CSCI-B555: Machine Learning **(Due: 05/02/20)**

Submission Assignment #3

Instructor: Zoran Tiganj *Name:* Shufan Lu, *Email:* shuflu@iu.edu

Claim: This work has been finished under the discussion with my study teammate, Xiaolei Lu. As references, I mainly used the course slides. For the codes, I use the the notebook file provided in the lecture as the template.

Question 1

(a) In feedforward network, there is no looping in the network and no connections between the hidden layers. The information only flows in the forward direction from the input layer, through the hidden layer and to the output layer.

Recurrent network has a feedback loop where data can be fed back into the input at some point before it is fed forward again for further processing and final output. And there is a matrix W that connects the previous hidden layers to current ones.

(b) For convolutional neural network, it calculates the convolution between nodes in a window in the featured map and a kernel with the window sliding over an entire image. The idea of parameter sharing manifests in the application of the same convolution kernel at each time step.

Recurrent network is focusing on time series of data. Each member of the output is a function of the previous members of the output.

(c) When W (the hidden layer to hidden layer transformation matrix) is not very close to 1, gradients propagated over many stages tend to either vanish or explode, making it hard to learn log-term dependencies.

LSTM network would help with it. Because in each LSTM cell, there are some more parameters and a system of gating units that controls the flow of information. For LSTM, the time scale of integration can change based on the input sequence, because the time constants are output by the model itself.

(d) The Python code is listed here.

```
1 import numpy as np
2 #import pylab as pl
3 import matplotlib.pyplot as pl
4 import random
5
```

```
xmin=0
xmax=6.28
num_points=100

noise=np.random.normal(0,0.1,num_points)
sinwave=np.sin(np.linspace(xmin,xmax,num_points))

data=sinwave+noise

#introduce a gaussian noise to the standard sine wave to make a
    noisy input
#which is more close to real case

pl.plot(sinwave,label='standard sinwave')
pl.plot(data,label='noisy sinwave')
pl.legend()
pl.show()
```

Listing 1: Data Preparation

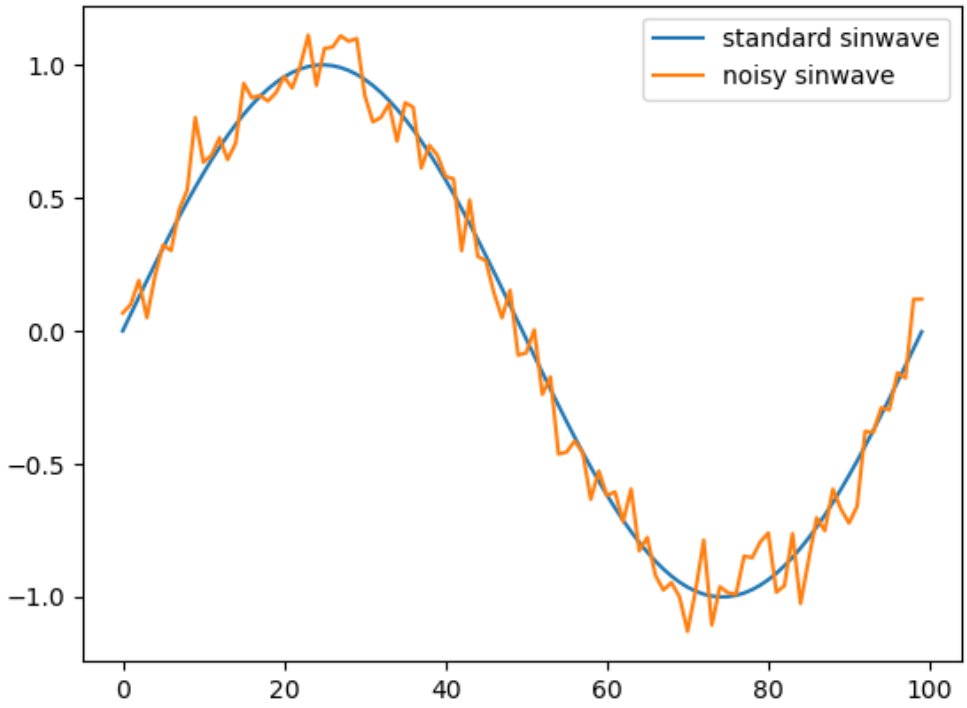


Figure 1: After data preparation, we get 100 training samples, that is in a full cycle of a noisy sine wave.

```
def sigmoid(s):
    return 1/(1+np.exp(-s))
def compute_error(pred,real):
    loss=0.5*(pred-real)**2
    return loss
class simpleRNN:
    def __init__(self):
        self.neurons=128
        ip_dim=1
        op_dim=1
        self.lr=0.01

        self.U=np.ones([ip_dim,self.neurons])
        self.W=np.random.randn(self.neurons,self.neurons)
        self.V=np.random.randn(self.neurons,op_dim)
        self.h=np.zeros((1,self.neurons))

    def feedforward(self,x):
        for i in range(x.shape[0]):
            self.h = sigmoid(np.dot(self.h,self.W)+np.dot(self.U,x[
    i]))
        self.yhat = np.dot(self.h,self.V)

    def backprop(self,y):
        self.dv=-np.dot((y-self.yhat),self.h)
        self.V=self.V-np.dot(self.lr,self.dv.transpose())
        self.h = np.zeros((1, self.neurons))

    def compute_loss(self,y):
        loss=compute_error(self.yhat,y)
        return loss

model=simpleRNN()

#above codes is to define all the necessary variables and functions
    in the RNN model

iterations=20000
bptt_steps=10
loss=np.zeros((iterations,1))
for i in range(iterations):
    start_time=random.randint(0,num_points-bptt_steps-2)
    model.feedforward(data[start_time:start_time+bptt_steps])
    model.backprop(data[start_time+bptt_steps+1])
    loss[i]=model.compute_loss(data[start_time+bptt_steps+1])
pl.plot(loss)
pl.xlabel('Iteration')
pl.ylabel('Error')
pl.show()

```

```
#calculate errors and showed how the error reduces with the
    iteration grows
```

Listing 2: Build the main structure for the RNN, calculate errors and show how the error reduces with the iteration grows.

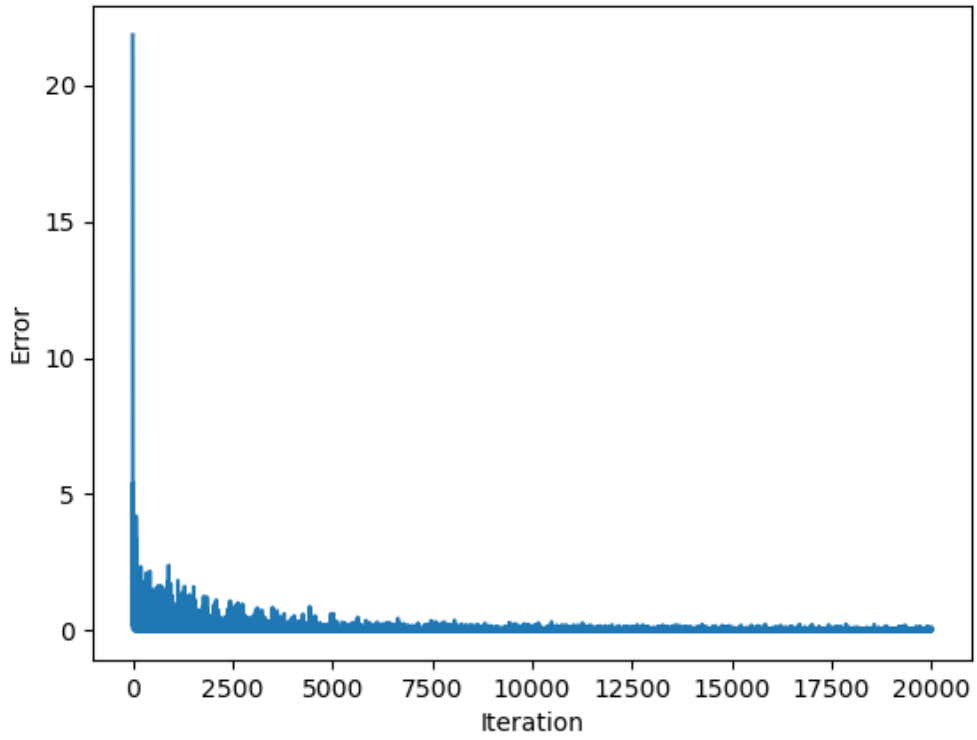
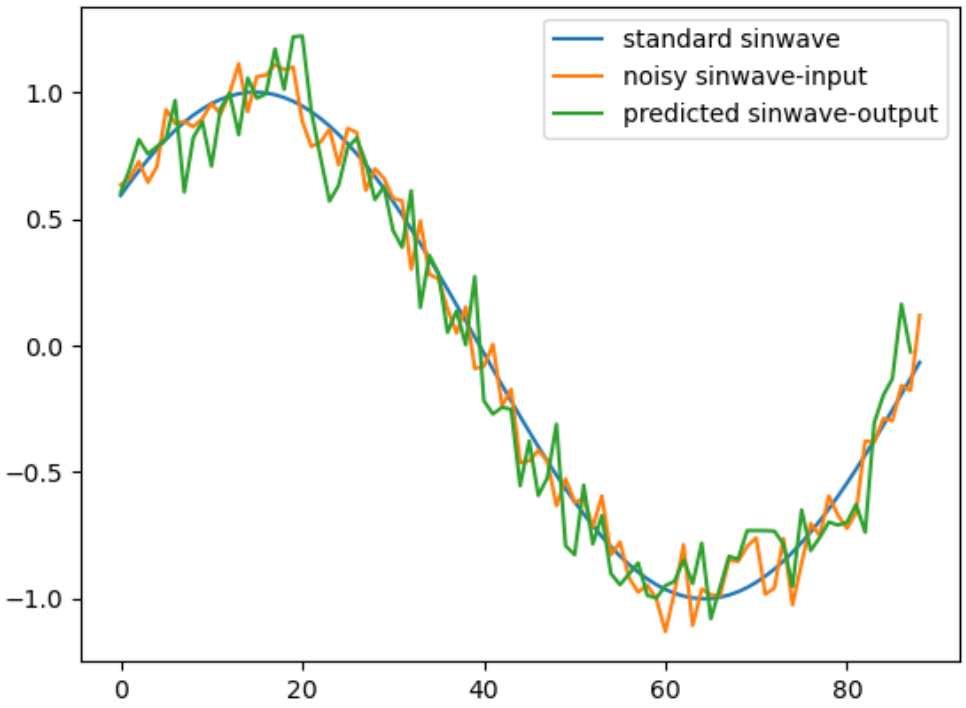


Figure 2: The relation of error and the iteration. We can see with the iteration time increase, the error is reduced significantly.

```
y=np.zeros((len(data)-bptt_steps-2,1))
for start_time in range(len(data)-bptt_steps-2):
    model.feedforward(data[start_time:start_time+bptt_steps])
    model.h = np.zeros((1, model.neurons))
    y[start_time] = model.yhat

#do the RNN prediction

pl.plot(sinwave[bptt_steps:len(data)-1],label='standard sinwave')
pl.plot(data[bptt_steps:len(data)-1],label='noisy sinwave-input')
pl.plot(y,label='predicted sinwave-output')
pl.legend()
```

```
pl.show()

#compare the prediction result with the ideal sine wave and the
    noisy input
```

Listing 3: Get the prediction result.



o

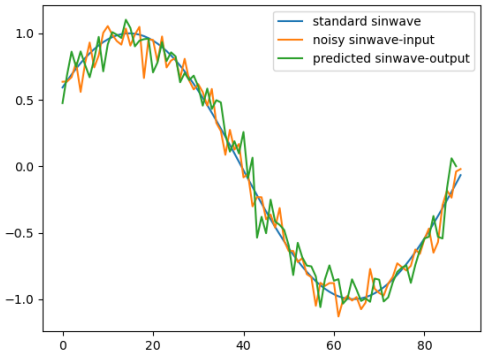
Figure 3: Compare the prediction result with the ideal sine wave and the noisy input. Here learning rate=0.01

From Fig.3 we can see it's quite a good fit.

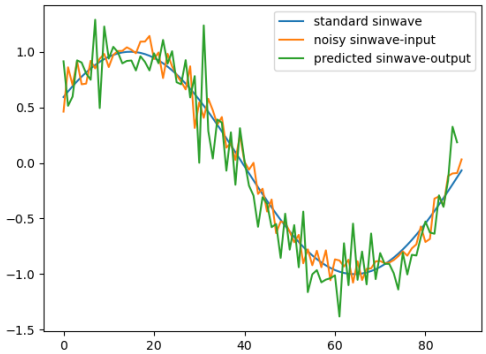
Fig.4 showed the prediction changes when adjusting learning rates to 0.02 and 0.005.

Fig.5 showed the prediction changes when adjusting the number of neurons in hidden layer to 50 and 240.

If we apply the back-propagation through time to not only V, but also U and W, the result can be improved.

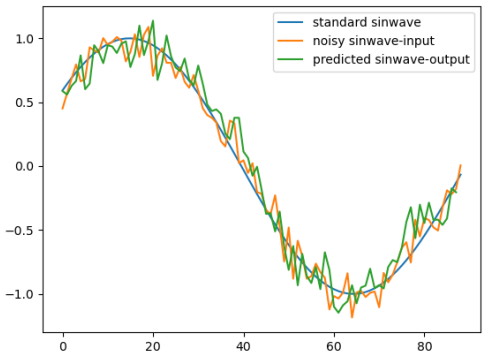


(a) The prediction with learning rate=0.02.

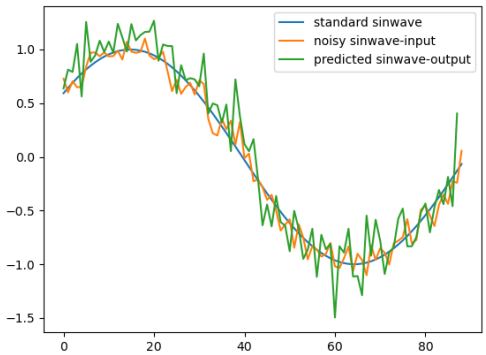


(b) The prediction with learning rate=0.005.

Figure 4: By changing the learning rates, the prediction result changes.



(a) The prediction with 50 neutrons in hidden layer.



(b) The prediction with 240 neutrons in hidden layer.

Figure 5: By changing the learning rates, the prediction result changes.