CNN-LSTM Parallel Moneymaker

In [1]:
```
import torch
import torch.optim
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader

import time
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
X_train, X_val, y_train, y_val, train_mean, val_mean, train_std, val_std
= torch.load("assets/all.pt")
```

In [3]:
```
print("X_train shape: \t\t", X_train.shape)
print("X_val shape: \t\t", X_val.shape)
print("y_train shape: \t\t", y_train.shape)
print("y_val shape: \t\t", y_val.shape)
```

```
X_train shape:          torch.Size([2542795, 122, 3])
X_val shape:            torch.Size([282732, 122, 3])
y_train shape:          torch.Size([2542795, 1])
y_val shape:            torch.Size([282732, 1])
```

In [4]:
```
# optimize for GPU if exists
if torch.cuda.is_available():
    X_train = X_train.cuda()
    y_train = y_train.cuda()
    X_val = X_val.cuda()
    y_val = y_val.cuda()
```

In [5]:
```
N, S, D = X_train.shape
```

Training a CNN

In [6]:
```
class Dataset(torch.utils.data.Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y

    def __len__(self):
        return len(self.X)

    def __getitem__(self, index):
        return self.X[index], self.y[index]
```

In [7]:
```
batch_size = 32
dataset = Dataset(X_train, y_train)
loader = DataLoader(dataset, batch_size, shuffle=True)
```

In [8]:
```
class CNNClassifier(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(CNNClassifier, self).__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, batch_first=True)
        self.drop = nn.Dropout()
        self.conv1 = nn.Conv1d(input_dim, hidden_dim, 7)
        self.pool = nn.MaxPool1d(3)
        self.flattened_dim = int((S - 7) / 3) * hidden_dim

        self.fc = nn.Linear(self.flattened_dim + hidden_dim, output_dim)

    def forward(self, x, h=None):
        # LSTM
        if type(h) == type(None):
            x1, hn = self.lstm(x)
        else:
            x1, hn = self.lstm(x, h.detach())
        x1 = x1[:, -1, :]
        x1 = self.drop(x1)

        # Conv
        x2 = torch.transpose(x, 1, 2)
        x2 = self.pool(F.relu(self.conv1(x2)))
        x2 = x2.view(-1, self.flattened_dim)

        x = torch.cat([x1, x2], dim=1)
        out = self.fc(x)
        return out
```

In [9]:
```
input_dim = 3
hidden_dim = 32
output_dim = 1
```

In [10]:
```
model = CNNClassifier(input_dim, hidden_dim, output_dim)
if torch.cuda.is_available():
    model = model.to("cuda")

criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters())
```

In [11]:
```
train_accs = []
val_accs = []
train_losses = []
val_losses = []
epoch = 0
```

Train the model

In [12]:
```
def pred_val(X_val, model):
    val_batch_size = 1000
    val_set_size = X_val.shape[0]
    preds = []
    with torch.no_grad():
        for i in range(0, val_set_size, val_batch_size):
            start = i
            end = min(i+val_batch_size, val_set_size)
            preds.append(model(X_val[start:end]))
    pred = torch.cat(preds, dim=0)
    return pred
```

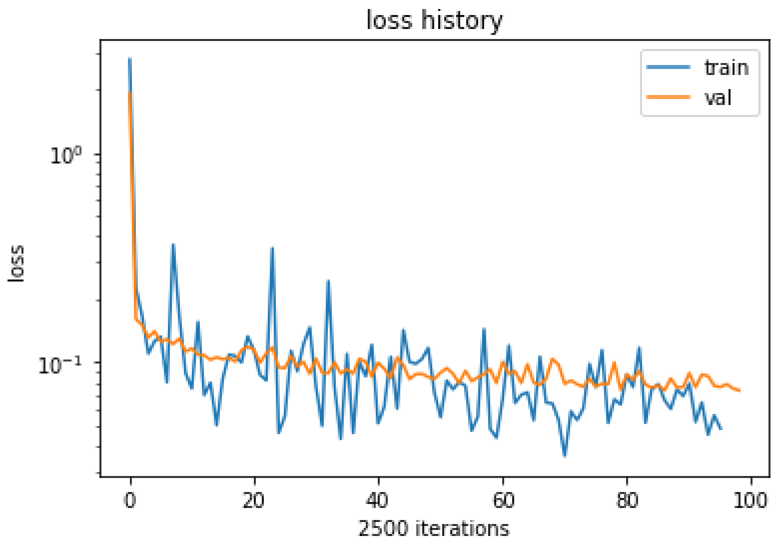
In [13]:
```
t0 = time.time()
num_epochs = 3
for ep in range(num_epochs):
    tstart = time.time()
    for i, data in enumerate(loader):
        model.train()
        print("{}/{}".format(i, int(X_train.shape[0] / batch_size)), end
='\r')
        optimizer.zero_grad()
        outputs = model(data[0])
        loss = criterion(outputs, data[1])
        loss.backward()
        optimizer.step()

        if i % 2500 == 0:
            with torch.no_grad():
                model.eval()
                train_losses.append(loss.item())
                pXval = pred_val(X_val, model)
                vloss = criterion(pXval, y_val)
                val_losses.append(vloss.item())
                torch.save({
                    'epoch': epoch,
                    'model_state_dict': model.state_dict(),
                    'optimizer_state_dict': optimizer.state_dict(),
                    'loss': loss,
                }, 'assets/partial_model.pt')
                print("training loss: {:<3.3f} \t val loss: {:<3.3f}".fo
rmat(loss, vloss))

    with torch.no_grad():
        model.eval()
        pXval = pred_val(X_val, model)
        vloss = criterion(pXval, y_val)
        val_losses.append(vloss.item())
        epoch += 1
        tend = time.time()
        print('epoch: {:<3d} \t time: {:<3.2f} \t val loss: {:<3.3f}'.fo
rmat(epoch,
                tend - tstart, vloss.item()))
time_total = time.time() - t0
print('Total time: {:4.3f}, average time per epoch: {:4.3f}'.format(time
_total, time_total / num_epochs))
```

```
training loss: 2.773     val loss: 1.913
training loss: 0.227     val loss: 0.161
training loss: 0.167     val loss: 0.151
training loss: 0.110     val loss: 0.131
training loss: 0.126     val loss: 0.141
training loss: 0.133     val loss: 0.126
training loss: 0.080     val loss: 0.129
training loss: 0.364     val loss: 0.122
training loss: 0.165     val loss: 0.130
training loss: 0.089     val loss: 0.113
training loss: 0.075     val loss: 0.117
training loss: 0.156     val loss: 0.109
training loss: 0.070     val loss: 0.108
training loss: 0.080     val loss: 0.103
training loss: 0.050     val loss: 0.106
training loss: 0.084     val loss: 0.104
training loss: 0.109     val loss: 0.105
training loss: 0.107     val loss: 0.101
training loss: 0.100     val loss: 0.114
training loss: 0.133     val loss: 0.119
training loss: 0.114     val loss: 0.115
training loss: 0.087     val loss: 0.100
training loss: 0.082     val loss: 0.110
training loss: 0.351     val loss: 0.118
training loss: 0.046     val loss: 0.095
training loss: 0.056     val loss: 0.094
training loss: 0.113     val loss: 0.107
training loss: 0.091     val loss: 0.096
training loss: 0.122     val loss: 0.101
training loss: 0.147     val loss: 0.088
training loss: 0.077     val loss: 0.104
training loss: 0.050     val loss: 0.089
epoch: 1         time: 576.77    val loss: 0.089
training loss: 0.244     val loss: 0.100
training loss: 0.076     val loss: 0.089
training loss: 0.043     val loss: 0.093
training loss: 0.110     val loss: 0.088
training loss: 0.046     val loss: 0.104
training loss: 0.101     val loss: 0.101
training loss: 0.086     val loss: 0.085
training loss: 0.121     val loss: 0.099
training loss: 0.051     val loss: 0.093
training loss: 0.061     val loss: 0.084
training loss: 0.106     val loss: 0.106
training loss: 0.060     val loss: 0.097
training loss: 0.142     val loss: 0.083
training loss: 0.101     val loss: 0.088
training loss: 0.098     val loss: 0.088
training loss: 0.104     val loss: 0.085
training loss: 0.117     val loss: 0.083
training loss: 0.072     val loss: 0.089
training loss: 0.055     val loss: 0.094
training loss: 0.082     val loss: 0.088
training loss: 0.074     val loss: 0.080
training loss: 0.081     val loss: 0.091
training loss: 0.077     val loss: 0.081
training loss: 0.047     val loss: 0.085
```

```
training loss: 0.055     val loss: 0.088
training loss: 0.144     val loss: 0.093
training loss: 0.048     val loss: 0.080
training loss: 0.044     val loss: 0.101
training loss: 0.067     val loss: 0.088
training loss: 0.120     val loss: 0.091
training loss: 0.064     val loss: 0.080
training loss: 0.070     val loss: 0.098
epoch: 2         time: 576.35    val loss: 0.080
training loss: 0.072     val loss: 0.078
training loss: 0.053     val loss: 0.084
training loss: 0.106     val loss: 0.104
training loss: 0.064     val loss: 0.098
training loss: 0.064     val loss: 0.079
training loss: 0.054     val loss: 0.082
training loss: 0.036     val loss: 0.079
training loss: 0.058     val loss: 0.076
training loss: 0.053     val loss: 0.083
training loss: 0.060     val loss: 0.077
training loss: 0.098     val loss: 0.079
training loss: 0.076     val loss: 0.079
training loss: 0.114     val loss: 0.100
training loss: 0.052     val loss: 0.074
training loss: 0.067     val loss: 0.087
training loss: 0.063     val loss: 0.083
training loss: 0.088     val loss: 0.091
training loss: 0.076     val loss: 0.078
training loss: 0.117     val loss: 0.076
training loss: 0.052     val loss: 0.078
training loss: 0.074     val loss: 0.073
training loss: 0.079     val loss: 0.084
training loss: 0.066     val loss: 0.076
training loss: 0.060     val loss: 0.076
training loss: 0.074     val loss: 0.089
training loss: 0.069     val loss: 0.076
training loss: 0.079     val loss: 0.087
training loss: 0.052     val loss: 0.086
training loss: 0.064     val loss: 0.077
training loss: 0.045     val loss: 0.076
training loss: 0.056     val loss: 0.079
training loss: 0.049     val loss: 0.075
epoch: 3         time: 574.98    val loss: 0.073
Total time: 1728.102, average time per epoch: 576.034
```

Training loss vs. validation loss

In [19]:
```
t_losses = [i for i in train_losses if i < 4000]
plt.plot(t_losses)
plt.plot(val_losses)
plt.title('loss history')
plt.xlabel('2500 iterations')
plt.ylabel('loss')
plt.yscale('log')
plt.legend(['train', 'val'])

plt.show()
```



Evaluate the model

In [16]:
```
X_train = X_train.cuda()
y_train = y_train.cuda()
X_val = X_val.cuda()
y_val = y_val.cuda()
```

In [17]:
```
model.eval()

pred = pred_val(X_val, model)
val_loss = criterion(pred, y_val).item()

print("Final model evaluation: ", val_loss)
```

```
Predicting 282000 - 282732...
Final model evaluation:  0.0734802633523941
```

One-step lag predictor

The one-step lag predictor simply outputs the last timestep in the input sequence. Our model should outperform the one-step lag predictor.

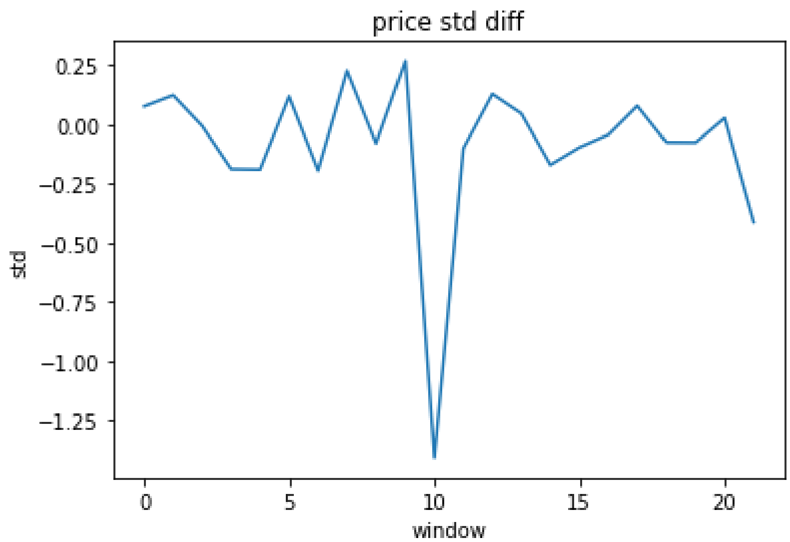
In [20]:
```
def one_step_lag_predictor(X):
    return X[:, -1, 0].unsqueeze(1)

p_val_naive = one_step_lag_predictor(X_val.cpu())
loss_naive = criterion(p_val_naive, y_val.cpu())

print("Loss from 1-step lag predictor:\t{}\nLoss from our model:\t\t{}".
format(loss_naive, val_loss))
```

```
Loss from 1-step lag predictor: 0.14193207025527954
Loss from our model:            0.0734802633523941
```

Standard deviation difference

In [21]:
```
# switch back to cpu for plotting
X_train = X_train.cpu()
y_train = y_train.cpu()
X_val = X_val.cpu()
y_val = y_val.cpu()
pred = pred.cpu()

# backprop components no longer needed
X_train = X_train.detach()
y_train = y_train.detach()
X_val = X_val.detach()
y_val = y_val.detach()
pred = pred.detach()
```

In [22]:
```
f1 = plt.figure()

ax1 = f1.add_subplot()
ax1.plot((pred - y_val)[500:522])
ax1.set_title('price std diff')
ax1.set(xlabel='window', ylabel='std')

plt.show()

# plt.plot((pred[:,3] - y_val.cpu()[:,3]).detach())
# plt.title('std difference')
# plt.plot([1, 2, 3])
```



In [24]:
```
# denormalize the data
pred_abs = pred * val_std[:,0].unsqueeze(1) + val_mean[:,0].unsqueeze(1)
y_val_abs = y_val.cpu() * val_std[:,0].unsqueeze(1) + val_mean[:,0].unsq
ueeze(1)
```

In [34]:
```
start_window = 500

fig, (ax1, ax2) = plt.subplots(2, figsize=(15, 10))
ax1.plot((pred_abs - y_val_abs)[start_window:start_window + 60])
l1, = ax2.plot(pred_abs[start_window:start_window + 60])
l1.set_label("predicted price")
l2, = ax2.plot(y_val_abs[start_window:start_window + 60])
l2.set_label("actual price")

plt.legend()
plt.show()
```