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

[Practical Part: Supplementary Tasks]

learn_bAIome
15-17 October 2024

Fatemeh Hadaeghi
Email: f.hadaeghi@uke.de

INSTITUTE OF COMPUTATIONAL NEUROSCIENCE
UNIVERSITY MEDICAL CENTER HAMBURG

UKE HAMBURG
bAIome

Supplementary Tasks Overview

i) Hyperparameters Fine-Tuning

- **Learning rate:** [0.001, 0.005, 0.01, 0.05]
- **Non-linearity:** ['tanh', 'relu']
- **Optimizer:** [Adam, SGD]

ii) Gradually Changing the Sequence Length

- Time-setps = 100, 200, 300

iii) Initialization

- Xavier initialization for input-to-hidden weights (W_ih) and hidden-to-hidden weights (W_hh) in Vanilla RNN

iv) Visualization

- **Neuron Activity**: Visualize the activity of neurons at the end of each epoch
- **Hidden Layer Weights**: Visualize the weights in the hidden layer

v) Reservoir Computing

- For **time-series prediction**, employ **ESNGenerator** using echoes

Helpers

```
class VanillaRNNWithXavierInit(nn.Module):
  def __init__(self, input_size, hidden_size, output_size):
     super(VanillaRNNWithXavierInit, 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)

     # Apply Xavier initialization to the RNN weights
     nn.init.xavier_uniform_(self.rnn.weight_ih_l0)  # Input to hidden layer weights
     nn.init.xavier_uniform_(self.rnn.weight_hh_l0)  # Hidden to hidden layer weights

  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
```

Helpers

```
def plot_neuron_activity(hidden_states, epoch, num_neurons=10):

    # Flatten the list if hidden_states is a list of lists
    if isinstance(hidden_states[0], list):
        print("Detected list of lists. Flattening hidden_states...")
        hidden_states = [hidden for epoch_states in hidden_states for hidden in epoch_states]

    # Check that hidden_states contains tensors
    assert isinstance(hidden_states[0], torch.Tensor), "Hidden states must be stored as tensors."

    # Number of neurons is determined by the hidden size (third dimension)
    num_neurons = min(num_neurons, hidden_states[0].shape[2])

    # Randomly select neurons to visualize
    selected_neurons = random.sample(range(hidden_states[0].shape[2]), num_neurons)
    plt.figure(figsize=(10, 6))
    # Plot neuron activity for the selected neurons
    for neuron in selected_neurons:
        neuron_activity = [hidden[0, 0, neuron].item() for hidden in hidden_states]  # First layer, first sample
        plt.plot(range(len(hidden_states)), neuron_activity, label=f'Neuron {neuron}')
```

Helpers

```
def save_hidden_states(model, input_sample, hidden_states):
  with torch.no_grad():
     model.eval()
     out, hidden = model.rnn(input_sample)  # Get hidden states from RNN
     hidden_states.append(hidden.detach().cpu())

def save_weights(model, weights_history):
  with torch.no_grad():
     model.eval()
     weights = model.rnn.weight_hh_l0  # Assuming one hidden layer RNN
     weights_history.append(weights.clone())
```

Helpers

```
def save_hidden_states(model, input_sample, hidden_states):
  with torch.no_grad():
     model.eval()
     out, hidden = model.rnn(input_sample)  # Get hidden states from RNN
     hidden_states.append(hidden.detach().cpu())

def save_weights(model, weights_history):
  with torch.no_grad():
     model.eval()
     weights = model.rnn.weight_hh_l0  # Assuming one hidden layer RNN
     weights_history.append(weights.clone())
```

Helpers

```
def plot_weight_evolution_animation(weights_history, save_as='gif'):

  # Convert list of tensors to numpy array for visualization
  weight_matrices = [weight.numpy() for weight in weights_history]
  # Create a figure
  fig, ax = plt.subplots(figsize=(12, 8))
  # Initialize a heatmap
  heatmap = sns.heatmap(weight_matrices[0], cmap="flare", cbar=False, ax=ax)
  plt.title('Weight Evolution at Epoch 1')
  plt.xlabel('Neurons')
  plt.ylabel('Neurons')

def update(frame):
    ax.clear()  # Clear the previous heatmap
    sns.heatmap(weight_matrices[frame], cmap="flare", cbar=(frame == len(weight_matrices) - 1), ax=ax)
    ax.set_title(f'Weight Evolution at Epoch {frame + 1}')
    ax.set_xlabel('Neurons')
    ax.set_ylabel('Neurons')
  # Create animation
  ani = FuncAnimation(fig, update, frames=len(weight_matrices), repeat=False)
  # Save the animation
  if save_as == 'gif':
    save_path = '/content/drive/My Drive/weight_evolution.gif'
    ani.save(save_path, writer='pillow', fps=2)
  elif save_as == 'video':
    save_path = '/content/drive/My Drive/weight_evolution.mp4'
    ani.save(save_path, writer='ffmpeg', fps=2)

  plt.close(fig)  # Close the figure window
```

Helpers

```
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)
      self.hidden_states = []  # Store hidden states

  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, (h_n, c_n) = self.lstm(x, (h0, c0))  # LSTM forward

      # Store only the hidden state (h_n) as a tensor
      self.hidden_states.append(h_n.detach().cpu().clone())

      out = self.fc(out[:, -1, :])  # Fully connected layer applied to the last time step
      return out

  def reset_hidden_states(self):
      """Reset hidden states after each epoch to avoid memory overflow."""
      self.hidden_states = []
```

Helpers

```
import numpy as np
from matplotlib import pyplot as plt
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split

from echoes import ESNGenerator
from echoes.datasets import load_mackeyglasst17
from echoes.plotting import plot_predicted_ts,
set_mystyle

set_mystyle()  # optional, set aesthetics


# Load and split data
mackey_ts = load_mackeyglasst17()
n_train_steps, n_test_steps = 2000, 2000
n_total_steps = n_train_steps + n_test_steps

y_train, y_test = train_test_split(
    mackey_ts,
    train_size=n_train_steps,
    test_size=n_test_steps,
    shuffle=False
)

esn = ESNGenerator(
    n_steps=n_test_steps,
    n_reservoir=200,
    spectral_radius=1.25,
    leak_rate=.4,
    random_state=42,
)

# Fit the model. Inputs is None because we only have
the target time series
esn.fit(X=None, y=y_train)

y_pred = esn.predict()
print("test r2 score", r2_score(y_test, y_pred))
```

```
# Plot training and test
plt.figure(figsize=(22, 5))
plt.plot(mackey_ts[: n_total_steps], 'steelblue', linewidth=5, label="target
system")
plt.plot(esn.training_prediction_, color="y", linewidth=1, label="training fit")
plt.plot(range(n_train_steps, n_total_steps), y_pred,'orange',
label="ESNGenerator")
plt.ylabel("oscillator value")
plt.xlabel('time')
lo, hi = plt.ylim()
plt.vlines(n_train_steps, lo-.05, hi+.05, linestyles='--')
plt.legend(fontsize='small')

# Plot test alone
plt.figure(figsize=(22, 5))
plt.subplot(1, 4, (1, 3))
plt.title("zoom into test")
plt.plot(y_test,
        color="steelblue",
        label="target system",
        linewidth=5.5)
plt.xlabel('time')

plt.plot(y_pred,
        linestyle='--',
        color="orange",
        linewidth=2,
        label="generative ESN",)
plt.ylabel("oscillator")
plt.xlabel('time')
plt.legend(fontsize='small')

plt.subplot(1, 4, 4)
plt.title(r"$W^{out}$ weights distribution")
plt.xlabel('weight')
plt.ylabel('frequency')
plt.hist(esn.W_out_.flat)
plt.tight_layout();
```