周报

Shixiang Yan

2020 年 2 月 6 日

目录

1 关于卫星天线仿真

我认为应该先从软件着手，对于 STK 我发现了有 Matlab，C，C++，C#，Java 的接口。综合考虑开发难度，开发工作量以及业务需求，Matlab 是最可靠的接口开发工具。本周我学习了 MATLAB 的相关知识，对其工作流程有了大致的了解。

对于总体设计，我的想法是先在一台计算机上把所需功能跑出来，然后再考虑分布式到多台电脑上并行计算。下周我准备利用 Matlab 开发一个简单的可以与 STK 进行交互可视化程序。

以下是我近期对 Matlab 学习所书写的部分代码：

1.1 结构

1.1.1 顺序

1.1.2 选择

```
%switch
input_num=1;
switch input_num
case -1
disp('negative 1');
case 0
disp('zero');
case 1
disp('positive 1');
otherwise
disp('other value');
end
```

1.1.3 循环

```
%while循环
n=1;
while prod(1:n)<1e100 %Á¬³Ë
n=n+1;
end
disp(n)
%for循环
for n=1:10
a(n)=2^n;
end
disp(a)
%%
tic
for ii=1:2000
for jj=1:2000
A(ii,jj)=ii+jj;
```

```
end
end
toc
%%
tic
A=zeros(2000,2000);
for ii=1:size(A,1)
for jj=1:size(A,2)
A(ii,jj)=ii+jj;
end
end
toc
```

1.2 matlab 声明函数

```
function result = fun2( a,b )
%UNTITLED2 此处显示有关此函数的摘要
%   此处显示详细说明
s=0;
for i=a:b
s=s+i;
end
result=s;
end
```

1.3 画图

1.3.1 正弦曲线

```
hold on
plot(cos(0:pi/20:pi*2));
plot(sin(0:pi/20:pi*2),'xg:');
hold off
```

如图 1:

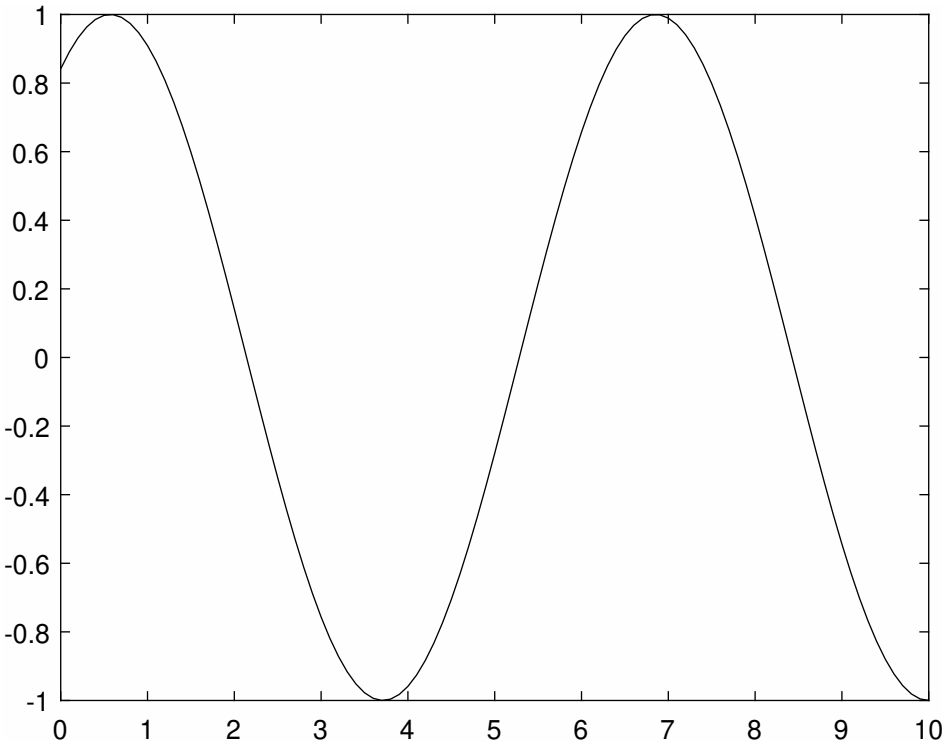

图 1: 正弦曲线

2 关于 Pytorch 模块的学习

2.1 numpy 和 pytorch

以下是近期我对 Pytorch 进行学习所书写的部分代码。

```
#官网说明书：https://pytorch.org/docs/stable/torch.html
import torch
import numpy as np

np_data = np.arange(6).reshape((2, 3))
torch_data = torch.from_numpy(np_data)
tensor2array = torch_data.numpy()
print(
'\nnumpy array:', np_data,          # [[0 1 2], [3 4 5]]
'\ntorch tensor:', torch_data,      #  0  1  2 \n 3  4  5    [torch.
                                  LongTensor of size 2x3]
'\ntensor to array:', tensor2array, # [[0 1 2], [3 4 5]]
)
```

```
# abs 绝对值计算
data = [-1, -2, 1, 2]
tensor = torch.FloatTensor(data)  # 转换成32位浮点 tensor
print(
'\nabs',
'\nnumpy: ', np.abs(data),          # [1 2 1 2]
'\ntorch: ', torch.abs(tensor)      # [1 2 1 2]
)

# sin   三角函数 sin
print(
'\nsin',
'\nnumpy: ', np.sin(data),      # [-0.84147098 -0.90929743  0.84147098
                                  0.90929743]
'\ntorch: ', torch.sin(tensor)  # [-0.8415 -0.9093  0.8415  0.9093]
)

# mean  均值
print(
'\nmean',
'\nnumpy: ', np.mean(data),          # 0.0
'\ntorch: ', torch.mean(tensor)     # 0.0
)

# matrix multiplication 矩阵点乘
data = [[1,2], [3,4]]
tensor = torch.FloatTensor(data)  # 转换成32位浮点 tensor
# correct method
print(
'\nmatrix multiplication (matmul)',
'\nnumpy: ', np.matmul(data, data),     # [[7, 10], [15, 22]]
'\ntorch: ', torch.mm(tensor, tensor)   # [[7, 10], [15, 22]]
)

# !!!!  下面是错误的方法 !!!!
data = np.array(data)
print(
'\nmatrix multiplication (dot)',
'\nnumpy: ', data.dot(data),        # [[7, 10], [15, 22]] 在numpy 中可行
# '\ntorch: ', tensor.dot(tensor)     # torch 会转换成 [1,2,3,4].dot([1,
                                      2,3,4) = 30.0
#'tensor.dot(tensor)',      torch 会转换成 [1,2,3,4].dot([1,2,3,4) = 30.
                                      0

# 变为
# '\ntorch:',torch.dot(tensor.dot(tensor))
```

```
)
\subsection{有关variable}
\begin{python}
  import torch
  from torch.autograd import Variable # torch 中 Variable 模块

  # 先生鸡蛋
  tensor = torch.FloatTensor([[1,2],[3,4]])
  # 把鸡蛋放到篮子里, requires_grad是参不参与误差反向传播, 要不要计算梯
                                    度
  variable = Variable(tensor, requires_grad=True)

  print(tensor)
  """
  1  2
  3  4
  [torch.FloatTensor of size 2x2]
  """

  print(variable)
  """
  Variable containing:
  1  2
  3  4
  [torch.FloatTensor of size 2x2]
  """

  t_out = torch.mean(tensor*tensor)       # x^2
  v_out = torch.mean(variable*variable)   # x^2
  print(t_out)
  print(v_out)    # 7.5

  v_out.backward()    # 模拟 v_out 的误差反向传递,反向传递求梯度。

  # 下面两步看不懂没关系, 只要知道 Variable 是计算图的一部分, 可以用来传
                                    递误差就好.
  # v_out = 1/4 * sum(variable*variable) 这是计算图中的 v_out 计算步骤
  # 针对于 v_out 的梯度就是, d(v_out)/d(variable) = 1/4*2*variable =
                                    variable/2

  print(variable.grad)    # 初始 Variable 的梯度
  '''
  0.5000  1.0000
  1.5000  2.0000
  '''
```

```
print(variable)     #  Variable 形式
"""
Variable containing:
1  2
3  4
[torch.FloatTensor of size 2x2]
"""

print(variable.data)    # tensor 形式
"""
1  2
3  4
[torch.FloatTensor of size 2x2]
"""

print(variable.data.numpy())    # numpy 形式
"""
[[ 1.  2.]
[ 3.  4.]]
"""
```

2.2 激活函数

```
import torch
import torch.nn.functional as F     # 激励函数都在这
from torch.autograd import Variable

# 做一些假数据来观看图像
x = torch.linspace(-5, 5, 200)  # x data (tensor), shape=(100, 1)
x = Variable(x)

x_np = x.data.numpy()   # 换成 numpy array, 出图时用,张量转numpy

# 几种常用的 激励函数
y_relu = torch.relu(x).data.numpy()
y_sigmoid = torch.sigmoid(x).data.numpy()
y_tanh = torch.tanh(x).data.numpy()
y_softplus = F.softplus(x).data.numpy()
# y_softmax = F.softmax(x)  softmax 比较特殊, 不能直接显示, 不过他是关于
                                概率的, 用于分类

import matplotlib.pyplot as plt  # python 的可视化模块, 我有教程 (https
                                 ://morvanzhou.github.io/tutorials/data
                                 -manipulation/plt/)
```

```
plt.figure(1, figsize=(8, 6))
plt.subplot(221)
plt.plot(x_np, y_relu, c='red', label='relu')
plt.ylim((-1, 5))
plt.legend(loc='best')

plt.subplot(222)
plt.plot(x_np, y_sigmoid, c='red', label='sigmoid')
plt.ylim((-0.2, 1.2))
plt.legend(loc='best')

plt.subplot(223)
plt.plot(x_np, y_tanh, c='red', label='tanh')
plt.ylim((-1.2, 1.2))
plt.legend(loc='best')

plt.subplot(224)
plt.plot(x_np, y_softplus, c='red', label='softplus')
plt.ylim((-0.2, 6))
plt.legend(loc='best')

plt.show()
```

2.3 Regressin

```
import torch
import matplotlib.pyplot as plt

x = torch.unsqueeze(torch.linspace(-1, 1, 100), dim=1)  \# x data (
                                    tensor), shape=(100, 1),数据维度进行扩
                                    充
y = x.pow(2) + 0.2*torch.rand(x.size())                 \# noisy y data
                                    (tensor), shape=(100, 1),生成与x形状相
                                    同的y

\# 画图
plt.scatter(x.data.numpy(), y.data.numpy())
plt.show()

class Net(torch.nn.Module):  \# 继承 torch 的 Module
def \_\_init\_\_(self, n\_feature, n\_hidden, n\_output):
super(Net, self).\_\_init\_\_()     \# 继承 \_\_init\_\_ 功能
\# 定义每层用什么样的形式
```

```
self.hidden = torch.nn.Linear(n\_feature, n\_hidden)   \# 隐藏层线性输出
                                    ,nn.Linear表示y=wx+b
self.predict = torch.nn.Linear(n\_hidden, n\_output)   \# 输出层线性输出

def forward(self, x):   \# 这同时也是 Module 中的 forward 功能
\# 正向传播输入值, 神经网络分析出输出值
x = torch.relu(self.hidden(x))      \# 激励函数(隐藏层的线性值)
x = self.predict(x)             \# 输出值
return x

net = Net(n\_feature=1, n\_hidden=10, n\_output=1)

print(net)  \# net 的结构
"""
Net (
(hidden): Linear (1 -> 10)
(predict): Linear (10 -> 1)
)
"""

\# optimizer 是训练的工具
optimizer = torch.optim.SGD(net.parameters(), lr=0.2)  \# 传入 net 的所
                                    有参数, 学习率
loss\_func = torch.nn.MSELoss()      \# 预测值和真实值的误差计算公式 (均
                                    方差)

for t in range(100):
prediction = net(x)     \# 喂给 net 训练数据 x, 输出预测值

loss = loss\_func(prediction, y)     \# 计算两者的误差

optimizer.zero\_grad()   \# 清空上一步的残余更新参数值
loss.backward()         \# 误差反向传播, 计算参数更新值
optimizer.step()        \# 将参数更新值施加到 net 的 parameters 上

\# import matplotlib.pyplot as plt

plt.ion()   \# 画图
plt.show()

for t in range(50):

prediction = net(x)  \# 喂给 net 训练数据 x, 输出预测值

loss = loss\_func(prediction, y)  \# 计算两者的误差
```

```
optimizer.zero\_grad()  \# 清空上一步的残余更新参数值
loss.backward()
optimizer.step()

\# 接着上面来
if t \% 10 == 0:
\# plot and show learning process
plt.cla()
plt.scatter(x.data.numpy(), y.data.numpy())
plt.plot(x.data.numpy(), prediction.data.numpy(), 'r-', lw=5)
plt.text(0.5, 0, 'Loss=\%.4f' \% loss.data.numpy(), fontdict=\{'size':
                                    20, 'color':  'red'\})
plt.pause(0.1)
```

2.4 classification

```
import torch
import matplotlib.pyplot as plt

# 假数据
n_data = torch.ones(100, 2)          # 数据的基本形态

x0 = torch.normal(2*n_data, 1)       # 类型0 x data (tensor), shape=(
                                    100, 2)
help(torch.normal)
print(x0)
y0 = torch.zeros(100)                # 类型0 y data (tensor), shape=(
                                    100, )
x1 = torch.normal(-2*n_data, 1)      # 类型1 x data (tensor), shape=(
                                    100, 1)
y1 = torch.ones(100)                 # 类型1 y data (tensor), shape=(
                                    100, )

# 注意 x, y 数据的数据形式是一定要像下面一样 (torch.cat 是在合并数据)
x = torch.cat((x0, x1), 0).type(torch.FloatTensor)  # FloatTensor = 32
                                    -bit floating
y = torch.cat((y0, y1), ).type(torch.LongTensor)    # LongTensor = 64-
                                    bit integer

plt.scatter(x.data.numpy()[:, 0], x.data.numpy()[:, 1], c=y.data.numpy
                                    (), s=100, lw=0, cmap='RdYlGn')
plt.show()

# 画图
```

```
# plt.scatter(x.data.numpy(), y.data.numpy())
# plt.show()

import torch
import torch.nn.functional as F     # 激励函数都在这

#搭建正向传递网络方式一:
class Net(torch.nn.Module):     # 继承 torch 的 Module
def __init__(self, n_feature, n_hidden, n_output):
super(Net, self).__init__()     # 继承 __init__ 功能
self.hidden = torch.nn.Linear(n_feature, n_hidden)   # 隐藏层线性输出
self.out = torch.nn.Linear(n_hidden, n_output)       # 输出层线性输出

def forward(self, x):
# 正向传播输入值, 神经网络分析出输出值
x = F.relu(self.hidden(x))      # 激励函数(隐藏层的线性值)
x = self.out(x)                 # 输出值, 但是这个不是预测值, 预测值还
                                需要再另外计算
return x

net = Net(n_feature=2, n_hidden=10, n_output=2) # 几个类别就几个
                                output

print(net)  # net 的结构
"""
Net (
(hidden): Linear (2 -> 10)
(out): Linear (10 -> 2)
)
"""
# #搭建正向传递方式二:
# net2 = torch.nn.Sequential(
#     torch.nn.Linear(2, 10),
#     torch.nn.ReLU(),
#     torch.nn.Linear(10, 2)
# )


# optimizer 是训练的工具
optimizer = torch.optim.SGD(net.parameters(), lr=0.02)  # 传入 net 的
                                所有参数, 学习率
# 算误差的时候, 注意真实值!不是! one-hot 形式的, 而是1D Tensor, (batch
                                ,)
# 但是预测值是2D tensor (batch, n_classes)
loss_func = torch.nn.CrossEntropyLoss()
```

```
for t in range(100):
out = net(x)     # 喂给 net 训练数据 x, 输出分析值

loss = loss_func(out, y)     # 计算两者的误差

optimizer.zero_grad()   # 清空上一步的残余更新参数值
loss.backward()         # 误差反向传播, 计算参数更新值
optimizer.step()        # 将参数更新值施加到 net 的 parameters 上



import matplotlib.pyplot as plt

plt.ion()   # 画图
plt.show()

# for t in range(100):
#
#     out = net(x)  # 喂给 net 训练数据 x, 输出分析值
#
#     loss = loss_func(out, y)  # 计算两者的误差
#     loss.backward()
#     optimizer.step()
#
#     # 接着上面来
#     if t % 2 == 0:
#         plt.cla()
#         # 过了一道 softmax 的激励函数后的最大概率才是预测值
#         prediction = torch.max(F.softmax(out), 1)[1]
#         pred_y = prediction.data.numpy().squeeze()
#         target_y = y.data.numpy()
#         plt.scatter(x.data.numpy()[:, 0], x.data.numpy()[:, 1], c=
                                pred_y, s=100, lw=0, cmap='RdYlGn')
#         accuracy = sum(pred_y == target_y)/200.  # 预测中有多少和真
                                实值一样
#         plt.text(1.5, -4, 'Accuracy=%.2f' % accuracy, fontdict={'
                                size': 20, 'color':  'red'})
#         plt.pause(0.1)
#
# plt.ioff()  # 停止画图
# plt.show()
```

2.5 save and load 参数

```
import torch
torch.manual_seed(1)    # reproducible

# 假数据
x = torch.unsqueeze(torch.linspace(-1, 1, 100), dim=1)  # x data (tensor
                                    ), shape=(100, 1)
y = x.pow(2) + 0.2*torch.rand(x.size())  # noisy y data (tensor), shape
                                    =(100, 1)

def save():
# 建网络
net1 = torch.nn.Sequential(
torch.nn.Linear(1, 10),
torch.nn.ReLU(),
torch.nn.Linear(10, 1)
)
optimizer = torch.optim.SGD(net1.parameters(), lr=0.5)
loss_func = torch.nn.MSELoss()

# 训练
for t in range(100):
prediction = net1(x)
loss = loss_func(prediction, y)

optimizer.zero_grad()
loss.backward()
optimizer.step()

torch.save(net1, 'net.pkl')  # 方式一：保存整个网络
torch.save(net1.state_dict(), 'net_params.pkl')   # 方式二：只保存网络中
                                    的参数（速度快，占内存少）

#这种方式将会提取整个神经网络，网络大的时候可能会比较慢.
def restore_net():
# restore entire net1 to net2
net2 = torch.load('net.pkl')
prediction = net2(x)

#这种方式将会提取所有的参数，然后再放到你的新建网络中.
def restore_params():
# 新建 net3
net3 = torch.nn.Sequential(
torch.nn.Linear(1, 10),
torch.nn.ReLU(),
```

```
    torch.nn.Linear(10, 1)
    )

    # 将保存的参数复制到 net3
    net3.load_state_dict(torch.load('net_params.pkl'))
    prediction = net3(x)
```