[Code Review] Multi-step Prediction & Pytorch Template

2023-2 URP | SKKU AAI 2021311828 Park Su Yeon

MULTI STEP PREDICTION TYPES

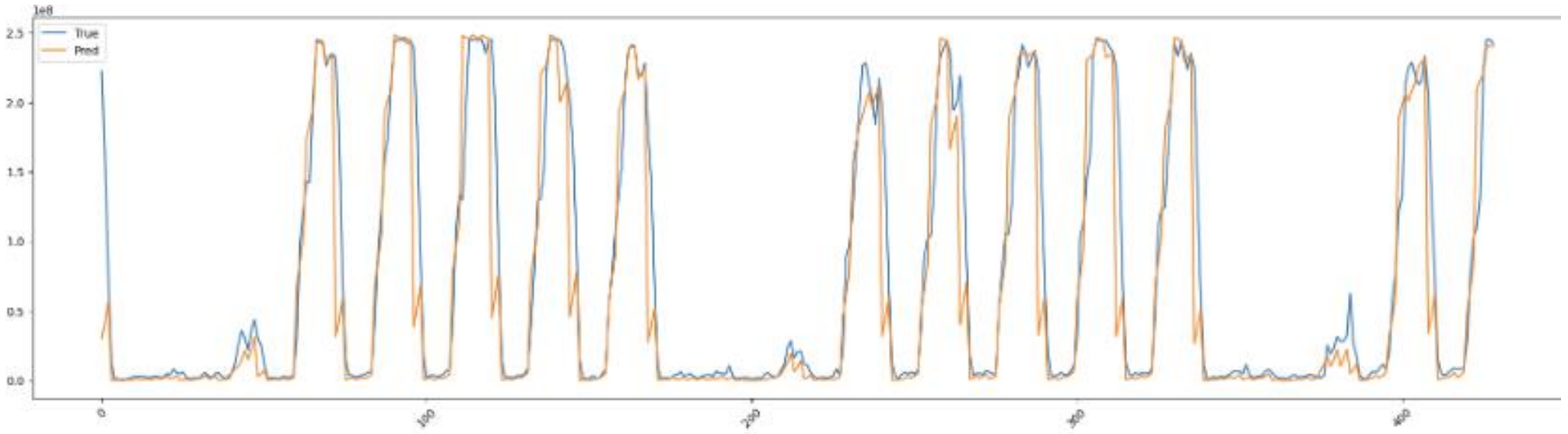
- Direct Multi step
  : 하나의 time step마다 별개의 모델을 생성
  - prediction(t) = model1(obs(t-1), obs(t-2), ..., obs(t-n))
  - prediction(t+1) = model2(obs(t-2), obs(t-3), ..., obs(t-n))
- Recursive Multi step
  : one-step 시계열 예측 모델로 여러 날을 예측. 이전 time-step의 예측된 값이 다음 time-step의 값을 예측하는 데에 input으로써 사용
  - prediction(t) = model(obs(t-1), obs(t-2), ..., obs(t-n))
  - prediction(t+1) = model(prediction(t), obs(t-1), ..., obs(t-n))

MULTI STEP PREDICTION TYPES

- Direct-Recursive Hybrid Multi step
  : 모델은 각각 예측하고자 하는 time step을 위해 구성. 각 모델은 예측할 때 이전 time step의 예측 값을 input 값으로 사용.
  - prediction(t) = model1(obs(t-1), obs(t-2), ..., obs(t-n))
  - prediction(t+1) = model2(prediction(t), obs(t-1), ..., obs(t-n))
- Multioutput
  : 예측해야 하는 시퀀스 전체를 한 번에 예측하는 것
  - prediction(t), prediction(t+1) = model(obs(t-1), obs(t-2), ..., obs(t-n))

RECURSIVE MULTI STEP PREDICTION

- step size: 3
- NRMSE: 0.1606



RECURSIVE MULTI STEP PREDICTION CODE REVIEW

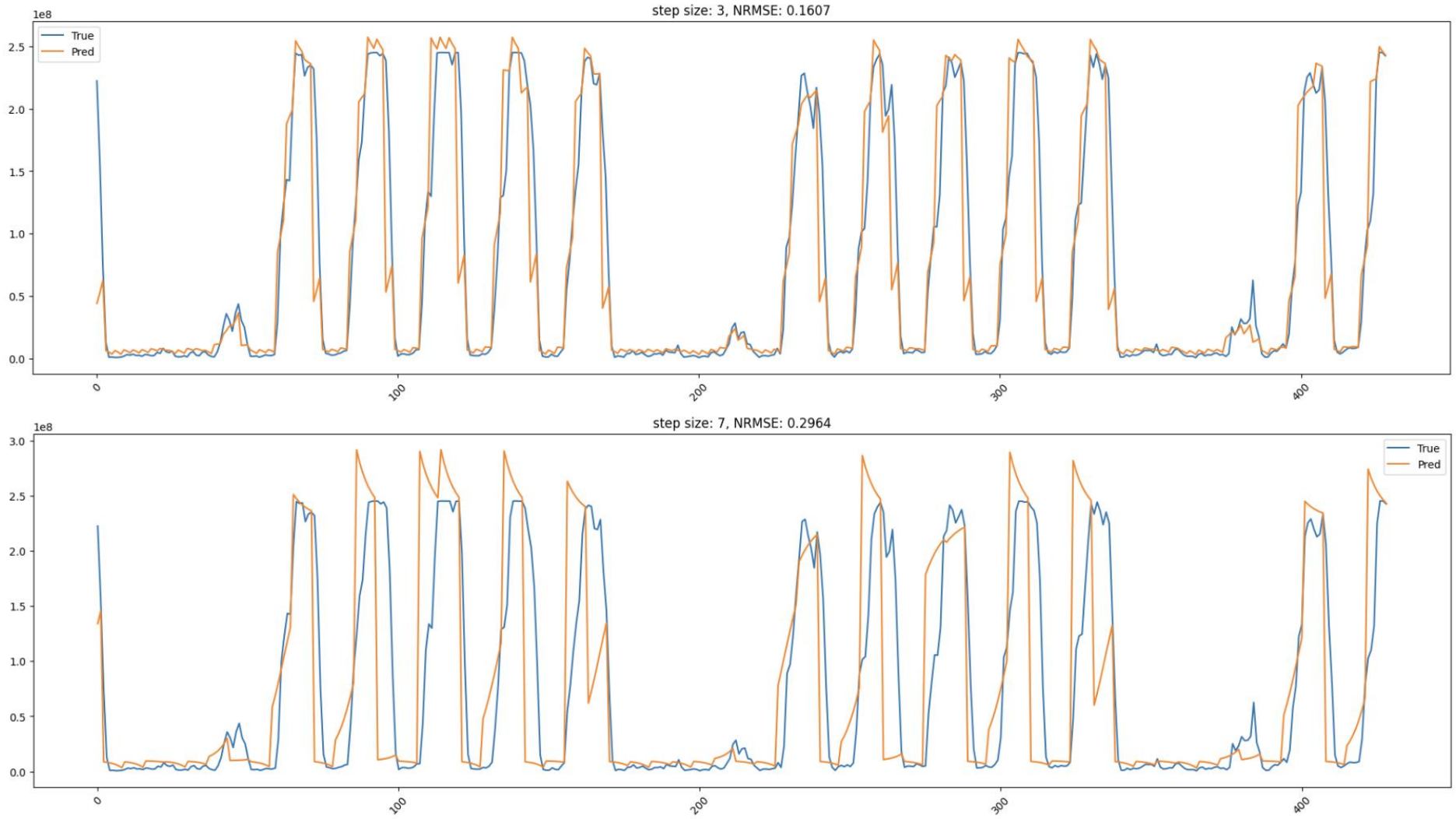
PREVIOUS

```
with torch.no_grad():
    test_seq = X_test[:1] # 첫번째 테스트 셋, 3차원
    preds = []
    for _ in range(len(X_test)):
        model.reset_hidden_state()
        y_test_pred = model(test_seq)
        pred = torch.flatten(y_test_pred).item()
        preds.append(pred)
        new_seq = test_seq.numpy().flatten()
        new_seq = np.append(new_seq, [pred]) # 시퀀스에 추가
        new_seq = new_seq[1:] # 추가된 값을 포함하여 seq_length 맞추기
        test_seq = torch.as_tensor(new_seq).view(1, sequence_len, 1).float()
```

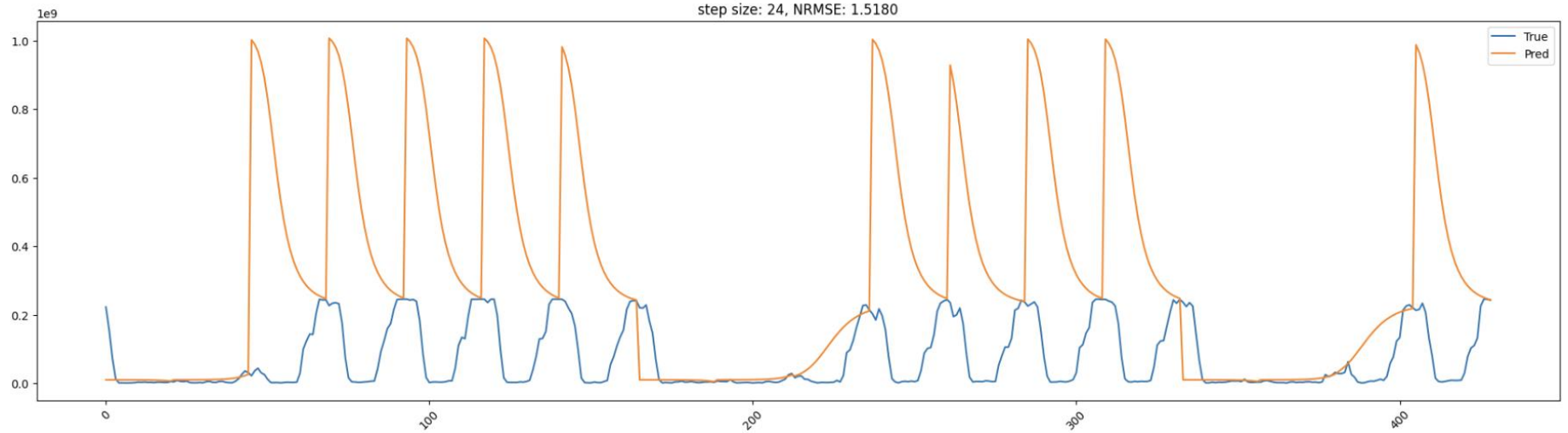
NOW

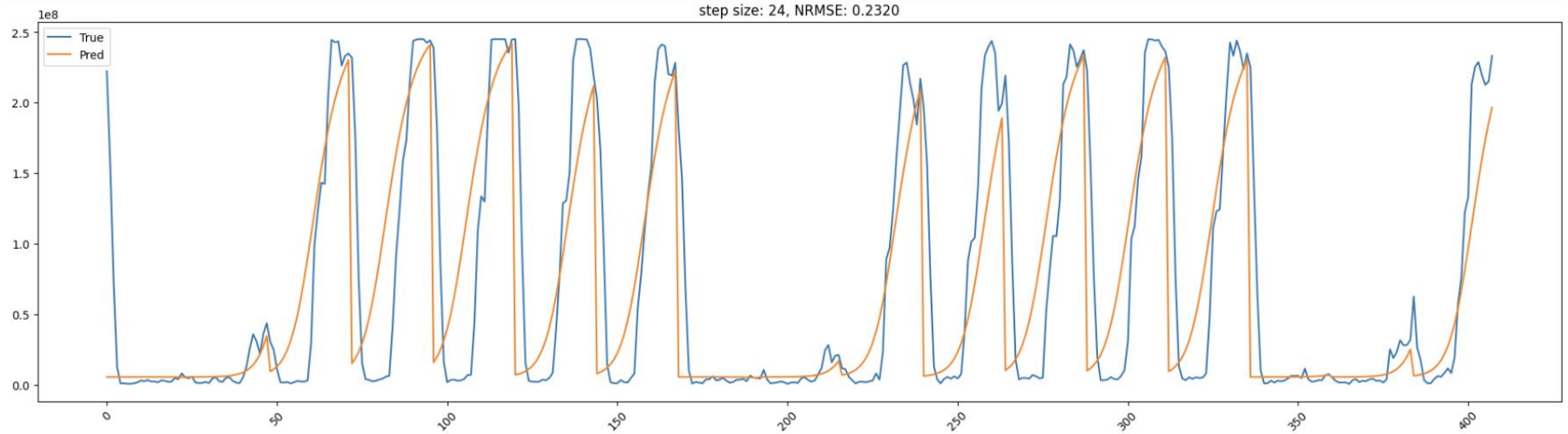
```
# Recursive Multi-step Prediction
step_size = 3
with torch.no_grad():
    test_seq = X_test[:1] # 첫번째 테스트 셋, 3차원
    preds = []
    for _ in range(1,len(X_test)+1):
        model.reset_hidden_state()
        y_test_pred = model(torch.unsqueeze(test_seq, 0))
        pred = torch.flatten(y_test_pred).item()
        preds.append(pred)
        if _ % step_size != 0:
          new_seq = test_seq.numpy().flatten()
          new_seq = np.append(new_seq, [pred]) # 시퀀스에 추가
          new_seq = new_seq[1:] # 추가된 값을 포함하여 seq_length가 3개가 되도록 맞추기
          test_seq = torch.as_tensor(new_seq).view(1,  sequence_len).float()
        else:
          test_seq = X_test[_+1:_+2]
```

RECURSIVE MULTI STEP PREDICTION EXPERIMENTS



RECURSIVE MULTI STEP PREDICTION EXPERIMENTS





PYTORCH TEMPLATE FOLDER STRUCTURE

```
pytorch-template/
│
├── train.py - main script to start training
├── test.py - evaluation of trained model
│
├── config.json - holds configuration for training
├── parse_config.py - class to handle config file and cli options
│
├── new_project.py - initialize new project with template files
│
├── base/ - abstract base classes
│   ├── base_data_loader.py
│   ├── base_model.py
│   └── base_trainer.py
│
├── data_loader/ - anything about data loading goes here
│   └── data_loaders.py
│
├── data/ - default directory for storing input data
│
├── model/ - models, losses, and metrics
│   ├── model.py
│   ├── metric.py
│   └── loss.py
│
├── saved/
│   ├── models/ - trained models are saved here
│   └── log/ - default logdir for tensorboard and logging output
│
├── trainer/ - trainers
│   └── trainer.py
│
├── logger/ - module for tensorboard visualization and logging
│   ├── visualization.py
│   ├── logger.py
│   └── logger_config.json
│
└── utils/ - small utility functions
    ├── util.py
    └── ...
```

- train.py
  : 모델을 학습을 위해 필요한 파일.
  parse_config.py의 from_args 메서드에서 객체를 반환받아 main 메서드를 수행하는 방식으로 학습 진행됨.
- Config.json
  : config 파일을 원하는 값으로 변경시키고, train.py 파일에 원하는 모델을 입력함으로써 다른 형식은 건드리지 않고 원하는 학습을 수행할 수 있음

PYTORCH TEMPLATE [TRAIN.PY]

```python
def main(config):
    logger = config.get_logger('train')

    # setup data_loader instances
    data_loader = config.init_obj('data_loader', module_data)
    valid_data_loader = data_loader.split_validation()

    # build model architecture, then print to console
    model = config.init_obj('arch', module_arch)
    logger.info(model)

    # prepare for (multi-device) GPU training
    device, device_ids = prepare_device(config['n_gpu'])
    model = model.to(device)
    if len(device_ids) > 1:
        model = torch.nn.DataParallel(model, device_ids=device_ids)

    # get function handles of loss and metrics
    criterion = getattr(module_loss, config['loss'])
    metrics = [getattr(module_metric, met) for met in config['metrics']]

    # build optimizer, learning rate scheduler. delete every lines containing lr_scheduler for disabling scheduler
    trainable_params = filter(lambda p: p.requires_grad, model.parameters())
    optimizer = config.init_obj('optimizer', torch.optim, trainable_params)
    lr_scheduler = config.init_obj('lr_scheduler', torch.optim.lr_scheduler, optimizer)

    trainer = Trainer(model, criterion, metrics, optimizer,
                      config=config,
                      device=device,
                      data_loader=data_loader,
                      valid_data_loader=valid_data_loader,
                      lr_scheduler=lr_scheduler)

    trainer.train()
```

```python
if __name__ == '__main__':
    args = argparse.ArgumentParser(description='PyTorch Template')
    args.add_argument('-c', '--config', default=None, type=str,
                      help='config file path (default: None)')
    args.add_argument('-r', '--resume', default=None, type=str,
                      help='path to latest checkpoint (default: None)')
    args.add_argument('-d', '--device', default=None, type=str,
                      help='indices of GPUs to enable (default: all)')

    # custom cli options to modify configuration from default values given in json file.
    CustomArgs = collections.namedtuple('CustomArgs', 'flags type target')
    options = [
        CustomArgs(['--lr', '--learning_rate'], type=float, target='optimizer;args;lr'),
        CustomArgs(['--bs', '--batch_size'], type=int, target='data_loader;args;batch_size')
    ]
    config = ConfigParser.from_args(args, options)
    main(config)
```

THANK YOU

Q&A