Exercise 9

January 8, 2020

1 Task 1 - Regularization

1.0.1 In the file noisy sinus.pt you will find a sine wave overlayed with Gaussian noise. The goal is to capture the sine wave without the noise in our model:

$$z_t = tanh(W_{xz}x_{t-1} + W_{zz}z_{t-1} + b_z)$$

$$x_t = tanh(W_{zx}z_t + b_x)$$

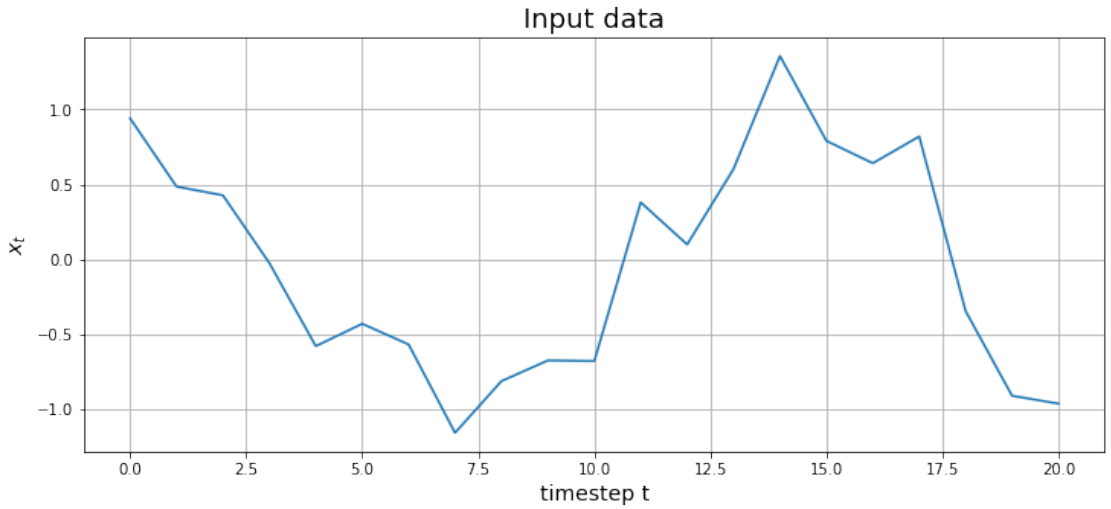
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp
from sklearn.linear_model import LinearRegression

import torch as tc

from rnn_template import rnn
```

In [2]:
```
# show input data
data = tc.load('noisy_sinus.pt')

fig1 = plt.subplots(figsize=[12,5])
plt.plot(data)
plt.xlabel('timestep t', fontsize = 14)
plt.ylabel(r'$x_t$', fontsize = 14)
plt.title('Input data', fontsize = 18)
plt.grid()
plt.show()
```

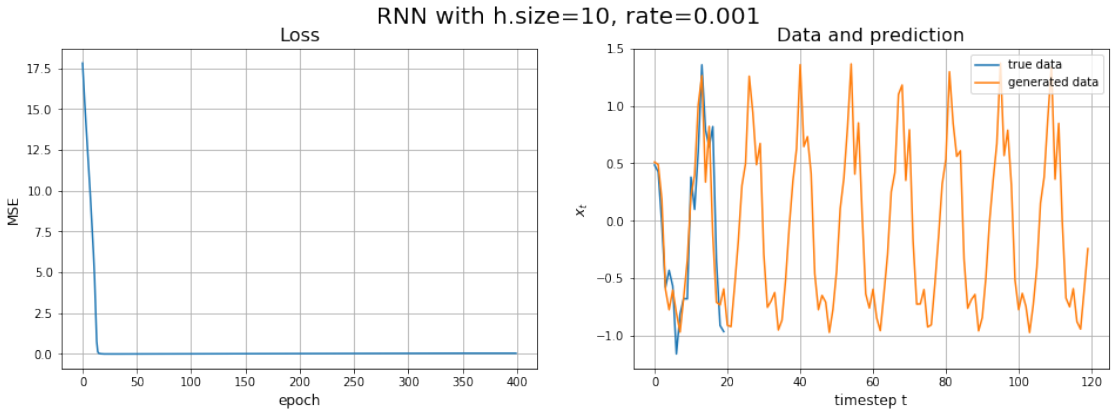


1.0.2 When optimizing the parameters of this network w.r.t. a mean squared error (MSE), why is this implicitly assuming Gaussian noise added to the outputs x_t?

Because by assuming gaussian-distributed errors, the minimization of MSE is equivalent to the maximization of the log-likelihood.

1.0.3 Train the network with 10 hidden states for 400 epochs. Plot the resulting predictions.

```
In [3]: rnn(lr = 0.001)
```



1.0.4 How does the overfitting manifest itself?

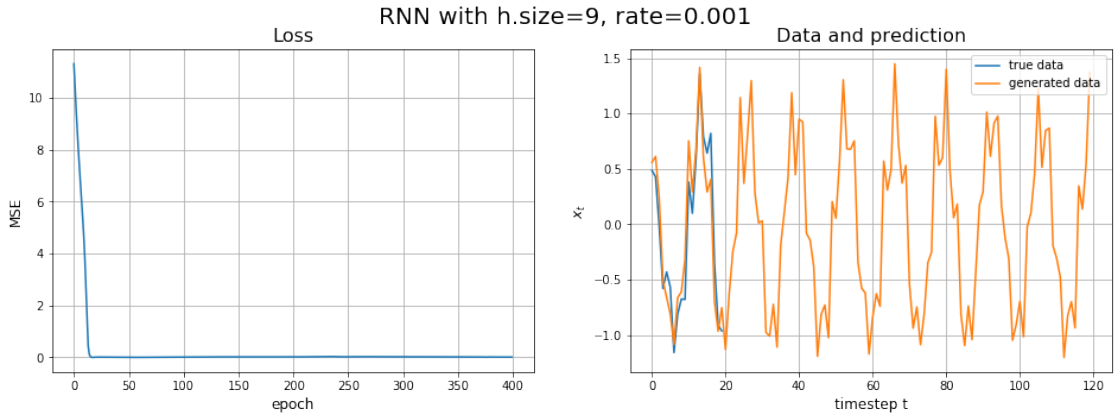
The predictions of the network follow the up and down trend of the noisy input data: this is a symptom of overfitting.

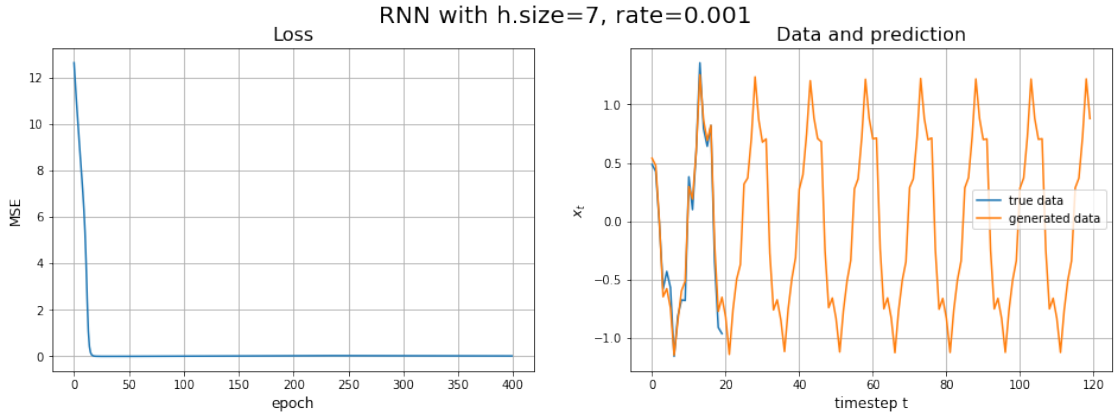
1.0.5 1) Reducing Model Capacity:

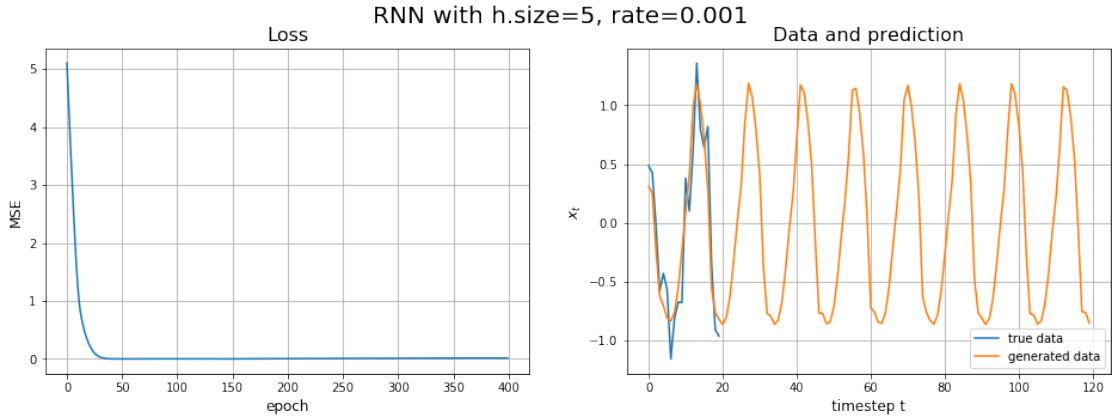
1.0.6 Reduce the number of hidden states until you capture a clean sine wave (without the model trying to reproduce the noise). Plot the predictions

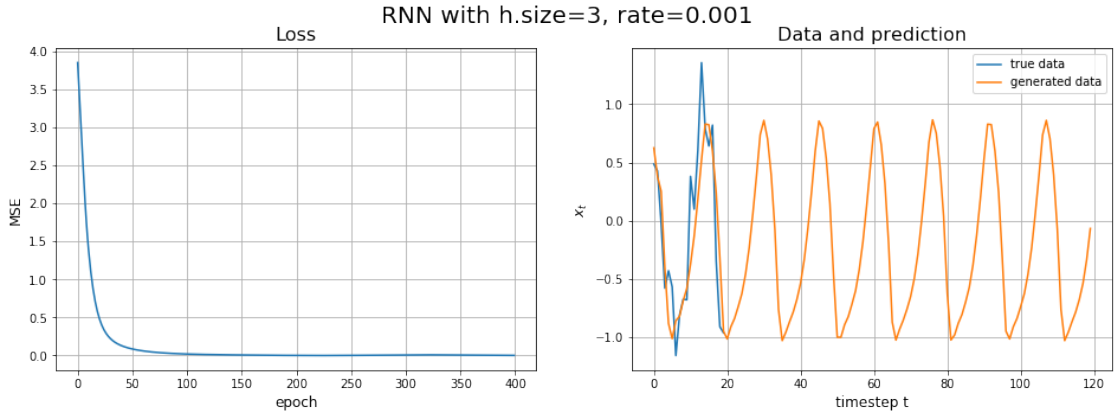
In [4]:
```
neurons = [9, 7, 5, 3]

    for neuron in neurons:
        rnn(hidden_size=neuron, lr = 0.001)
```





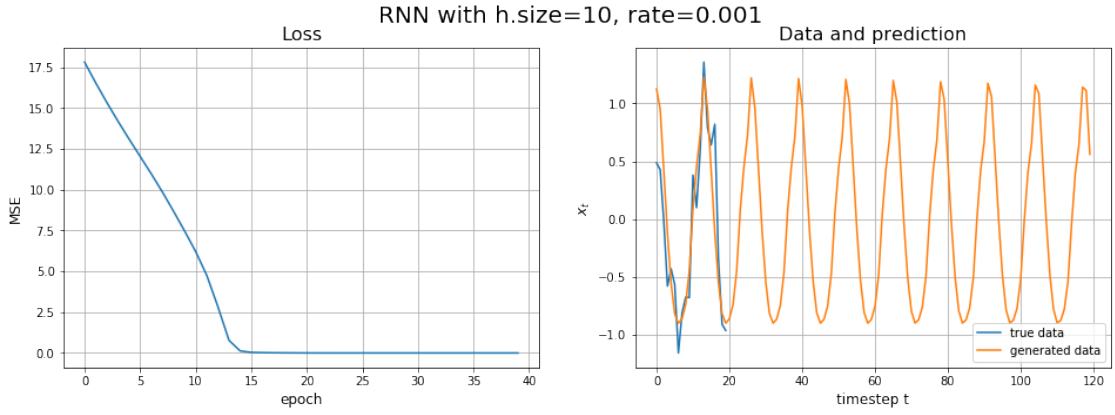




1.0.7 2) Early Stopping:

1.0.8 Reduce the number of training epochs, again until the model does not overfit. Plot the predictions.

```
In [5]: rnn(lr=0.001, epochs = 40)
```

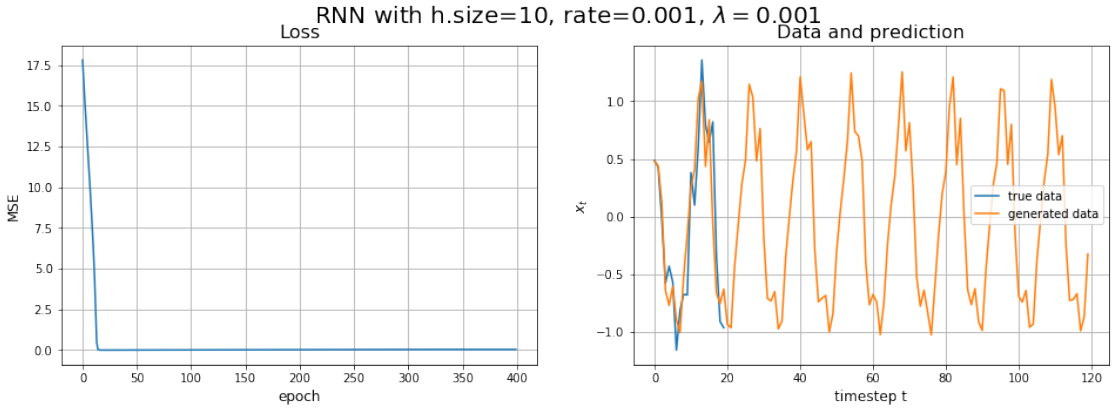


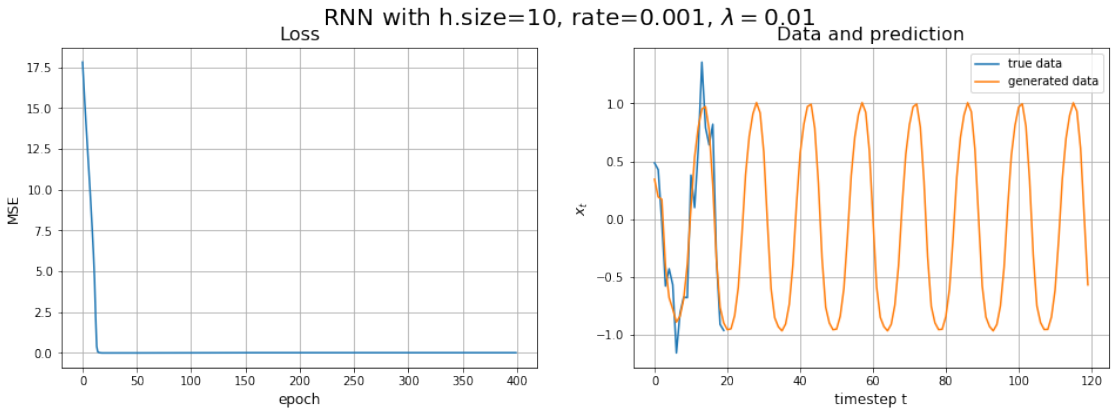
1.0.9 3) Weight Decay:

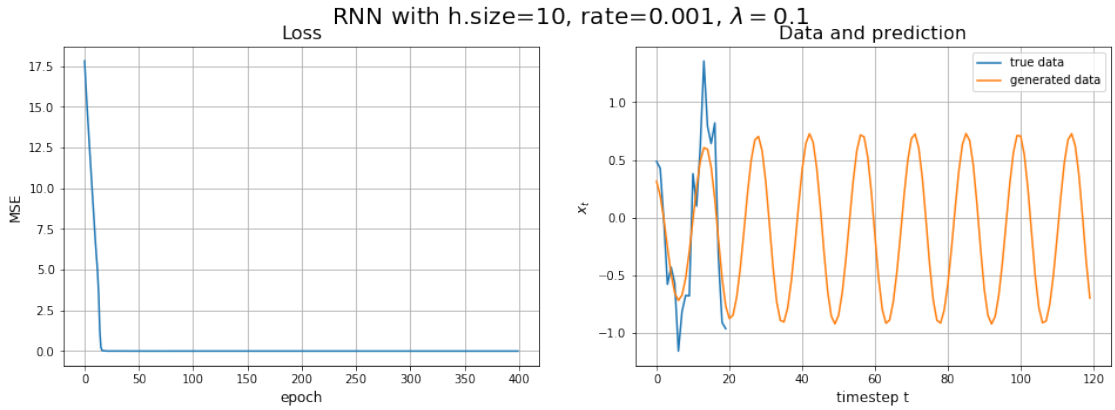
1.0.10 Regularize the model with a constraint on the magnitude of the weights in the loss. Try different settings for the value of λ. Plot the predictions for different λ.

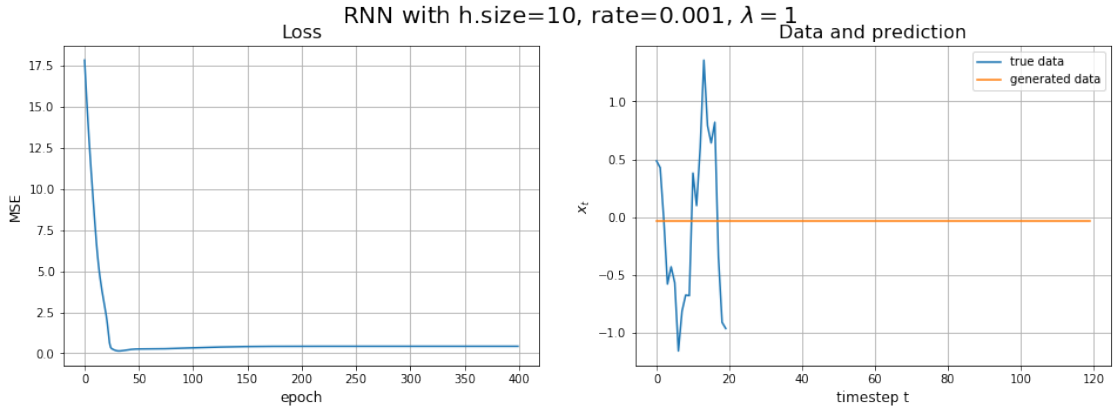
In [7]:
```
lmbds = [0.001, 0.01, 0.1, 1]

for lmbd in lmbds:
    rnn(lr=0.001, lmbd = lmbd)
```









1.0.11 What happens if regularized too strongly?

By regularizing the model with a too strong constraint ($\lambda = 1$) on the magnitude of the weights, the network is no longer able to reproduce the output.

1.0.12 In which sense can weight decay and early stopping be seen as equivalent?

These two techniques can be seen as equivalent since they try to avoid the excess in minimization of the training error, which can lead to poor generalization properties.

rnn_template.py

```
import torch as tc
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from matplotlib import pyplot as plt

# RNN class
class RNN(nn.Module):

    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        # define the network modules here
        # e.g. self.layer = nn.Linear(6, 5)
        self.hidden_size = hidden_size

        self.w_xz = nn.Parameter( tc.Tensor(input_size, hidden_size) )
        self.w_zz = nn.Parameter( tc.Tensor(hidden_size, hidden_size) )
        self.w_zx = nn.Parameter( tc.Tensor(hidden_size, input_size) )

        self.b_z = nn.Parameter( tc.Tensor(hidden_size) )
        self.b_x = nn.Parameter( tc.Tensor(input_size) )

        self.init_weights()


    def init_weights(self):
        tc.manual_seed(2) # for reproducibility

        for name, p in self.state_dict().items():
            nn.init.uniform_(p.data)


    def forward(self, inp, hidden):
        # instantiate modules here
        # e.g. output = self.layer(inp)

        hidden = tc.tanh( tc.mm(inp,self.w_xz) +
            tc.mm(hidden,self.w_zz) + self.b_z )
        output = tc.mm(hidden,self.w_zx) + self.b_x

        return output, hidden


    def get_prediction(self, inp, T):
```

```
        hidden = tc.zeros((1, self.hidden_size))
        predictions = []

        for i in range(T):  # predict for longer time than the training data
            prediction, hidden = self.forward(inp, hidden)
            inp = prediction
            predictions.append(prediction.data.numpy().ravel()[0])

        return predictions


    def train(self, x, y, lr, epochs, lmbd):
        if lmbd == 0:
            optimizer = optim.Adam(self.parameters(), lr=lr)
        else:
            optimizer = optim.Adam(self.parameters(), lr=lr, weight_decay = lmbd)

        losses = []

        for i in range(epochs):
            hidden = tc.zeros((1, self.hidden_size))
            for j in range(x.size(0)):
                optimizer.zero_grad()
                input_ = x[j:(j+1)]
                target = y[j:(j+1)]
                (prediction, hidden) = self.forward(input_, hidden)
                loss = (prediction - target).pow(2).sum()/2

                loss.backward(retain_graph=True)
# retain, because of BPTT (next lecture)
                optimizer.step()
                #losses.append(loss)
            losses.append(loss)

        return losses


# define rnn function to be called
def rnn(hidden_size = 10, lr=0.01, epochs = 400, lmbd = 0):

    # load data
    data = tc.load('noisy_sinus.pt')
    x = tc.FloatTensor(data[:-1])
    y = tc.FloatTensor(data[1:])

    # create RNN model
    model = RNN(input_size=1, hidden_size=hidden_size, output_size=1)
```

```
# train RNN model
losses = model.train(x, y, lr, epochs, lmbd)


# plots
fig1 = plt.subplots(figsize=[16,5])

# print title
s_lr = ('%f' % lr).rstrip('0').rstrip('.')
if lmbd == 0:
    plt.suptitle('RNN␣with␣h.size=%i,␣rate=%s' %(hidden_size, s_lr), fontsize =
else:
    s_lmbd = ('%f' % lmbd).rstrip('0').rstrip('.')
    plt.suptitle(r'RNN␣with␣h.size=%i,␣rate=%s,␣$\lambda=%s$' %(hidden_size, s_
plt.subplots_adjust(hspace = 0.3)

# plot MSE
plt.subplot(1,2,1)
plt.plot(losses)
plt.title('Loss', fontsize = 16)
plt.xlabel('epoch', fontsize = 12)
plt.ylabel('MSE', fontsize = 12)
plt.grid()

# plot predictions over true data
plt.subplot(1,2,2)
predictions = model.get_prediction(inp=x[0:1],T=6*x.size(0))
plt.plot(data[1:], label='true␣data')
plt.plot(predictions, label='generated␣data')
plt.xlabel('timestep␣t', fontsize = 12)
plt.ylabel(r'$x_t$', fontsize = 12)
plt.title('Data␣and␣prediction', fontsize = 16)
plt.grid()
plt.legend()

plt.show(fig1)
```