汇报人：**B20041231章春阳**

进度汇报

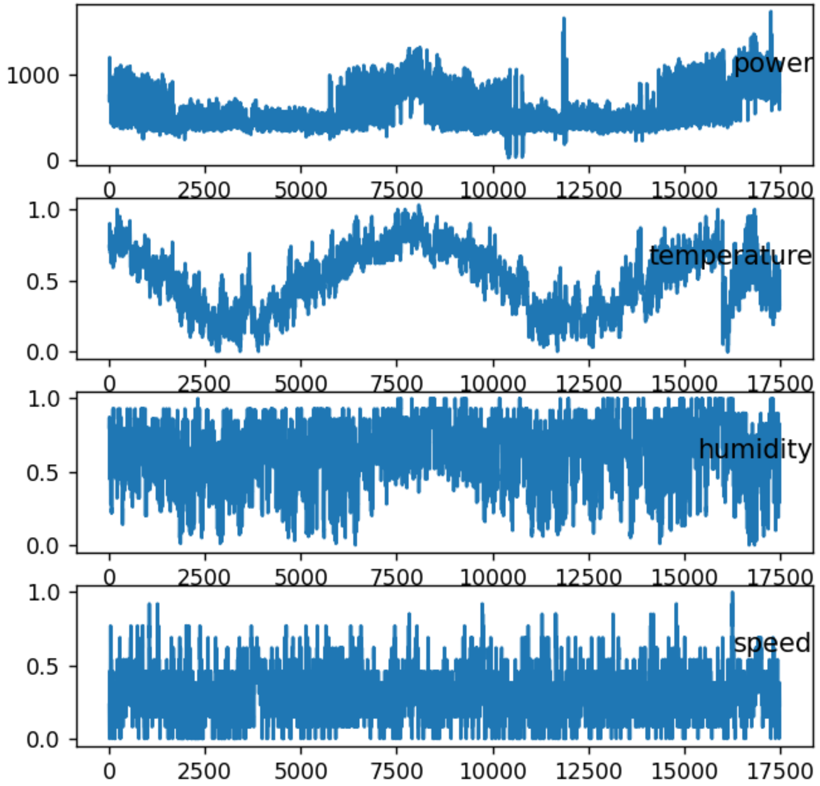
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipisicing elit,

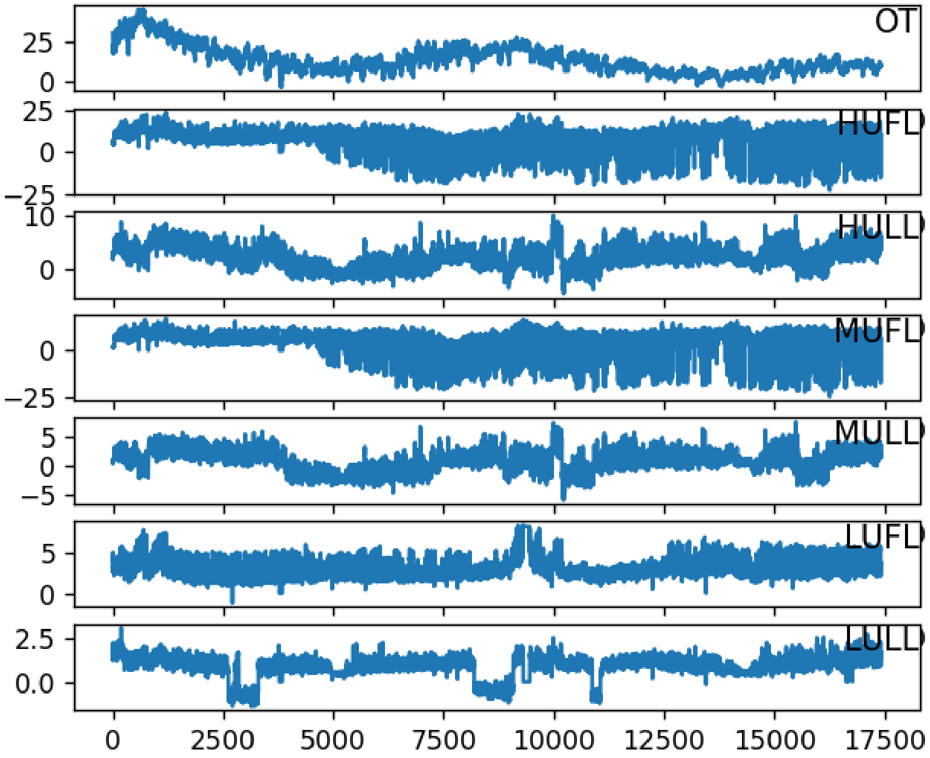
01 AMR模型负载预测

原数据集回顾

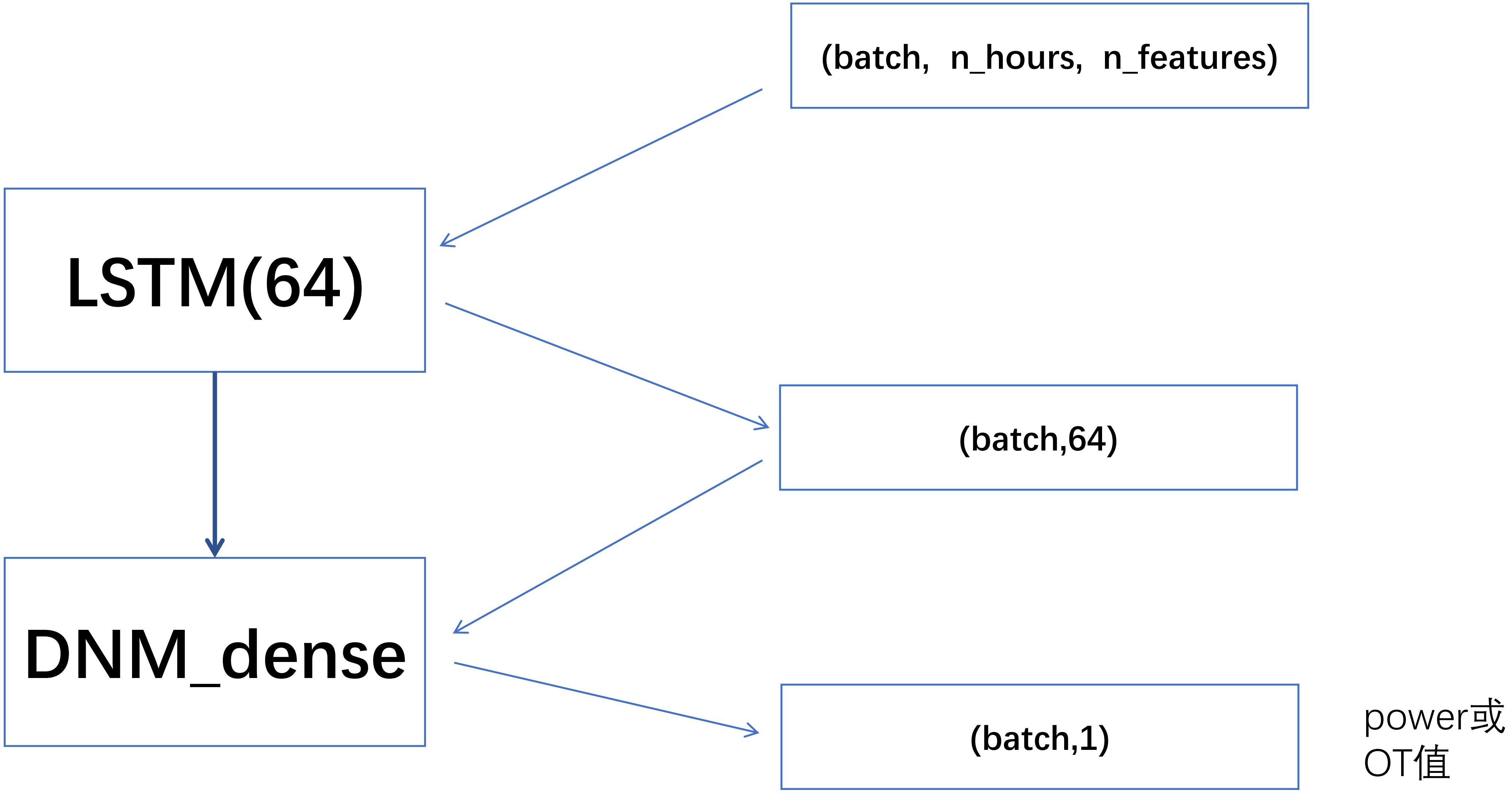
1.power数据集，使用温度、湿度等四个特征，把power作为标签进行训练预测的数据集。



2.OT数据集，使用OT、HUFL、HULL等七个特征，把OT作为标签进行训练预测的数据集。



原模型回顾


(batch，n_hours，n_features)
LSTM(64)
(batch,64)
DNM_dense
(batch,1)
power或
OT值

原模型结果

```
Average Runtime over 30 runs: 31.654 seconds
Average MAE over 30 runs: 63.585
Average MAPE over 30 runs: 0.099
Average RMSE over 30 runs: 111.145
Average R2 over 30 runs: 0.793
```

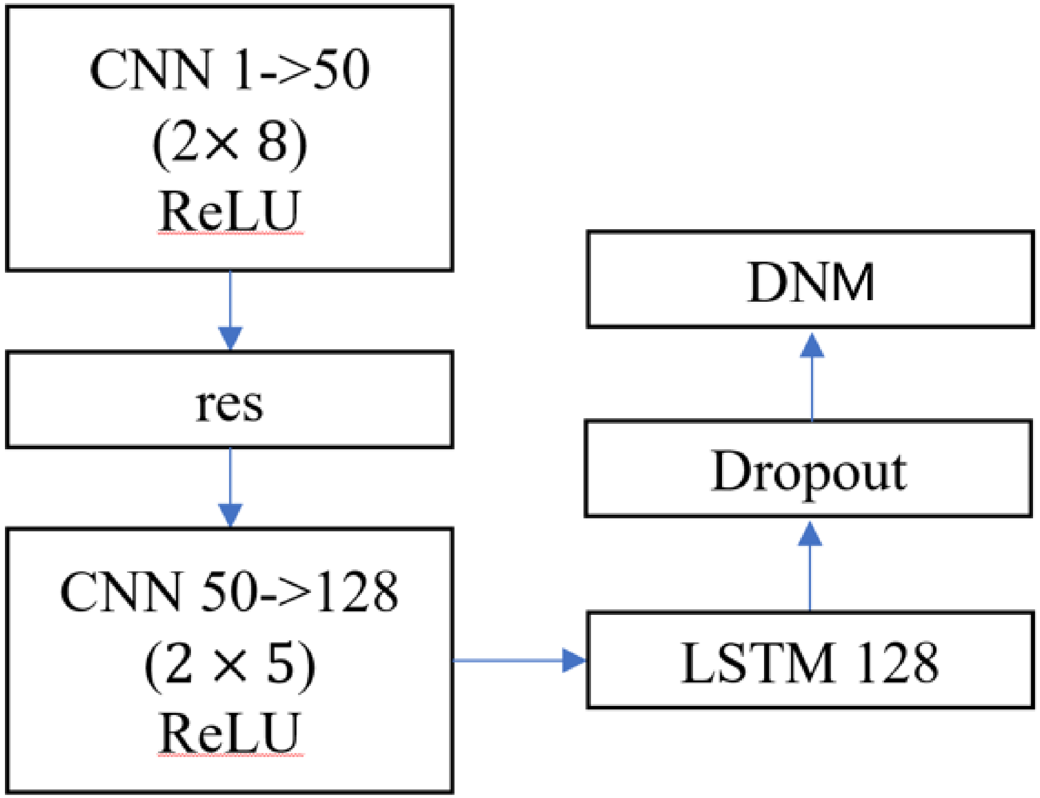
DNM Power

```
Average Runtime over 30 runs: 47.302 seconds
Average MAE over 30 runs: 0.730
Average MAPE over 30 runs: 0.163
Average RMSE over 30 runs: 0.903
Average R2 over 30 runs: 0.948
```

DNM OT

AMR模型

原模型用于无线电信号处理，其主要使用的网络名为MynewNN，由两层CNN，一个res残差单元，一层LSTM，加上随机失活，最后由DNM线性层输出，用于分类，其分类数量可以由其中参数n_classes确定。




```
def forward(self, input):
    # input2 = input[:, :, 0, :]#[40
    # input3 = input[:, :, 1, :]
    x = self.conv1(input)  # ([400,
    x = x+input
    x = self.conv5(x) #([400, 128, 1
    x = x.squeeze(axis=2)  # ([400,
    x = x.permute([0, 2, 1])# ([400,
    # encoder_output = self.transfor
    # decoder_output = self.transfor
    x, _ = self.lstm3(x)  # ([400, 1
    x = x[:, -1, :]  # 取LSTM的最后一个
    x = self.dropout(x)
    x = self.fc(x)
    return x
```

尝试将AMR模型应用在负载数据集上

修改MynewNN网络

MynewNN网络所用框架为PyTorch，继承自无线电信号处理库rfml中的Model父类，主要传递参数input_samples和n_classes。考虑到处理负载数据并没有用到input_samples参数，便直接修改父类为torch.nn.Module,并传递n_classes和时间步长参数time_stride参数。

```
from rfml.nn.model import Model
```

```
class MynewNN(Model):
    def __init__(self, input_samples: int, n_classes: int):
        super(MynewNN, self).__init__(input_samples=input_samples, n_classes=n_classes)
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 50, kernel_size=(2, 8), padding="same"),
            nn.ReLU()
        )
        self.transformer1 = TransformerBlock(embed_dim=64, num_heads=4, ff_dim=16)
```

↓

```
class MynewNN(nn.Module):
    def __init__(self, n_classes, time_stride):
        super(MynewNN, self).__init__()
```

修改MynewNN网络

前向传播依旧按照CNN-->res-->CNN-->LSTM+Dropout-->DNM的网络架构，经预处理后的负载时序数据单个样本的形状为(time_stride, n_features), 因此改变CNN卷积核的大小为(time_stride, 1), 对每个特征的不同时间步进行卷积运算，并符合后续LSTM层的输入尺寸要求。

```
self.conv1 = nn.Sequential(
    nn.Conv2d(1, 50, kernel_size=(time_stride, 1), padding="same"),
    nn.ReLU()
)

self.conv5 = nn.Sequential(
    nn.Conv2d(50, 128, kernel_size=(time_stride, 1)),
    nn.ReLU()
)
```

由于是回归预测问题，输出的节点为1，即n_classes分类数也为1。

```
net = MynewNN(n_classes=1, time_stride=3)
```

数据输入和训练

原DataFrame负载数据经过归一化、series_to_supervised函数处理变为时序数据，因为模型的接口为CNN，将时序数据的形状改变为(batch_size, 1, time_stride, n_features)，1为通道数。以power数据为例，预处理完的训练数据形状为(10495, 1, 3, 4)，训练标签形状为(10495, 1)，配置模型参数，选择优化器和损失函数之后，一次性输入数据进行训练。

成功开始训练，从loss值可以看到模型在训练过程中逐渐收敛。

```
#构建模型
model = MynewNN(n_classes=1,time_stride=n_hours)
optimizer = optim.Adam(model.parameters(), lr=lr)
criteon = nn.MSELoss()

for epoch in range(epoches):
    logits = model(train_X)
    loss = criteon(logits, train_y)

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

```
runtime0 epoch0 loss:3.859121561050415
runtime0 epoch1 loss:0.14393675327301025
runtime0 epoch2 loss:0.04446866363286972
runtime0 epoch3 loss:0.0427021384239l968
runtime0 epoch4 loss:0.04968875274062157
runtime0 epoch5 loss:0.03961871936917305
```

torch数据集的创建

考虑到原模型使用keras框架分批次对负载数据进行预测，而MynewNN用的是PyTorch，需要构建相应的torch的负载Dataset，并使用torch.utils.data.DataLoader对其进行分批加载，以便对比前后模型运行的性能指标。
继承torch.utils.data.Dataset类，编写其中__getitem__和__len__函数，获取单个负载数据样本、标签和数据集总长度，完成torch数据集的创建。

```
class Power_ds(Dataset):
    def __init__(self, n_hours, n_features, data_type):
        super(Power_ds, self).__init__()

        self.data_type = data_type
        self.n_hours = n_hours
        self.n_features = n_features
        self.values = self.preprocessing()

        if self.data_type == 'train':
            n_hours = self.n_hours
            n_features = self.n_features
            n_train_samples = int(len(self.values) * 0.6)
            train = self.values[:n_train_samples, :]
            n_obs = n_hours * n_features
```

AMR模型的训练与评估

与DNM模型保持相同的批次数量，使用DataLoader加载训练数据，不分批次加载测试数据，保持相同的运行次数、epoch数开始训练，并记录相应的运行结果。

```
train_ds = Power_ds(3, 4, 'train')
test_ds = Power_ds(3, 4, 'test')

train_loader = DataLoader(train_ds, batch_size=batchsize, shuffle=False)
test_loader = DataLoader(test_ds, batch_size=len(test_ds), shuffle=False)
```

```
mape_results = []
rmse_results = []
R2_results = []
mae_results = []
runtime_results = []
```

AMR模型的训练与评估

测试结果:

```
Average Runtime over 30 runs: 40.288 seconds
Average MAE over 30 runs: 118.782
Average MAPE over 30 runs: 0.176
Average RMSE over 30 runs: 179.785
Average R2 over 30 runs: 0.433
```

AMR power

对比:

模型	Runtime	MAE	MAPE	RMSE	R2
DNM	31.7s	63.59	0.1	111.15	0.793
AMR	40.3s	118.78	0.176	179.78	0.433

power

```
Average Runtime over 30 runs: 86.314 seconds
Average MAE over 30 runs: 8.083
Average MAPE over 30 runs: 2.062
Average RMSE over 30 runs: 8.986
Average R2 over 30 runs: -4.439
```

AMR OT

模型	Runtime	MAE	MAPE	RMSE	R2
DNM	47.3s	0.73	0.163	0.9	0.948
AMR	86.3s	8.08	2.062	8.99	-4.439

OT

AMR模型的训练与评估

对比:

模型	Runtime	MAE	MAPE	RMSE	R2
DNM	31.7s	63.59	0.1	111.15	0.793
AMR	40.3s	118.78	0.176	179.78	0.433

power

模型	Runtime	MAE	MAPE	RMSE	R2
DNM	47.3s	0.73	0.163	0.9	0.948
AMR	86.3s	8.08	2.062	8.99	-4.439

OT

可以看到AMR模型对于负载时序数据的拟合效果并不是很好，尤其是在OT这样的多特征时序预测上的效果更是逆天。

尝试的改良方法

以power为例

1.改变DNM输出层树突神经元数量M：负载数据集的噪声相比无线电数据可能会小一些，尝试改变M来增强其拟合能力。

M	Runtime	MAE	MAPE	RMSE	R2
20	45.2s	118.73	0.173	185.9	0.397
10	40.3s	118.78	0.176	179.8	0.433
5	39.7s	102.50	0.150	157.9	0.579
1	38.9s	100.29	0.143	154.1	0.602

2.减小网络复杂度：负载数据集的复杂度可能比无线电要小，尝试降低网络复杂度，删除一层CNN和res

M	Runtime	MAE	MAPE	RMSE	R2
10	33.9s	90.8	0.134	142.9	0.657

尝试的改良方法

以power为例

只用一层CNN的最好效果:

M	Runtime	MAE	MAPE	RMSE	R2
1	33.1s	88.3	0.129	136.9	0.685

如果把最后一层DNM换成线性层，模型的效果在两个数据集上的效果都有很大的提升，但是这样就不算AMR了。

```
Average Runtime over 10 runs: 44.019 seconds
Average MAE over 10 runs: 82.561
Average MAPE over 10 runs: 0.118
Average RMSE over 10 runs: 131.287
Average R2 over 10 runs: 0.710
```

```
Average Runtime over 10 runs: 71.135 seconds
Average MAE over 10 runs: 1.543
Average MAPE over 10 runs: 0.406
Average RMSE over 10 runs: 2.014
Average R2 over 10 runs: 0.728
```

没有考虑网络处理数据的逻辑，是否会破坏数据结构等，只是致力于让数据与网络“对口”。后续可以尝试把最后一层用于分类的DNM改回用于回归，像keras里的DNM_Dense那样，看看效果会不会变好。

THANK YOU

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.