实验三-LSTM实验

实验内容

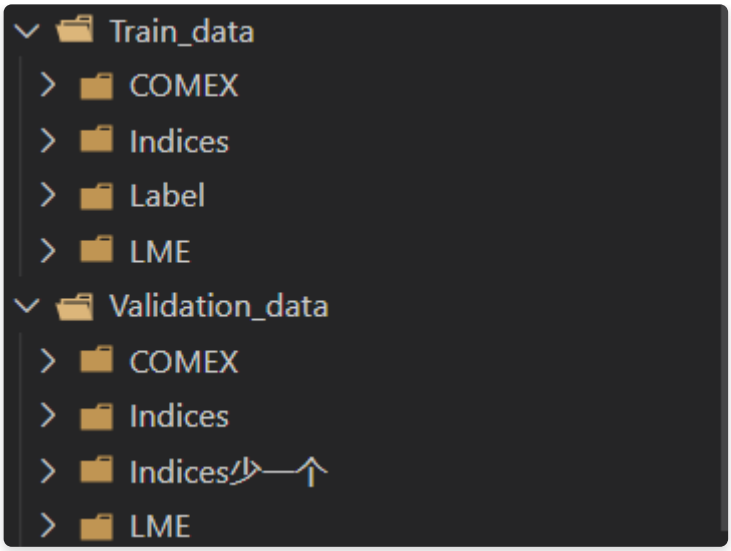
- 预测铜、铝、铅、镍、锌以及锡大宗商品的价格走势
- 分别预测其价格1天、20天、60天的涨跌
- 预测指标为三个时间段预测的准确率
- 每个时间段计算六个金属预测

数据集处理方法和实现方案

主要功能实现见`MyDataLoader.py`文件

数据集文件分类

实现中，为了方便处理，将提供的数据集分类如下:

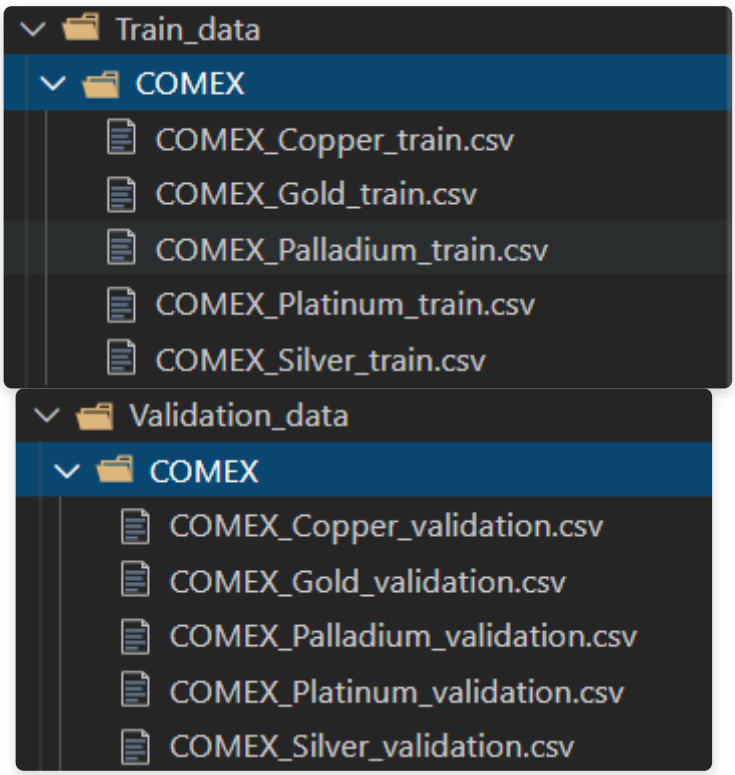


原始数据集分类目录

Train_data和Validation_data两个文件下分别存储用于训练和测试的数据集，每个数据集根据前缀放在不同的文件夹下，例如以LME开头的训练文件，都将放在Train_data/LME目录中，详情可查看提交文件。

文件索引方式

鉴于上述的文件存储方式，可以发现每个文件夹下的文件名，仅有一部分不同，例如**Train_data/COMEX**文件下的五个文件，如下图所示:



文件索引示例

这五个文件仅仅是金属名称不同，文件中内容的结构也相似。事实上，以COMEX开头的是个文件，只要两个属性$kind, usage$,这里
$kind \in \{Copper, Gold, Palladium, Platinum, Silver\}, usage \in \{train, validation\}$，就可以根据他们的组合，获得这十个文件。

同理，对于Indices，Label，LME，LME_3M开头的文件，也有相似的特征。

数据处理实现

数据读取

训练集数据和标签读取，测试集数据读取

- 根据前面所述的文件索引方式，项目中实现了以下几个函数，详情见`MyDataLoader.py`文件。

```
def load_COMEX_Data(kind = "Copper", usage = "train")
def load_Indices_Data(kind = "NKY", usage = "train")
def load_LME_Data(kind = "Copper", usage = "train")
def load_LME_3M_Data(kind = "Copper", usage = "train")
def load_LME_Label(kind = "Copper", seq = "1d")
```

    - **load_COMEX_Data**：根据输入的参数可以读取以COMEX为开头的文件
    - **load_Indices_Data**：根据输入的参数可以读取以Indices为开头的文件
    - **load_LME_Data:** 根据输入的参数可以读取以LME为开头但不包含3M的文件
    - **load_LME_3M_Data**：根据输入的参数可以读取以LME_3M为开头的文件
    - **load_LME_Label**：根据输入的参数可以读取以Label为开头的文件
- 为了方便所有数据检查，前面所述方式读取的文件经过处理后，都将会保存在**DataFolders**文件夹下，详情可查看提交目录
- 为了方便所有数据的读取，项目中实现了前面功能函数的多次调用，以**字典**形式返回同一类前缀的数据，可以根据**金属名**获取数据，详情见`MyDataLoader.py`文件。

```
def load_COMEX_Train_Validation()
def load_Indices_Train_Validation()
def load_LME_Train_Validation()
def load_LME_3M_Train_Validation()
def load_LME_Label_1d()
def load_LME_Label_20d()
def load_LME_Label_60d()
```

    - **load_COMEX_Train_Validation**：根据输入的参数可以读取以COMEX为开头的所有文件，包括训练集和测试集
    - **load_Indices_Train_Validation**：根据输入的参数可以读取以Indices为开头的文件，包括训练集和测试集
    - **load_LME_Train_Validation**：根据输入的参数可以读取以LME为开头但不包含3M的文件，包括训练集和测试集
    - **load_LME_3M_Train_Validation**：根据输入的参数可以读取以LME_3M为开头的文件，包括训练集和测试集
    - **load_LME_Label_1d**：根据输入的参数可以读取以Label为开头的文件,且用于1d预测的训练集标签
    - **load_LME_Label_20d**：根据输入的参数可以读取以Label为开头的文件，且用于20d预测的训练集标签
    - **load_LME_Label_60d**：根据输入的参数可以读取以Label为开头的文件，且用于60d预测的训练集标签

测试集标签读取

- 测试集的标签是所有金属，所有预测周期的数据都混合在了一起，因此这这部分的数据处理需要单独处理。
- 项目中实现了`load_Validation_Label`函数用于读取测试集每个金属的每个周期的标签数据，代码如下，详情见`MyDataLoader.py`文件。

```
def load_Validation_Label():
    Validation_Label_name = ["raw_id", "label"]
    Validation_Label_path = "Validation_data" + "/" +
"validation_label_file" + ".csv"
    Validation_Label = pd.read_csv(Validation_Label_path,skiprows = 1,
names = Validation_Label_name)

    Label_name = ["Aluminium","Copper","Lead","Nickel","Tin","Zinc"]
    Seq_name =  ["1d","20d","60d"]

    s = {}
    for label_name in Label_name:
        for seq_name in Seq_name:
            p = "LME" + label_name + "-validation-" + seq_name + "-.*"
            t =
pd.DataFrame(Validation_Label.loc[Validation_Label["raw_id"].str.contains
(p)])
            t["raw_id"] = t["raw_id"].apply(lambda x: x[-10:])
            t.rename(columns={'raw_id':'date'},inplace=True)
            s[label_name + seq_name] = t
            outFolderName = "Validation_data" +
"/Split_Validation_Label/" + label_name + "_" + seq_name +
"_split_handler.csv"
            t.to_csv(outFolderName,index = False,sep=',')
    return  s
```

- 文件处理结果将会根据金属名字和预测周期两个属性，将处理文件保存到**Validation_data/Split_Validation_Label**下中，具体可查看提交目录
- 为方便多次调用，文件处理仍返回一个字典，存储所有的测试集标签，可以根据**金属名**和**预测周期**获取，例如以字符串"Copper" + "1d"为索引，将可以获得**金属铜预测周期为1天的测试集标签。**

至此 ,所有提供的数据将全部处理完毕。

数据清洗

在数据处理过程中，会出现空值等情况，另一方面的，不同特征的尺度差异较大，为方便处理，对数据处理如下

- 对于某一天的某一个属性值为空的情况，数据将会直接丢弃
- 为模型处理，所有的数据，通过可以通过**max-min**归一化，经过观察数据差异，对此进行了一些更改，表达式如下，实现代码见`MyDataLoader.py`文件

$$scaler = x_i - mean(x)/(max(x) = min(x)) \quad (1)$$

LSTM模型原理及实现方案

主要功能实现见`LSTM_Batch_MultiLayer.py`文件

实验中默认了batch_first = True，dropout = 0，num_direction = 1。

实验中实现了可以支持批处理，多层的lstm。

LSTM实现

参考pytorch的官方文档如下，对于多层的LSTM，只需要在$n \geq 2$时，输入更改为第$n-1$层的输出h_t即可。

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
i_t &= \sigma(W_{ii} x_t + b_{ii} + W_{hi} h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if} x_t + b_{if} + W_{hf} h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig} x_t + b_{ig} + W_{hg} h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io} x_t + b_{io} + W_{ho} h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

where h_t is the hidden state at time *t*, c_t is the cell state at time *t*, x_t is the input at time *t*, h_{t-1} is the hidden state of the layer at time *t-1* or the initial hidden state at time *0*, and i_t , f_t , g_t , o_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function, and $\odot$ is the Hadamard product.

In a multilayer LSTM, the input $x_t^{(l)}$ of the l -th layer ($l >= 2$) is the hidden state $h_t^{(l-1)}$ of the previous layer multiplied by dropout $\delta_t^{(l-1)}$ where each $\delta_t^{(l-1)}$ is a Bernoulli random variable which is 0 with probability `dropout`.

a multi-layer long short-term memory (LSTM)

实现的LSTM类简要说明

代码详情见`LSTM_Batch_MultiLayer.py`中的`MyLSTM`类

```
You, seconds ago | 1 author (You)
class MyLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers = 1, batch_first = True):···

    def reset_weigths(self):···

    def forward(self, input, hx = None):  ···
```

实现的MyLSTM主要函数

- **__init__**:初始化
  该部分除了对输入输出等参数设置以外，还注册了输出层隐藏层的权重和偏置参数，其具体设置如下所示:

- **~LSTM.weight_ih_l[k]** – the learnable input-hidden weights of the k^{th} layer (*W_ii|W_if|W_ig|W_io*), of shape (*4*hidden_size, input_size*) for *k = 0*. Otherwise, the shape is (*4*hidden_size, num_directions * hidden_size*)
- **~LSTM.weight_hh_l[k]** – the learnable hidden-hidden weights of the k^{th} layer (*W_hi|W_hf|W_hg|W_ho*), of shape (*4*hidden_size, hidden_size*)
- **~LSTM.bias_ih_l[k]** – the learnable input-hidden bias of the k^{th} layer (*b_ii|b_if|b_ig|b_io*), of shape (*4*hidden_size*)
- **~LSTM.bias_hh_l[k]** – the learnable hidden-hidden bias of the k^{th} layer (*b_hi|b_hf|b_hg|b_ho*), of shape (*4*hidden_size*)

LSTM输入层，输出层的参数设置

实现中的核心代码如下:

```
# input _layer 输入层参数定义
self.weight_ih_l0 = Parameter(Tensor(4 * self.hidden_size,
self.num_direction * self.input_size))
self.weight_hh_l0 = Parameter(Tensor(4 * self.hidden_size,
self.hidden_size))
self.bias_ih_l0 = Parameter(Tensor(4 * hidden_size))
self.bias_hh_l0 = Parameter(Tensor(4 * hidden_size))

# hidden layer 隐含层参数定义
for i in range(1, num_layers):
    weight_ih_li = Parameter(Tensor(4 * self.hidden_size,
self.num_direction * self.hidden_size))
    weight_hh_li = Parameter(Tensor(4 * self.hidden_size,
self.hidden_size))
    bias_ih_li = Parameter(Tensor(4 * self.hidden_size))
    bias_hh_li = Parameter(Tensor(4 * self.hidden_size))
    self.register_parameter('weight_ih_l' + str(i) , weight_ih_li)
    self.register_parameter('weight_hh_l' + str(i) , weight_hh_li)
    self.register_parameter('bias_ih_l' + str(i) , bias_ih_li)
    self.register_parameter('bias_hh_l' + str(i) , bias_hh_li)
```

- **reset_weights**:

  这部分采用pytorch的均匀分布的方式进行权重的初始化即可。

```
def reset_weigths(self):
    stdv = 1.0 / math.sqrt(self.hidden_size)
    for weight in self.parameters():
        init.uniform_(weight, -stdv, stdv)
```

- **forward**:

  MyLSTM的前向传递，公式原理已经说明。

  实现中的核心代码如下:

```
hidden_seq = [] # 存储结果

for seq in range(seq_size): # 依次传入序列
    x_t = input[:, seq, :].t()
```

```
    i, f, g, o = self.split["i"], self.split["f"], self.split["g"],
self.split["o"]

    for tp in range(self.num_layers): # 依次计算每一层的输出
        h_tp = h_t[tp,:,:].t().clone()
        c_tp = c_t[tp,:,:].t().clone()

        i_t = torch.sigmoid(self.weight_ih[tp][i] @ x_t +
self.bias_ih[tp][i].unsqueeze(0).t() + self.weight_hh[tp][i] @ h_tp +
self.bias_hh[tp][i].unsqueeze(0).t())
        f_t = torch.sigmoid(self.weight_ih[tp][f] @ x_t +
self.bias_ih[tp][f].unsqueeze(0).t() + self.weight_hh[tp][f] @ h_tp +
self.bias_hh[tp][f].unsqueeze(0).t())
        g_t =    torch.tanh(self.weight_ih[tp][g] @ x_t +
self.bias_ih[tp][g].unsqueeze(0).t() + self.weight_hh[tp][g] @ h_tp +
self.bias_hh[tp][g].unsqueeze(0).t())
        o_t = torch.sigmoid(self.weight_ih[tp][o] @ x_t +
self.bias_ih[tp][o].unsqueeze(0).t() + self.weight_hh[tp][o] @ h_tp +
self.bias_hh[tp][o].unsqueeze(0).t())

        c_tp = f_t * c_tp + i_t * g_t
        h_tp = o_t * torch.tanh(c_tp)

        x_t = h_tp # 隐含层输入修正

        c_tp = c_tp.t().unsqueeze(0)
        h_tp = h_tp.t().unsqueeze(0)
        h_t[tp,:,:] = h_tp
        c_t[tp,:,:] = c_tp
    hidden_seq.append(h_tp)
hidden_seq = torch.cat(hidden_seq, dim=0)
hidden_seq = torch.transpose(hidden_seq, 0, 1)
```

LSTM验证

- 模型的验证可以通过直接运行`LSTM_Batch_MultiLayer.py`文件查看（也可以通过运行后面搭建的实验模型，更改其中的lstm模型为官方的LSTM进行比较）。
- 运行`LSTM_Batch_MulLtiLayer.py`的文件的话，该函数不固定随机种子，以保证功能的正确性。

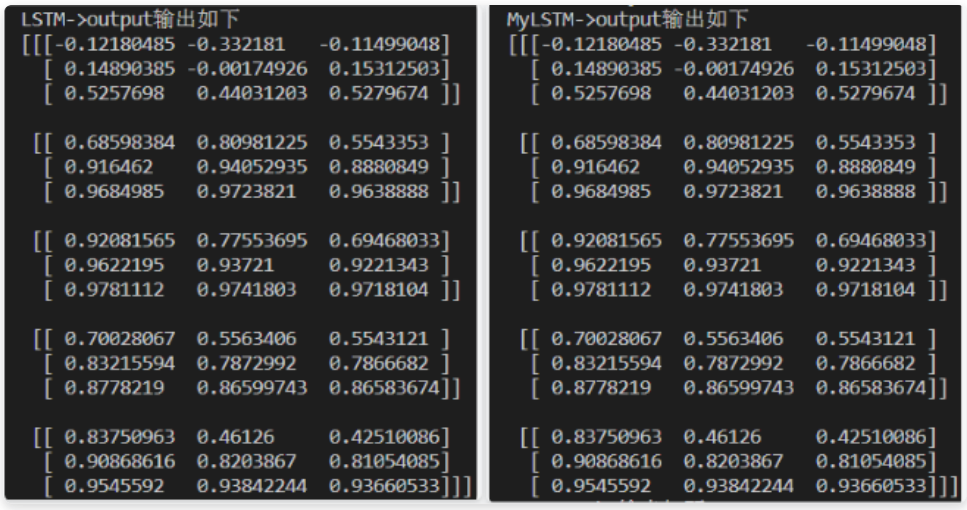
```
input = torch.randn(5, 3, 2)
h0 = torch.randn(2, 5, 3)
c0 = torch.randn(2, 5, 3)
rnn = nn.LSTM(input_size = 2, hidden_size = 3, num_layers = 2,
batch_first = True)
print("LSTM库的输出")
reset_weigths(rnn)
output, (hn, cn) = rnn(input, (h0, c0))
print("LSTM->output输出如下")
print(output.detach().numpy())
print("LSTM->hn输出如下")
print(hn.detach().numpy())
print("LSTM->cn输出如下")
print(cn.detach().numpy())

print("\n")
```

```
myrnn = MyLSTM(input_size = 2, hidden_size = 3, num_layers = 2,
batch_first = True)
print("自己实现的MyLSTM类的输出")
reset_weigths(myrnn)
myoutput, (myhn, mycn) = myrnn(input, (h0, c0))
print("MyLSTM->output输出如下")
print(myoutput.detach().numpy())
print("MyLSTM->hn输出如下")
print(myhn.detach().numpy())
print("MyLSTM->cn输出如下")
print(mycn.detach().numpy())
```

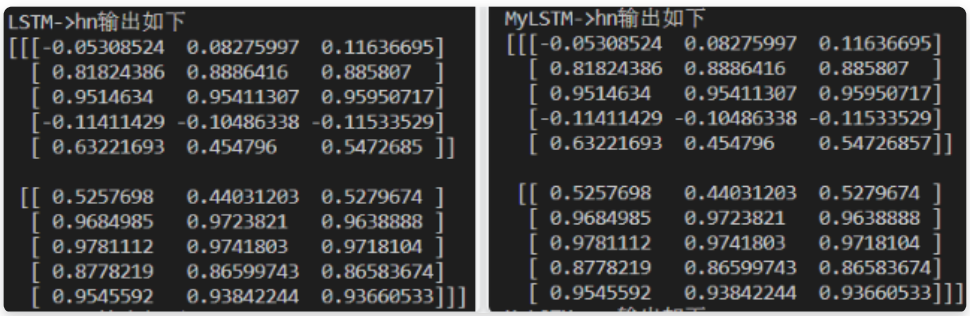
测试程序中，创建了shape为(5,3,2)的输入，通过分别调用nn.LSTM和MyLSTM进行计算，例如当随机种子为10时，终端输出将会为:（可以通过调用实现的set_seed函数确定随机种子）

- 调用nn.LSTM和MyLSTM的output结果:



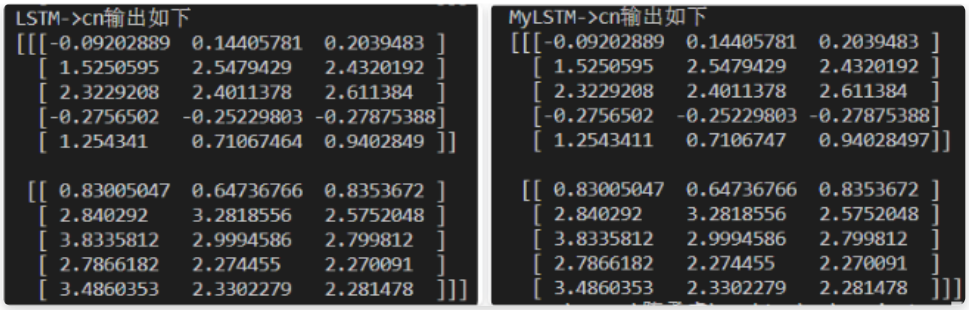
LSTM_Batch_MulLtiLayer终端输出1(随机种子为10时)

- 调用nn.LSTM和MyLSTM的hn结果:



LSTM_Batch_MulLtiLayer终端输出2(随机种子为10时)

- 调用nn.LSTM和MyLSTM的cn结果:



LSTM_Batch_MulLtiLayer终端输出3(随机种子为10时)

可以看出MyLSTM和nn.LSTM的输出一致。

预测模型构建和训练

代码详情见`MyLSTM_Stock.py`中的`LSTM_Stock`类

模型首先经过一个lstm机，随后通过全连接层输出。

```
class LSTM_Stock(nn.Module):
    def __init__(self, input_size=8, hidden_size=32, num_layers=1 , output_size=1 , batch_first=True):···

    def forward(self, x):···

    def train(self, args, train_loader, criterion, optimizer):···

    def train_test(self, args, train_loader):···

    def test_test(self, args, test_loader, file_name):···
```

Lenet5卷积网络

- `_init_`:初始化，实现网络输入，隐含层维度，隐含层层数，输出尺度等参数的设置，以及构建lstm子层
- `forward`: 前向传递数据
- `train`: 模型的训练
- `train_test`: 训练好的模型上，检查训练集的准确率
- `test_test`: 训练好的模型上，检查测试集的准确率

  具体实现方式请查看`MyLSTM_Stock.py`文件

实验模型和运行

平台说明

- 开发工具: VSCode 1.50.1
- OS: Windows_NT x64 10.0.18363
- 编程语言: Python3.7.6
- 显卡: GeForce RTX 2060

LSTM模型文件

- MyLSTM_Stock.py：内容在前面已经说明，直接运行该文件即可。

六个金属的1d预测

代码详情见`main_1d.py`文件，直接运行该文件即可。

随机种子固定为10；

迭代次数:400； 隐含层层数:2；输入特征数: 42；隐含层维度256；

学习率: 0.0001；序列长度:14；批长度:16；

```
# 固定随机种子
set_seed(10)
# 参数设置
args.epochs = 400
args.layers = 2
args.input_size = 42
args.hidden_size = 256
args.lr = 0.0001
args.sequence_length = 14
args.batch_size  = 16
```

main_1d参数设置情况

终端输出如下，平均准确率为 55.12%：

```
------------训练集--------------
金属次序: Copper Aluminium, Lead, Nickel, Tin, Zinc
训练集样本个数:
2949 2949 2949 2949 2949 2949
训练集正确预测个数:
2588 1979 2045 1952 1886 2095
训练集准确率:
87.76% 67.11% 69.35% 66.19% 63.95% 71.04%
平均准确率:
70.9
------------测试集--------------
金属次序: Copper Aluminium, Lead, Nickel, Tin, Zinc
测试集样本个数:
205 205 205 205 205 205
测试集正确预测个数:
117 110 113 117 116 105
测试集准确率:
57.07 53.66 55.12 57.07 56.59 51.22
平均准确率:
55.12
base: 53.16, baseline: 55.01
model with acc 55.12% is saved!
```

main_1d运行后的终端输出

六个金属的20d预测

代码详情见`main_20d.py`文件，直接运行该文件即可。

随机种子固定为10；

迭代次数:150； 隐含层层数:1；输入特征数: 66；隐含层维度256；

学习率: 0.0001；序列长度:14；批长度:16；

```
# 固定随机种子
set_seed(10)
# 参数设置
args.epochs = 150
args.layers = 1
args.input_size = 66
args.hidden_size = 256
args.lr = 0.0001
args.sequence_length = 14
args.batch_size  = 16
```

main_20d参数设置情况

终端输出如下，平均准确率为 59.12%：

```
------------训练集--------------
金属次序: Copper Aluminium, Lead, Nickel, Tin, Zinc
训练集样本个数:
971 971 971 971 971 971
训练集正确预测个数:
819 597 650 659 549 652
训练集准确率:
84.35% 61.48% 66.94% 67.87% 56.54% 67.15%
平均准确率:
67.39
------------测试集--------------
金属次序: Copper Aluminium, Lead, Nickel, Tin, Zinc
测试集样本个数:
212 212 212 212 212 212
测试集正确预测个数:
134 98 132 95 150 143
测试集准确率:
63.21 46.23 62.26 44.81 70.75 67.45
平均准确率:
59.12
base: 63.57, baseline: 70.29
```

main_20d运行后的终端输出

六个金属的60d预测

代码详情见`main_60d.py`文件，直接运行该文件即可。

随机种子固定为10；

迭代次数:150； 隐含层层数:1；输入特征数: 72；隐含层维度132；

学习率: 0.0001；序列长度:14；批长度:8；

```
# 固定随机种子
set_seed(10)
# 参数设置
args.epochs = 150
args.layers = 1
args.input_size = 72
args.hidden_size = 132
args.lr = 0.0001
args.sequence_length = 14
args.batch_size  = 8
```

main_60d参数设置情况

终端输出如下，平均准确率为 44.95%：

```
------------训练集--------------
金属次序: Copper Aluminium, Lead, Nickel, Tin, Zinc
训练集样本个数:
801 801 801 801 801 801
训练集正确预测个数:
721 621 720 586 478 709
训练集准确率:
90.01% 77.53% 89.89% 73.16% 59.68% 88.51%
平均准确率:
79.8
------------测试集--------------
金属次序: Copper Aluminium, Lead, Nickel, Tin, Zinc
测试集样本个数:
142 142 142 142 142 142
测试集正确预测个数:
74 87 49 48 54 71
测试集准确率:
52.11 61.27 34.51 33.8 38.03 50.0
平均准确率:
44.95
base: 63.97, baseline: 77.01
```

main_60d运行后的终端输出

附加文件说明

- Parse.py: 用于捕获终端输入，由于main文件固定了各项输入参数，因此该文件主要是提供了一个方便修改参数的args
- Util.py:包含了实现的几个工具函数

    - Mydataset类:继承于Dataset，与DataLoader并用进行数据的封装
    - set_seed:设置随机种子的函数
    - split_data_label:数据和标签分离
    - split_data_label_merge:多金属同时预测时数据和标签分离
- 提交目录中主要文件
    - DataFolders: 运行过程中生成的中间文件
    - Train_data,Validation_data: 训练集和测试集数据
    - LSTM_Batch_MultiLayer.py: LSTM模型实现
    - MyDataLoader: 数据处理功能实现
    - MyLSTM_Stock: 预测模型
    - Parse,Util:辅助类
    - main_1d,main_20d,main60d:金属预测的main函数
    - report.md,report.pdf: 实验报告的markdown和pdf版本，markdown版本会便于阅读。