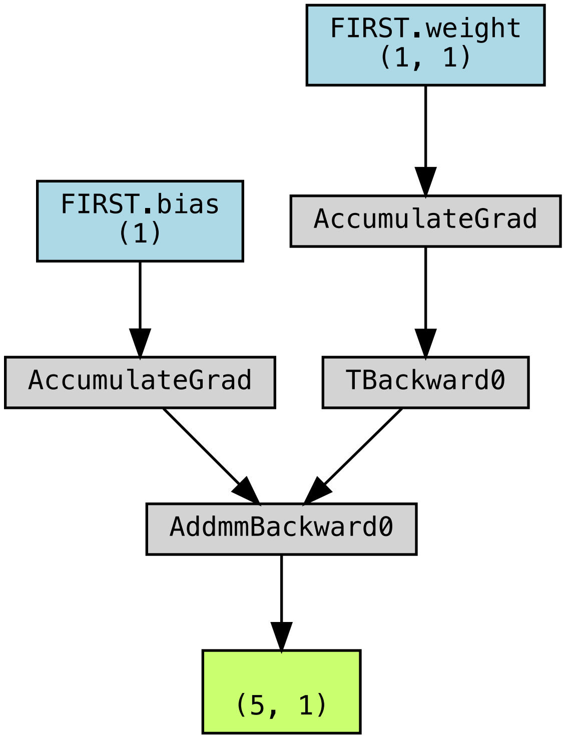
Mode	Excludes operations from being recorded in backward graph	Skips additional autograd tracking overhead	Tensors created while the mode is enabled can be used in grad-mode later	Examples
default			✓	Forward pass
no-grad	✓		✓	Optimizer updates
inference	✓	✓		Data processing, model evaluation

A basic neural net

Let's look at a simple network, that doesn't even include activations:

In [83]:
```
net = nn.Sequential()
net.add_module("FIRST",nn.Linear(1,1))

x = th.rand(5,1) # Note batch axis with batch size 5
y = net(x)

make_dot(y,params=dict(net.named_parameters()))
```

Out[83]:

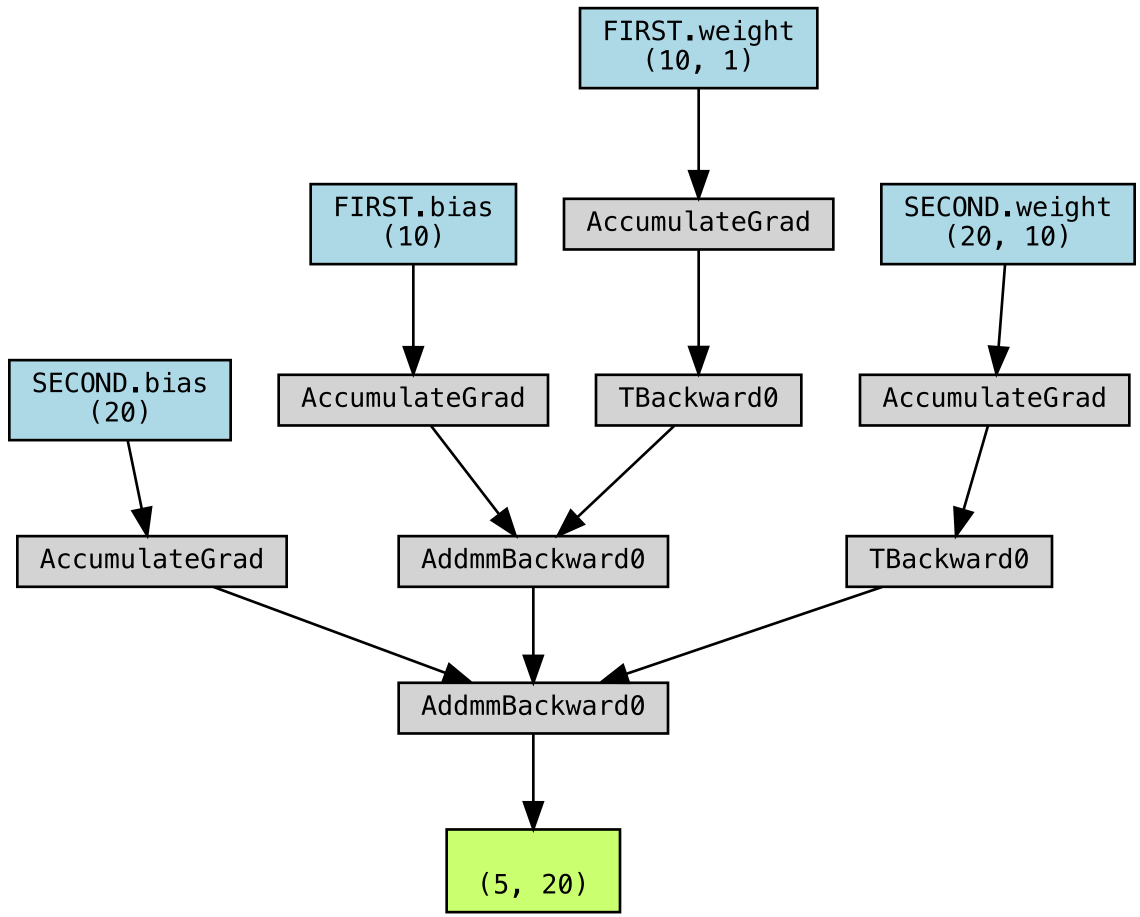

FIRST.weight
(1, 1)
FIRST.bias
(1)
AccumulateGrad
AccumulateGrad
TBackward0
AddmmBackward0
(5, 1)

- Its weights and biases appear
- The shape of parameters is shown

In [84]:

```
x = th.rand(5,1)

net = nn.Sequential()
net.add_module("FIRST",nn.Linear(1,10))
net.add_module("SECOND",nn.Linear(10,20))

y = net(x)
make_dot(y,params=dict(net.named_parameters()))
```

Out[84]:



Now we can see two weights and two biases. Let's check a many-layer network with different sized layers.

In [85]:

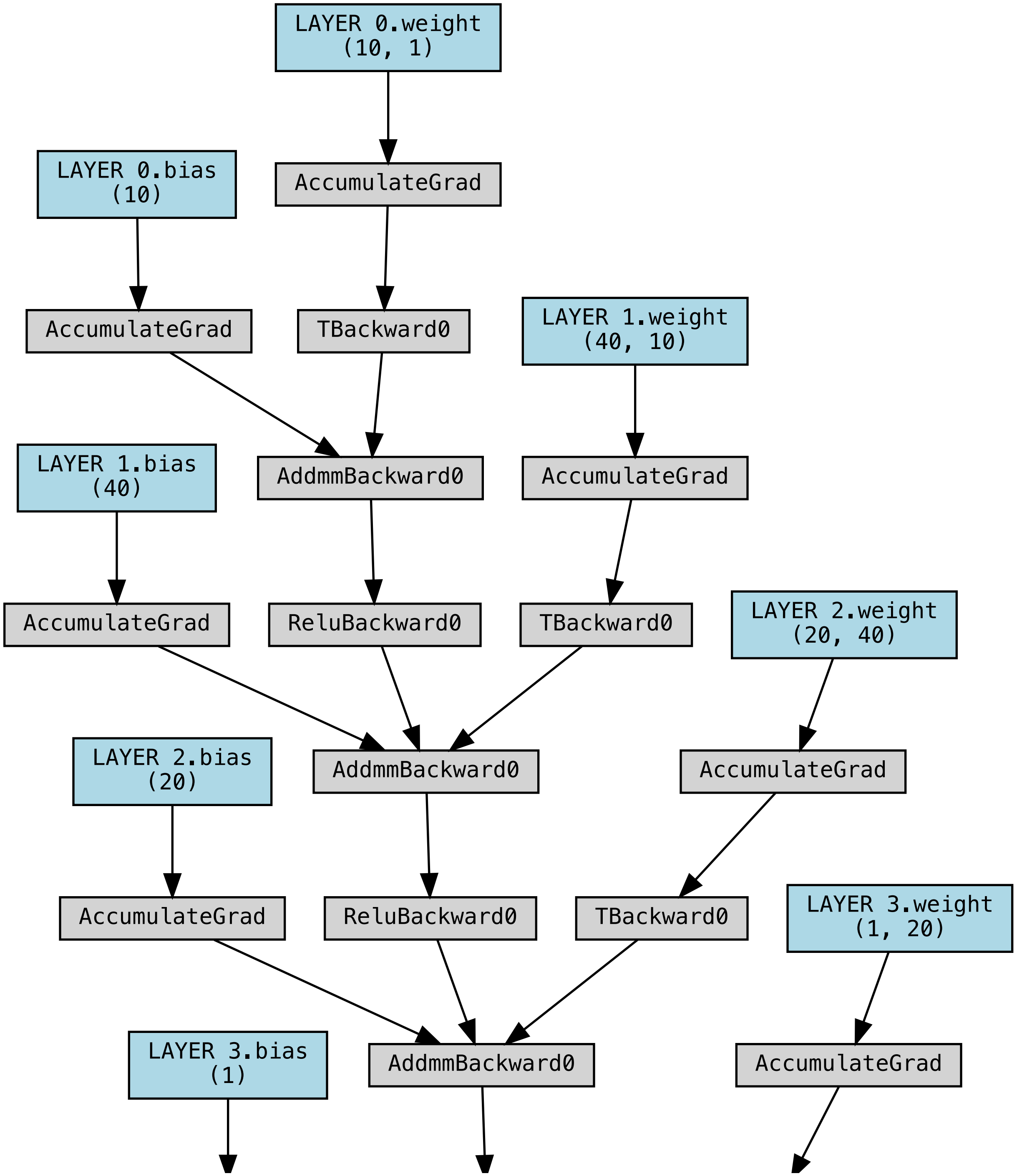
```python
x = th.rand(5,1)

net = nn.Sequential()
sizes = [1,10,40,20,1]

for i,(n_in,n_out) in enumerate(zip(sizes[:-1],sizes[1:])):
    net.add_module(f"LAYER {i}",nn.Linear(n_in,n_out))
    net.add_module(f"ACTIVATION {i}",nn.ReLU())
print(net)
y = net(x)
```

```
Sequential(
  (LAYER 0): Linear(in_features=1, out_features=10, bias=True)
  (ACTIVATION 0): ReLU()
  (LAYER 1): Linear(in_features=10, out_features=40, bias=True)
  (ACTIVATION 1): ReLU()
  (LAYER 2): Linear(in_features=40, out_features=20, bias=True)
  (ACTIVATION 2): ReLU()
  (LAYER 3): Linear(in_features=20, out_features=1, bias=True)
  (ACTIVATION 3): ReLU()
)
```

In [86]:

```
make_dot(y,params=dict(net.named_parameters()))
```

Out[86]:



LAYER 0.weight
(10, 1)
LAYER 0.bias
(10)
AccumulateGrad
AccumulateGrad
TBackward0
LAYER 1.weight
(40, 10)
LAYER 1.bias
(40)
AddmmBackward0
AccumulateGrad
AccumulateGrad
ReluBackward0
TBackward0
LAYER 2.weight
(20, 40)
LAYER 2.bias
(20)
AddmmBackward0
AccumulateGrad
AccumulateGrad
ReluBackward0
TBackward0
LAYER 3.weight
(1, 20)
LAYER 3.bias
(1)
AddmmBackward0
AccumulateGrad

Small detail: `TBackward` is a reflection of the fact that calculations are implemented as follows:

$$X^{n+1} = X^n \cdot W^T + b$$

such that the batch axis comes first in X.

Tape-based autograd

Pytorch is watching

- This form of automatic differentiation is called Tape-Based Autograd. All operations on tensors that have the `<thing>.requires_grad == True` are recorded.
- But how can this help us compute the gradients we need for training?
- How does backward relate to forward?
- What is the difference between `grad` and `.backward()` ?

The chain rule

You might remember from calculus:

Suppose:

$x(t)$

$y(t)$

$f(x, y)$

$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial x} * \frac{\partial x}{\partial t} + \frac{\partial f}{\partial y} * \frac{\partial y}{\partial t}$

Consider the following situation:

- f(x,y) is a 'height' over a 2-d landscape
- x and y are coordinate space
- x(t) and y(t) describe a path through time

How does the height (f) change over time?

It will receiving contributions due to how steep f is in the x direction ($\frac{\partial f}{\partial x}$), and the y direction ($\frac{\partial f}{\partial y}$).

It will also receiving contributions due to how fast x is changing ($\frac{\partial x}{\partial t}$) and how fast y is changing ($\frac{\partial x}{\partial t}$)

The single-variable chain rule proves that the two contributions through the path of x multiply with each other.

The multi-variable chain rule is simply an expression of the fact that *differentiation is linear,* so the contributions from each path sum together.

In [87]:

```
x = np.linspace(0,3*np.pi,100)[:,np.newaxis]
y = np.linspace(0,3*np.pi,100)
x,y = np.broadcast_arrays(x,y)

x.shape,y.shape
```

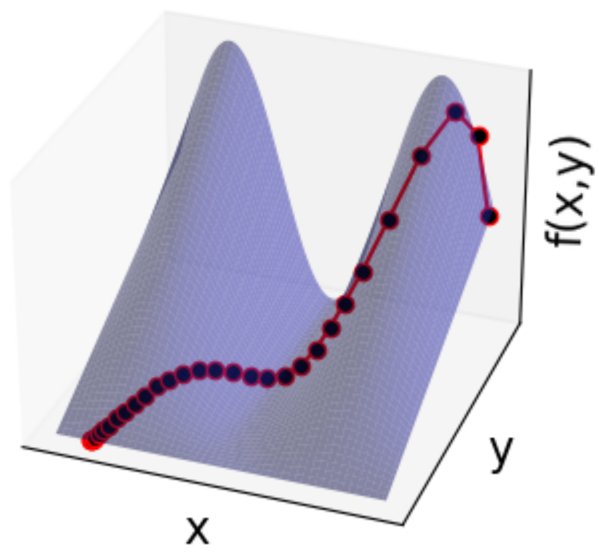
Out[87]:

```
((100, 100), (100, 100))
```

In [88]:

```
def f(x,y):
    return np.sin(x)*y + y
all_f = f(x,y)

t = np.linspace(0,5**1/2,30)**2
path_x = np.cos(t/4)+3*t/2
path_y = np.sin(t**2/4)+3*t/2
path_f = f(path_x,path_y)
```

In [90]:
```
plt.sca(ax);plt.show()
```



Consider the following situation:

- f(x,y) is a 'height' over a 2-d landscape
- x and y are coordinate space
- x(t) and y(t) describe a path through time

How does the height (f) change over time?

It will receiving contributions due to how steep f is in the x direction ($\frac{\partial f}{\partial x}$), and the y direction ($\frac{\partial f}{\partial y}$).

It will also receiving contributions due to how fast x is changing ($\frac{\partial x}{\partial t}$) and how fast y is changing ($\frac{\partial x}{\partial t}$)

The single-variable chain rule proves that the two contributions through the path of x multiply with each other.

The multi-variable chain rule is simply an expression of the fact that *differentiation is linear,* so the contributions from each path sum together.

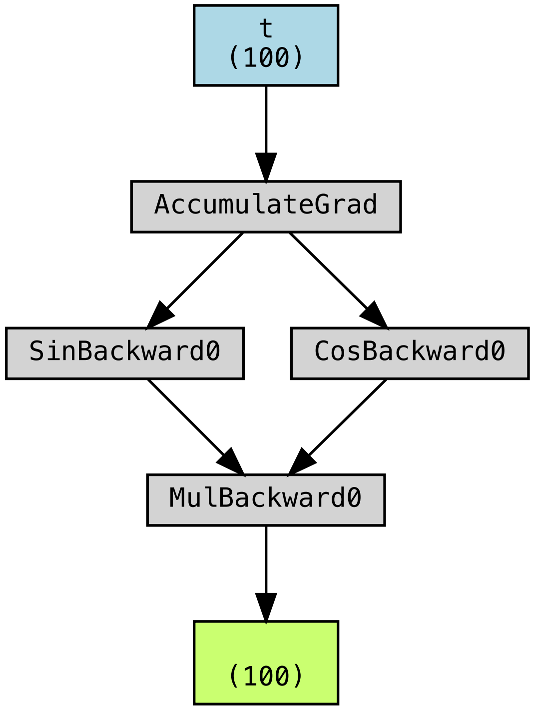
Revisiting autograd graphs

Revisiting autograd graphs

In [91]:
```
t = th.linspace(0,3,100)
t.requires_grad_(True)
x = t.sin()
y = t.cos()
f_all = x*y
```

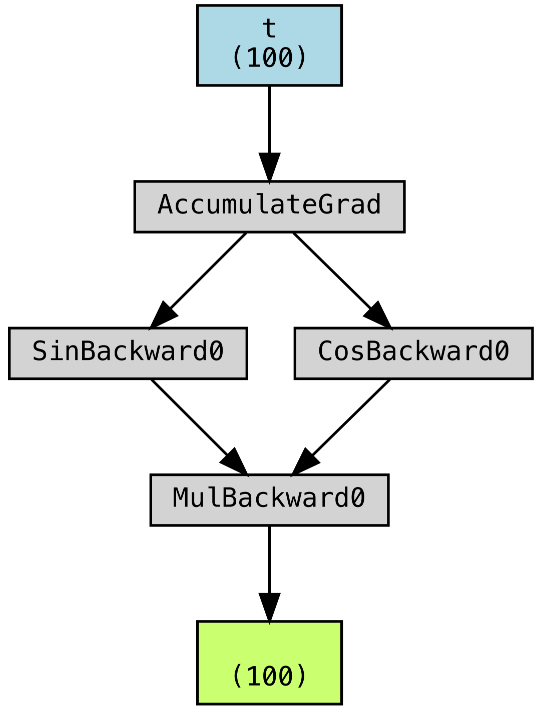
In [92]:
```
make_dot(f_all,params={"t":t})
```

Out[92]:



In [92]:
```
make_dot(f_all,params={"t":t})
```

Out[92]:



If you apply the chain rule recursively, you find that *every possible path* of multiplications contributes to the gradient.

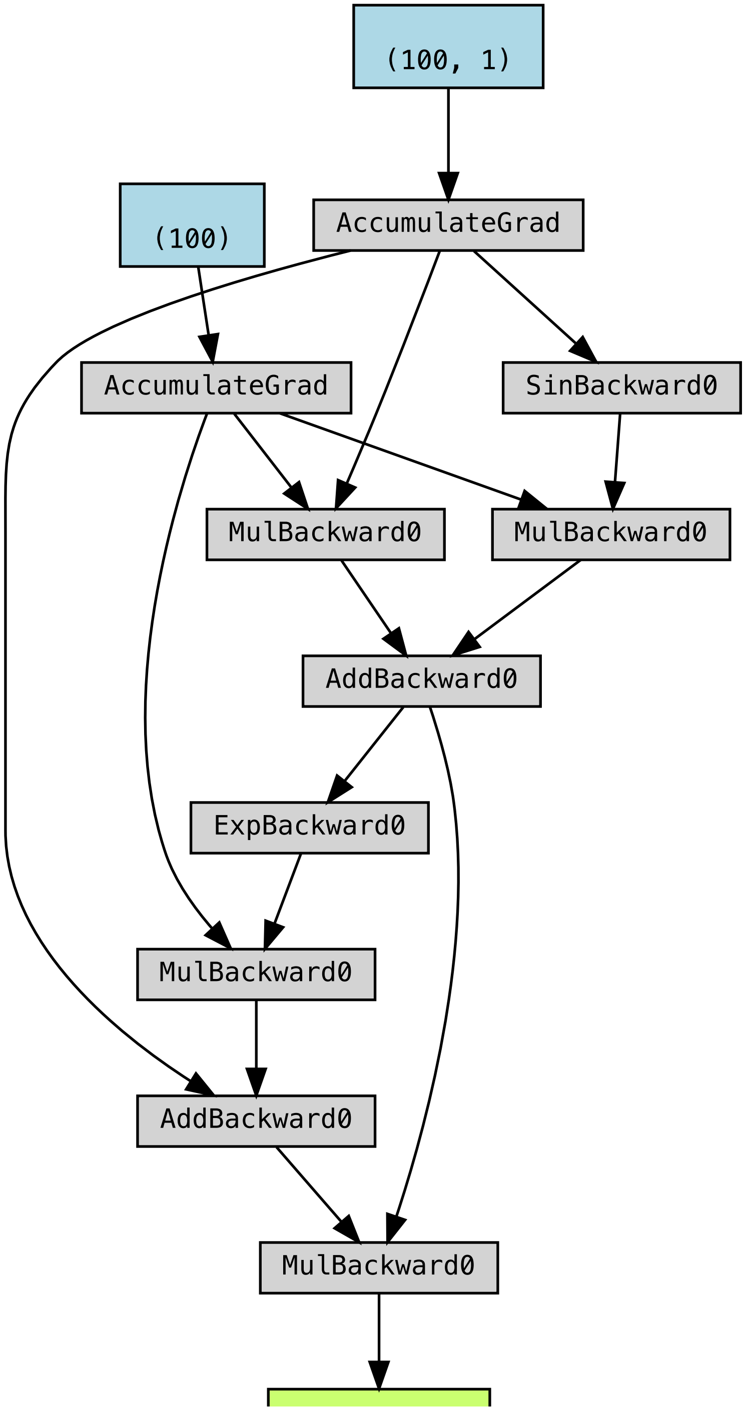
Writing out each path could be quite a lot. It tends to be what happens if you ask a human to expand the formulas for the derivatives out by hand.

In [93]:
```
def f(x,y):
    z = th.sin(x)*y + y*x
    return (th.exp(z)*y+x)*z

# By the way, you can add axes just like in numpy with np.newaxis
x = th.linspace(0,3*np.pi,100)[:,np.newaxis]
y = th.linspace(0,3*np.pi,100)
x.requires_grad_(True)
y.requires_grad_(True)
f_all = f(x,y)
```

In [94]:

```
make_dot(f_all)
```

Out[94]:



(100, 1)
(100)
AccumulateGrad
AccumulateGrad
SinBackward0
MulBackward0
MulBackward0
AddBackward0
ExpBackward0
MulBackward0
AddBackward0
MulBackward0

A recipe for tape-based autograd

However, all of the multiplications *below* a given node will be equal to a single (tensor-valued) result!

If we consider a final loss function $\mathcal{L}$ and an intermediate node k, the gradients to all pieces before k only need to know the *total contribution* of the gradient $\frac{\partial L}{\partial k}$. We can walk backwards through all k, in reverse order of the forward pass to compute L, and determine the full gradient value for each node *without traversing the graph explicitly*.

All we need is:

1. Have our underlying functions know about a related `backward` function. The backward function receives the gradient of the cost for the output of the function, and computes one contribution of the gradient for the inputs.

let $G_\theta := \frac{\partial L}{\partial \theta}$ be a concrete value of the gradient with respect to a cost we are interested in.

$w = f(x, y) \implies$

$G_x = G_w * \frac{\partial f}{\partial x}(x, y)$

$G_y = G_w * \frac{\partial f}{\partial y}(x, y)$

2. There are some proofs that the backwards functions take about the same number as operations as the forwards functions, but only IF we are allowed to store the results of the forward calculations. If you are doing autograd in a tape-based implementation like pytorch, this means that your functions are not pure.

3. Walk through the backwards functions in the reverse order. Every time you reach a certain result, *add* that contribution to the gradient and store it in `x.grad` (This is why you have to do the `zero_grad` step in training.)

When you are done replaying operations in reverse, all of the possible paths will have been implicitly summed, and every tensor will have the correct result in its `.grad` attribute

Autograd recipe

1. Link forward operations to their backward (adjoint) operations
2. Save necessary forward computations to enable efficient backward operations
3. Walk through the graph backward, adding results that target the same tensor

These steps are not horrendously complicated, but it is very convenient to have a library on hand with lots of functions and links to the corresponding backwards functions.

micrograd is an instructive implementation of tape-based autograd in less than 100 lines for the core algorithm.

Difference between `grad` and `backward`

1. `torch.autograd.grad` is useful if you only need the derivative of certain inputs, not every input. It doesn't store any information on `.grad` .

Difference between `grad` and `backward`

1. `torch.autograd.grad` is useful if you only need the derivative of certain inputs, not every input. It doesn't store any information on `.grad`.

In [95]:
```
x = th.Tensor([1,2,3]).requires_grad_(True)
y = 3*x.sum()
gx = grad(y,[x])[0]
print("gx:",gx)
print("x.grad:",x.grad)
```

```
gx: tensor([3., 3., 3.])
x.grad: None
```

2. `torch.autograd.backward` will accumulate gradients for *all* inputs that require grad.
   - the `Tensor.backward` method is equivalent to this
   - the result is ADDED ONTO the `.grad` attribute of the tensor
   - this is why we have calls to `zero_grad()` in the tbaining loops, so that we are adding the gradient onto zero instead of onto the gradient from previous calculations.

Note that using either of these functions will *erase* the autograd tape information:

2. `torch.autograd.backward` will accumulate gradients for *all* inputs that require grad.
   - the `Tensor.backward` method is equivalent to this
   - the result is ADDED ONTO the `.grad` attribute of the tensor
   - this is why we have calls to `zero_grad()` in the tbaining loops, so that we are adding the gradient onto zero instead of onto the gradient from previous calculations.

Note that using either of these functions will *erase* the autograd tape information:

In [96]:

```
x = th.Tensor([1,2,3]).requires_grad_(True)
y = 3*x.sum()

print("First call works!")
y.backward()
print("Second call breaks!")
with suppress_traceback():
    y.backward()
```

```
First call works!
Second call breaks!

RuntimeError: Trying to backward through the graph a second time
(or directly access saved tensors after they have already been f
reed). Saved intermediate values of the graph are freed when you
call .backward() or autograd.grad(). Specify retain_graph=True i
f you need to backward through the graph a second time or if you
need to access saved tensors after calling backward.
```

Retaining the autograd tape

1. There's no *need* in the algorithm to delete the graph when you do a backwards computation.
- However, you typically don't need to use the same graph more than once, and once the graph is consumed, pytorch can free the memory associated with intermediate variables.
- This is much like using `tensor.Detach`; Pytorch will do these things, but only if you ask.
2. You can pass `retain_graph=True` to prevent consuming the graph.
- Doing so without some care will likely lead to memory leaks and eventually an out of memory error.

In [97]:

```
x = th.Tensor([1,2,3]).requires_grad_(True)
y = 3*x.sum()

print("First call works:")
grad_1=y.backward(retain_graph=True)
print("First call result:",x.grad)
print("Second call works:")
y.backward()
print("Second call result:",x.grad)
```

```
First call works:
First call result: tensor([3., 3., 3.])
Second call works:
Second call result: tensor([6., 6., 6.])
```

In [97]:
```python
x = th.Tensor([1,2,3]).requires_grad_(True)
y = 3*x.sum()

print("First call works:")
grad_1=y.backward(retain_graph=True)
print("First call result:",x.grad)
print("Second call works:")
y.backward()
print("Second call result:",x.grad)
```

```
First call works:
First call result: tensor([3., 3., 3.])
Second call works:
Second call result: tensor([6., 6., 6.])
```

The result is twice as much the second time because of the accumulation to `.grad` .

Integrating your custom operation with autograd.

Note: This is a good learning experience, but is ***Almost Always Not Necessary***.

(The one time I have used this is because I have a specific sparse, three-way tensor-tensor-matrix contraction operation in my neural network for atomistic systems, and although it can be implemented with pure pytorch operations, that implementation used far more memory than needed. I used numba to perform the calculation more efficiently, and linked it into pytorch with a custom autograd operation)

In [98]:

```
class MySineFunction(th.autograd.Function):
    @staticmethod # Methods are static, do not take `self`
    def forward(ctx, x): # ctx variable acts like "self"
        ## Perform the forward.
        ## Note that autograd is OFF inside an autograd.Function
        ## because YOU are supplying the backward implementation.
        y = th.sin(x)
        ## ctx is a context object for storing information
        ## for backward computation
        ctx.x = x
        return y

    @staticmethod
    def backward(ctx, grad_output):
        ## Note that autograd is ON by default in the backward pass.
        x = ctx.x
        dLdy = grad_output #  the gradients w.r.t a scalar
        # dL/dx = dL/dy * (dy/dx)
        # In this case, (dy/dx) = cos(x)
        dLdx = dLdy * th.cos(x)
        return dLdx

# the apply method of `Function` is what you want to use.
mysine = MySineFunction.apply
```

Note that autograd is ON by default in the backward pass. This allows you to implement more complicated things like infinitely differentiable functions. For example, you can make a set of sine and cosine functions using numpy code instead of pytorch, and link them both to each other.

Again, *usually you don't need to do this*. I'm presenting for the value of learning a bit about the internals, it is *not* a routine thing to do in pytorch.

Now we can use `mysine` like any other pytorch function.

Checking the output

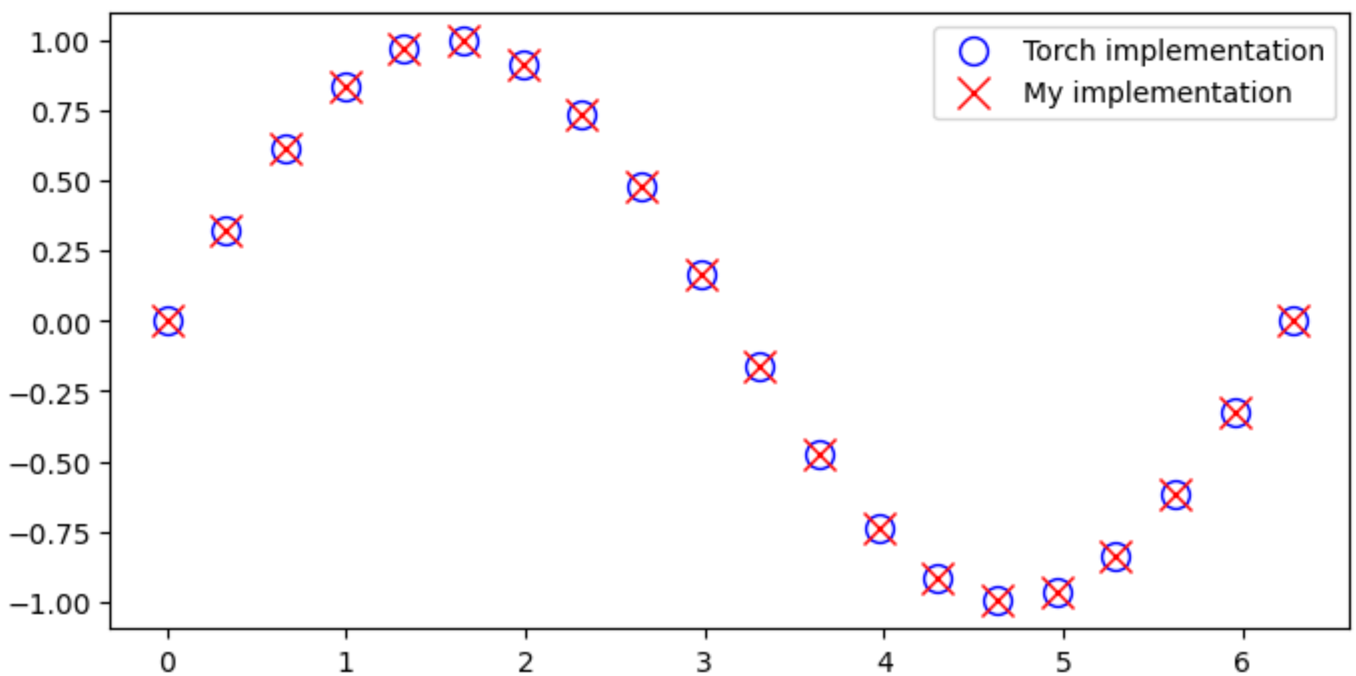
Checking the output

In [99]:
```
x = th.linspace(0,2*np.pi,20,dtype=th.float64).requires_grad_(True)

th_sin_x = th.sin(x)
my_sin_x = mysine(x)
```

Checking the output

In [99]:
```
x = th.linspace(0,2*np.pi,20,dtype=th.float64).requires_grad_(True)

th_sin_x = th.sin(x)
my_sin_x = mysine(x)
```

In [101]:
```
plt.sca(ax); plt.show()
```



In [102]:
```
x = th.linspace(0,2*np.pi,20,dtype=th.float64).requires_grad_(True)

th_sin_x = th.sin(x)
my_sin_x = mysine(x)

print("Arrays are equal:",(my_sin_x == th_sin_x).all().item())
print(my_sin_x)
```

```
Arrays are equal: True
tensor([ 0.0000e+00,  3.2470e-01,  6.1421e-01,  8.3717e-01,  9.6
940e-01,
         9.9658e-01,  9.1577e-01,  7.3572e-01,  4.7595e-01,  1.6
459e-01,
        -1.6459e-01, -4.7595e-01, -7.3572e-01, -9.1577e-01, -9.9
658e-01,
        -9.6940e-01, -8.3717e-01, -6.1421e-01, -3.2470e-01, -2.4
493e-16],
       dtype=torch.float64, grad_fn=<MySineFunctionBackward>)
```

In [102]:
```
x = th.linspace(0,2*np.pi,20,dtype=th.float64).requires_grad_(True)

th_sin_x = th.sin(x)
my_sin_x = mysine(x)

print("Arrays are equal:",(my_sin_x == th_sin_x).all().item())
print(my_sin_x)
```

```
Arrays are equal: True
tensor([ 0.0000e+00,  3.2470e-01,  6.1421e-01,  8.3717e-01,  9.6
940e-01,
         9.9658e-01,  9.1577e-01,  7.3572e-01,  4.7595e-01,  1.6
459e-01,
        -1.6459e-01, -4.7595e-01, -7.3572e-01, -9.1577e-01, -9.9
658e-01,
        -9.6940e-01, -8.3717e-01, -6.1421e-01, -3.2470e-01, -2.4
493e-16],
       dtype=torch.float64, grad_fn=<MySineFunctionBackward>)
```

Note how the `grad_fn` points at something interesting called `MySineFunctionBackward`.

Let's examine how the gradient behavior looks

Let's examine how the gradient behavior looks

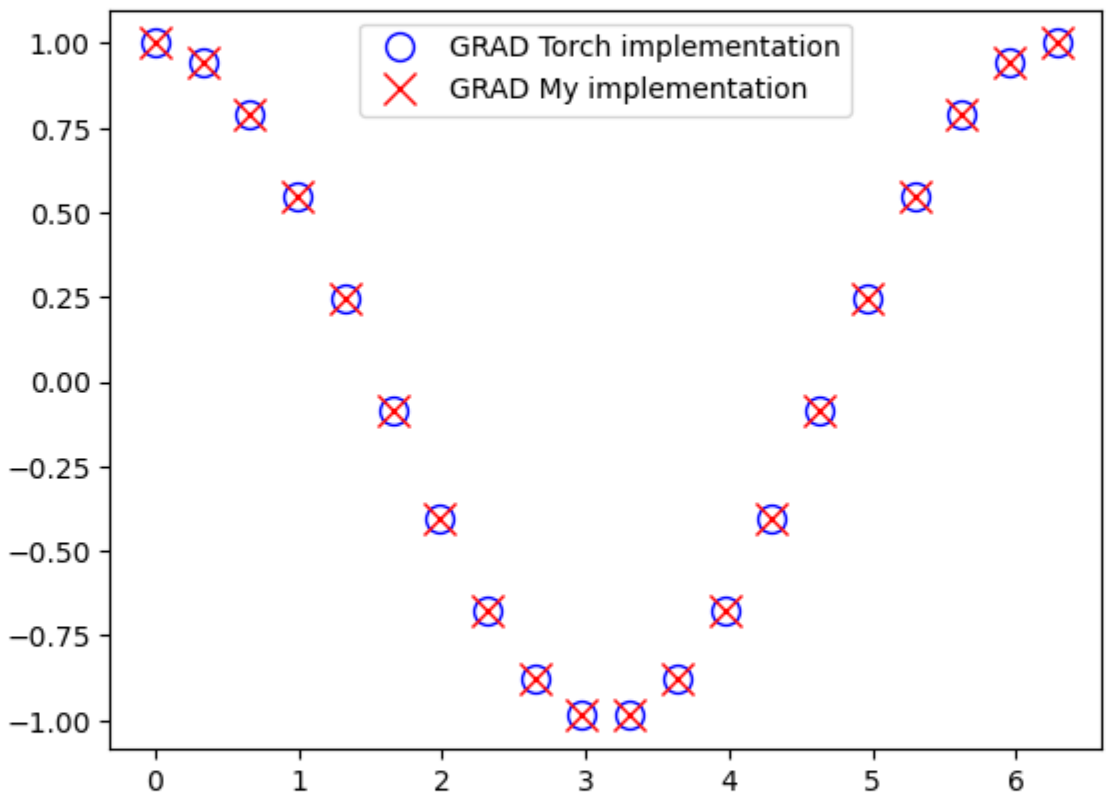
In [103]:

```
x = th.linspace(0,2*np.pi,20,dtype=th.float64).requires_grad_(True)
th_sin_x = th.sin(x)
my_sin_x = mysine(x)

gx_th_sin= grad(th_sin_x.sum(),[x])[0]
gx_my_sin = grad(my_sin_x.sum(),[x])[0]

print("Grad arrays are equal:",(gx_th_sin==gx_my_sin).all().item())
```

```
Grad arrays are equal: True
```

In [104]:

```
# Plot them to take a look:
plt.plot(x.detach().numpy(),gx_th_sin.detach().numpy(),
         marker='o',mfc=[1,1,1,1],ms=10,lw=0,c='b',
         label = "GRAD Torch implementation")
plt.plot(x.detach().numpy(),gx_my_sin.detach().numpy(),
         marker='x',ms=12,lw=0,c='r',
         label='GRAD My implementation')
plt.legend()
plt.show()
```



Looks good! As an exercise, play around with MySineFunction. Can you make MyTanhFunction? Can you make this function work via `np.sin` or `math.sin`?

Checking Gradients

Although pytorch tries to make it hard to "shoot yourself in the foot" with autograd, it is still possible. If your model is not working right, you can check gradients through a function by comparing with finite differences.

- Finite differences are much slower.
- They rely on a finite but small ϵ parameter. This means that:
  - There is error introduced in the calculation.
  - `float32` dtypes are often not acccurate enough to get good finite differences, so do the computation in `float64`.
- However, because finite differences only rely on the forward computation and the definition of the derivative, it is typically robust; it is "too dumb to fail"

Checking gradients numerically is extremely useful for debugging 'weird' problems.

You can check gradients with `torch.autograd.gradcheck`!

In [105]:

```
from torch.autograd import gradcheck

# Use float64 inputs for gradcheck V
x = th.linspace(0,2*np.pi,10,dtype=th.float64).requires_grad_(True)

# If this returns True, then all is well.
gradcheck(mysine,[x])
```

Out[105]:

```
True
```

In [105]:

```
from torch.autograd import gradcheck

# Use float64 inputs for gradcheck V
x = th.linspace(0,2*np.pi,10,dtype=th.float64).requires_grad_(True)

# If this returns True, then all is well.
gradcheck(mysine,[x])
```

Out[105]:

```
True
```

It's that easy!

What it looks like when you break autograd:

What it looks like when you break autograd:

In [106]:

```
def broken_f(x,y):
    """This function is broken w.r.t autograd!"""
    yp = y.detach().numpy()
    output = x*th.from_numpy(yp) # A differentiable variable went out o
    return output

x = th.ones(4,dtype=th.float64).requires_grad_(True)
y = th.sin(x)
with suppress_traceback():
    gradcheck(broken_f,[x,y])
```

```
GradcheckError: Jacobian mismatch for output 0 with respect to i
nput 1,
numerical:tensor([[1.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 1.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 1.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 1.0000]], dtype=torch.float64)
analytical:tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]], dtype=torch.float64)
```

In [107]:

```python
with suppress_traceback():
    gradcheck(broken_f,[x,y])
```

```
GradcheckError: Jacobian mismatch for output 0 with respect to i
nput 1,
numerical:tensor([[1.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 1.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 1.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 1.0000]], dtype=torch.float64)
analytical:tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]], dtype=torch.float64)
```

In [107]:
```
with suppress_traceback():
    gradcheck(broken_f,[x,y])
```

```
GradcheckError: Jacobian mismatch for output 0 with respect to i
nput 1,
numerical:tensor([[1.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 1.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 1.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 1.0000]], dtype=torch.float64)
analytical:tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]], dtype=torch.float64)
```

Here, the numerical jacobian of df/dy with respect to input 1, that is, `y`, is the identity matrix. But the "analytic" one, that is, the one coming from autograd, is all zeros.

Note: gradcheck will run f many times. If you have a print statement in your function, it will spam the terminal.

Differences from other autograd implementations

- `jax` uses a clever ahead-of-time functional programming approach, however, it is a little more fragile and easier to break than pytorch's tape-based approach.
- `autograd` is a famous python package that brought a foothold to the tape-based approach. Both Pytorch and Jax followed autograd's lead to provide a numpy-like interface.

- `theano` and `tensorflow` work very differently from pytorch:
    - You program the model by first defining the graph, and then plugging in values later.
    - This is a *declarative programming* paradigm, rather than an an *imperative programming* paradigm. If you haven't heard of the difference, you have *probably* always been using an imperative language.
    - Tensorflow now has a mode that works more like pytorch from a user perspective.
    - Theano will additionally optimize the graph in a lot of usesful ways.
    - The downside of this is that it is much harder to add extra operations to the graph -- instead of using the python print function to print an intermediate value, the graph has to have a `print node` of some kind.

- In languages other than python, *code macros* are a powerful tool that allow one to differentiate source code directly. This has advantages, because a compiler can then optimize very thoroughly, and one does not need to implement as many backward functions. This is the approach of the Julia package `zygote.jl`.
- `Keras` is a wrapper library -- it does not implement any of this stuff itself, and is only focused on making it easier to build NNs using a backend engine like pytorch or tensorflow.
    - in my opinion, pytorch is nearly as easy as keras
    - `skorch` provides `sklearn` -like wrappers for pytorch
    - `ignite` provides high-level training workflows.

Conclusions

- Pytorch offers a numpy/scipy-like array of operations in tandem with GPU support.
- The key feature for all neural network libraries is automatic differentiation
- Pytorch implements this using a tape-based approach of recording live metadata.
- Autograd is a recursive implementation of the chain rule.
- You can add custom operations to pytorch autograd.

Conclusions

- Pytorch offers a numpy/scipy-like array of operations in tandem with GPU support.
- The key feature for all neural network libraries is automatic differentiation
- Pytorch implements this using a tape-based approach of recording live metadata.
- Autograd is a recursive implementation of the chain rule.
- You can add custom operations to pytorch autograd.

In [108]:

```
print("Done")
```

```
Done
```