**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)

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

# 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()

A close up of a logo

Description automatically generated

# 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()

A close up of a logo

Description automatically generated

# 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])

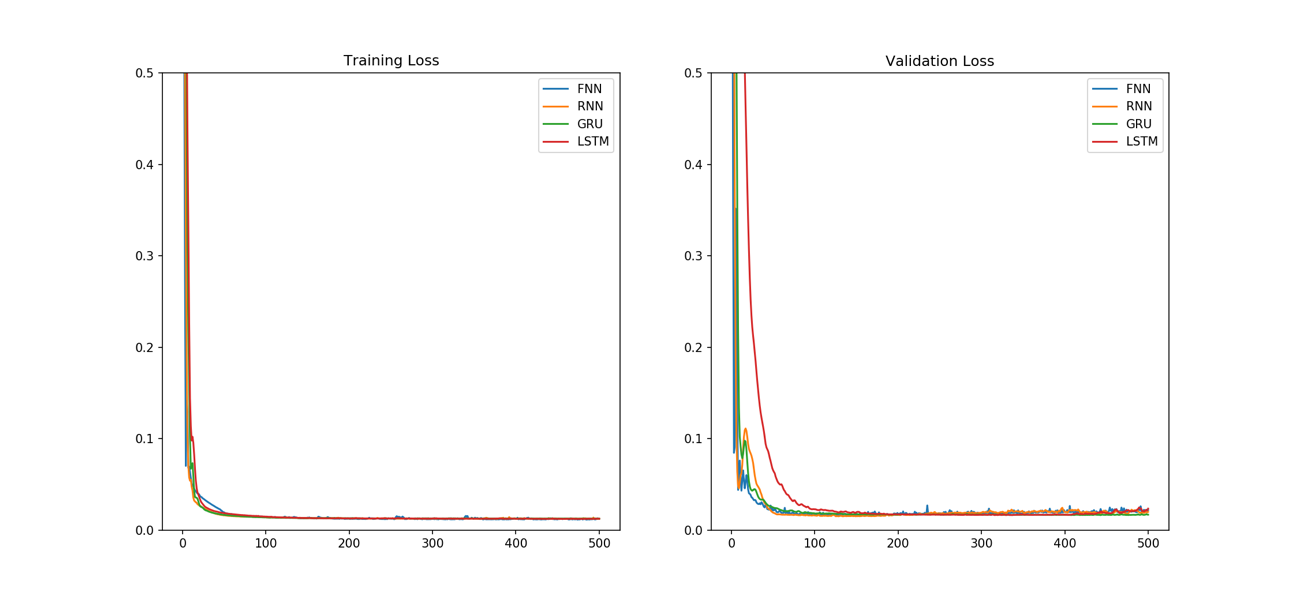
ax1.set\_ylim([0, 0.5])

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 |

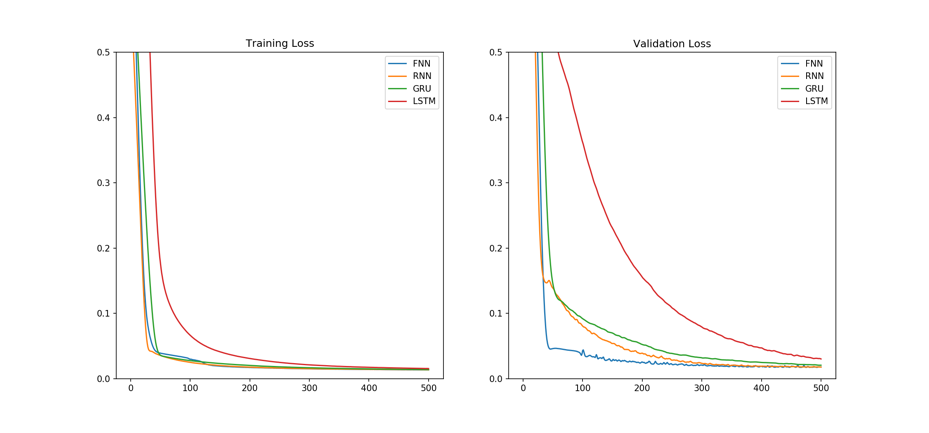


**Tuning**

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).



A screenshot of a social media post

Description automatically generated

A screenshot of a cell phone

Description automatically generated

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

A close up of a map

Description automatically generated

A close up of a map

Description automatically generated

A close up of a map

Description automatically generated

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%])

A screenshot of a social media post

Description automatically generatedA screenshot of a social media post

Description automatically generated

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

A close up of a map

Description automatically generated

A close up of a map

Description automatically generated

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.