04 Numerical Optimization

May 28, 2021

Information: *Brief introduction to gradient descent, how gradient descent is supported in pytorch, convex functions, and some numerical considerations to kept in mind*

Written by: *Zihao Xu*

Last update date: *05.28.2021*

1 Gradient Descent Optimization

1.1 Motivation

- Most ML/DL algorithms involve **optimization** of some sort.
  - Optimization refers to the task of either **minimizing** or maximizing some function $f(\mathbf{x})$ by altering $\mathbf{x}$
  - Usually phrase most optimization problems in terms of minimizing $f(\mathbf{x})$
  - Maximization may be accomplished via s minimization algorithm by minimizing $-f(\mathbf{x})$
- Usually the function we want to minimize is called the **objective function**, or **criterion**. In ML/DL contexts, the name **loss function** is often used.
  - As mentioned in introduction, a loss function quantifies the *distance* between the **real** and **predicted** value of the target.
  - Usually be a non-negative number where smaller values are better and perfect predictions incur a loss of 0
  - Usually denoted as $L(\boldsymbol{\theta})$ where $\boldsymbol{\theta}$ is usually the parameter of ML/DL models
- Usually denote the value that minimizes a function with a superscript $*$
  - $\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\text{argmin}} L(\boldsymbol{\theta})$
- Most ML/DL algorithms are so complex that it is difficult or impossible to find the closed form solution for the optimization problem
  - Use numerical optimization method instead
- One common algorithm is **gradient descent**, other optimization algorithms are
  - Expectation Maximization
  - Sampling-based optimization
  - Greedy optimization

1.2 Definition

- **Definition**:
  - A **first-order iterative** optimization algorithm for finding **local minimum** of a **differential** function.

    * The idea is to take **repeated steps** in the opposite direction of the **gradient** of the function at the current point, because this is the direction of steepest descent.
    * As it calculates the **first-order** derivative, it requires the objective function to be **differential**
    * Converge when first-order derivative is zero, which only ensures reaching **local minimum** for general functions
- **Theory**:
  - Based on the observation that if the multi-variable function $F(\mathbf{x})$ is defined and differentiable in a neighborhood of a point $\mathbf{a}$, then $F(\mathbf{x})$ decreases **fastest** if one goes from $\mathbf{a}$ in the direction of the negative gradient of F at $\mathbf{a}$, which is $-\nabla F(\mathbf{a})$. It follows that if

    $$\mathbf{a}_{n+1} = \mathbf{a}_n - \gamma \nabla F(\mathbf{a}_n)$$

    for a $\gamma \in \mathbb{R}_+$ small enough, then

    $$F(\mathbf{a}_n) \geq F(\mathbf{a}_{n+1})$$

- Simple form of **vanilla gradient descent** (GD):
  1. Start at random parameter $\boldsymbol{\theta}$
  2. Repeat until converged
     - $\mathbf{d} \leftarrow -\nabla L(\boldsymbol{\theta})$
     - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \mathbf{d}^T$
  - α is called **learning rate** or **step size**

1.3 Select appropriate learning rate

- Too large α leads to instability and even divergence
- Too small α leads to slow convergence
- **Steepest gradient descent** use line search to compute the best α
  1. Start at random parameter $\boldsymbol{\theta}$
  2. Repeat until converged
     - $\mathbf{d} \leftarrow -\nabla L(\boldsymbol{\theta})$
     - $\alpha^* \leftarrow \underset{\alpha}{\operatorname{argmin}}\{L(\boldsymbol{\theta} + \alpha \mathbf{d}^T)\}$
     - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^* \mathbf{d}^T$
- **Adaptive learning rates** may help, but not always
  - $\alpha = \frac{1}{t}$, approaches 0 but can cover an infinite distance since $\lim_{a \to \infty} \sum_{t=1}^{a} \frac{1}{t} = \infty$
- **Coordinate Descent** update one parameter at a time
  - Removes problem of selecting step size
  - Each update can be very fast, but lots of updates

1.4 Slow convergence due to Poor Conditioning

- **Conditioning** refers to how rapidly a function changes with respect to small changes in its inputs.
- Consider the function

  $$f(x) = \mathbf{A}^{-1}\mathbf{x}$$

  When $\mathbf{A} \in \mathbb{R}^{n \times n}$ has an eigenvalue decomposition, its **condition number** is

  $$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|$$

This is the ratio of the magnitude of the largest and smallest eigenvalue

- A problem with a **low condition number** is said to be **well-conditioned**, while a problem with a high condition number is said to be ill-conditioned
  - In non-mathematical terms, an ill-conditioned problem is one where, for a small change in the inputs there is a large change in the answer or dependent variable, which means the correct solution to the equation becomes hard to find
  - Condition number is a property of the problem
- **Gradient descent** is very sensitive to **condition number** of the problem
  - No good choice of step size. Tiny change in one variable could lead to great change in dependent variable.
- **Solutions:**
  - **Newton's method:** Correct for local second derivative. (Sphere the ellipse)
    - Too much computation and too difficult to implement
  - **Alternative methods**:
    - Preconditioning: Easy, but tends to be ad-hoc, not so robust
    - Momentum

1.5 Compute Loss Gradient

- Take the **mean square error** as an example:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} L_{MSE}(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \left\{ \frac{1}{N} \sum_{i=1}^{N} \left\| \mathbf{y}_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i) \right\|^2 \right\} \\ &= \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \left\{ (\mathbf{y}_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i))^T (\mathbf{y}_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i)) \right\}\end{aligned}$$

  Use the chain rule and scale-by-vector matrix calculus identity that

$$\frac{\partial \mathbf{x}^T \mathbf{x}}{\partial \mathbf{x}} = 2\mathbf{x}^T$$

  We can get

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} L_{MSE}(\boldsymbol{\theta}) &= \frac{2}{N} \sum_{i=1}^{N} (\mathbf{y}_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i))^T \nabla_{\boldsymbol{\theta}} (\mathbf{y}_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i)) \\ &= \frac{2}{N} \sum_{i=1}^{N} (\mathbf{y}_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i))^T \nabla_{\boldsymbol{\theta}} (-f_{\boldsymbol{\theta}}(\mathbf{x}_i)) \\ &= -\frac{2}{N} \sum_{i=1}^{N} (\mathbf{y}_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i))^T \nabla_{\boldsymbol{\theta}} (f_{\boldsymbol{\theta}}(\mathbf{x}_i))\end{aligned}$$

- The result of the gradient usually includes three parts:
  - Sum over training data. It consists of a lot of computations but the way of computation is relatively easy and straight forward
  - Prediction error term such as $\mathbf{y}_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i)$ in MSE, which is usually easy to get
  - Gradient of inference function $\nabla_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}(\mathbf{x}_i))$, which is difficult to solve
    - Enabled by automatic differentiation built into modern domain specific languages such as Pytorch, Tensorflow, …
    - For neural networks, this is known as **back propagation**

2 Automatic Differentiation via Pytorch

2.1 Data Manipulation via Pytorch

- The n-dimensional array is usually called the tensor
- *tensor* class in Pytorch is similar to *NumPy*'s *ndarray* with several additional features
  - GPU is well-supported to accelerate the computation whereas *NumPy* only supports CPU computation
  - *tensor* class supports automatic differentiation
- The Pytorch documentation shows the full attributes

2.1.1 Create a tensor

- To get started, import **torch**. Although it's called Pytorch, we should import **torch** instead of pytorch

[1]:
```
import torch
# Check the version of a module
print(torch.__version__)
```

```
1.8.1
```

- Common ways to creating a tensor
  - torch.arange(start,end,step)
  - torch.zeros(shape)
  - torch.ones(shape)
  - torch.randn(shape)
  - torch.tensor(elements)

[2]:
```
torch.arange(0, 12, 1)
```

[2]:
```
tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

[3]:
```
torch.zeros((3, 4))
```

[3]:
```
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
```

[4]:
```
torch.ones((2, 5))
```

[4]:
```
tensor([[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]])
```

[5]:
```
torch.randn(10)
```

[5]:
```
tensor([ 0.4958,  0.2206, -0.3878, -0.4165,  0.2741, -0.1352,  0.8250, -0.3746,
        -0.8352, -1.5164])
```

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

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

- One can access a tensor's shape by viewing the **shape** attribute

[7]:
```
torch.ones((4, 5)).shape
```

[7]:
```
torch.Size([4, 5])
```

- **reshape** method can change the shape of a tensor without altering either the number of elements or their values.
  - No need to manually specify every dimension
  - Can place -1 for the dimension that we would like tensors to automatically infer

[8]:
```
torch.arange(12).reshape(3, 4)
```

[8]:
```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

[9]:
```
torch.arange(12).reshape(3, -1)
```

[9]:
```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

2.1.2 Type of a tensor

- Usually, a tensor is created as **tensor.float32** (32-bit floating point) by default. One can view its type in the **dtype** attribute
  - When creating a tensor using **torch.tensor**, tensor with all integers would be created as **torch.int64** (64-bit signed integer)
- Full tensor types can be viewed in the documentation

[10]:
```
torch.ones(10).dtype
```

[10]:
```
torch.float32
```

[11]:
```
torch.tensor([[1, 2, 3], [4, 5, 6]]).dtype
```

[11]:
```
torch.int64
```

[12]:
```
# Only add one dot after the first element
torch.tensor([[1., 2, 3], [4, 5, 6]]).dtype
```

[12]:
```
torch.float32
```

- One can assign the wanted type when creating the tensor by setting the **dtype** attribute to
  - *torch.float32*, 32-bit floating point
  - *torch.float64*, 64-bit floating point

  - *torch.uint8*, 8-bit unsigned integer
  - *torch.int8*, 8-bit signed integer
  - *torch.int32*, 32-bit signed integer
  - *torch.int64*, 64-bit signed integer
  - *torch.bool*, Boolean

[13]:
```
torch.ones(10, dtype=torch.float64).dtype
```

[13]:
```
torch.float64
```

[14]:
```
torch.ones(10, dtype=torch.uint8).dtype
```

[14]:
```
torch.uint8
```

[15]:
```
torch.ones(10, dtype=torch.int32).dtype
```

[15]:
```
torch.int32
```

- One can also construct the type of a tensor from list or numpy array using the method:
  - *FloatTensor*, 32-bit floating point
  - *DoubleTensor*, 64-bit floating point
  - *ByteTensor*, 8-bit unsigned integer
  - *CharTensor*, 8-bit signed integer
  - *IntTensor*, 32-bit signed integer
  - *LongTensor*, 64-bit signed integer
  - *BoolTensor*, Boolean

[16]:
```
torch.DoubleTensor([1, 2, 3]).dtype
```

[16]:
```
torch.float64
```

[17]:
```
torch.ByteTensor([1, 2, 3]).dtype
```

[17]:
```
torch.uint8
```

2.1.3 Use GPU for tensor computation

- Unless otherwise specified, a new tensor will be stored in main memory and designated for CPU-based computation
- One can check which device the tensor is designated for by viewing the **device** attribute

[18]:
```
torch.ones(10).device
```

[18]:
```
device(type='cpu')
```

- One can always set the create a device if a GPU supporting cuda is available and use **to(device)** method to determine the device on which a tensor is or will be allocated
  - Assign the **device** parameter when creating a tensor also works

[19]:
```
cuda0 = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
cuda0
```

[19]:
```
device(type='cuda', index=0)
```

[20]:
```
torch.ones(5, device=cuda0)
```

[20]:
```
tensor([1., 1., 1., 1., 1.], device='cuda:0')
```

[21]:
```
# If available, indexing the cuda also works
torch.ones(5, device=0)
```

[21]:
```
tensor([1., 1., 1., 1., 1.], device='cuda:0')
```

[22]:
```
# If avaiable, the string also works
torch.ones(5, device="cuda:0")
```

[22]:
```
tensor([1., 1., 1., 1., 1.], device='cuda:0')
```

[23]:
```
# Use the to method the move a tensor
x = torch.ones(5).to(cuda0)
x
```

[23]:
```
tensor([1., 1., 1., 1., 1.], device='cuda:0')
```

[24]:
```
# One can also move a tensor from GPU to CPU
x.to("cpu").device
```

[24]:
```
device(type='cpu')
```

2.1.4 Operations

- Common standard arithmetic operators have all been lifted to element-wise operations

[25]:
```
x = torch.tensor([1., 2., 4., 8.])
c = 1.
x + c, x * c, x**c
```

[25]:
```
(tensor([2., 3., 5., 9.]), tensor([1., 2., 4., 8.]), tensor([1., 2., 4., 8.]))
```

[26]:
```
x = torch.tensor([1., 2., 4., 8.])
y = torch.tensor([4., 3., 2., 1.])
x + y, x * y, x**y
```

[26]:
```
(tensor([5., 5., 6., 9.]),
 tensor([4., 6., 8., 8.]),
 tensor([ 1.,  8., 16.,  8.]))
```

[27]:
```
torch.exp(x)
```

[27]:
```
tensor([2.7183e+00, 7.3891e+00, 5.4598e+01, 2.9810e+03])
```

- Matrix multiplication is also supported

[28]:
```
A = torch.randn((4, 3))
B = torch.randn((3, 5))
# Two ways of matrix multiplication
torch.mm(A, B), A @ B
```

[28]:
```
(tensor([[ 3.4776, -1.6009, -0.0927, -3.0731, -1.6719],
         [-2.5072,  1.3272, -0.5220,  0.5175,  1.5722],
         [ 0.9626, -0.3774,  0.2813,  0.3111, -0.5749],
         [ 1.0100,  1.6795, -1.0543, -0.5897,  1.0897]]),
 tensor([[ 3.4776, -1.6009, -0.0927, -3.0731, -1.6719],
         [-2.5072,  1.3272, -0.5220,  0.5175,  1.5722],
         [ 0.9626, -0.3774,  0.2813,  0.3111, -0.5749],
         [ 1.0100,  1.6795, -1.0543, -0.5897,  1.0897]]))
```

[29]:
```
A = torch.randn((5, 4))
B = torch.randn((5, 4))
# Two ways of elementwise multiplication
torch.mul(A, B), A * B
```

[29]:
```
(tensor([[ 2.7632e-01, -2.8037e-01,  1.3489e-01,  8.0120e-02],
         [-2.8262e-01,  5.2607e-01, -6.7521e-03,  1.0018e+00],
         [ 4.8200e-02,  1.9570e+00, -2.8883e-03, -1.2611e-01],
         [-4.6326e-01,  2.0817e+00, -2.9168e+00,  7.9631e-02],
         [-3.0559e+00,  4.7762e-01, -5.6648e-01, -4.0203e-02]]),
 tensor([[ 2.7632e-01, -2.8037e-01,  1.3489e-01,  8.0120e-02],
         [-2.8262e-01,  5.2607e-01, -6.7521e-03,  1.0018e+00],
         [ 4.8200e-02,  1.9570e+00, -2.8883e-03, -1.2611e-01],
         [-4.6326e-01,  2.0817e+00, -2.9168e+00,  7.9631e-02],
         [-3.0559e+00,  4.7762e-01, -5.6648e-01, -4.0203e-02]]))
```

- We can also **concatenate** multiple tensors together, stacking them end-to-end to form a larger tensor. We just need to provide a list of tensors and tell the system along which axis to concatenate.

[30]:
```
A = torch.arange(0, 12, 1).reshape((3, 4))
B = torch.ones((3, 4))
# dim stands for the index of dimension in which the tensors are concatenated
torch.cat((A, B), dim=0), torch.cat((A, B), dim=1)
```

[30]:
```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [ 1.,  1.,  1.,  1.],
         [ 1.,  1.,  1.,  1.],
         [ 1.,  1.,  1.,  1.]]),
```

```
tensor([[ 0.,  1.,  2.,  3.,  1.,  1.,  1.,  1.],
        [ 4.,  5.,  6.,  7.,  1.,  1.,  1.,  1.],
        [ 8.,  9., 10., 11.,  1.,  1.,  1.,  1.]]))
```

- Also, we can construct a binary tensor via logical statements

[31]:
```
A = torch.arange(0, 12, 1).reshape((3, 4))
B = 5 * torch.ones((3, 4))
A == B, A > B
```

[31]:
```
(tensor([[False, False, False, False],
         [False,  True, False, False],
         [False, False, False, False]]),
 tensor([[False, False, False, False],
         [False, False,  True,  True],
         [ True,  True,  True,  True]]))
```

2.1.5 Broadcasting Mechanism

- Under certain conditions, even shapes differ, we can still perform **element-wise** operations by invoking the **broadcasting mechanism**.
  - First, expand one or both arrays by copying elements appropriately so that after this transformation, the two tensors have the same shape.
  - Second, carry out the element-wise operation on the resulting arrays
- In most cases, we broadcast along an axis where an array initially only has length 1

[32]:
```
a = torch.arange(3).reshape(3, 1)
b = torch.arange(2).reshape(1, 2)
a, b, a + b
```

[32]:
```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]),
 tensor([[0, 1],
         [1, 2],
         [2, 3]]))
```

2.1.6 Indexing and Slicing

- As in standard Python lists, we can access elements according to their relative position to the end of the list by using negative indices

[33]:
```
X = torch.arange(16).reshape(4, 4)
X, X[-1], X[1:3]
```

[33]:
```
(tensor([[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11],
```

```
        [12, 13, 14, 15]]),
 tensor([12, 13, 14, 15]),
 tensor([[ 4,  5,  6,  7],
        [ 8,  9, 10, 11]]))
```

- Can also index using binary tensor

[34]:
```
A = torch.arange(0, 12, 1).reshape((3, 4))
B = 5 * torch.ones((3, 4))
A, B, A[A > B]
```

[34]:
```
(tensor([[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]]),
 tensor([[5., 5., 5., 5.],
         [5., 5., 5., 5.],
         [5., 5., 5., 5.]]),
 tensor([ 6,  7,  8,  9, 10, 11]))
```

2.2 Automatic Calculation of Gradients

- In practice, based on our designed model, the system builds a **computational graph**, tracking which data combined through which operations to produce the output. Automatic differentiation enables the system to subsequently **backpropagate gradients**.
  - Here *backpropagate* simply means to trace through the computational graph, filling in the partial derivatives with respect to each parameter

2.2.1 Intuition

- All computation can be broken into simple components
  - sum
  - multiply
  - exponential
  - convolution
  - …
- Derivatives for each simple component can be derived mathematically
- Derivatives for **any composition** can be derived via **chain rule**
- That is to say, for a function

$$f = f(x_1, x_2, \cdots, x_n)$$

Start from the chain rule

$$\frac{\partial f}{\partial t} = \sum_{j=1}^{n} \frac{\partial f}{\partial x_j} \frac{\partial x_j}{\partial t}$$

And decompose f into simple components such as

$$f = g_1(g_2(\cdots g_M(x_1, x_2, \cdots, x_n)))$$

Where all the functions $g(\bullet)$ are simple components like sum, multiplication …, we can always get the derivative respect to time

$$\frac{\partial f}{\partial t}$$

by **composition of simple operations** of

$$\frac{\partial x_i}{\partial t},\ i = 1, 2, \cdots, n$$

2.2.2 Automatic Differentiation - Forward Mode

- **Notice:** Here is only my naive understanding about automatic differentiation since I haven't found some detailed mathematical explanation about it.
- One naive way of find the partial derivative respect to i-th variable $\frac{\partial f}{\partial x_i}$ in the function

$$f = f(x_1, x_2, \cdots, x_n)$$

the simplest way is to **construct** the chain rule ($\frac{\partial f}{\partial t}$ is actually meaningless in this context)

$$\frac{\partial f}{\partial t} = \sum_{j=1}^{n} \frac{\partial f}{\partial x_j} \frac{\partial x_j}{\partial t}$$

by setting

$$\frac{\partial x_j}{\partial t} = \begin{cases} 1 & j = i \\ 0 & j \neq i \end{cases}$$

we get

$$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial x_i}$$

- That is to say, for each **decomposed simple operation**, the computer does not only calculate the **output**, but also simultaneously tracks the **derivatives of output respect to time** based on the input value and input derivative, thus **step by step** getting the final derivative respect to time, which is equal to the partial derivative
- Since the computation is **straight forward**, this method to calculate partial derivative is called **forward mode**.
- For example, for a function

$$f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$$

Decompose the function into simple operation nodes

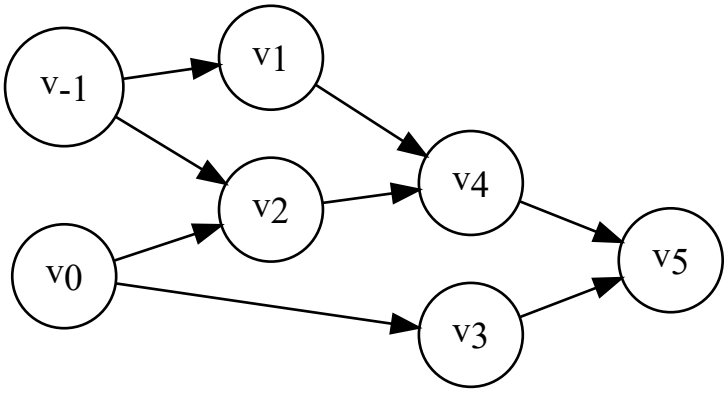
$$\begin{aligned} v_{-1} &= x_1 \\ v_0 &= x_2 \\ v_1 &= \ln v_{-1} \\ v_2 &= v_{-1} \times v_0 \\ v_3 &= \sin v_0 \\ v_4 &= v_1 + v_2 \\ v_5 &= v_4 - v_3 \\ y &= v_5 \end{aligned}$$

The forward mode computation graph is shown below

[35]:
```
from graphviz import Digraph
```

```
f = Digraph('ComputationGraph')
f.attr(rankdir='LR')
f.attr('node', shape='circle')
f.node('v_-1', label='<v<sub>-1</sub>>')
f.node('v_0', label='<v<sub>0</sub>>')
f.node('v_1', label='<v<sub>1</sub>>')
f.node('v_2', label='<v<sub>2</sub>>')
f.node('v_3', label='<v<sub>3</sub>>')
f.node('v_4', label='<v<sub>4</sub>>')
f.node('v_5', label='<v<sub>5</sub>>')
f.edge('v_-1', 'v_1')
f.edge('v_-1', 'v_2')
f.edge('v_0', 'v_2')
f.edge('v_0', 'v_3')
f.edge('v_1', 'v_4')
f.edge('v_2', 'v_4')
f.edge('v_4', 'v_5')
f.edge('v_3', 'v_5')
f
```

[35]:



- To find the **partial derivate** $\frac{\partial f}{\partial x_1}$ when $x_1 = 2, x_2 = 5$, compute along the **forward evaluation trace** first:

$$
\begin{aligned}
v_{-1} = \quad & x_1 = && 2 \\
v_0 = \quad & x_2 = && 5 \\
v_1 = \quad & \ln v_{-1} = && \ln 2 \\
v_2 = \quad & v_{-1} \times v_0 = && 2 \times 5 \\
v_3 = \quad & \sin v_0 = && \sin 5 \\
v_4 = \quad & v_1 + v_2 = && 0.693 + 10 \\
v_5 = \quad & v_4 - v_3 = && 10.693 + 0.959 \\
y = \quad & v_5 = && 11.652
\end{aligned}
$$

Then set $\dot{x_1} = 1,\ \dot{x_2} = 0$ and compute along the **forward derivative trace** (maybe

also computed simultaneously along with the forward evaluation trace)

$$\begin{aligned}
\dot{v}_{-1} = \quad & \dot{x}_1 = & 1 \\
\dot{v}_0 = \quad & \dot{x}_2 = & 0 \\
\dot{v}_1 = \quad & \dot{v}_{-1}/v_{-1} = & 1/2 \\
\dot{v}_2 = \quad & \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1} = & 1 \times 5 + 0 \times 2 \\
\dot{v}_3 = \quad & \dot{v}_0 \times \cos v_0 = & 0 \times \cos 5 \\
\dot{v}_4 = \quad & \dot{v}_1 + \dot{v}_2 = & 0.5 + 5 \\
\dot{v}_5 = \quad & \dot{v}_4 - \dot{v}_3 = & 5.5 - 0 \\
\dot{y} = \quad & \dot{v}_5 = & 5.5
\end{aligned}$$

Therefore, the result is

$$\frac{\partial f}{\partial x_1} = \frac{\partial f}{\partial t} = 5.5$$

If $\frac{\partial f}{\partial x_2}$ is also interested, the **forward derivative trace** needs to be computed again when setting $\dot{x}_1 = 0, \dot{x}_2 = 1$

- **Comments:**
  - For a function $f : \mathbb{R}^n \Rightarrow \mathbb{R}^m$ with input $\mathbf{x} \in \mathbb{R}^n$ and output $\mathbf{y} \in \mathbb{R}^m$, needs to calculate n times along the **forward derivative trace** for the partial derivatives

    $$\frac{\partial \mathbf{y}}{\partial x_j}, \ \ j = 1, 2, \cdots, n$$

    by sequentially setting

    $$\frac{\partial x_j}{\partial t} = \begin{cases} 1 & j = i \\ 0 & j \neq i \end{cases}, \ \ i = 1, 2, \cdots, n$$

    and only one time along the **forward evaluation trace** for the m-dimension output values

    $$y_1, y_2, \cdots, y_m$$

  - Get the partial derivatives of **all outputs** respect to **one variable** after computing along the derivative trace once
  - **High efficiency** when $n << m$
  - **Low efficiency** when $n >> m$, which is common in machine learning and deep learning

2.2.3 Automatic differentiation - Reverse Mode

- Motivated by the **low efficiency** of *forward mode* when the derivatives of multiple variables are interested
  - In machine learning and deep learning, usually the dimension of input parameters are huge (such as 10^6) while the dimension of output is usually 1 (focusing on the loss function which produce a scalar)
- To find the partial derivative respect to i-th variable $\frac{\partial f}{\partial x_i}$ in the function

$$f = f(x_1, x_2, \cdots, x_n)$$

we can always decompose the function into simple operation nodes

$$v_1, v_2, \cdots, v_p$$

such that we can find some nodes $v_c, c \in [1, M]$ to compose the derivative

$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial v_{c_q}} \frac{\partial v_{c_q}}{\partial v_{c_{q-1}}} \cdots \frac{\partial v_{c_2}}{\partial v_{c_1}} \frac{\partial v_{c_1}}{\partial x_i}$$

Denote

$$\bar{v}_c = \frac{\partial y}{\partial v_c}$$

The idea is to first compute following the **forward evaluation trace** same as that in *forward mode* while recording the **relationship** (such as the differential equation) between nodes, then sequentially calculate

$$\bar{v}_{c_q} = \frac{\partial y}{\partial v_{c_q}}, \bar{v}_{c_{q-1}} = \bar{v}_{c_q} \frac{\partial v_{c_q}}{\partial v_{c_{q-1}}}, \cdots, \frac{\partial f}{\partial x_i} = \bar{v}_{c_1} \frac{\partial v_{c_1}}{\partial x}$$

- The example is still to find the partial derivatives at $x_1 = 2, x_2 = 5$ of the function

  $$f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$$

  Similarly, decompose the function into simple operation nodes

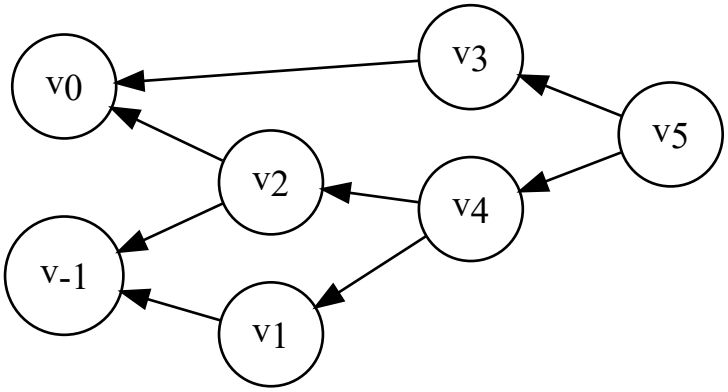
  $$\begin{aligned} v_{-1} &= x_1 \\ v_0 &= x_2 \\ v_1 &= \ln v_{-1} \\ v_2 &= v_{-1} \times v_0 \\ v_3 &= \sin v_0 \\ v_4 &= v_1 + v_2 \\ v_5 &= v_4 - v_3 \\ y &= v_5 \end{aligned}$$

  The **forward evaluation computation graph** would be the same of that in *forward mode*. Here show the **reverse adjoint computation graph**

[36]:
```
f = Digraph('ComputationGraph')
f.attr(rankdir='RL')
f.attr('node', shape='circle')
f.node('v_-1', label='<v<sub>-1</sub>>')
f.node('v_0', label='<v<sub>0</sub>>')
f.node('v_1', label='<v<sub>1</sub>>')
f.node('v_2', label='<v<sub>2</sub>>')
f.node('v_3', label='<v<sub>3</sub>>')
f.node('v_4', label='<v<sub>4</sub>>')
f.node('v_5', label='<v<sub>5</sub>>')
f.edge('v_1', 'v_-1')
f.edge('v_2', 'v_-1')
```

```
f.edge('v_2', 'v_0')
f.edge('v_3', 'v_0')
f.edge('v_4', 'v_1')
f.edge('v_4', 'v_2')
f.edge('v_5', 'v_4')
f.edge('v_5', 'v_3')
f
```

[36]:



- First, compute along the **forward evaluation trace**

$$
\begin{aligned}
v_{-1} = \quad & x_1 = && 2 \\
v_0 = \quad & x_2 = && 5 \\
v_1 = \quad & \ln v_{-1} = && \ln 2 \\
v_2 = \quad & v_{-1} \times v_0 = && 2 \times 5 \\
v_3 = \quad & \sin v_0 = && \sin 5 \\
v_4 = \quad & v_1 + v_2 = && 0.693 + 10 \\
v_5 = \quad & v_4 - v_3 = && 10.693 + 0.959 \\
y = \quad & v_5 = && 11.652
\end{aligned}
$$

Then compute along the **reverse adjoint trace**

$$
\begin{aligned}
\bar{v}_5 &= \frac{\partial y}{\partial v_5} &&= 1 \\
\bar{v}_4 &= \bar{v}_5 \frac{\partial v_5}{\partial v_4} &&= \bar{v}_5 \times 1 = 1 \\
\bar{v}_3 &= \bar{v}_5 \frac{\partial v_5}{\partial v_3} &&= \bar{v}_5 \times (-1) = -1 \\
\bar{v}_1 &= \bar{v}_4 \frac{\partial v_4}{\partial v_1} &&= \bar{v}_4 \times 1 = 1 \\
\bar{v}_2 &= \bar{v}_4 \frac{\partial v_4}{\partial v_2} &&= \bar{v}_4 \times 1 = 1 \\
\bar{v}_0 &= \bar{v}_2 \frac{\partial v_2}{\partial v_0} + \bar{v}_3 \frac{\partial v_3}{\partial v_0} &&= \bar{v}_2 \times v_{-1} + \bar{v}_3 \times \cos v_0 = 1.716 \\
\bar{v}_{-1} &= \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} + \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}} &&= \bar{v}_1 \times \frac{1}{v_{-1}} + \bar{v}_2 \times v_0 = 5.5
\end{aligned}
$$

Therefore, the result is

$$
\begin{aligned}
\frac{\partial f}{\partial x_1} &= \bar{v}_{-1} = 5.5 \\
\frac{\partial f}{\partial x_2} &= \bar{v}_0 = 1.716
\end{aligned}
$$

- **Comments:**
  - For a function $f : \mathbb{R}^n \Rightarrow \mathbb{R}^m$ with input $\mathbf{x} \in \mathbb{R}^n$ and output $\mathbf{y} \in \mathbb{R}^m$, needs to calculate m times along the **reverse adjoint trace**
  - Get the partial derivatives of **one output** respect to **all variables** after computing along the reverse adjoint trace once
    - Fits the idea of calculating **gradients**
  - **High efficiency** when $n >> m$
  - **Low efficiency** when $n << m$
  - Frequently used in ML/DL tasks and is supported by PyTorch, Tensorflow, …

2.2.4 Computation graph and AutoGrad in PyTorch

- As mentioned above, PyTorch supports **reverse mode automatic differentiation**
  - PyTorch automatically creates a computation graph if **requires_grad=True**
  - For a given variable with **requires_grad=True**, identify the operations and record the required information for reverse adjoint trace computation
  - Can be checked by viewing the **requires_grad** attributes of independent variables
  - Use the **make_dots** method from **torchviz** module to visualize the computation graph
    - Need support from **graphviz**

[37]:
```
# The default setting is requires_grad = False
x = torch.tensor(5.)
y = 3*x**2+x
```

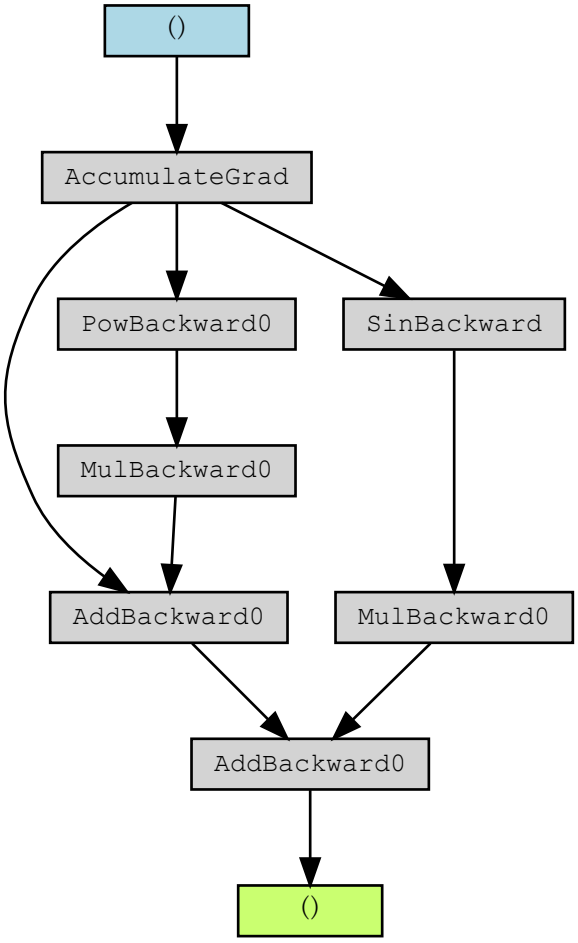
```
print(x,x.requires_grad)
print(y,y.requires_grad)
```

```
tensor(5.) False
tensor(80.) False
```

[38]:
```
# Use the torchviz module for visualization
from torchviz import make_dot
x = torch.tensor(5., requires_grad=True)
y = 3*x**2+x+4*torch.sin(x)
# Independent variable would show the requires_grad attribute
print(x,x.requires_grad)
# Dependent variable would show the grad_fn attribute
print(y,y.requires_grad)
make_dot(y)
```

```
tensor(5., requires_grad=True) True
tensor(76.1643, grad_fn=<AddBackward0>) True
```

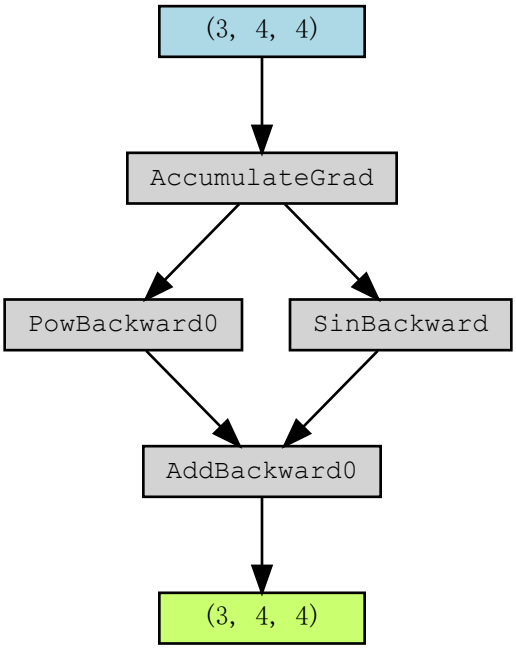
[38]:



- Notice the **grad_fn** attribute. Tensor use this attribute to do the backwards computation.
- When the dimension of input tensor is greater than one, the dimension of each step would also show on the graph
  - Very useful when building neural networks

[39]:
```
x = torch.randn((3,4,4),requires_grad=True)
y = x**2 + torch.sin(x)
make_dot(y)
```

[39]:



- One can access the gradient by calling **backward()** method
- After invoke **backward()** method on the dependent variable we are interested in, the partial derivative would automatically be returned to the **grad** attribute of variables contributed to that dependent variable
- For **independent** variables, we can directly call **backward()** method and see the gradients
- In the following case:

$$z = x_1^2 + x_2^2$$

The partial derivatives at $x_1 = 5, x_2 = 2$ are

$$\frac{\partial z}{\partial x_1} = 2x_1 = 10, \frac{\partial z}{\partial x_2} = 2x_2 = 4$$

[40]:
```
x_1 = torch.tensor(5.,requires_grad=True)
x_2 = torch.tensor(2.,requires_grad=True)
z = x_1**2 + x_2**2
z.backward()
```

```
print(x_1,x_1.grad)
print(x_2,x_2.grad)
print(z)
```

```
tensor(5., requires_grad=True) tensor(10.)
tensor(2., requires_grad=True) tensor(4.)
tensor(29., grad_fn=<AddBackward0>)
```

- For **intermediate dependent** variables (*non-leaf tensor* in pytorch documentation) used in computation, the gradient value is usually not considered and would be cleared after the computation along reverse adjoint trace.
- If the value is indeed needed, the **retain_grad()** method should be invoked

[41]:
```
x_1 = torch.tensor(5.,requires_grad=True)
x_2 = torch.tensor(2.,requires_grad=True)
y = 0.5*x_2**2
y.retain_grad()
z = x_1**2 + y
z.backward()
print(x_1,x_1.grad)
print(x_2,x_2.grad)
print(y,y.grad)
print(z)
```

```
tensor(5., requires_grad=True) tensor(10.)
tensor(2., requires_grad=True) tensor(2.)
tensor(2., grad_fn=<MulBackward0>) tensor(1.)
tensor(27., grad_fn=<AddBackward0>)
```

- It needs to be noticed that the gradients accumulate when calling **backward()**

[42]:
```
x = torch.tensor(5.,requires_grad=True)
for ii in range(2):
    y = 3*x**2
    y.backward()
    print(x,x.grad)
    print(y)
```

```
tensor(5., requires_grad=True) tensor(30.)
tensor(75., grad_fn=<MulBackward0>)
tensor(5., requires_grad=True) tensor(60.)
tensor(75., grad_fn=<MulBackward0>)
```

- Therefore, when calculating the gradients repeated in loops, we usually need to **zero** the gradients before calling **backward()** method, by calling **zero_()** method

[43]:
```
x = torch.tensor(5.,requires_grad=True)
for ii in range(2):
```

```
    try:
        x.grad.zero_()
    except Exception as e:
        print(e)
    y = 3*x**2
    y.backward()
    print(x,x.grad)
    print(y)
```

```
'NoneType' object has no attribute 'zero_'
tensor(5., requires_grad=True) tensor(30.)
tensor(75., grad_fn=<MulBackward0>)
tensor(5., requires_grad=True) tensor(30.)
tensor(75., grad_fn=<MulBackward0>)
```

- Sometimes, we would need to call the **backward()** method multiple times, like using MSE loss and Cross-Entropy loss separately for localization and detection in CV. In this case, **retain_graph** should be set to **True** except in the last **backward()** method if any **intermediate dependent variables** are used.

[44]:
```
x = torch.tensor(5.,requires_grad=True)
y = x**2
z_1 = x + y
z_2 = x**2 + y
z_3 = x**3 + y
z_1.backward(retain_graph=True)
z_2.backward(retain_graph=True)
z_3.backward()
print(x,x.grad)
```

```
tensor(5., requires_grad=True) tensor(116.)
```

- Generally speaking, PyTorch can compute gradients for any number of parameters and any complex functions, as long as the decomposed operation is differentiable

[45]:
```
x = torch.arange(5.).requires_grad_(True)
y = torch.sum(x**2)
y.backward()
print(x)
print(y)
print(x.grad)
```

```
tensor([0., 1., 2., 3., 4.], requires_grad=True)
tensor(30., grad_fn=<SumBackward0>)
tensor([0., 2., 4., 6., 8.])
```

[46]:
```
x = torch.arange(5.).requires_grad_(True)
y = torch.mean(torch.log(x**2+1)+5*x)
y.backward()
```

```
print(x)
print(y)
print(x.grad)
```

```
tensor([0., 1., 2., 3., 4.], requires_grad=True)
tensor(11.4877, grad_fn=<MeanBackward0>)
tensor([1.0000, 1.2000, 1.1600, 1.1200, 1.0941])
```

[ ]: