Introduction to Recurrent Neural Networks (RNNs) and their Applications (in biomedical signal processing)

[Practical Part: Codes Explained]

learn_bAIome
15-17 October 2024

Fatemeh Hadaeghi
Email: f.hadaeghi@uke.de

INSTITUTE OF COMPUTATIONAL NEUROSCIENCE
UNIVERSITY MEDICAL CENTER HAMBURG
UKE HAMBURG
bAIome

Libraries Overview

PyTorch

- A deep learning library known for its dynamic computational graph and ease of use.
- **Key Features**: Autograd, neural network modules (torch.nn), and GPU support for acceleration.
- **Use in Course**: Implementing and training RNNs, GRUs, LSTMs.

NumPy

- A fundamental package for scientific computing with Python.
- **Key Features**: Arrays, linear algebra operations, random number generation.
- **Use in Course**: Efficient data manipulation, signal processing, and numerical computations.

scikit learn

- A machine learning library offering simple and efficient tools.
- **Key Features**: Preprocessing functions, model evaluation metrics, and various ML algorithms.
- **Use in Course**: Splitting data into train-validation-test sets, model evaluation, and basic ML models.

matplotlib

- A comprehensive library for creating static, animated, and interactive visualizations.
- **Key Features**: Plotting time-series data, visualizing learning curves and evaluation metrics.
- **Use in Course**: Visualizing data, signal trends, model performance.

Signal Processing Pipeline: Key Steps

Load, Clean and Preprocess Data → **Prepare Train-Validation-Test Sets** → **Design and Train the Model** → **Inference** → **Performance Evaluation**

Load, Clean and Preprocess Data

- Load raw signals
- Remove outliers, handle missing values, and smooth noisy signals.
- Use techniques like interpolation, median filters, or low-pass filters.
- Normalize, scale, or standardize signal data
- Feature extraction (e.g., extracting key frequencies from signals).

Prepare Train-Validation-Test Sets

Split the data into **training**, **validation**, and **test** sets.

Design and Train the Model

- Design models like RNNs, LSTMs, or GRU.
- Train the model using **PyTorch** or other deep learning frameworks.
- Use **backpropagation through time (BPTT)** and optimize using **Adam** or **SGD**.

Inference

- Run the model on new data (test set or real-time signals).
- Evaluate how well the model generalizes to unseen data.

Performance Evaluation

- Use metrics like **MSE**, **accuracy**, **precision**, and **recall** depending on the task.
- Plot learning curves using **Matplotlib** or evaluation reports using **Scikit-learn**.

Load, Clean and Preprocess Data

- Smoothen and Normalize your data (if necessary)
- Partition the signals into overlapping segments (if necessary)
- Apply time-delay embedding if necessary (if necessary)
- Convert the signal with any formats to Pytorch tensors

Note: Participants implementing their code in Google Colab must 1) upload the dataset to their Google Drive and 2) follow the steps below to enable access to the data.

```
from google.colab import drive
drive.mount('/content/drive')
cd drive/My\ Drive
```

```
# Imports
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import torch
from sklearn.model_selection import train_test_split
import torch.optim as optim
import matplotlib.pyplot as plt
```

Load, Clean and Preprocess Data

- Smoothen and Normalize your data (if necessary)
- Partition the signals into overlapping segments (if necessary)
- Apply time-delay embedding if necessary (if necessary)
- Convert the signal with any formats to Pytorch tensors

```
# Normalize data
data = (data - np.mean(data)) / np.std(data)
print(data.shape)

# Create sequences
def create_dataset(series, time_steps):
   X, y = [], []
   for i in range(len(series) - time_steps):
       X.append(series[i:i + time_steps])
       y.append(series[i + time_steps])
   return np.array(X), np.array(y)

# Convert to Numpy if it's a Pandas object
if isinstance(data, pd.Series) or isinstance(data, pd.DataFrame):
   data = data.values

time_steps = 50  # Number of time steps (sliding window)
X, y = create_dataset(data, time_steps)
```

Load, Clean and Preprocess Data

- Smoothen and Normalize your data (if necessary)
- Partition the signals into overlapping segments (if necessary)
- Apply time-delay embedding if necessary (if necessary)
- Convert the signal with any formats to Pytorch tensors

```
# Convert data to PyTorch tensors and reshape for RNN input (samples/batch size, time steps, features)
X = np.reshape(X, (X.shape[0], X.shape[1], 1))
X_tensor = torch.from_numpy(X).float()
y_tensor = torch.from_numpy(y).float()
```

Prepare Train-Validation-Test sets

- **Train:** to train the model
- **Validation:** to fine-tune hyperparameters
- **Test:** for final model evaluation

```
X_train, X_temp, y_train, y_temp = train_test_split(X_tensor, y_tensor, test_size=0.3,
random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,
random_state=42)

print(f"Training set: {X_train.shape}, {y_train.shape}")
print(f"Validation set: {X_val.shape}, {y_val.shape}")
print(f"Test set: {X_test.shape}, {y_test.shape}")
```

Design the Model

```
# Define the vanilla RNN model
class VanillaRNN(nn.Module):
   def __init__(self, input_size, hidden_size, output_size):
       super(VanillaRNN, self).__init__()
       self.hidden_size = hidden_size
       self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
       self.fc = nn.Linear(hidden_size, output_size)

   def forward(self, x):
       h0 = torch.zeros(1, x.size(0), self.hidden_size).to(x.device) # Initial hidden state
       out, _ = self.rnn(x, h0)
       out = self.fc(out[:, -1, :]) # Take the output from the last time step
       return out
```

Design the Model

```python
# Define the LSTM model
class LSTMModel(nn.Module):
   def __init__(self, input_size, hidden_size, output_size):
       super(LSTMModel, self).__init__()
       self.hidden_size = hidden_size
       self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
       self.fc = nn.Linear(hidden_size, output_size)

   def forward(self, x):
       h0 = torch.zeros(1, x.size(0), self.hidden_size).to(x.device)  # Initial hidden state
       c0 = torch.zeros(1, x.size(0), self.hidden_size).to(x.device)  # Initial cell state
       out, _ = self.lstm(x, (h0, c0))  # LSTM forward
       out = self.fc(out[:, -1, :])  # Fully connected layer applied to the last time step
       return out
```

Design the Model

```
# Define the GRU model
class GRUModel(nn.Module):
 def __init__(self, input_size, hidden_size, output_size):
     super(GRUModel, self).__init__()
     self.hidden_size = hidden_size
     self.gru = nn.GRU(input_size, hidden_size, batch_first=True)
     self.fc = nn.Linear(hidden_size, output_size)

 def forward(self, x):
     h0 = torch.zeros(1, x.size(0), self.hidden_size).to(x.device)  # Initial hidden state
     out, _ = self.gru(x, h0)  # GRU forward
     out = self.fc(out[:, -1, :])  # Fully connected layer applied to the last time step
     return out
```

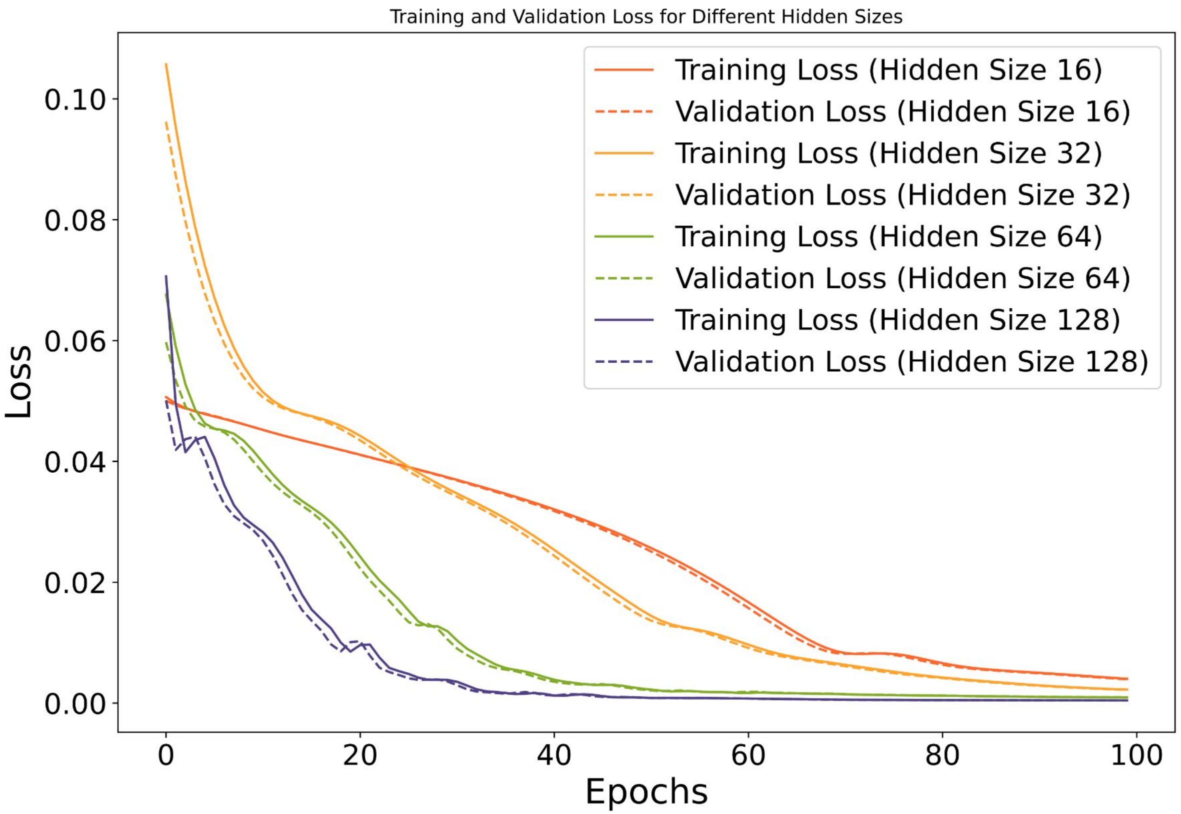
Train the Model, part 1

```
# Set initial parameters

input_size = 1
output_size = 1
hidden_sizes = [16, 32, 64, 128] # Hidden layer sizes to tune

# Initialize a dictionary to store the losses for each hidden layer size
losses_per_hidden_size = {}

# Set initial parameters
best_model = None
best_val_loss = float('inf')
hidden_sizes = [16, 32, 64, 128] # Hidden layer sizes to tune

# Loop over different hidden sizes
for hidden_size in hidden_sizes:
   model = VanillaRNN(input_size, hidden_size, output_size)
   criterion = nn.MSELoss()
   optimizer = optim.Adam(model.parameters(), lr=0.001)

   # Initialize lists for storing epoch losses
   train_loss_epoch = []
   val_loss_epoch = []

   # Training loop
   epochs = 100
   for epoch in range(epochs):
       model.train()
       optimizer.zero_grad()
       output = model(X_train)
       loss = criterion(output, y_train)
       loss.backward()
       optimizer.step()
```

Train the Model, part 2 (see Notebook)

```
        # Append training loss for this epoch
        train_loss_epoch.append(loss.item())

        # Validation
        model.eval()
        val_output = model(X_val)
        val_loss = criterion(val_output, y_val)
        val_loss_epoch.append(val_loss.item())

        # Save the best model
        if val_loss.item() < best_val_loss:
            best_val_loss = val_loss.item()
            best_model = model  # Save the best model

        if epoch % 10 == 0:
            print(f"Epoch {epoch+1}/{epochs}, Hidden Size: {hidden_size}, Training
Loss: {loss.item()}, Validation Loss: {val_loss.item()}")

   # Store the training and validation losses for this hidden size
   losses_per_hidden_size[hidden_size] = (train_loss_epoch, val_loss_epoch)

# Plot training and validation loss for each hidden size
my_color = ['#FF662A', '#FFA22A', '#82AC26', '#4F3F84']
plt.figure(figsize=(12, 8))
for i, hidden_size in enumerate(hidden_sizes):
   train_loss, val_loss = losses_per_hidden_size[hidden_size]
   plt.plot(train_loss, color=my_color[i], label=f'Training Loss (Hidden Size
{hidden_size})', linestyle='solid')
   plt.plot(val_loss, color=my_color[i], label=f'Validation Loss (Hidden Size
{hidden_size})', linestyle='dashed')

plt.title('Training and Validation Loss for Different Hidden Sizes')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

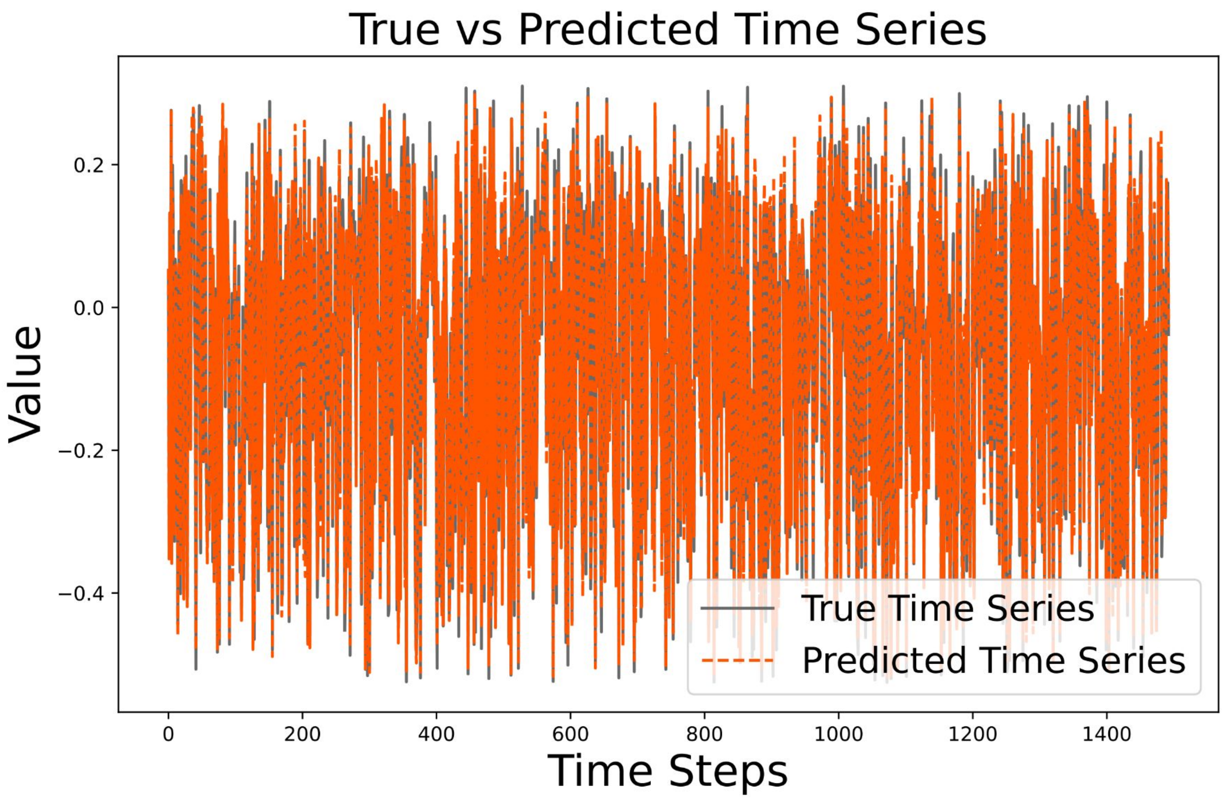
Train the Model



Inference and Performance Evaluation

```
best_model.eval()
test_output = best_model(X_test)
test_loss = criterion(test_output, y_test)

print(f"Test Loss: {test_loss.item()}")

# Convert tensors to numpy arrays for plotting
y_test_np = y_test.detach().numpy()
test_output_np = test_output.detach().numpy()

# Plot the true vs predicted time series
save_path = '/content/drive/My Drive/predicted_vs_true_timeseries.pdf'
plt.figure(figsize=(10, 6))
plt.plot(y_test_np, label='True Time Series', color='#696969')
plt.plot(test_output_np, label='Predicted Time Series', color='#FF5500', linestyle='dashed')
plt.title('True vs Predicted Time Series')
plt.xlabel('Time Steps')
plt.ylabel('Value')
plt.legend()

# Save the figure as a PDF file
plt.savefig(save_path, format='pdf')

print(f"Figure saved as PDF to: {save_path}")
```

Inference and Performance Evaluation



Assignments

- Conduct a comparative analysis of the computation time for three distinct models: vanilla RNN, LSTM, and GRU
- Gradually increase size of data segments and perform the task
- Implement Xavier initialization
- For long time series implement gradient clipping