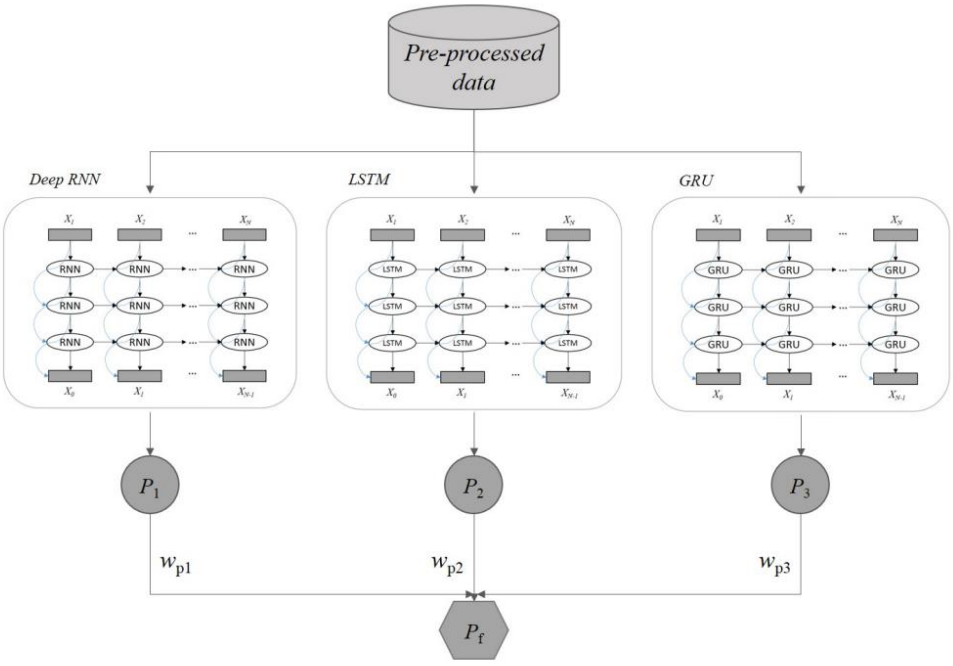
DERN: Deep Ensemble Learning Model for Short and Long-Term Prediction of Baltic Dry Index

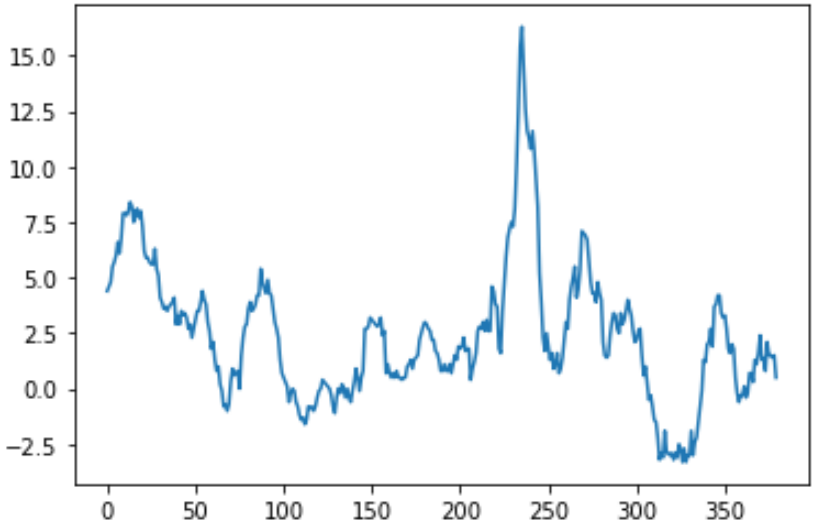
본 논문에서 말하는 Baltic Dry Index(BDI)는 global shipping 과 trade activity를 나타내기 위해 일반적으로 사용되는 지표(indicator)이다. 이것은 다양한 경제활동에 영향을 미치기 때문에 정확한 예측이 필요하다. 하지만 BDI의 volatility, non-stationarity, and complexity 특징으로 인해 정확한 예측에 있어서 어려움이 존재한다. 따라서 본 논문에서는 BDI의 short and long-term의 정확한 예측을 위해서 **sequential deep-learning 모델인 RNN, LSTM, and GRU를 활용한 Ensemble deep-learning** 방법을 제안한다.

Proposed Method



본 논문에서 제안하는 방법의 전체 구조는 위와 같다. Pre-processed data를 활용해 RNN, LSTM, GRU 모델에 각각 독립적으로 학습을 진행한다. 해당 모델들이 학습 데이터에 대해서 Converge 되었다면, test phase에서는 각 모델의 예측치 P_1, P_2, P_3에 대해서 가중치 w_{p1}, w_{p2}, w_{p3} 을 활용해 BDI의 next value인 P_f 를 산출한다. w_{p1}, w_{p2}, w_{p3}는 기본적인 forward- and back-propagation을 활용하는 neural network를 통해 학습된다. 위 구조를 사용해 one-step-ahead(short-term) 와 multi-step-ahead(long-term) 예측 2가지를 수행한다. 이번 Paper representation에서는 개별 모델을 사용한 결과와 논문에서 제안하는 RNN+LSTM+GRU 모델의 성능 비교를 중점으로 진행하겠습니다.

사용 데이터



먼저 논문에서 사용한 데이터는 open-source가 아니기 때문에, 논문에서 사용한 데이터와 비슷한 데이터를 가져와 실험을 진행하였다. 데이터의 총 길이는 380이다.

데이터 load

```
torch.cuda.is_available()
is_cuda = torch.cuda.is_available()

if is_cuda:
    device = torch.device("cuda")
    print("GPU is available")

else:
    device = torch.device("cpu")
    print("GPU not available, CPU used")

path = "C:/Users/KJW/Desktop/비즈니스/dataset/"

data = pd.read_csv(path + "CPIH_Non_seasonal_food.csv")
all_data = data["value"].values.astype(float)

scaler = MinMaxScaler(feature_range = (0,1))

data_normalized = torch.FloatTensor(scaler.fit_transform(all_data.reshape(-1,1))).view(-1)

test_size = 20
train_size = int(len(data_normalized)) - test_size

train_data_normalized = data_normalized[:-test_size]
test_data_normalized = data_normalized[-test_size:]

train_X = train_data_normalized[:-1]
train_Y = train_data_normalized[1:]
```

개별 Model 생성

```
class RNN(nn.Module):
    def __init__(self , input_size , hidden_size , ouput_size , num_layers):
        super(RNN , self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.num_layers = num_layers

        self.RNN = nn.RNN(input_size , hidden_size , num_layers = num_layers)
        self.linear = nn.Linear(hidden_size , output_size)

    def forward(self , x):

        rnn_out , _ = self.RNN(x.view(len(x),1,-1))
        predictions_rnn = self.linear(rnn_out.view(len(x),-1))

        return predictions_rnn.view(len(x))


class LSTM(nn.Module):
    def __init__(self , input_size , hidden_size , ouput_size , num_layers):
        super(LSTM , self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.num_layers = num_layers

        self.LSTM = nn.LSTM(input_size , hidden_size , num_layers = num_layers)
        self.linear = nn.Linear(hidden_size , output_size)

    def forward(self , x):

        lstm_out , _ = self.LSTM(x.view(len(x),1,-1))
        predictions_lstm = self.linear(lstm_out.view(len(x),-1))

        return predictions_lstm.view(len(x))


class GRU(nn.Module):
    def __init__(self , input_size , hidden_size , output_size , num_layers):
        super(GRU, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.num_layers = num_layers

        self.GRU = nn.GRU(input_size , hidden_size , num_layers = num_layers)
        self.linear = nn.Linear(hidden_size , output_size)

    def forward(self , x):

        gru_out , _ = self.GRU(x.view(len(x),1,-1))
        predictions_gru = self.linear(gru_out.view(len(x),-1))

        return predictions_gru.view(len(x))
```

먼저 각 모델 RNN, LSTM, GRU에 대해서 독립적으로 학습해야 하기 때문에 각 모델에 대한 Class를 하나씩 생성해서 사용해야 한다.

개별 Model 학습 및 학습 결과 저장

```
def train(model_name):

    input_size = 1
    hidden_size = 32
    output_size = 1
    num_layers = 2
    epochs = 1000

    if model_name == "RNN":
        model = RNN(input_size  , hidden_size , output_size , num_layers).to(device)
    elif model_name == "LSTM":
        model = LSTM(input_size , hidden_size , output_size , num_layers).to(device)
    elif model_name == "GRU":
        model = GRU(input_size , hidden_size , output_size , num_layers).to(device)

    loss_function = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters() , lr = 0.001)

    entire_loss = list()

    for i in range(epochs):

        model.zero_grad()

        y_pred = model(train_X.to(device)).to(device)
        loss = loss_function(y_pred , train_Y.to(device))

        loss.backward()
        optimizer.step()

        entire_loss.append(loss)

        if i%10 == 1:

            print(f'Epoch: {i:3} loss: {loss:10.8f}')

    torch.save(model.state_dict() , "C:/Users/KJW/Desktop/비즈니스/model_" + model_name + ".pt")

    return entire_loss , y_pred

entire_loss , y_pred =  train("GRU")
```

각 모델을 학습하여 Ensemble을 위한 가중치를 다시 학습해서 Ensemble-model을 만들어야 하는데 먼저 각 모델을 하나씩 학습한 뒤에 학습한 모델의 parameters를 저장하여 Ensemble-model을 만들 때 하나씩 load하여 사용하였다. 각 모델의 parameters는 논문과 똑같이 사용하였고, hidden-size는 32, layer의 개수는 2, Epochs은 1000으로 학습을 진행하였다[hidden-size는 너무 클 필요가 없다고 생각하여 32개로 줄여서 사용하였음]

위와 같이 개별 모델의 학습 결과를 저장하여 예측을 진행할 때 load하여 사용할 수 있다. 개별 모델을 저장하지 않고 python내에 그대로 둔다면 memory out 문제가 생길 수 있기 때문에 저장한 뒤 불러와서 inference하는 것이 효율적이다.

개별 학습 모델 load

```
input_size = 1
hidden_size = 32
output_size = 1
num_layers = 2

model_RNN = RNN(input_size  , hidden_size  , output_size, num_layers).to(device)
model_LSTM = LSTM(input_size  , hidden_size  , output_size, num_layers).to(device)
model_GRU = GRU(input_size  , hidden_size  , output_size, num_layers).to(device)

model_RNN.load_state_dict(torch.load("C:/Users/KJW/Desktop/비즈니스/model_RNN.pt"))
model_LSTM.load_state_dict(torch.load("C:/Users/KJW/Desktop/비즈니스/model_LSTM.pt"))
model_GRU.load_state_dict(torch.load("C:/Users/KJW/Desktop/비즈니스/model_GRU.pt"))
```

먼저 각 모델의 파라미터를 넣어줄 모델 껍데기를 생성한 다음 저장해 놓은 파라미터들을 load 하여 넣어준다.

DERN 모델 생성

```
class DERN(nn.Module):
    def __init__(self, model_RNN, model_LSTM , model_GRU):
        super(DERN, self).__init__()
        self.model_RNN = model_RNN
        self.model_LSTM = model_LSTM
        self.model_GRU = model_GRU
        self.linear_weights = nn.Linear(3, 1)

    def forward(self, x):
        rnn_out = self.model_RNN(x)
        lstm_out = self.model_LSTM(x)
        gru_out = self.model_GRU(x)

        blend_output = torch.cat([rnn_out.view(len(x),1) , lstm_out.view(len(x),1) , gru_out.view(len(x),1)] , dim = 1)

        weighted_blend_output = self.linear_weights(blend_output)


        return weighted_blend_output.view(len(x))
```

학습이 완료된 개별 모델을 불러와 예측을 한 뒤 각 예측치에 가중치를 곱하여 새로운 예측치를 생성하는 모델이다. Torch.cat을 이용해 개별 예측치를 하나로 합쳐주고 nn.Linear를 사용해 가중치를 곱해준다[bias = False로 사용해야 함]

해당 모델의 학습을 통해 가중치 또한 학습시켜줄 수 있다.

DERN 모델 학습

```
epochs = 500

Ensemble = DERN(model_RNN , model_LSTM , model_GRU).to(device)

loss_function = nn.MSELoss()
optimizer = torch.optim.Adam(Ensemble.parameters() , lr = 0.001)

entire_loss = list()

for i in range(epochs):

    Ensemble.zero_grad()

    y_pred = Ensemble(train_X.to(device)).to(device)
    loss = loss_function(y_pred , train_Y.to(device))

    loss.backward()
    optimizer.step()

    entire_loss.append(loss)

    if i%10 == 1:

        print(f'Epoch: {i:3} loss: {loss:10.8f}')
```

실험 결과

Method	Test size: 40			Test size: 30			Test size: 20		
	RMSE	MAE	MAPE	RMSE	MAE	MAPE	RMSE	MAE	MAPE
RNN	5.1772	5.0051	0.5885	5.0333	4.9887	0.5709	5.1123	5.0541	0.5785
LSTM	4.4732	4.0255	0.5343	4.2331	4.1881	0.5221	3.9912	3.7865	0.4994
GRU	4.6731	4.2215	0.5541	4.1211	4.0015	0.5311	3.8778	3.6312	0.4811
DERN	**3.1234**	**3.4313**	**0.4743**	**3.0231**	**3.1314**	**0.4133**	**2.8221**	**2.5333**	**0.4001**

논문에서 제안하는 방법인 DERN과 개별 모델(RNN, LSTM, GRU) 간의 성능 평가를 진행하였다. 성능 평가를 위한 지표로는 RMSE, MAE, MAPE를 사용하였고, 예측하는 데이터의 개수는 20개 ~ 40개로 바꿔 실험하였다. 실험 결과 제안하는 방법인 DERN의 성능이 가장 좋은 것을 볼 수 있다.

결론

본 논문에서는 개별 모델들의 학습 결과를 바탕으로 각 모델에 적절한 가중치를 곱하여 최종 예측치를 산출하는 방법을 사용하였다. 논문의 결과와 맞게 제안하는 방법의 성능이 더 좋은 것을 확인할 수 있었다. Deep-Learning Ensemble을 sequence 모델에 적용할 수 있는 방법으로 모델마다 각기 다른 Skip-connection을 사용해 여러 개의 모델을 만들어 하나의 결과를 산출하는 방법도 적용해 볼 수 있다. Deep-learning을 Ensemble할 수 있는 다양한 접근 방법을 시도해 볼 수 있을 것 같다.