# TERA HW#4

2023 年 11 月 3 日

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import statsmodels.api as sm
from sklearn.linear_model import Lasso, Ridge
from sklearn.tree import DecisionTreeRegressor as RT
from sklearn.ensemble import RandomForestRegressor as RF
from sklearn.svm import SVR
import multiprocess as mp
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import PolynomialFeatures
```

## § 1 CUDA

安装 CUDA 支持下的 pytorch，目前版本支持 CUDA12.1。安装成功之后，尝试使用类似下列命令进行验证： torch.cuda.is_available()

[26]:
```
# show CUDA information
print(f'PyTorch version: {torch.__version__}')
print('*'*40)
print(f'_CUDA version: ')
!nvcc --version
print('*'*40)
print(f'CUDNN version: {torch.backends.cudnn.version()}')
print(f'Available GPU devices: {torch.cuda.device_count()}')
print(f'Device Name: {torch.cuda.get_device_name()}')
```

```
PyTorch version: 2.1.0+cu121
**************************************
_CUDA version:
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Fri_Sep__8_19:56:38_Pacific_Daylight_Time_2023
Cuda compilation tools, release 12.3, V12.3.52
Build cuda_12.3.r12.3/compiler.33281558_0
**************************************
CUDNN version: 8801
Available GPU devices: 1
Device Name: NVIDIA GeForce RTX 4060 Laptop GPU
```

## § 2 Feedford Neural Network

建立一个最简单的 Feedforwar Neural Net，参数自己选择。重复 InnerCode==3 的股票的滚动窗口预测实验。汇报 MSFE 并和之前的预测结果进行比较。
注意：该题目无需使用 GPU 并行。

[3]:
```
dat = pd.read_csv('play_data.csv')
IN = dat.InnerCode == 3
dat_selected = dat[IN].copy()
dat_selected.loc[:, "Return"] = dat_selected["ClosePrice"].pct_change()
dat_selected.loc[:, "Return"] = dat_selected["Return"].fillna(0)
```

[ ]:
```
dat_selected # show dataset
```

### § 2.1 Repeat the rolling window exercise using classic machine learning algorithms

[5]:
```
import warnings

# Suppress RuntimeWarning
warnings.filterwarnings("ignore", category=RuntimeWarning)

# define main execution function
def forecast_fun(data):
    import numpy as np
```

```
import pandas as pd
import statsmodels.api as sm
from sklearn.linear_model import Lasso, Ridge
from sklearn.tree import DecisionTreeRegressor as RT
from sklearn.ensemble import RandomForestRegressor as RF
from sklearn.svm import SVR
np.random.seed(1111)

# ready data
yt, xt, ye, xe = data
# ols
ols = sm.OLS(yt,xt).fit()
f1 = ols.predict(xe)
# lasso
lasso_model = Lasso()
lasso_model.fit(xt, yt)
f2 = lasso_model.predict(xe)
# ridge
ridge_model = Ridge()
ridge_model.fit(xt, yt)
f3 = ridge_model.predict(xe)
# regression tree
tree_model = RT(min_samples_leaf=10)
tree_model.fit(xt, yt)
f4 = tree_model.predict(xe)
# random forest regression
forest_model = RF(max_features=3, random_state=1)
forest_model.fit(xt, yt)
f5 = forest_model.predict(xe)
# SVR (Linear Kernel)
linear_svr_model = SVR(kernel='linear')
linear_svr_model.fit(xt, yt)
f6 = linear_svr_model.predict(xe)
# SVR (Gaussian Kernel)
gaussian_svr_model = SVR(kernel='rbf')
gaussian_svr_model.fit(xt, yt)
```

```
    f7 = gaussian_svr_model.predict(xe)
    # merge results and deliver output
    f = np.hstack((f1.values,f2,f3,f4,f5,f6,f7))
    ERRt = f-np.tile(ye,7)
    return ERRt
```

[6]:
```
%%time

# Main execution
if __name__ == '__main__':
    # start worker pool
    num_cores = mp.cpu_count()
    pool = mp.Pool(processes=num_cores)
    # Generate data partitions
    WL = 1000
    Result = []
    y = dat_selected["Return"]
    x = dat_selected.iloc[:,3:-2]
    x.iloc[:,[4,5,7,8]] = np.log(x.iloc[:,[4,5,7,8]])
    x = x.shift(1)
    x = x.iloc[1:-1,:]
    y = y.iloc[1:-1]
    x.replace([np.inf, -np.inf, np.nan], 0, inplace=True)
    y.replace([np.inf, -np.inf, np.nan], 0, inplace=True)
    x = sm.add_constant(x)
    T = y.size
    datafull = []
    for t in range(T-WL):
        INt = np.arange(t,t+WL)
        INe = [t+WL]
        yt = y.iloc[INt]
        xt = x.iloc[INt,:]
        ye = y.iloc[INe]
        xe = x.iloc[INe,:]
        datafull.append([yt,xt,ye,xe])
    # parallel estimation
```

```
    results = pool.map(forecast_fun, datafull)
    ERR = np.array(results)
    Result.append(np.mean(ERR ** 2,axis=0))

    pool.close()
    pool.join()
```

```
CPU times: total: 953 ms
Wall time: 38 s
```

[7]:
```
# print evaluation results
VN = ['LM','Lasso','Ridge','RT','RF','SVR-L','SVR-G']
R = pd.DataFrame(Result)
R.columns = VN
R
```

[7]:

| | LM | Lasso | Ridge | RT | RF | SVR-L | SVR-G |
|---|---|---|---|---|---|---|---|
| 0 | 0.000531 | 0.00052 | 0.000526 | 0.000672 | 0.0006 | 0.000806 | 0.000607 |

## § 2.2 Feedforward Neural Network

[8]:
```
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_sizes, output_size):
        super(NeuralNet, self).__init__()
        self.layers = nn.ModuleList()
        sizes = [input_size] + hidden_sizes + [output_size]
        for i in range(len(sizes) - 1):
            self.layers.append(nn.Linear(sizes[i], sizes[i + 1]))

    def forward(self, x):
        for layer in self.layers[:-1]:
            x = torch.relu(layer(x))
        x = torch.sigmoid(self.layers[-1](x))
        return x

# convert data to tensor
X = torch.tensor(x.values, dtype=torch.float32).to("cuda")
Y = torch.tensor(y.values.reshape(-1, 1), dtype=torch.float32).to("cuda")
```

```
X.device
```

[8]:
```
device(type='cuda', index=0)
```

### § 2.2.1 One layer shallow network

[9]:
```
# Instantiate the model
input_size = X.shape[1]
output_size = 1
hidden_sizes = [3]
model = NeuralNet(input_size, hidden_sizes, output_size).to("cuda")   # move␣
 ↪the model to GPU
print(model)
```

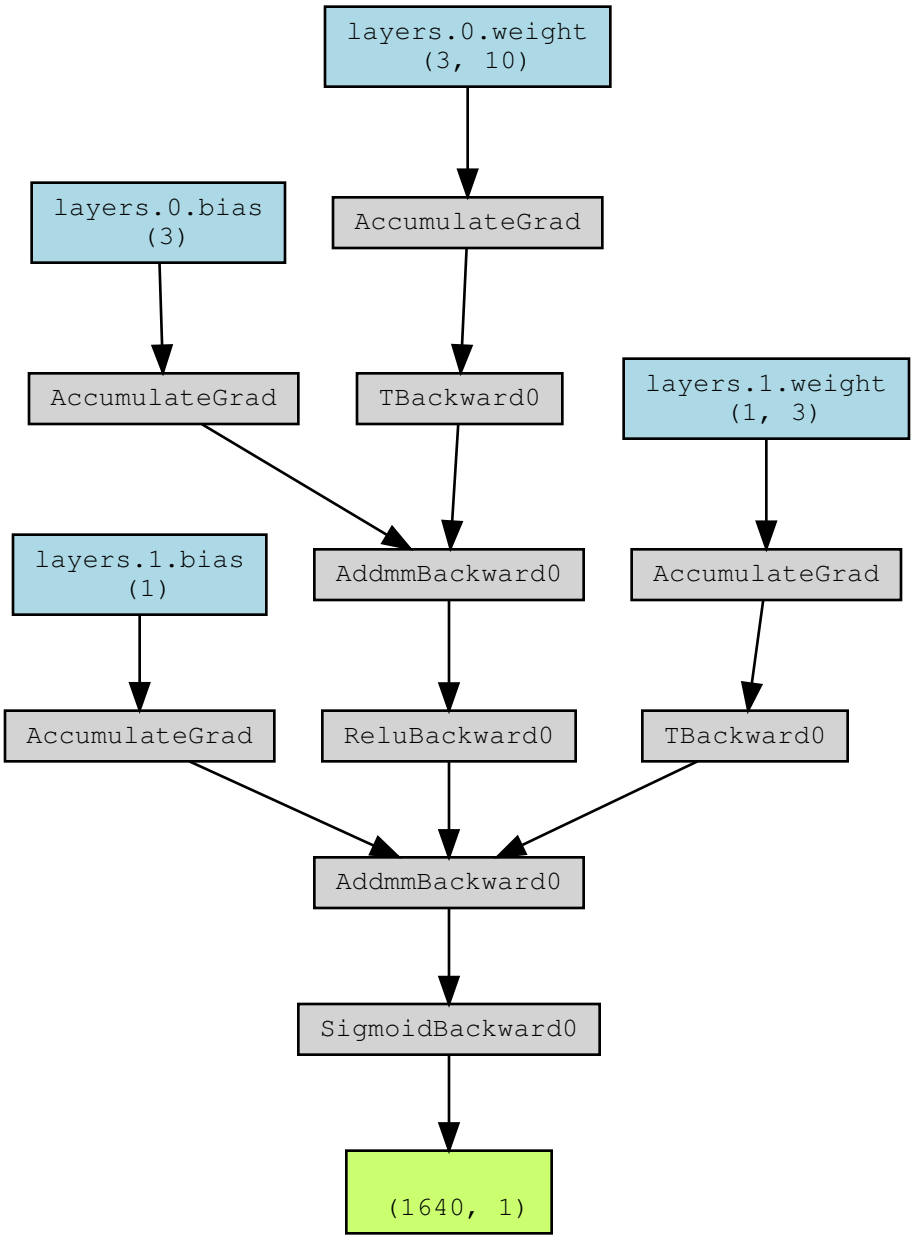
```
NeuralNet(
  (layers): ModuleList(
    (0): Linear(in_features=10, out_features=3, bias=True)
    (1): Linear(in_features=3, out_features=1, bias=True)
  )
)
```

[10]:
```
# Loss and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters())

# Training the model
num_epochs = 100
for epoch in range(num_epochs):
    # Forward pass
    outputs = model(X)
    loss = criterion(outputs, Y)

    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 10 == 0:
```

```
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

```
Epoch [10/100], Loss: 0.0251
Epoch [20/100], Loss: 0.0066
Epoch [30/100], Loss: 0.0026
Epoch [40/100], Loss: 0.0016
Epoch [50/100], Loss: 0.0012
Epoch [60/100], Loss: 0.0010
Epoch [70/100], Loss: 0.0009
Epoch [80/100], Loss: 0.0009
Epoch [90/100], Loss: 0.0008
Epoch [100/100], Loss: 0.0008
```

[11]:
```
# Visualize the model architecture
from torchviz import make_dot
_ = make_dot(model(X), params=dict(model.named_parameters())).render("01",␣
↪format="svg")
```

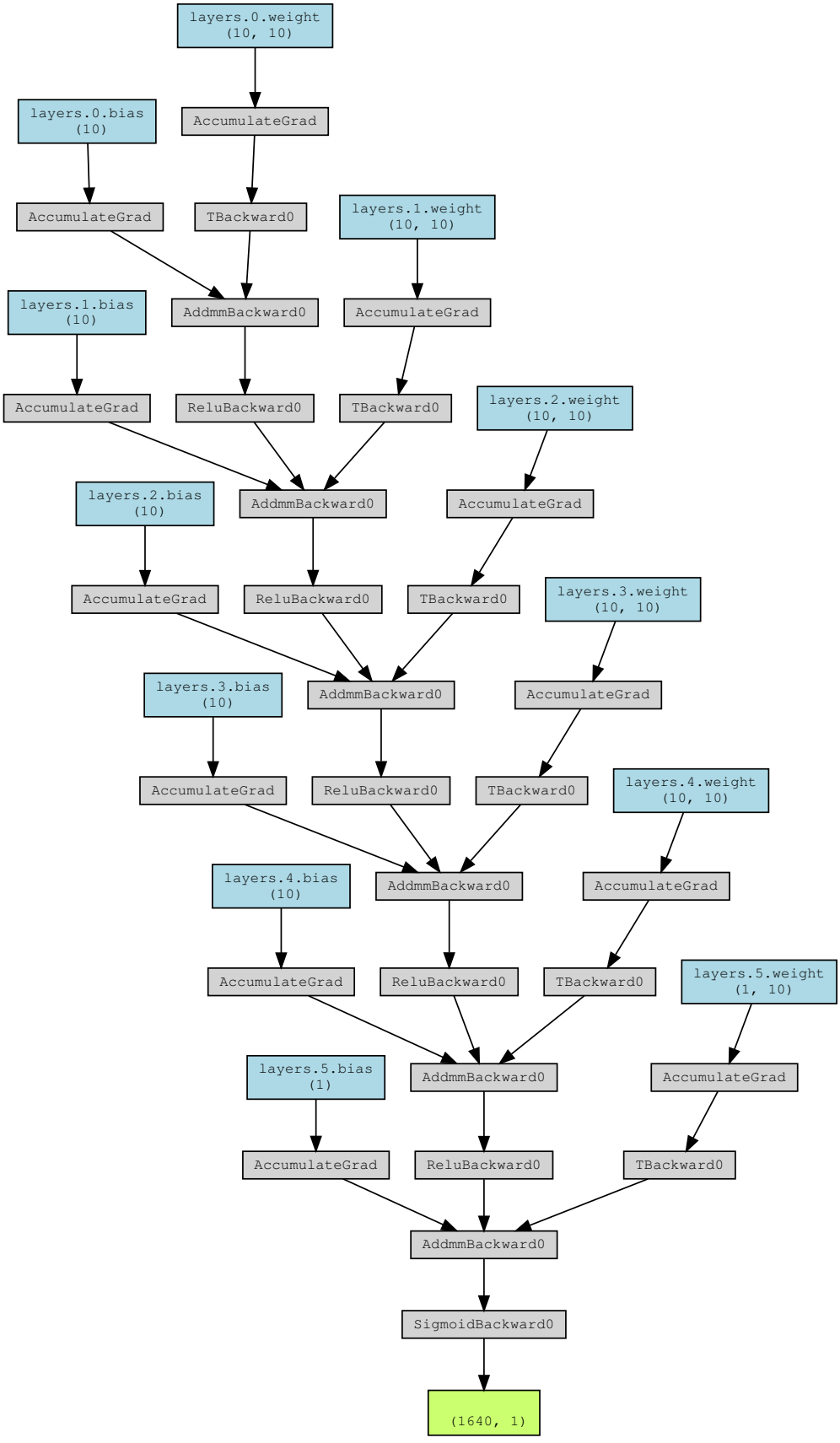



### § 2.2.2 More Complicated NN Model

[21]:
```
# Instantiate the model
input_size = X.shape[1]
output_size = 1
hidden_sizes = [10] * 5
model = NeuralNet(input_size, hidden_sizes, output_size).to("cuda")
print(model)
```

```
# Loss and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters())

# Training the model
num_epochs = 100
for epoch in range(num_epochs):
    # Forward pass
    outputs = model(X)
    loss = criterion(outputs, Y)

    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Visualize the model architecture
_ = make_dot(model(X), params=dict(model.named_parameters())).render("02",␣
 ↪format="svg")
```

```
NeuralNet(
  (layers): ModuleList(
    (0-4): 5 x Linear(in_features=10, out_features=10, bias=True)
    (5): Linear(in_features=10, out_features=1, bias=True)
  )
)
```


layers.0.weight
(10, 10)
layers.0.bias
(10)
AccumulateGrad
AccumulateGrad
TBackward0
layers.1.weight
(10, 10)
layers.1.bias
(10)
AddmmBackward0
AccumulateGrad
AccumulateGrad
ReluBackward0
TBackward0
layers.2.weight
(10, 10)
layers.2.bias
(10)
AddmmBackward0
AccumulateGrad
AccumulateGrad
ReluBackward0
TBackward0
layers.3.weight
(10, 10)
layers.3.bias
(10)
AddmmBackward0
AccumulateGrad
AccumulateGrad
ReluBackward0
TBackward0
layers.4.weight
(10, 10)
layers.4.bias
(10)
AddmmBackward0
AccumulateGrad
AccumulateGrad
ReluBackward0
TBackward0
layers.5.weight
(1, 10)
layers.5.bias
(1)
AddmmBackward0
AccumulateGrad
AccumulateGrad
ReluBackward0
TBackward0
AddmmBackward0
SigmoidBackward0
(1640, 1)

## § 2.3 CPU vs. CUDA

[13]:
```
%%time
N = 10000
a = torch.ones(N,N,device="cpu")
a.device
```

```
CPU times: total: 203 ms
Wall time: 48.1 ms
```

[13]:
```
device(type='cpu')
```

[14]:
```
%%time
a = torch.ones(N,N,device="cuda")
a.device
```

```
CPU times: total: 15.6 ms
Wall time: 26.6 ms
```

[14]:
```
device(type='cuda', index=0)
```

### § 2.3.1 out-of-sample forecasting exercise

[15]:
```
import warnings

# Suppress RuntimeWarning
warnings.filterwarnings("ignore", category=RuntimeWarning)

# define main execution function
def forecast_fun(data):
    import numpy as np
    import pandas as pd
    from sklearn.linear_model import Lasso, Ridge
    from sklearn.preprocessing import StandardScaler
    import torch
    import torch.nn as nn
    import torch.optim as optim
    np.random.seed(1111)

    class NeuralNet(nn.Module):
```

```
    def __init__(self, input_size, hidden_sizes, output_size):
        super(NeuralNet, self).__init__()
        self.layers = nn.ModuleList()
        sizes = [input_size] + hidden_sizes + [output_size]
        for i in range(len(sizes) - 1):
            self.layers.append(nn.Linear(sizes[i], sizes[i + 1]))

    def forward(self, x):
        for layer in self.layers[:-1]:
            x = torch.relu(layer(x))
        x = torch.sigmoid(self.layers[-1](x))
        return x

# ready data
yt, xt, ye, xe = data
# lasso
lasso_model = Lasso()
lasso_model.fit(xt, yt)
f1 = lasso_model.predict(xe)

# NN-simple
scaler = StandardScaler()
X = torch.tensor(scaler.fit_transform(xt.values), dtype=torch.float32)
Xe = torch.tensor(scaler.transform(xe.values), dtype=torch.float32)
Y = torch.tensor(yt.values.reshape(-1, 1), dtype=torch.float32)
# Instantiate the model
input_size = X.shape[1]
output_size = 1
hidden_sizes = [3]
model = NeuralNet(input_size, hidden_sizes, output_size)
# Loss and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters())
# Training the model
num_epochs = 100
for epoch in range(num_epochs):
```

```
        # Forward pass
        outputs = model(X)
        loss = criterion(outputs, Y)
        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    with torch.no_grad():
        f2 = model(Xe).item()

    # NN-complicate
    hidden_sizes2 = [10] * 5
    model2 = NeuralNet(input_size, hidden_sizes2, output_size)
    # Loss and optimizer
    criterion2 = nn.MSELoss()
    optimizer2 = optim.Adam(model2.parameters())
    # Training the model
    for epoch in range(num_epochs):
        # Forward pass
        outputs = model2(X)
        loss = criterion2(outputs, Y)
        # Backward and optimize
        optimizer2.zero_grad()
        loss.backward()
        optimizer2.step()
    with torch.no_grad():
        f3 = model2(Xe).item()



    # merge results and deliver output
    f = np.hstack((f1,f2,f3))
    ERRt = f-np.tile(ye,len(f))
    return ERRt
```

[16]:
```
%%time
```

```
# Main execution
if __name__ == '__main__':
    # start worker pool
    num_cores = mp.cpu_count()
    pool = mp.Pool(processes=num_cores)
    # Generate data partitions
    WL = 1000
    Result = []
    y = dat_selected["Return"]
    x = dat_selected.iloc[:,3:-2]
    x.iloc[:,[4,5,7,8]] = np.log(x.iloc[:,[4,5,7,8]])
    x = x.shift(1)
    x = x.iloc[1:-1,:]
    y = y.iloc[1:-1]
    x.replace([np.inf, -np.inf, np.nan], 0, inplace=True)
    y.replace([np.inf, -np.inf, np.nan], 0, inplace=True)
    x = sm.add_constant(x)
    T = y.size
    datafull = []
    for t in range(T-WL):
        INt = np.arange(t,t+WL)
        INe = [t+WL]
        yt = y.iloc[INt]
        xt = x.iloc[INt,:]
        ye = y.iloc[INe]
        xe = x.iloc[INe,:]
        datafull.append([yt,xt,ye,xe])
    # parallel estimation
    results = pool.map(forecast_fun, datafull)
    ERR = np.array(results)
    Result.append(np.mean(ERR ** 2,axis=0))

    pool.close()
    pool.join()

# print evaluation results
```

```
VN = ['Lasso','NN-simple','NN-complicate']
R = pd.DataFrame(Result)
R.columns = VN
R
```

```
CPU times: total: 953 ms
Wall time: 27 s
```

[16]:

| | Lasso | NN-simple | NN-complicate |
|---|---|---|---|
| 0 | 0.00052 | 0.157204 | 0.060114 |

### § 2.3.2 Explore activation functions

[17]:
```
%%time

import warnings

# Suppress RuntimeWarning
warnings.filterwarnings("ignore", category=RuntimeWarning)

# define main execution function
def forecast_fun(data):
    import numpy as np
    import pandas as pd
    from sklearn.linear_model import Lasso, Ridge
    from sklearn.preprocessing import StandardScaler
    import torch
    import torch.nn as nn
    import torch.optim as optim
    np.random.seed(1111)

    class NeuralNet(nn.Module):
        def __init__(self, input_size, hidden_sizes, output_size):
            super(NeuralNet, self).__init__()
            self.layers = nn.ModuleList()
            sizes = [input_size] + hidden_sizes + [output_size]
            for i in range(len(sizes) - 1):
                self.layers.append(nn.Linear(sizes[i], sizes[i + 1]))
```

```
    def forward(self, x):
        for layer in self.layers[:-1]:
            x = torch.relu(layer(x))
        x = torch.sigmoid(self.layers[-1](x))
        return x

class NeuralNet2(nn.Module):
    def __init__(self, input_size, hidden_sizes, output_size):
        super(NeuralNet2, self).__init__()
        self.layers = nn.ModuleList()
        sizes = [input_size] + hidden_sizes + [output_size]
        for i in range(len(sizes) - 1):
            self.layers.append(nn.Linear(sizes[i], sizes[i + 1]))

    def forward(self, x):
        for layer in self.layers[:-1]:
            x = torch.relu(layer(x))
        x = self.layers[-1](x)
        return x

class NeuralNet3(nn.Module):
    def __init__(self, input_size, hidden_sizes, output_size):
        super(NeuralNet3, self).__init__()
        self.layers = nn.ModuleList()
        sizes = [input_size] + hidden_sizes + [output_size]
        for i in range(len(sizes) - 1):
            self.layers.append(nn.Linear(sizes[i], sizes[i + 1]))

    def forward(self, x):
        for layer in self.layers[:-1]:
            x = torch.relu(layer(x))
        x = torch.tanh(self.layers[-1](x))
        return x

# ready data
```

```
    yt, xt, ye, xe = data
    # lasso
    lasso_model = Lasso()
    lasso_model.fit(xt, yt)
    f1 = lasso_model.predict(xe)

    # NN-default
    scaler = StandardScaler()
    X = torch.tensor(scaler.fit_transform(xt.values), dtype=torch.float32)
    Xe = torch.tensor(scaler.transform(xe.values), dtype=torch.float32)
    Y = torch.tensor(yt.values.reshape(-1, 1), dtype=torch.float32)
    # Instantiate the model
    input_size = X.shape[1]
    output_size = 1
    hidden_sizes = [10] * 5
    model = NeuralNet(input_size, hidden_sizes, output_size)
    # Loss and optimizer
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters())
    # Training the model
    num_epochs = 100
    for epoch in range(num_epochs):
        # Forward pass
        outputs = model(X)
        loss = criterion(outputs, Y)
        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    with torch.no_grad():
        f2 = model(Xe).item()

    # NN-linear
    model2 = NeuralNet2(input_size, hidden_sizes, output_size)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model2.parameters())
```

```
    for epoch in range(num_epochs):
        outputs = model2(X)
        loss = criterion(outputs, Y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    with torch.no_grad():
        f3 = model2(Xe).item()

    # NN-tanh
    model3 = NeuralNet3(input_size, hidden_sizes, output_size)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model3.parameters())
    for epoch in range(num_epochs):
        outputs = model3(X)
        loss = criterion(outputs, Y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    with torch.no_grad():
        f4 = model3(Xe).item()



    # merge results and deliver output
    f = np.hstack((f1,f2,f3,f4))
    ERRt = f-np.tile(ye,len(f))
    return ERRt

# Main execution
if __name__ == '__main__':
    # start worker pool
    num_cores = mp.cpu_count()
    pool = mp.Pool(processes=num_cores)
    # Generate data partitions
    WL = 1000
    Result = []
```

```
    y = dat_selected["Return"]
    x = dat_selected.iloc[:,3:-2]
    x.iloc[:,[4,5,7,8]] = np.log(x.iloc[:,[4,5,7,8]])
    x = x.shift(1)
    x = x.iloc[1:-1,:]
    y = y.iloc[1:-1]
    x.replace([np.inf, -np.inf, np.nan], 0, inplace=True)
    y.replace([np.inf, -np.inf, np.nan], 0, inplace=True)
    x = sm.add_constant(x)
    T = y.size
    datafull = []
    for t in range(T-WL):
        INt = np.arange(t,t+WL)
        INe = [t+WL]
        yt = y.iloc[INt]
        xt = x.iloc[INt,:]
        ye = y.iloc[INe]
        xe = x.iloc[INe,:]
        datafull.append([yt,xt,ye,xe])
    # parallel estimation
    results = pool.map(forecast_fun, datafull)
    ERR = np.array(results)
    Result.append(np.mean(ERR ** 2,axis=0))

    pool.close()
    pool.join()

# print evaluation results
VN = ['Lasso','NN-default','NN-Linear','NN-tanh']
R = pd.DataFrame(Result)
R.columns = VN
R
```

```
CPU times: total: 1.11 s
Wall time: 45.7 s
```

[17]:

| | Lasso | NN-default | NN-Linear | NN-tanh |
|---|---|---|---|---|
| 0 | 0.00052 | 0.062623 | 0.001287 | 0.001062 |

### § 2.3.3 Regularization

[18]:

```
%%time

import warnings

# Suppress RuntimeWarning
warnings.filterwarnings("ignore", category=RuntimeWarning)

# define main execution function
def forecast_fun(data):
    import numpy as np
    import pandas as pd
    from sklearn.linear_model import Lasso, Ridge
    from sklearn.preprocessing import StandardScaler
    import torch
    import torch.nn as nn
    import torch.optim as optim
    np.random.seed(1111)

    class NeuralNet(nn.Module):
        def __init__(self, input_size, hidden_sizes, output_size,␣
↪dropout_rate=0):
            super(NeuralNet, self).__init__()
            self.layers = nn.ModuleList()
            sizes = [input_size] + hidden_sizes + [output_size]

            for i in range(len(sizes) - 2):
                self.layers.append(nn.Linear(sizes[i], sizes[i + 1]))
                self.layers.append(nn.ReLU())
                self.layers.append(nn.Dropout(dropout_rate))  # Adding dropout␣
↪after activation

            # Output layer with no activation function (linear)
```

```
            self.layers.append(nn.Linear(sizes[-2], sizes[-1]))

        def forward(self, x):
            for layer in self.layers:
                x = layer(x)
            return x

    # ready data
    yt, xt, ye, xe = data
    # lasso
    lasso_model = Lasso()
    lasso_model.fit(xt, yt)
    f1 = lasso_model.predict(xe)

    # NN-linear
    scaler = StandardScaler()
    X = torch.tensor(scaler.fit_transform(xt.values), dtype=torch.float32)
    Xe = torch.tensor(scaler.transform(xe.values), dtype=torch.float32)
    Y = torch.tensor(yt.values.reshape(-1, 1), dtype=torch.float32)
    # Instantiate the model
    input_size = X.shape[1]
    output_size = 1
    hidden_sizes = [10] * 5
    model = NeuralNet(input_size, hidden_sizes, output_size)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters())
    num_epochs = 100
    for epoch in range(num_epochs):
        outputs = model(X)
        loss = criterion(outputs, Y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    with torch.no_grad():
        f2 = model(Xe).item()
```

```
    # NN-linear with l2-norm
    optimizer2 = optim.Adam(model.parameters(), weight_decay=0.01)
    for epoch in range(num_epochs):
        outputs = model(X)
        loss = criterion(outputs, Y)
        optimizer2.zero_grad()
        loss.backward()
        optimizer2.step()
    with torch.no_grad():
        f3 = model(Xe).item()

    # NN-linear with l2-norm and early stopping
    num_epochs2 = 50
    for epoch in range(num_epochs2):
        outputs = model(X)
        loss = criterion(outputs, Y)
        optimizer2.zero_grad()
        loss.backward()
        optimizer2.step()
    with torch.no_grad():
        f4 = model(Xe).item()

    # NN-linear with l2-norm, early stopping, and dropout
    model2 = NeuralNet(input_size, hidden_sizes, output_size, dropout_rate = 0.
↪1)
    optimizer3 = optim.Adam(model2.parameters(), weight_decay=0.01)
    for epoch in range(num_epochs2):
        outputs = model2(X)
        loss = criterion(outputs, Y)
        optimizer3.zero_grad()
        loss.backward()
        optimizer3.step()
    with torch.no_grad():
        f5 = model2(Xe).item()
```

```
    # merge results and deliver output
    f = np.hstack((f1,f2,f3,f4,f5))
    ERRt = f-np.tile(ye,len(f))
    return ERRt

# Main execution
if __name__ == '__main__':
    # start worker pool
    num_cores = mp.cpu_count()
    pool = mp.Pool(processes=num_cores)
    # Generate data partitions
    WL = 1000
    Result = []
    y = dat_selected["Return"]
    x = dat_selected.iloc[:,3:-2]
    x.iloc[:,[4,5,7,8]] = np.log(x.iloc[:,[4,5,7,8]])
    x = x.shift(1)
    x = x.iloc[1:-1,:]
    y = y.iloc[1:-1]
    x.replace([np.inf, -np.inf, np.nan], 0, inplace=True)
    y.replace([np.inf, -np.inf, np.nan], 0, inplace=True)
    x = sm.add_constant(x)
    T = y.size
    datafull = []
    for t in range(T-WL):
        INt = np.arange(t,t+WL)
        INe = [t+WL]
        yt = y.iloc[INt]
        xt = x.iloc[INt,:]
        ye = y.iloc[INe]
        xe = x.iloc[INe,:]
        datafull.append([yt,xt,ye,xe])
    # parallel estimation
    results = pool.map(forecast_fun, datafull)
    ERR = np.array(results)
    Result.append(np.mean(ERR ** 2,axis=0))
```

```
    pool.close()
    pool.join()

# print evaluation results
VN = ['Lasso','NN-linear','NN-linear-l2','NN-linear-l2-es','NN-linear-l2-es-dp']
R = pd.DataFrame(Result)
R.columns = VN
R
```

```
CPU times: total: 1.14 s
Wall time: 51.3 s
```

[18]:

| | Lasso | NN-linear | NN-linear-l2 | NN-linear-l2-es | NN-linear-l2-es-dp |
|---|---|---|---|---|---|
| 0 | 0.00052 | 0.001369 | 0.000565 | 0.000538 | 0.008889 |

## § 3 Convolutional Networks

[19]:
```
%%time

import warnings

# Suppress RuntimeWarning
warnings.filterwarnings("ignore", category=RuntimeWarning)

# define main execution function
def forecast_fun(data):
    class ConvNet(nn.Module):
        def __init__(self, input_size, output_size, out_channels, kernel_size,␣
↪stride=1):
            super(ConvNet, self).__init__()
            self.conv1 = nn.Conv1d(in_channels=1, out_channels=out_channels,␣
↪kernel_size=kernel_size)
            self.relu = nn.ReLU()
            self.flatten = nn.Flatten()
            self.fc = nn.Linear(out_channels * (input_size - kernel_size + 1),␣
↪output_size)
```

```
            self.tanh = nn.Tanh()

        def forward(self, x):
            x = self.conv1(x)
            x = self.relu(x)
            x = self.flatten(x)
            x = self.fc(x)
            x = self.tanh(x)
            return x

    # Ready data
    yt, xt, ye, xe = data

    # Lasso
    lasso_model = Lasso()
    lasso_model.fit(xt, yt)
    f1 = lasso_model.predict(xe)

    # CNN
    scaler = StandardScaler()
    X = torch.tensor(scaler.fit_transform(xt.values), dtype=torch.float32).
↪to("cuda")
    Xe = torch.tensor(scaler.transform(xe.values), dtype=torch.float32).
↪to("cuda")
    Y = torch.tensor(yt.values.reshape(-1, 1), dtype=torch.float32).to("cuda")

    # Instantiate the model
    input_size = X.shape[1]
    output_size = 1
    out_channels = 2 ** 10     # usually set at powers of 2
    kernel_size = 5            # usually set at odd number
    model = ConvNet(input_size, output_size, out_channels, kernel_size).
↪to("cuda")
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), weight_decay=0.01)
```

```
    # Training the model
    num_epochs = 100
    for epoch in range(num_epochs):
        # Forward pass
        outputs = model(X.unsqueeze(1))  # Add channel dimension for 1D␣
↪convolution
        loss = criterion(outputs, Y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Remove the channel dimension for prediction
    Xe = Xe.unsqueeze(1)
    with torch.no_grad():
        f2 = model(Xe).item()

    # merge results and deliver output
    f = np.hstack((f1,f2))
    ERRt = f-np.tile(ye,len(f))
    return ERRt


# Main Estimation
WL = 1000
Result = []
y = dat_selected["Return"]
x = dat_selected.iloc[:,3:-2]
x.iloc[:,[4,5,7,8]] = np.log(x.iloc[:,[4,5,7,8]])
x = x.shift(1)
x = x.iloc[1:-1,:]
y = y.iloc[1:-1]
x.replace([np.inf, -np.inf, np.nan], 0, inplace=True)
y.replace([np.inf, -np.inf, np.nan], 0, inplace=True)
x = sm.add_constant(x)
T = y.size
ERR = []
```

```
for t in range(T-WL):
    INt = np.arange(t,t+WL)
    INe = [t+WL]
    yt = y.iloc[INt]
    xt = x.iloc[INt,:]
    ye = y.iloc[INe]
    xe = x.iloc[INe,:]
    results = forecast_fun([yt,xt,ye,xe])
    ERR.append(results)

# print evaluation results
VN = ['Lasso','CNN']
Result = [np.mean(np.array(ERR) ** 2,axis=0)]
R = pd.DataFrame(Result)
R.columns = VN
R
```

```
CPU times: total: 44.9 s
Wall time: 1min 26s
```

[19]:

| | Lasso | CNN |
|---|---|---|
| 0 | 0.00052 | 0.000642 |