وزارت علوم، تحقیقات و فناوری
دانشگاه تحصیلات تکمیلی علوم پایه
گاوازنگ، زنجان

دانشکده علوم رایانه و فناوری اطلاعات

درس: یادگیری ژرف (Deep Learning)
تمرین ۳

استاد: دکتر رزاقی

دانشجو: امیرحسین صفری

شماره دانشجویی: ۱۴۰۱۴۱۲۱

بهار ۱۴۰۲

سوال اول

کد با وارد کردن(import) کتابخانه‌های لازم شروع می‌شود:
numpy برای عملیات عددی،
torch برای ساخت و آموزش مدل‌ها،
torch.nn برای تعریف ماژول‌های شبکه عصبی،
torch.autograd برای تمایز خودکار،
matplotlib.pyplot برای ترسیم نتایج،
sklearn preprocessing برای اعمال پیش پردازش روی داده‌ها،
و sklearn.metrics برای محاسبه RMSE

```
[ ]  # Import libraries
     # Notice that it is important that which libraries you use, so you should import
     # libraries just here in your code
     import pandas as pd
     import matplotlib.pyplot as plt
     import numpy as np

     from sklearn import preprocessing
     from sklearn.metrics import mean_squared_error

     import torch
     import torch.nn as nn
     from torch.autograd import Variable
```

لود کردن دیتا از فایل مربوطه

```
[ ]  # Read data
     df = pd.read_csv('time_series.csv')
     df
```

	<TICKER>	<PER>	<DATE>	<TIME>	<OPEN>	<HIGH>	<LOW>	<CLOSE>	<VOL>
0	US1.NVDA	D	20101004	0	11.24	11.430	11.01	11.23	18184874
1	US1.NVDA	D	20101005	0	11.48	11.500	11.29	11.32	18266877
2	US1.NVDA	D	20101006	0	11.32	11.370	10.67	10.78	25753399
3	US1.NVDA	D	20101007	0	10.82	10.840	10.38	10.70	18732301
4	US1.NVDA	D	20101008	0	10.65	10.950	10.51	10.86	16998198
...	...	...	...	...	...	...	...	...	...
2762	US1.NVDA	D	20210927	0	217.16	217.750	213.28	216.57	625939

دو ستون "<TICKER>"و "<PER>" شامل داده‌های غیر عددی و غیر ضروری برای ساخت مدل پیش‌بینی ما هستند؛ لذا آن‌ها را از دیتاست حذف می‌کنیم.

```
[ ]  features = []
     for feature in df.columns:
       if feature not in ["<TICKER>", "<PER>"] :
         features.append(feature)
```

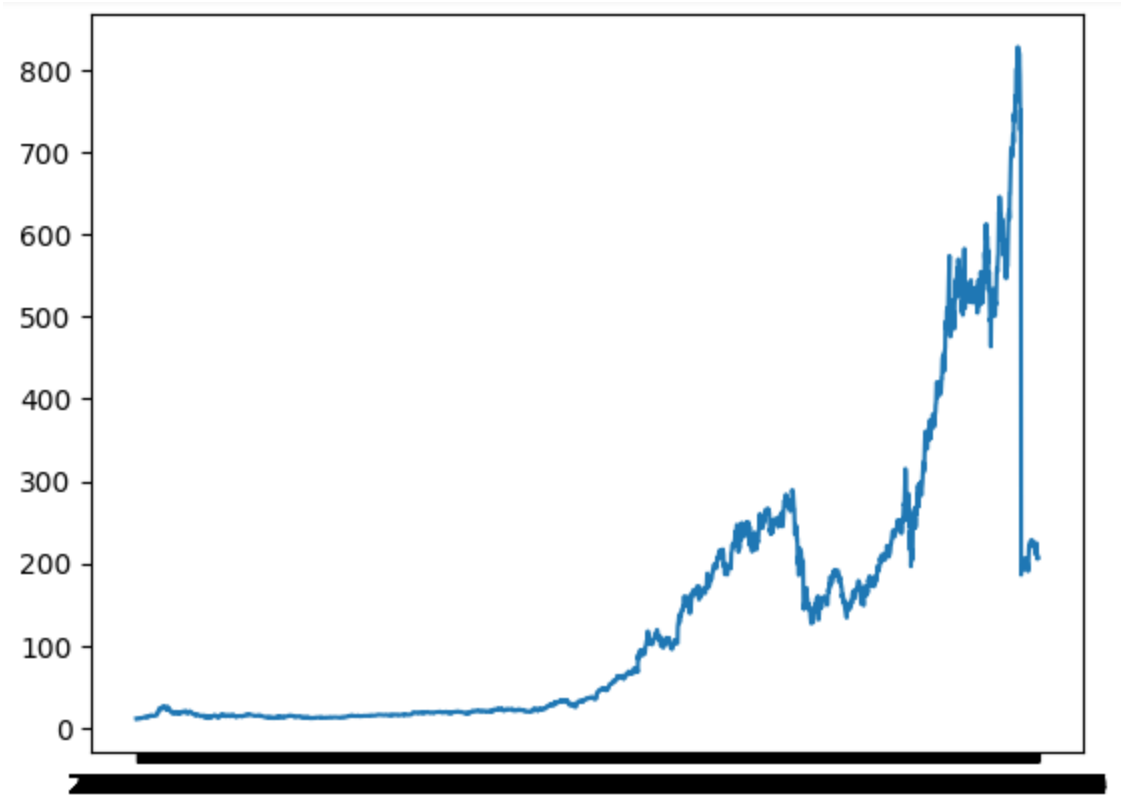
```
[ ]  features
```

```
['<DATE>', '<TIME>', '<OPEN>', '<HIGH>', '<LOW>', '<CLOSE>', '<VOL>']
```

```
[ ]  df = df[features]
```

با توجه به اینکه تاریخ‌ها به صورت یک عدد int هشت رقمی می‌باشند؛ لذا آن‌ها را به string تبدیل نموده تا بتوان برای هر روز یک ستون قیمت داشته باشیم. سپس نمودار قیمت هر روز را رسم می‌کنیم.

```
[ ]  # Plot close prices ("<CLOSE>") based on dates ("<DATE>")

     date = df["<DATE>"].to_numpy()
     date = [str(d) for d in date]

     close = df["<CLOSE>"].to_numpy()

     plt.plot(date, close)
```



دیتای خود را قبل از اعمال به مدل، به کمک کتابخانه‌ی MinMaxScaler پیش‌پردازش می‌کنیم تا اعداد دیتاست ما بین بازه‌ی [0 , 1] قرار گیرند. (MinMaxScaler از فرمول زیر استفاده می‌کند.)

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

```
[ ]  data = df.values
     min_max_scaler = preprocessing.MinMaxScaler()
     x_scaled = min_max_scaler.fit_transform(data)
     df = pd.DataFrame(x_scaled)
```

تابعی برای ایجاد داده‌های آموزشی و آزمایشی داده‌های موجودیو طول توالی

```
[ ]  # function to create train, test data given stock data and sequence length
     def load_data(stock, look_back):
         data_raw = stock.values # convert to numpy array
         data = []

         # create all possible sequences of length look_back
         for index in range(len(data_raw) - look_back):
             data.append(data_raw[index: index + look_back])

         data = np.array(data);
         test_set_size = int(np.round(0.2*data.shape[0]));
         train_set_size = data.shape[0] - (test_set_size);

         x_train = data[:train_set_size,:-1,:]
         y_train = data[:train_set_size,-1,:]

         x_test = data[train_set_size:,:-1]
         y_test = data[train_set_size:,-1,:]

         return [x_train, y_train, x_test, y_test]
```

نمایی از داده‌های پیش‌پردازش شده

```
[ ]  x_train

array([[[1.81823141e-05, 0.00000000e+00, 8.13166007e-04, ...,
         3.60772800e-04, 9.78880650e-05, 3.04233503e-01],
        [2.72734711e-05, 0.00000000e+00, 2.06325703e-04, ...,
         0.00000000e+00, 0.00000000e+00, 2.21044304e-01],
        [3.63646281e-05, 0.00000000e+00, 0.00000000e+00, ...,
         1.61725738e-04, 1.95776130e-04, 2.00497853e-01],
        ...,
        [2.02732802e-03, 0.00000000e+00, 5.09745855e-03, ...,
         5.26230671e-03, 5.26148349e-03, 7.36563434e-02],
        [2.03641918e-03, 0.00000000e+00, 5.27951064e-03, ...,
         5.56087730e-03, 5.17583144e-03, 5.41963022e-02],
        [2.04551033e-03, 0.00000000e+00, 5.19455300e-03, ...,
         5.49867509e-03, 5.17583144e-03, 4.96619083e-02]],

       ...,

       [[8.13704010e-01, 0.00000000e+00, 1.94832148e-01, ...,
         1.92851722e-01, 1.95164330e-01, 1.46660749e-01],
        [8.13713101e-01, 0.00000000e+00, 1.92174187e-01, ...,
         1.91110061e-01, 1.93500232e-01, 1.25760824e-01],
        [8.13740375e-01, 0.00000000e+00, 1.85632449e-01, ...,
```

سپس، random seed برای اطمینان از تکرارپذیری نتایج تنظیم می شود. این کار با تنظیم random seed برای Numpy و PyTorch انجام می شود.

```
[ ]  # Build model
     ####################
     # you can change these parameters to get better result
     input_dim = 1
     hidden_dim = 32
     num_layers = 2
     output_dim = 1

     # Set the random seed for reproducibility
     np.random.seed(0)
     torch.manual_seed(0)
```

این قسمت مدل LSTM را با استفاده از کلاس nn.Module ارائه شده توسط PyTorch تعریف می کند. (constructor) سازنده __init__ مدل را که شامل تعداد لایه های پنهان (num_layers)، بعد پنهان (hidden_dim) و بعد خروجی (output_dim) مقداردهی اولیه می کند.

ویژگی lstm نمونه‌ای از ماژول LSTM از nn.LSTM است که بعد ورودی، بعد پنهان، تعداد لایه‌ها و batch_first=True را می‌گیرد تا نشان دهد که تنسور(tensor) های ورودی و خروجی شکل (batch_dim, seq_dim, feature_dim) خواهند داشت).

ویژگی fc نمایانگر لایه کاملاً متصل است که وضعیت پنهان را به بعد خروجی ترسیم می کند.

```
[ ]  # Here we define our model as a class
     class LSTM(nn.Module):
         def __init__(self, input_dim, hidden_dim, num_layers, output_dim):
             super(LSTM, self).__init__()
             # Hidden dimensions
             self.hidden_dim = hidden_dim

             # Number of hidden layers
             self.num_layers = num_layers

             # batch_first=True causes input/output tensors to be of shape
             # (batch_dim, seq_dim, feature_dim)
             self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers, batch_first=True)

             # Readout layer
             self.fc = nn.Linear(hidden_dim, output_dim)
```

تابع forward گذر رو به جلو مدل LSTM را تعریف می کند. با توجه به یک تنسور(tensor) ورودی x، حالت پنهان و حالت سلول با صفر مقدار دهی اولیه می شوند. سپس ورودی با استفاده از ماژول lstm در لایه های LSTM منتشر می شود و خروجی آخرین مرحله زمانی از لایه خطی fc عبور می کند تا خروجی نهایی به دست آید.

```
    def forward(self, x):
        # Initialize hidden state with zeros
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_dim).requires_grad_()

        # Initialize cell state with zeros
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_dim).requires_grad_()

        # Propagate input through the LSTM layers
        out, _ = self.lstm(x, (h0, c0))

        # Pass the output of the last time step through the linear layer
        out = self.fc(out[:, -1, :])

        return out

    def backward(self, optimizer, criterion, inputs, labels):
        optimizer.zero_grad()  # Clear gradients
        outputs = self.forward(inputs)  # Forward pass
        loss = criterion(outputs, labels)  # Compute loss
        loss.backward()  # Backward pass
        optimizer.step()  # Update weights

        return loss.item()
```

تابع Backward یک روش کاربردی در کلاس LSTM است که یک پاس به عقب (backpropagation) را برای به روز رسانی وزن های مدل بر اساس ورودی ها و برچسب های ارائه شده انجام می دهد.

optimizer.zero_grad: این خط گرادیان تمام تانسورهای بهینه شده را پاک می کند. مهم است که قبل از محاسبه گذر به عقب، گرادیان ها را صفر کنیم، زیرا PyTorch به طور پیش فرض گرادیان ها را در پاس های رو به عقب بعدی جمع می کند.

outputs = self.forward(inputs): این خط با فراخوانی متد forward، یک گذر رو به جلو از مدل LSTM انجام می دهد. ورودی های تنسور ورودی را می گیرد و پیش بینی های خروجی را برمی گرداند.

loss = criterion(outputs, labels): این خط loss بین خروجی های پیش بینی شده (خروجی ها) و برچسب هایی که می‌دانیم درست هستند (برچسب ها) را محاسبه می‌کند. criterion معمولاً یک تابع loss است، مانند میانگین مربعات خطا (nn.MSEloss)، و loss را بر اساس خروجی های پیش بینی شده و برچسب های هدف محاسبه می کند.

loss.backward (): این خط گذر به عقب (backpropagation) را برای محاسبه گرادیان پارامترهای مدل با توجه به loss انجام می دهد. گرادیان ها به طور خودکار توسط مکانیزم autograd در PyTorch محاسبه می شوند.
optimizer.step(): این خط پارامترهای مدل (وزن‌ها و بایاس‌ها) را بر اساس گرادیان‌های محاسبه شده و الگوریتم بهینه سازی انتخاب شده به روز می‌کند. بهینه ساز نمونه ای از کلاس بهینه ساز است (به عنوان مثال torch.optim.Adam) و step() پارامترها را با استفاده از قانون بروزرسانی بهینه ساز به روز می کند.

همینطور برای کلاس GRU، کد کلاس GRU را به عنوان زیر کلاس nn.Module تعریف می کند. این یک مدل (GRU Gated Recurrent Unit) برای پیش‌بینی توالی استفاده شده است.
همانند کلاس LSTM، سازنده (__init__) معماری مدل را مقداردهی اولیه می کند، شامل بعد ورودی (input_dim)، بعد پنهان (hidden_dim)، تعداد لایه ها (num_layers) و بعد خروجی (output_dim). ویژگی gru نمونه ای از ماژول GRU از nn.GRU است. بعد ورودی، بعد پنهان، تعداد لایه‌ها و batch_first=True را می‌گیرد تا نشان دهد که تانسورهای ورودی و خروجی شکل (batch_dim، seq_dim، feature_dim) خواهند داشت. ویژگی fc نمایانگر لایه کاملاً متصل است که وضعیت پنهان را به بعد خروجی ترسیم می کند.

```python
class GRU(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, output_dim):
        super(GRU, self).__init__()
        # Hidden dimensions
        self.hidden_dim = hidden_dim

        # Number of hidden layers
        self.num_layers = num_layers

        # batch_first=True causes input/output tensors to be of shape
        # (batch_dim, seq_dim, feature_dim)
        self.gru = nn.GRU(input_dim, hidden_dim, num_layers, batch_first=True)

        # Readout layer
        self.fc = nn.Linear(hidden_dim, output_dim)
```

متد forward گذر رو به جلو مدل GRU را تعریف می کند. با توجه به تانسور ورودی x، حالت پنهان h0 با صفر مقداردهی می شود. سپس ورودی با استفاده از ماژول gru در لایه های GRU منتشر می شود. خروجی آخرین مرحله زمانی با استفاده از [:, -1, :]out استخراج می شود و از لایه خطی fc عبور می کند تا خروجی نهایی به دست آید.

همینطور برای backward GRU، متد backward یک تابع است که یک گذر به عقب (backpropagation) را برای بروزرسانی وزن های مدل GRU بر اساس ورودی ها و برچسب های ارائه شده انجام می دهد.

```
    def forward(self, x):
        # Initialize hidden state with zeros
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_dim).requires_grad_()

        # Propagate input through the GRU layers
        out, _ = self.gru(x, h0)

        # Pass the output of the last time step through the linear layer
        out = self.fc(out[:, -1, :])

        return out

    def backward(self, optimizer, criterion, inputs, labels):
        optimizer.zero_grad()  # Clear gradients
        outputs = self.forward(inputs)  # Forward pass
        loss = criterion(outputs, labels)  # Compute loss
        loss.backward()  # Backward pass
        optimizer.step()  # Update weights

        return loss.item()
```

این آرایه ها با استفاده از torch.from_numpy() به تانسور PyTorch تبدیل می شوند.

```
[ ]  X_train = torch.from_numpy(x_train).float()
     y_train = torch.from_numpy(y_train).float()
```

این کد مدل LSTM را برای آموزش مقداردهی اولیه می کند. کلاس LSTM با بعد ورودی مشخص شده، بعد پنهان، تعداد لایه ها و بعد خروجی نمونه سازی می شود.
بعد ورودی و خروجی برابر با ۷ در نظر گرفته شده است تا هر خروجی برای هر ستون تنظیم شود و بتوان میزان دقت را در نهایت محاسبه نمود.
به متغیر lstm_criterion تابع loss میانگین مربعات خطا nn.MSELoss() اختصاص داده شده است. این تابع loss برای اندازه گیری اختلاف بین پیش بینی های مدل و مقادیر هدف در طول آموزش استفاده می شود.
متغیر lstm_optimizer به Adam optimizer (torch.optim.Adam) با نرخ یادگیری 0.01 اختصاص داده شده است.
سپس حلقه آموزشی را برای مدل LSTM آغاز می کند که برای تعداد مشخصی اجرا می شود که آن را روی ۱۰۰ تنظیم کرده‌ایم.
در هر دور، متد lstm_model.backward() فراخوانی می شود تا یک تکرار آموزشی را انجام دهد. این روش بهینه ساز، loss، داده های ورودی (X_train) و داده های هدف (y_train) را به عنوان آرگومان می گیرد. پاس رو به جلو را انجام می دهد، loss را محاسبه می‌کند، پاس رو به عقب را انجام می دهد و وزن های مدل را با استفاده از بهینه ساز به روز می کند. در نهایت، مقدار loss را برای آن تکرار برمی‌گرداند.
مقدار ضرر در متغیر lstm_loss ذخیره می‌شود. در نهایت دور فعلی، تعداد کل دوره‌ها و مقدار loss را نمایش می‌دهد.

```
[ ]  # Train models
     # Build and train the LSTM model
     lstm_model = LSTM(input_dim=7, hidden_dim=32, num_layers=2, output_dim=7)
     lstm_criterion = nn.MSELoss()
     lstm_optimizer = torch.optim.Adam(lstm_model.parameters(), lr=0.01)

     epochs = 100
     for epoch in range(epochs):
         lstm_loss = lstm_model.backward(lstm_optimizer, lstm_criterion, X_train, y_train)
         print('LSTM - Epoch {}/{}, Loss: {:.4f}'.format(epoch + 1, 100, lstm_loss))
```

```
LSTM - Epoch 43/100, Loss: 0.0023
LSTM - Epoch 44/100, Loss: 0.0023
LSTM - Epoch 45/100, Loss: 0.0023
LSTM - Epoch 46/100, Loss: 0.0022
LSTM - Epoch 47/100, Loss: 0.0022
LSTM - Epoch 48/100, Loss: 0.0022
LSTM - Epoch 49/100, Loss: 0.0021
LSTM - Epoch 50/100, Loss: 0.0021
LSTM - Epoch 51/100, Loss: 0.0020
LSTM - Epoch 52/100, Loss: 0.0020
```

همین روند برای gru model

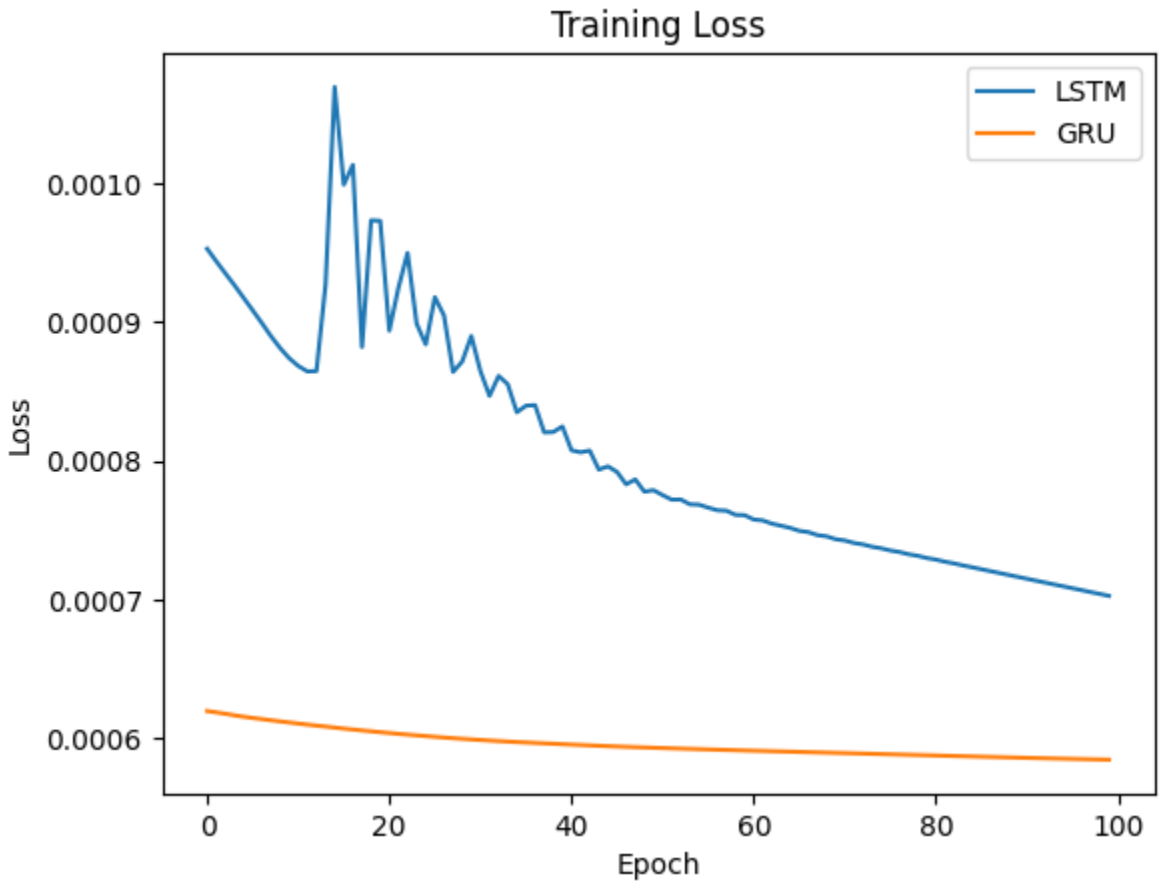
```
# Build and train the GRU model
gru_model = GRU(input_dim=7, hidden_dim=32, num_layers=2, output_dim=7)
gru_criterion = nn.MSELoss()
gru_optimizer = torch.optim.Adam(gru_model.parameters(), lr=0.01)

for epoch in range(100):
    gru_loss = gru_model.backward(gru_optimizer, gru_criterion, X_train, y_train)
    print('GRU - Epoch {}/{}, Loss: {:.4f}'.format(epoch + 1, 100, gru_loss))
```

```
GRU - Epoch 43/100, Loss: 0.0015
GRU - Epoch 44/100, Loss: 0.0014
GRU - Epoch 45/100, Loss: 0.0013
GRU - Epoch 46/100, Loss: 0.0012
GRU - Epoch 47/100, Loss: 0.0012
GRU - Epoch 48/100, Loss: 0.0011
GRU - Epoch 49/100, Loss: 0.0011
GRU - Epoch 50/100, Loss: 0.0010
```

مدل‌های بالا برای مشاهده اینکه هر کدام از مدل‌ها به درستی کار می‌کنند یک بار آموزش داده شدند و میزان loss به میزان 0.0006 و 0.0008 در نهایت رسید که میزان رضایت‌بخشی است؛ لذا پیش‌پردازش اثرگذار بوده است (بدون پیش‌پردازش نتایج خوبی دریافت نمی‌شد).

مدل‌ها با ذخیره‌ی میزان loss آن‌ها دوباره آموزش می‌دهیم. در نهایت نمودار loss بر حسب epoch را رسم می‌کنیم.

```
[ ]  # Plot loss based on epochs

     # Initialize lists to store losses
     lstm_losses = []
     gru_losses = []

     for epoch in range(100):
         lstm_loss = lstm_model.backward(lstm_optimizer, lstm_criterion, X_train, y_train)
         lstm_losses.append(lstm_loss)

         gru_loss = gru_model.backward(gru_optimizer, gru_criterion, X_train, y_train)
         gru_losses.append(gru_loss)

         print('Epoch {}/{}, LSTM Loss: {:.4f}, GRU Loss: {:.4f}'.format(epoch + 1, 100, lstm_loss, gru_loss))

     # Plot the loss
     plt.plot(range(len(lstm_losses)), lstm_losses, label='LSTM')
     plt.plot(range(len(gru_losses)), gru_losses, label='GRU')
     plt.xlabel('Epoch')
     plt.ylabel('Loss')
     plt.title('Training Loss')
     plt.legend()
     plt.show()
```



مدل آموزش‌دیده LSTM با استفاده از eval() به حالت ارزیابی تغییر می‌کند، که عملیات خاصی مانند dropout را غیرفعال می‌کند. پیش بینی ها بر روی داده های آزمایشی با استفاده از مدل آموزش دیده انجام می شود. پیش‌بینی ها به آرایه های numpy تبدیل می شوند.
در نهایت، RMSE بین لیبل درستی (y_train) و پیش‌بینی‌های LSTM با استفاده از تابع mean_squared_error از sklearn.metrics محاسبه می‌شود. مقدار RMSE چاپ می شود.
همین فرآیند را می توان برای مدل GRU برای مقایسه عملکرد مدل های LSTM و GRU اعمال کرد. (همانطور که مشاهده می‌شود مدل GRU کمی بهتر عمل کرده است.)

```python
# make predictions

X_test = torch.from_numpy(x_test).float()

# Set models to evaluation mode
lstm_model.eval()
gru_model.eval()

# Make predictions using the LSTM model
with torch.no_grad():
    lstm_predictions = lstm_model(X_test)

# Make predictions using the GRU model
with torch.no_grad():
    gru_predictions = gru_model(X_test)

# Convert predictions to numpy arrays
lstm_predictions = lstm_predictions.numpy()
gru_predictions = gru_predictions.numpy()
```

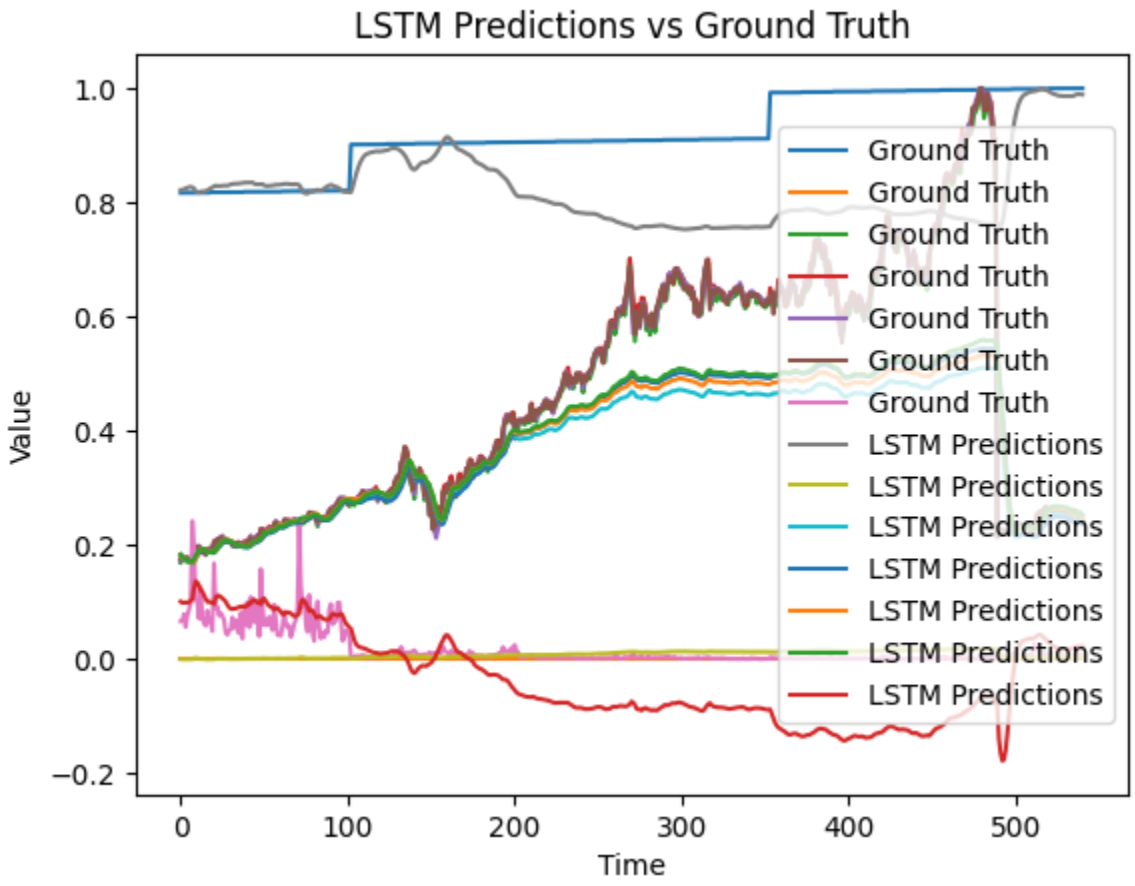
```python
# Calculate root mean squared error

# Calculate RMSE for LSTM predictions
lstm_rmse = np.sqrt(mean_squared_error(y_test, lstm_predictions))

# Calculate RMSE for GRU predictions
gru_rmse = np.sqrt(mean_squared_error(y_test, gru_predictions))

# Print the RMSE values
print('LSTM RMSE:', lstm_rmse)
print('GRU RMSE:', gru_rmse)
```

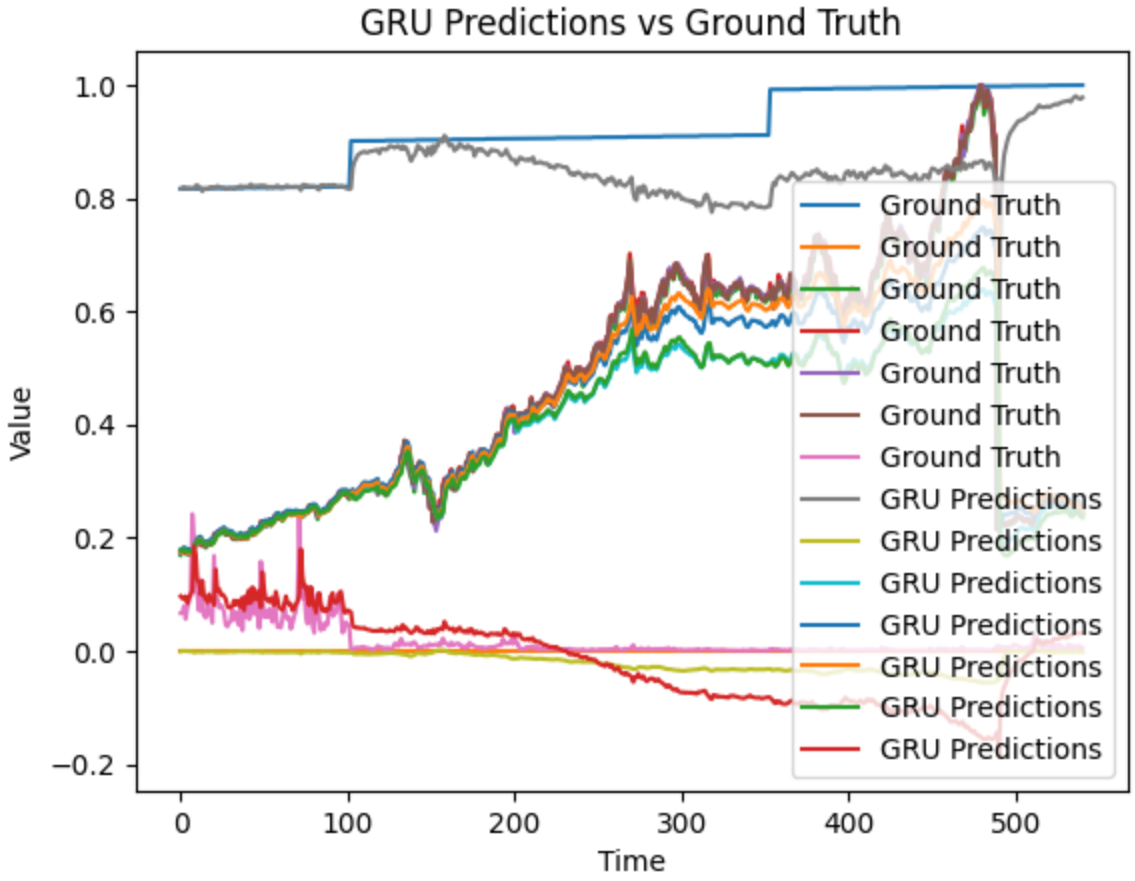
```
LSTM RMSE: 0.12328612898833713
GRU RMSE: 0.08024550593221959
```

در این قطعه کد، کد لازم را برای تجسم نتایج پیش‌بینی برای هر دو مدل LSTM و GRU اضافه شده است.
با استفاده از Matplotlib، می توانیم دو نمودار مجزا ایجاد کنیم: یکی برای پیش بینی های LSTM و دیگری برای پیش بینی های GRU. هر نمودار شامل مقادیر لیبل درستی (y_test) و همچنین پیش‌بینی‌های مربوطه خواهد بود. مقادیر محور x برای ترسیم به صورت x_axis با استفاده از تابع range() بر اساس طول y_test تعریف می‌شوند. پس از رسم داده ها، برچسب ها، عنوان و legend، نمودارها با استفاده از plt.show() نمایش داده می شوند.

```
[ ]  # Visualising the prediction results and compare LSTM and GRU models

     # Define the x-axis values for plotting
     x_axis = range(len(y_test))

     # Plot the ground truth and predictions for LSTM
     plt.plot(x_axis, y_test, label='Ground Truth')
     plt.plot(x_axis, lstm_predictions, label='LSTM Predictions')
     plt.xlabel('Time')
     plt.ylabel('Value')
     plt.title('LSTM Predictions vs Ground Truth')
     plt.legend()
     plt.show()

     # Plot the ground truth and predictions for GRU
     plt.plot(x_axis, y_test, label='Ground Truth')
     plt.plot(x_axis, gru_predictions, label='GRU Predictions')
     plt.xlabel('Time')
     plt.ylabel('Value')
     plt.title('GRU Predictions vs Ground Truth')
     plt.legend()
     plt.show()
```

نتایج:


LSTM Predictions vs Ground Truth
Value
Time
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
100
200
300
400
500
Ground Truth
Ground Truth
Ground Truth
Ground Truth
Ground Truth
Ground Truth
Ground Truth
LSTM Predictions
LSTM Predictions
LSTM Predictions
LSTM Predictions
LSTM Predictions
LSTM Predictions
LSTM Predictions


GRU Predictions vs Ground Truth
Value
Time
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
100
200
300
400
500
Ground Truth
Ground Truth
Ground Truth
Ground Truth
Ground Truth
Ground Truth
Ground Truth
GRU Predictions
GRU Predictions
GRU Predictions
GRU Predictions
GRU Predictions
GRU Predictions
GRU Predictions

قسمت دوم
(تنها قسمت‌های ToDo)

در این قطعه کد، بخش رمزگذار(encoder) ابعاد داده ها را به صورت متوالی به ترتیب زیر کاهش می دهد:

```
28*28 = 784 ==> 128 ==> 64 ==> 36 ==> 18 ==> 9
```

که تعداد گره های ورودی 784 است که به 9 گره در فضای پنهان کدگذاری می شوند. در حالی که، در بخش رمزگشا(decoder)، ابعاد داده ها به صورت خطی به اندازه ورودی اصلی افزایش می یابد تا ورودی را بازسازی کند.

```
9 ==> 18 ==> 36 ==> 64 ==> 128 ==> 784 ==> 28*28 = 784
```

که ورودی نمایش فضای نهفته 9 گره و خروجی ورودی بازسازی شده 28*28 است.

رمزگذار با گره های 28*28 در یک لایه Linear و سپس یک لایه ReLU شروع می شود و تا زمانی ادامه می یابد که ابعاد به 9 گره کاهش یابد. رمزگشا از این 9 نمایش داده برای بازگرداندن تصویر اصلی با استفاده از معکوس معماری رمزگذار استفاده می کند. معماری رمزگشا از یک لایه Sigmoid برای محدوده مقادیر فقط بین 0 و 1 استفاده می کند.

```
[ ]  # Creating a PyTorch class
     # 28*28 ==> 9 ==> 28*28
     class AE(torch.nn.Module):
         def __init__(self):
             super().__init__()

             # Building an linear encoder with Linear
             # layer followed by Relu activation function
             # 784 ==> 9
             self.encoder = torch.nn.Sequential(
                 torch.nn.Linear(28 * 28, 128),
                 torch.nn.ReLU(),
                 torch.nn.Linear(128, 64),
                 torch.nn.ReLU(),
                 torch.nn.Linear(64, 36),
                 torch.nn.ReLU(),
                 torch.nn.Linear(36, 18),
                 torch.nn.ReLU(),
                 torch.nn.Linear(18, 9)
             )
```

```
            # Building an linear decoder with Linear
            # layer followed by Relu activation function
            # The Sigmoid activation function
            # outputs the value between 0 and 1
            # 9 ==> 784
            self.decoder = torch.nn.Sequential(
                torch.nn.Linear(9, 18),
                torch.nn.ReLU(),
                torch.nn.Linear(18, 36),
                torch.nn.ReLU(),
                torch.nn.Linear(36, 64),
                torch.nn.ReLU(),
                torch.nn.Linear(64, 128),
                torch.nn.ReLU(),
                torch.nn.Linear(128, 28 * 28),
                torch.nn.Sigmoid()
            )
```

ما مدل را با استفاده از تابع میانگین مربعات خطا تأیید می کنیم و از Adam Optimizer با نرخ یادگیری 0.1 و کاهش وزن 10^(-8) استفاده می کنیم. (مدل برای استفاده از GPU باید به مود دستگاه مورد استفاده برود.)

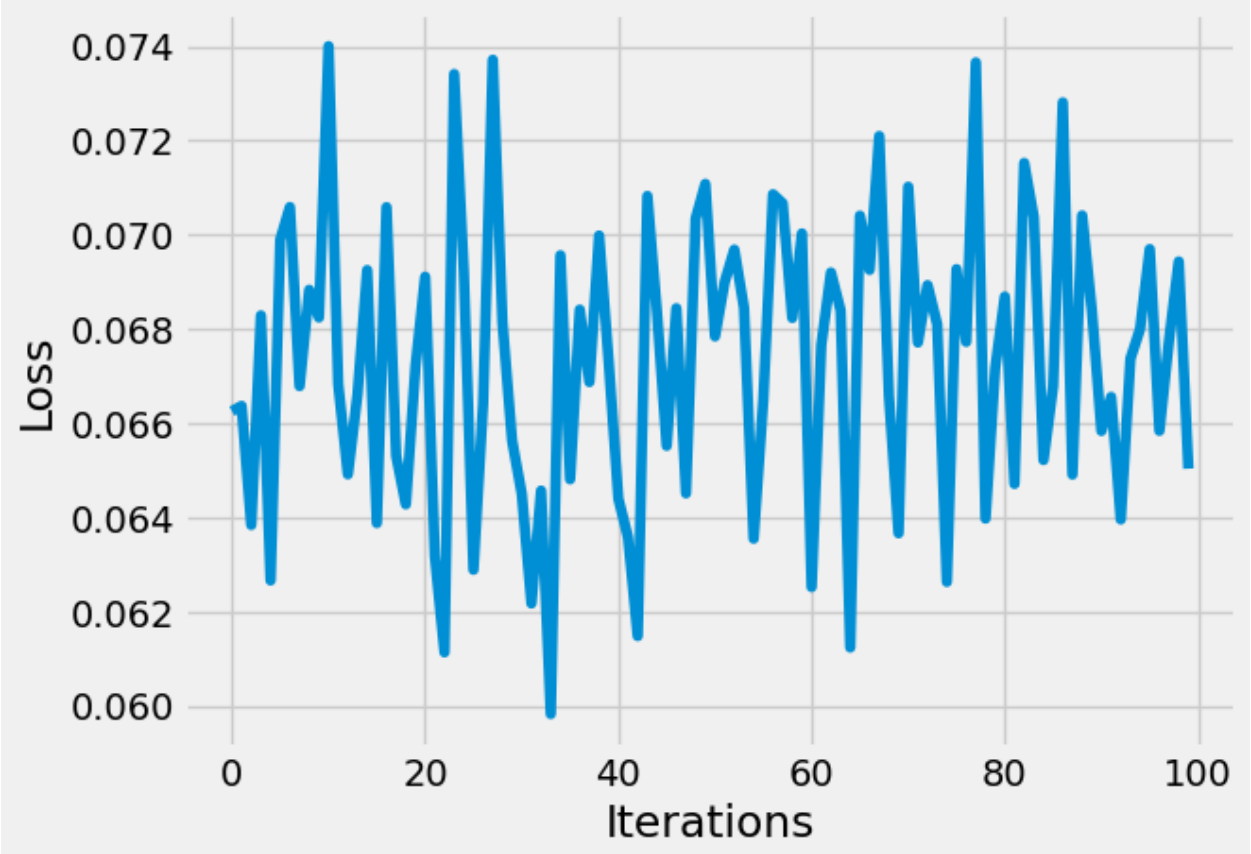
```
[ ]  # Model Initialization
     model = AE()
     model = model.to(device)

     # Validation using MSE Loss function
     loss_function = torch.nn.MSELoss()

     # Using an Adam Optimizer with lr = 0.1
     optimizer = torch.optim.Adam(model.parameters(),
                                  lr = 1e-1,
                                  weight_decay = 1e-8)
```

خروجی با ارسال به عنوان پارامتر به کلاس Model() محاسبه می شود و تانسور نهایی در یک لیست خروجی ذخیره می شود. تصویر به (-1، 784) و به عنوان یک پارامتر به کلاس Autoencoder ارسال می شود، که به نوبه خود یک تصویر بازسازی شده را برمی گرداند (تصویر نیز باید به مود دستگاه مورد استفاده باید تبدیل شود). تابع loss با استفاده از تابع MSEloss محاسبه و رسم شده است. تصویر اصلی و تصویر بازسازی شده از لیست خروجی ها جدا شده و به یک آرایه NumPy برای رسم تصاویر تبدیل می شوند.

```
[ ]  epochs = 20
     outputs = []
     losses = []
     for epoch in range(epochs):
         print(f"epoch = {epoch}")
         for (image, _) in loader:

           # Reshaping the image to (-1, 784)
           image = image.reshape(-1, 28*28).to(device)

           # Output of Autoencoder
           reconstructed = model(image)

           # Calculating the loss function
           loss = loss_function(reconstructed, image)

           # The gradients are set to zero,
           # the gradient is computed and stored.
           # .step() performs parameter update
           optimizer.zero_grad()
           loss.backward()
           optimizer.step()
```

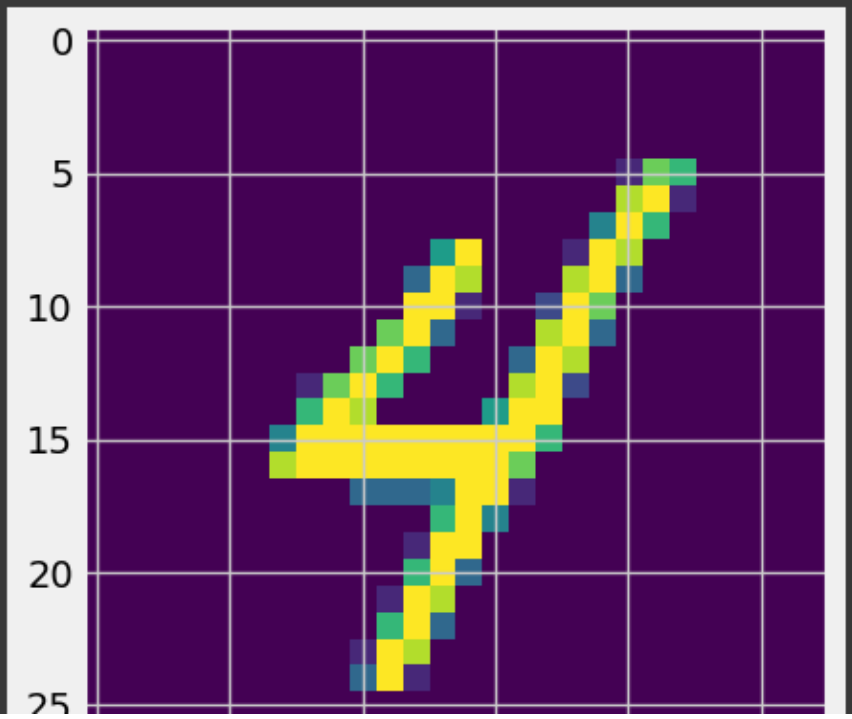
سپس نمودار loss برابری ۱۰۰ مقدار نهایی آن رسم می‌کنیم.

```
          # Storing the losses in a list for plotting
          losses.append(loss.to('cpu').detach().numpy())
        outputs.append((epochs, image, reconstructed))

    # Defining the Plot Style
    plt.style.use('fivethirtyeight')
    plt.xlabel('Iterations')
    plt.ylabel('Loss')

    # Plotting the last 100 values
    plt.plot(losses[-100:])
```



اولین آرایه تصویر ورودی و اولین آرایه تصویر ورودی بازسازی شده با استفاده از plt.imshow() رسم شده است (برای نمایش تصویر ورودی و خروجی باید تصاویر به مود cpu تبدیل شوند).

```
[ ]  for i, item in enumerate(image):
       # Reshape the array for plotting
       item = item.reshape(-1, 28, 28).to('cpu')
       plt.imshow(item[0])
```




```
for i, item in enumerate(reconstructed):
  item = item.reshape(-1, 28, 28).to('cpu').detach().numpy()
  plt.imshow(item[0])
```