hw5-hopfield

March 12, 2023

[8]:
```python
import numpy as np
```

Training (Constructing Weights)

[9]:
```python
states = np.array([[+1, -1, +1, -1, +1], [-1, +1, +1, -1, +1]])

# Number of states
N = len(states[0])
M = len(states)
# Construct the weight matrix
W =  np.zeros((N, N))

for i in range(N):
    for j in range(N):
        if i != j:
            for s in range(M):
                W[i, j] += states[s, i] * states[s, j]
            W[i, j] /= N
W_diag = np.zeros((N, N))
np.fill_diagonal(W_diag, 0)
W = W + W_diag

np.fill_diagonal(W, 0)
```

Probing Pattern

[10]:
```python
initial_input_state = np.array([+1, +1, +1, +1, +1])
```

Dynamic Evolution

[11]:
```python
# Hard-limiting non-linearity function
f_h = np.vectorize(lambda x: 1 if x >= 0 else -1)

# Dynamic evolution
def evolve(order, log=False):
    t = 0
    last_U = None
    U = initial_input_state.copy()
```

```
    if log:
        print('U({}) = {}'.format(t, U))

    # Until convergence
    while not np.array_equal(U, last_U):
        t += 1
        last_U = U.copy()
        # Update the nodes in order
        for i in order:
            # Be sure to zero-index

            # YOUR CODE HERE
            i -= 1
            U_i = 0
            for j in range(N):
                if i != j:
                    U_i += W[i, j] * U[j]
            U_i = f_h(U_i)
            U[i] = U_i

            if log:
                print('U_{}({}) = {}'.format(i, t, U[i-1]))

        # Log U for this iteration
        if log:
            print('U({}) = {}'.format(t, U))
            None

    return U
```

[12]:
```
U = evolve([3, 1, 5, 2, 4], log=True)
print(U)
```

```
U(0) = [1 1 1 1 1]
U_2(1) = 1
U_0(1) = 1
U_4(1) = 1
U_1(1) = -1
U_3(1) = 1
U(1) = [-1  1  1 -1  1]
U_2(2) = 1
U_0(2) = 1
U_4(2) = -1
U_1(2) = -1
U_3(2) = 1
U(2) = [-1  1  1 -1  1]
[-1  1  1 -1  1]
```

[13]:

```
U = evolve([2, 4, 3, 5, 1], log=True)
print(U)
```

```
U(0) = [1 1 1 1 1]
U_1(1) = 1
U_3(1) = 1
U_2(1) = -1
U_4(1) = -1
U_0(1) = 1
U(1) = [ 1 -1  1 -1  1]
U_1(2) = 1
U_3(2) = 1
U_2(2) = -1
U_4(2) = -1
U_0(2) = 1
U(2) = [ 1 -1  1 -1  1]
[ 1 -1  1 -1  1]
```

[ ]:

```
```

hw5-periodicblankkk

March 12, 2023

[1]:
```
import string
import random
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import numpy as np
```

Prepare for Dataset

[2]:
```
# Get a random sequence of sine curve.
def get_random_seq():
    seq_len     = 128  # The length of an input sequence.
    # Sample a sequence.
    t = np.arange(0, seq_len)
    a = 2*np.pi*1.0/seq_len
    b = 2*np.pi*np.random.rand()*5
    seq = np.sin(a*t+b)
    return seq

# Sample a mini-batch including input tensor and target tensor.
def get_input_and_target():
    seq    = get_random_seq()
    input  = torch.tensor(seq[:-1]).float().view(-1,1,1) # Input sequence.
    target = torch.tensor(seq[1:]).float().view(-1,1,1)  # Target sequence.
    return input, target
```

Choose a Device

[3]:
```
# If there are GPUs, choose the first one for computing. Otherwise use CPU.
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
# If 'cuda:0' is printed, it means GPU is available.
```

```
cpu
```

Network Definition

[9]:
```
class Net(nn.Module):
    def __init__(self):
```

```
        # Initialization.
        super(Net, self).__init__()
        self.input_size  = 1
        self.hidden_size = 100
        self.output_size = 1
        self.rnn = nn.RNNCell(self.input_size, self.hidden_size)
        self.fc = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input, hidden):
        """ Forward function.
              input:  Input. It refers to the x_t in homework write-up.
              hidden: Previous hidden state. It refers to the h_{t-1}.
            Returns (output, hidden) where output refers to y_t and
                     hidden refers to h_t.
        """
        # Forward function.
        hidden = self.rnn(input, hidden)
        output = self.fc(hidden)
        return output, hidden###### To be filled ######
        ###### To be filled ######

        return ###### To be filled ######

    def init_hidden(self):
        # Initial hidden state.
        # 1 means batch size = 1.
        return torch.zeros(1, self.hidden_size).to(device)

net = Net()      # Create the network instance.
net.to(device)  # Move the network parameters to the specified device.
```

[9]:
```
Net(
  (rnn): RNNCell(1, 100)
  (fc): Linear(in_features=100, out_features=1, bias=True)
)
```

Training Step and Evaluation Step

[14]:
```
# Training step function.
def train_step(net, opt, input, target):
    """ Training step.
        net:    The network instance.
        opt:    The optimizer instance.
        input:  Input tensor.  Shape: [seq_len, 1, 1].
        target: Target tensor. Shape: [seq_len, 1].
    """
    seq_len = input.shape[0]    # Get the sequence length of current input.
```

```
    hidden = net.init_hidden()  # Initial hidden state.
    net.zero_grad()             # Clear the gradient.
    loss = 0                    # Initial loss.

    for t in range(seq_len):    # For each one in the input sequence.
        output, hidden = net(input[t], hidden)
        loss += loss_func(output, target[t])

    loss.backward()             # Backward.
    opt.step()                  # Update the weights.

    return loss.item() / seq_len       # Return the average loss w.r.t sequence␣
 ↪length.
```

[15]:
```
# Evaluation step function.
def eval_step(net, predicted_len=100):
    # Initialize the hidden state, input and the predicted sequence.
    hidden        = net.init_hidden()
    init_seq      = get_random_seq()
    init_input    = torch.tensor(init_seq).float().view(-1,1,1).to(device)
    predicted_seq = []

    # Use initial points on the curve to "build up" hidden state.
    for t in range(len(init_seq) - 1):
        output, hidden = net(init_input[t], hidden)

    # Set current input as the last character of the initial string.
    input = init_input[-1]

    # Predict more points after the initial string.
    for t in range(predicted_len):
        # Get the current output and hidden state.
        output, hidden = net(input, hidden)

        # Add predicted point to the sequence and use it as next input.
        predicted_seq.append(output.item())

        # Use the predicted point to generate the input of next round.
        input = output

    return init_seq, predicted_seq
```

Training Procedure

[16]:
```
# Number of iterations.
iters       = 200    # Number of training iterations.
print_iters = 10     # Number of iterations for each log printing.
```

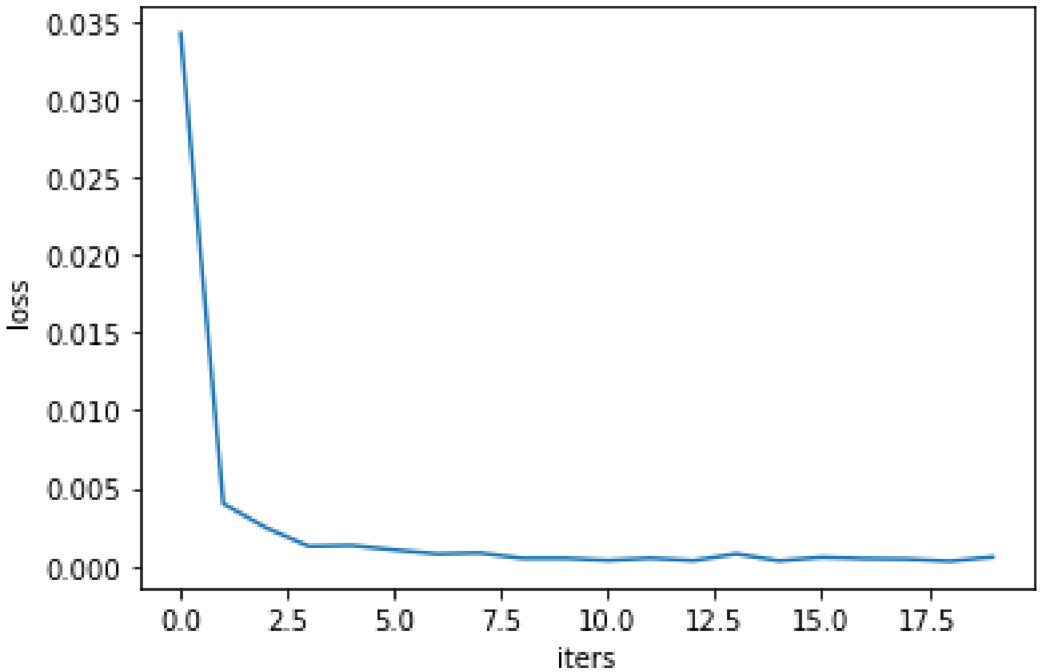
```python
# The loss variables.
all_losses = []
loss_sum   = 0

# Initialize the optimizer and the loss function.
opt       = torch.optim.Adam(net.parameters(), lr=0.005)
loss_func = nn.MSELoss()

# Training procedure.
for i in range(iters):
    input, target = get_input_and_target()            # Fetch input and target.
    input, target = input.to(device), target.to(device) # Move to GPU memory.
    loss      = train_step(net, opt, input, target)   # Calculate the loss.
    loss_sum += loss                                  # Accumulate the loss.

    # Print the log.
    if i % print_iters == print_iters - 1:
        print('iter:{}/{} loss:{}'.format(i, iters, loss_sum / print_iters))
        #print('generated sequence: {}\n'.format(eval_step(net)))

        # Track the loss.
        all_losses.append(loss_sum / print_iters)
        loss_sum = 0
```

```
iter:9/200 loss:0.034246294000956023
iter:19/200 loss:0.004006280462572894
iter:29/200 loss:0.002469993318159749
iter:39/200 loss:0.0012767110459917176
iter:49/200 loss:0.0013323496763161789
iter:59/200 loss:0.0010404377178413662
iter:69/200 loss:0.0007833188033010078
iter:79/200 loss:0.0008453899334500154
iter:89/200 loss:0.0004985172814858241
iter:99/200 loss:0.000501039854478179
iter:109/200 loss:0.00036802953915802516
iter:119/200 loss:0.0005075956071455646
iter:129/200 loss:0.0003546518229652107
iter:139/200 loss:0.000789939034290201
iter:149/200 loss:0.00034005130516497165
iter:159/200 loss:0.0005675784422187354
iter:169/200 loss:0.000480396749349091
iter:179/200 loss:0.0004481071840942375
iter:189/200 loss:0.0003149818130365506
iter:199/200 loss:0.0005868568532462195
```

Training Loss Curve

[17]:

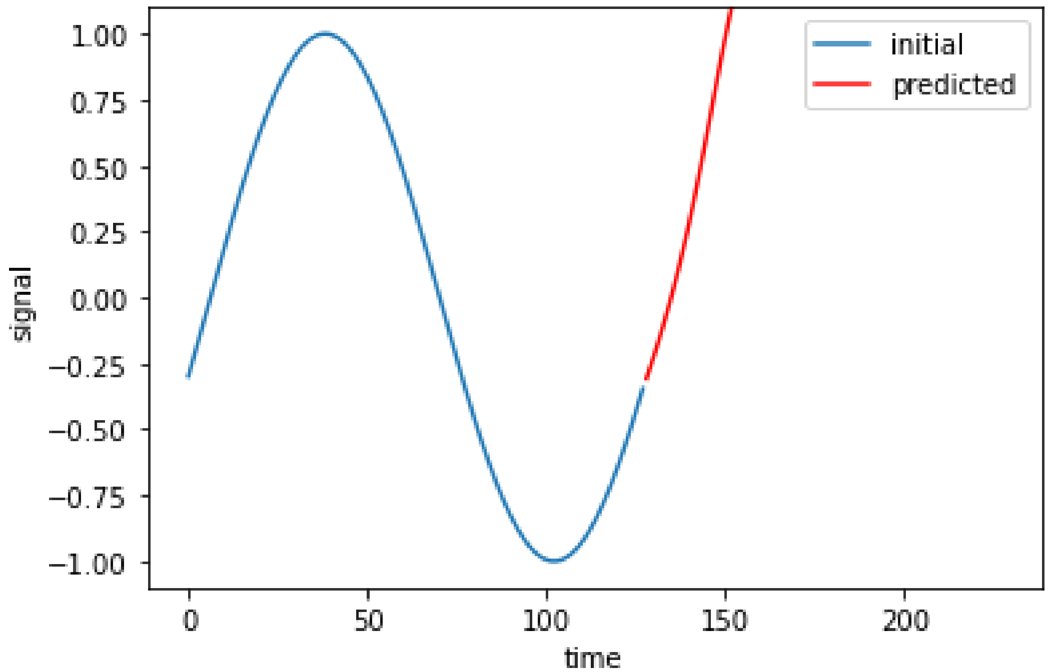
```
plt.xlabel('iters')
plt.ylabel('loss')
plt.plot(all_losses)
plt.show()
```



Evaluation: A Sample of Generated Sequence

[19]:

```
init_seq, predicted_seq = eval_step(net, predicted_len=100)
init_t      = np.arange(0, len(init_seq))
predicted_t = np.arange(len(init_seq), len(init_seq)+len(predicted_seq))
plt.plot(init_t, init_seq, label='initial')
plt.plot(predicted_t, predicted_seq, color='red', label='predicted')
plt.legend()
plt.ylim([-1.1, 1.1])
plt.xlabel('time')
plt.ylabel('signal')
plt.show()
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
signal
0
50
100
150
200
time
initial
predicted

[ ]: