NEURAL NETWORK

Define Model Classes

We first define the network model classes. Each model inherits the initialization function from corresponding models in PyTorch NN Module. In the initialization, we construct hidden layers, RNN layer and output layer for FNN, RNN, GRU and LSTM model respectively with input layer size. And we define forward propagation function for them to generate output. We use ReLU as activation function in FNN and use unsqueeze method to add dimensions to the input data.

Python Code

```
class FNN_net(torch.nn.Module):
    def __init__(self, feature_size, hidden_size1, hidden_size2, output_size):
        super(FNN_net, self).__init__()
        self.hidden1 = torch.nn.Linear(feature_size, hidden_size1)  # Hidden layer 1
        self.hidden2 = torch.nn.Linear(hidden_size1, hidden_size2)  # Hidden layer 2
        self.output = torch.nn.Linear(hidden_size2, output_size)    # Output layer
        self.feature_size = feature_size

    def forward(self, x):
        x = torch.relu_(self.hidden1(x))
        x = torch.relu_(self.hidden2(x))
        x = self.output(x)
        return x

class RNN_net(torch.nn.Module):
    def __init__(self, feature_size, hidden_size, output_size):
        super(RNN_net, self).__init__()
        self.rnn = torch.nn.RNN(input_size = feature_size, hidden_size = hidden_size, num_layers = 1, batch_first = 
True)  # RNN layer
        self.output = torch.nn.Linear(hidden_size, output_size)  # Output layer

    def forward(self, x):
        x, _ = self.rnn(x.unsqueeze(2))
        x = self.output(x[:, -1, :])
        return x

class GRU_net(torch.nn.Module):
    def __init__(self, feature_size, hidden_size, output_size):
        super(GRU_net, self).__init__()
        self.rnn = torch.nn.GRU(input_size = feature_size, hidden_size = hidden _size, num_layers = 1, batch_first = 
True)  # RNN layer
        self.output = torch.nn.Linear(hidden _size, output_size)  # Output layer

    def forward(self, x):
        x, _ = self.rnn(x.unsqueeze(2))
        x = self.output(x[:, -1, :])
        return x
```

```
class LSTM_net(torch.nn.Module):
    def __init__(self, feature_size, hidden_size, output_size):
        super(LSTM_net, self).__init__()
        self.rnn = torch.nn.LSTM(input_size = feature_size, hidden_size = hidden_size, num_layers = 1, batch_first =
True)  # RNN layer
        self.output = torch.nn.Linear(hidden_size, output_size)  # Output layer

    def forward(self, x):
        x, _ = self.rnn(x.unsqueeze(2))
        x = self.output(x[:, -1, :])
        return x
```

Define Functions

Here we define two function for training and testing. Firstly, we choose a loss function, which is MSELoss in this case. In train function, the optimizer chooses a way to update the weight in order to converge to find the best weights in this neural network. The learning rate will be tuned later at the tuning section. We loop over the data serval times and log the losses for training and validation set. Similar to training the neural network, we also need to load batches of test data and collect the outputs.

Python Code

```
criterion = torch.nn.MSELoss()

def train(model, num_epoch, dataloader):
    optimizer = optim.Adam(model.parameters(), lr = 0.01)
    loss_train_list = []  # record training loss
    loss_vali_list = []   # record validation loss

    for epoch in range(num_epoch):  # loop over the dataset multiple times
        running_loss = 0.0
        batch_num = 0

        for _, data in enumerate(dataloader, 0):
            inputs, labels = data
            optimizer.zero_grad()

            outputs = model(inputs)
            loss = criterion(outputs, labels)  # get loss
            loss.backward()  # back propagation, get gradients of loss
            optimizer.step()  # optimize one step
            running_loss += inputs.shape[0] * loss.item()
            batch_num += inputs.shape[0]

        loss_train = running_loss / batch_num
        loss_vali = test(model, dl_validation_data)
```

```python
        loss_train_list.append(loss_train)
        loss_vali_list.append(loss_vali)

    print('Finished Training')
    return loss_train_list, loss_vali_list

def test(model, dataloader):
    with torch.no_grad():
        running_loss = 0.0
        batch_num = 0
        for i, data in enumerate(dataloader):
            inputs, labels = data
            outputs = model(inputs)  # forward propagation, get outputs
            loss = criterion(outputs, labels)  # get loss
            running_loss += inputs.shape[0] * loss.item()
            batch_num += inputs.shape[0]
    return (running_loss / batch_num)
```

Prepare Data

After loading the data, we firstly plot the price path. Then we normalize the stock price, and again, plot the normalized price path. Then we slice the data into training, validation and testing data set with following data size. And we set lag to 10 days.

Training	60%
Validation	15%
Testing	25%

We also create tensor datasets from these three datasets, and then create data loader out of them.

Python Code

```python
#   Load data
df_stock_data = pd.read_csv('Zhu-Zhengyu-data.csv')[["Date", "Adj Close"]]
df_stock_data["Date"] = pd.to_datetime(df_stock_data["Date"])
df_stock_data.set_index("Date", inplace = True)
ax = plt.axes()
plt.plot(df_stock_data, color = "#3F5161")
plt.savefig("Stock_Price.png", dpi = 150)
plt.show()
```

```
#   Normalization
df_stock_data_norm = (df_stock_data - np.mean(df_stock_data)) / np.std(df_stock_data)
ax = plt.axes()
plt.plot(df_stock_data_norm, color = "#3F5161")
plt.savefig("Stock_Price_Normalized.png", dpi = 150)
plt.show()
```


```
#   Prepare training and testing set
# Training: 2016-2018, Testing: 2019
narray_training_data_raw = df_stock_data_norm.loc[ : dtm.date(2019, 1, 1)]
narray_testing_data = df_stock_data_norm.loc[dtm.date(2019, 1, 1) : ]

# Training: 55%, Validation 20%, Testing 25%
slice_line = round(narray_training_data_raw.shape[0] * 3 / 4)
narray_training_data = narray_training_data_raw.iloc[ : slice_line]
narray_validation_data = narray_training_data_raw.iloc[slice_line : ]
narray_training_data = narray_training_data.to_numpy()
narray_validation_data = narray_validation_data.to_numpy()
narray_testing_data = narray_testing_data.to_numpy()

lag = 10

narray_training_data = np.concatenate([narray_training_data[i: i + lag + 1].reshape(1, -1) for i in
range(len(narray_training_data) - lag)], 0)
narray_validation_data = np.concatenate([narray_validation_data[i: i + lag + 1].reshape(1, -1) for i in
range(len(narray_validation_data) - lag)], 0)
narray_testing_data = np.concatenate([narray_testing_data[i: i + lag + 1].reshape(1, -1) for i in
range(len(narray_testing_data) - lag)], 0)

tensor_training_data = torch.from_numpy(narray_training_data).float()
tensor_validation_data = torch.from_numpy(narray_validation_data).float()
tensor_testing_data = torch.from_numpy(narray_testing_data).float()

#   Create Tensor Dataset
td_training_data = Data.TensorDataset(tensor_training_data[:, 0:-1], tensor_training_data[:, -1:])
td_validation_data = Data.TensorDataset(tensor_validation_data[:, 0:-1], tensor_validation_data[:, -1:])
td_testing_data = Data.TensorDataset(tensor_testing_data[:, 0:-1], tensor_testing_data[:, -1:])
```

```
#   Create Dataloaders
dl_training_data = Data.DataLoader(td_training_data, batch_size = 256, shuffle = True, num_workers = 0)
dl_validation_data = Data.DataLoader(td_validation_data, batch_size = 32, shuffle = True, num_workers = 0)
dl_testing_data = Data.DataLoader(td_testing_data, batch_size = 32, shuffle = True, num_workers = 0)
```

Training & Testing

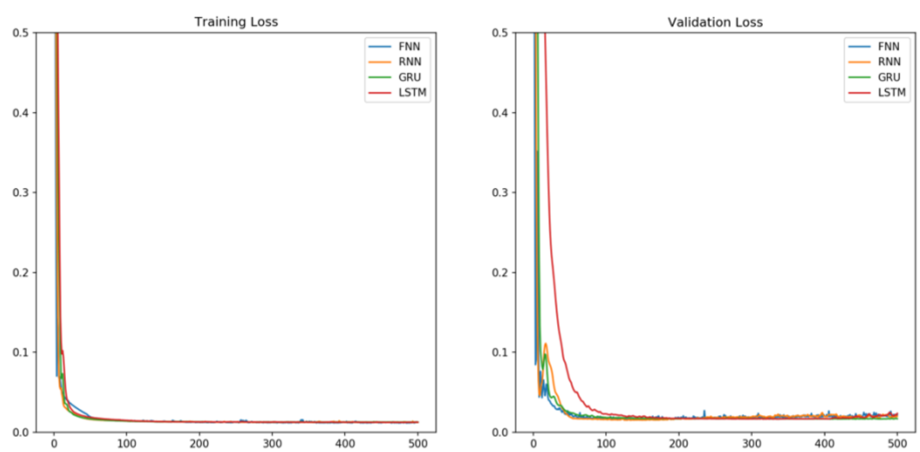
We first set the hidden layer size to 8 for all models. Then we train each model and get the loss plots for training and validation.

Python Code

```
#   Training
fnn = FNN_net(10, 8, 8, 1)
rnn = RNN_net(1, 8, 1)
gru = GRU_net(1, 8, 1)
lstm = LSTM_net(1, 8, 1)

net_names = ["FNN", "RNN", "GRU", "LSTM"]
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 7))
ax1.set_title("Training Loss")
ax2.set_title("Validation Loss")

for i, net in enumerate([fnn, rnn, gru, lstm]):
    print("Training", net_names[i])
    loss_train, loss_vali = train(net, 500, dl_training_data)
    ax1.plot(range(1, 1 + len(loss_train)), loss_train, label = net_names[i])
    ax2.plot(range(1, 1 + len(loss_vali)), loss_vali, label = net_names[i])
ax1.set_ylim([0, 0.5])
ax2.set_ylim([0, 0.5])
ax1.legend()
ax2.legend()
plt.show()
```



We print the loss table to show the loss in training, validation and testing set. The loss values are quite similar among the models. And it shows a lag from the real price to predicted price.

```
#   Testing
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(22, 7))
ax1.set_title("Prediction on Training set")
ax2.set_title("Prediction on Validation set")
ax3.set_title("Prediction on Test set")

ax1.plot(tensor_training_data[500 : 550, -1], label = "Origin", color = "#3F5161")
ax2.plot(tensor_validation_data[:, -1], label = "Origin", color = "#3F5161")
ax3.plot(tensor_testing_data[:, -1], label = "Origin", color = "#3F5161")

df_loss_testing = pd.DataFrame(columns = ["Net", "Training", "Validation", "Testing"], index = [0, 1, 2, 3])

count = 0
for i, net in enumerate([rnn, gru, lstm, fnn]):
    ax1.plot(net(tensor_training_data[500 : 550, : -1])[ : , ].squeeze().data.numpy(), label = net_names[i])
    ax2.plot(net(tensor_validation_data[ : , : -1])[ : , 0].squeeze().data.numpy(), label = net_names[i])
    ax3.plot(net(tensor_testing_data[ : , : -1])[ : , 0].squeeze().data.numpy(), label = net_names[i])

    df_loss_testing.iloc[count]["Net"] = net_names[i]
    df_loss_testing.iloc[count]["Training"] = test(net, dl_training_data)
    df_loss_testing.iloc[count]["Validation"] = test(net, dl_validation_data)
    df_loss_testing.iloc[count]["Testing"] = test(net, dl_testing_data)
    count += 1

print(df_loss_testing)
df_loss_testing.to_csv("Loss_Testing.csv")
ax1.legend()
ax2.legend()
ax3.legend()
plt.savefig("Prediction.png", dpi = 150)
plt.show()
```

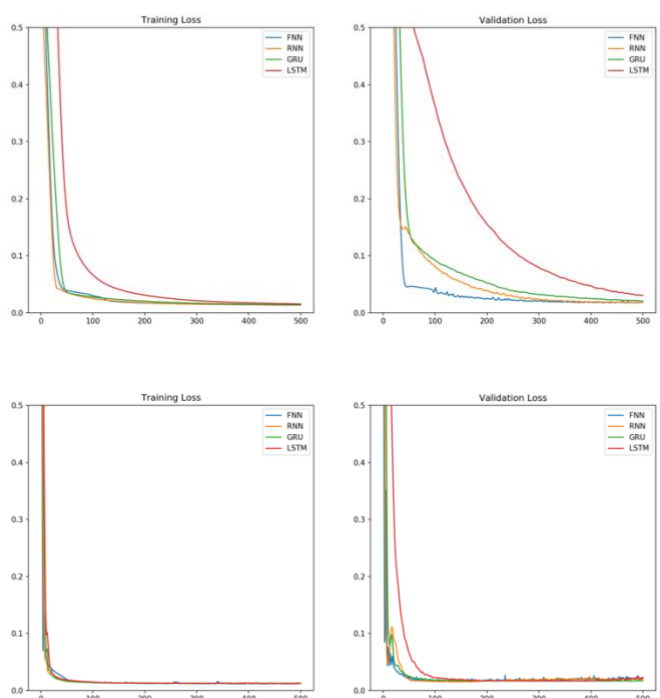
Net	Training	Validation	Testing
FNN	0.01243	0.01996	0.01107
RNN	0.01261	0.01688	0.01074
GRU	0.01263	0.02347	0.01082
LSTM	0.01249	0.02184	0.01244

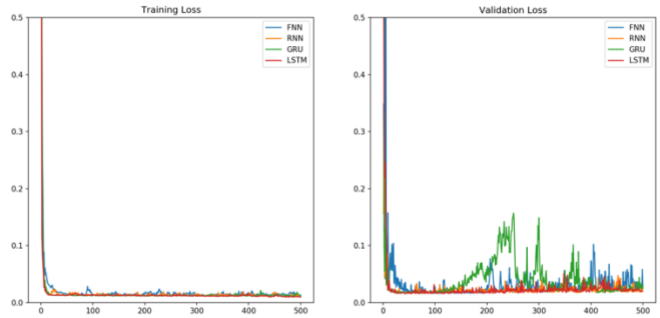
Tuning

We first change the learning rate to 0.001, 0.01 and 0.1. And get the following results. From the loss table we find that all the performances are quite good and there's no underfitting and overfitting. And the testing performance of lr = 0.01 is better than others. we find that the loss graph of 0.001 and 0.01 is decreasing more smoothly than 0.1.

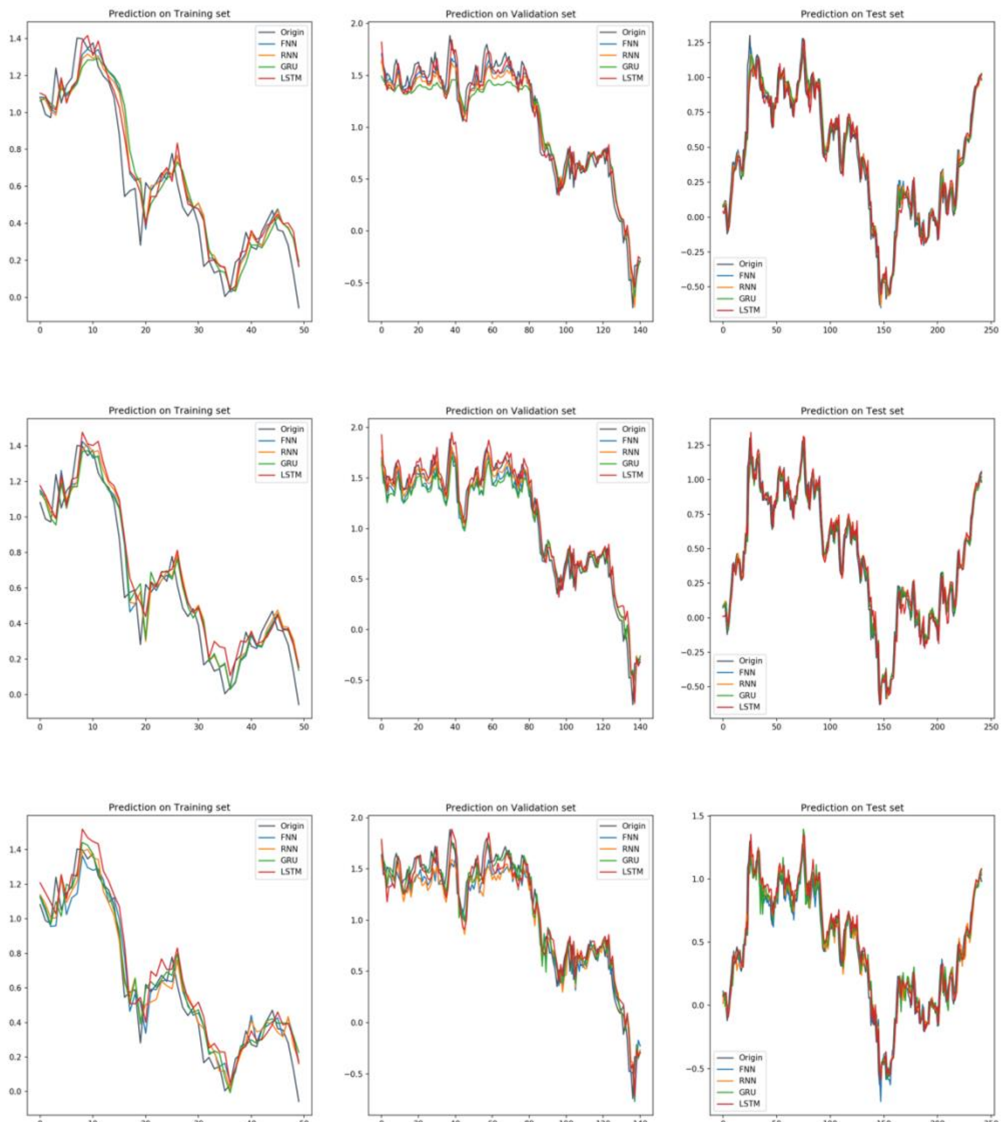
	Training			**Validation**			**Testing**		
LR	**0.001**	**0.01**	**0.1**	**0.001**	**0.01**	**0.1**	**0.001**	**0.01**	**0.1**
FNN	0.01372	0.01243	0.01070	0.01766	0.01996	0.02592	0.01087	0.01107	0.01194
RNN	0.01390	0.01261	0.01084	0.02059	0.01688	0.02925	0.01104	0.01074	0.01274
GRU	0.01542	0.01263	0.00974	0.03016	0.02347	0.01992	0.01301	0.01082	0.01237
LSTM	0.01344	0.01249	0.01224	0.01786	0.02184	0.02426	0.01239	0.01244	0.01351

Loss graph (lr = 0.001, 0.01, 0.1).





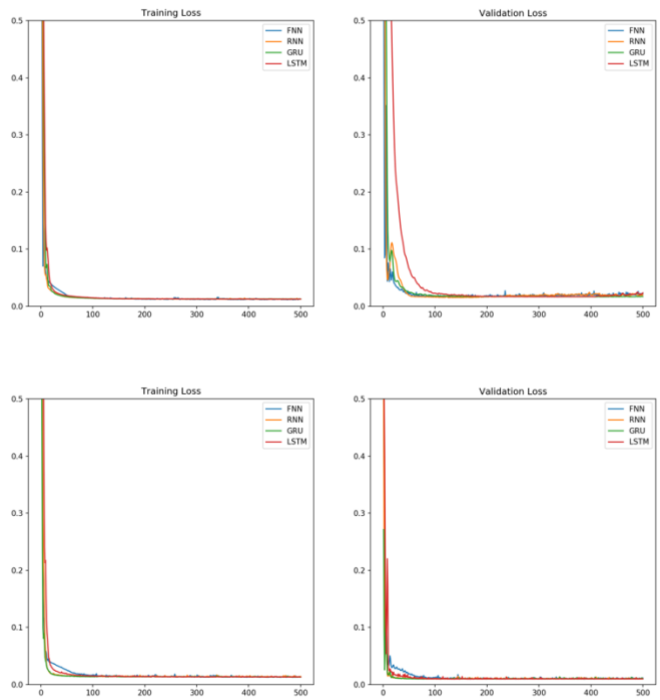
Prediction graph (lr = 0.001, 0.01, 0.1).



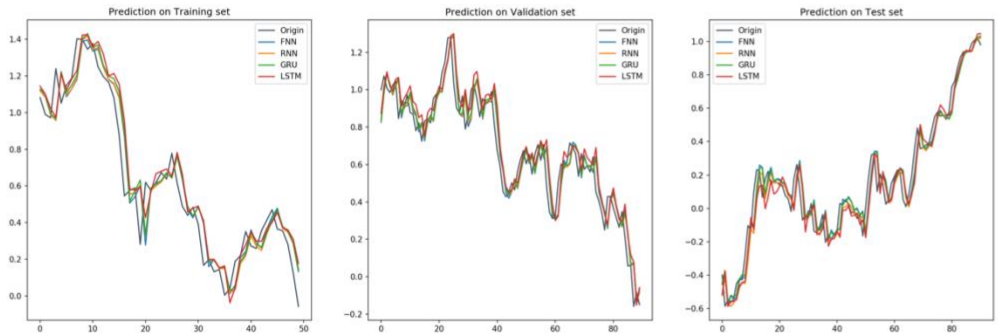
Then we tune the hyperparameter, we change the dataset size from 60%, 15%, 25% to 80%, 10%, 10%. And we have the following result that the losses from validation and testing set after tuning are lower than the previous ones.

Net	Training	Training (Tuned)	Validation	Validation (Tuned)	Testing	Testing (Tuned)
FNN	0.01243	0.01370	0.01996	0.00986	0.01107	0.00868
RNN	0.01261	0.01299	0.01688	0.00977	0.01074	0.00948
GRU	0.01263	0.01316	0.02347	0.0100	0.01082	0.00865
LSTM	0.01249	0.01341	0.02184	0.01146	0.01244	0.00993

Loss graph (set size = [60%, 15%, 25%] and [80%, 10%, 10%])



Prediction graph (set size = [60%, 15%, 25%] and [80%, 10%, 10%])



Since the loss in training set and testing set are quite close, and in a good range. There's no need to tune the network size.