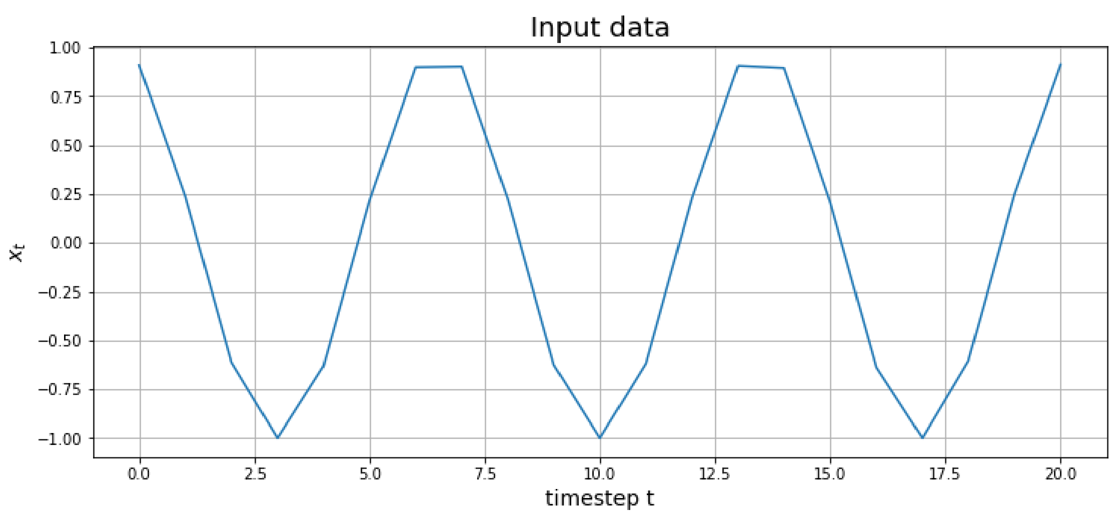
Time Series Analysis and Recurrent Neural Network
Giacomo Barzon - 3626438
Exercise 8

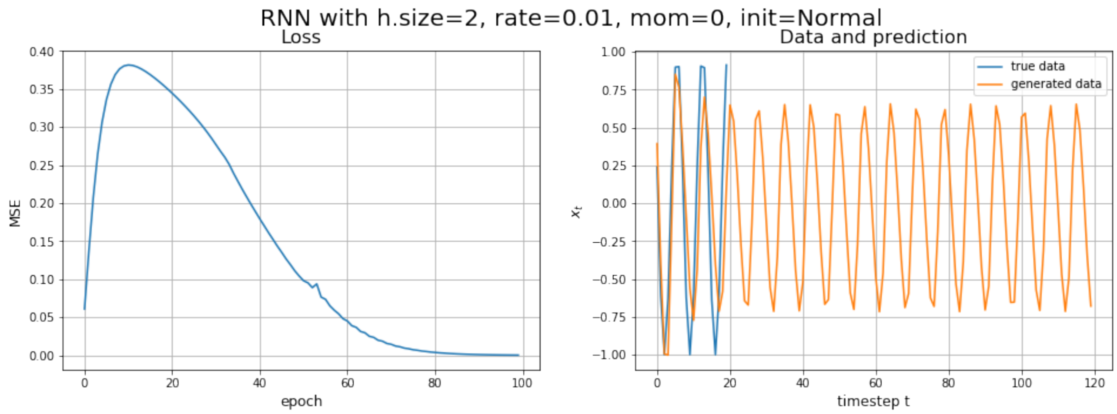
December 18, 2019

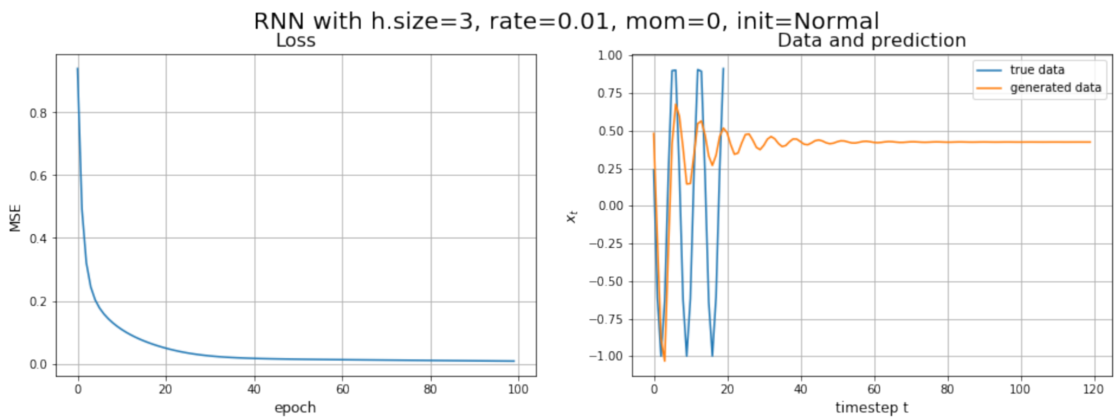
1 Task 1 - Learning Dynamics

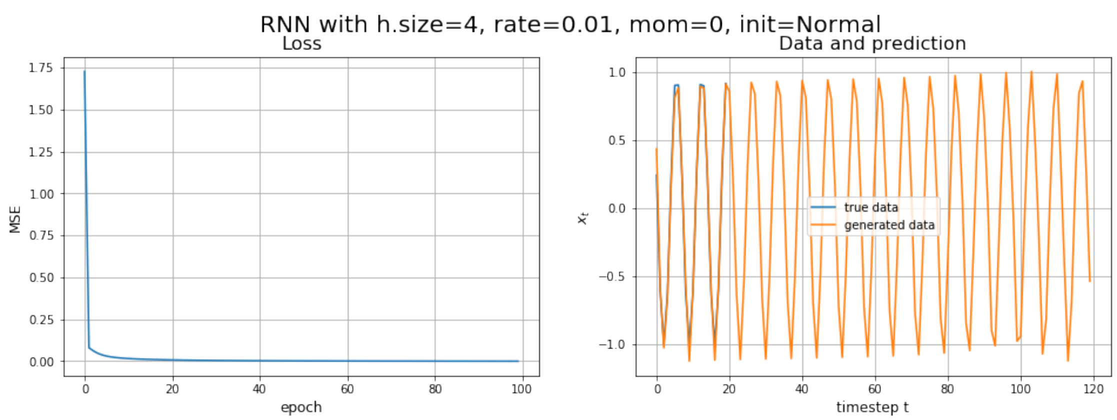
```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import scipy as sp
        from sklearn.linear_model import LinearRegression

        import torch as tc

        from rnn_template import rnn

In [2]: data = tc.load('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()
```

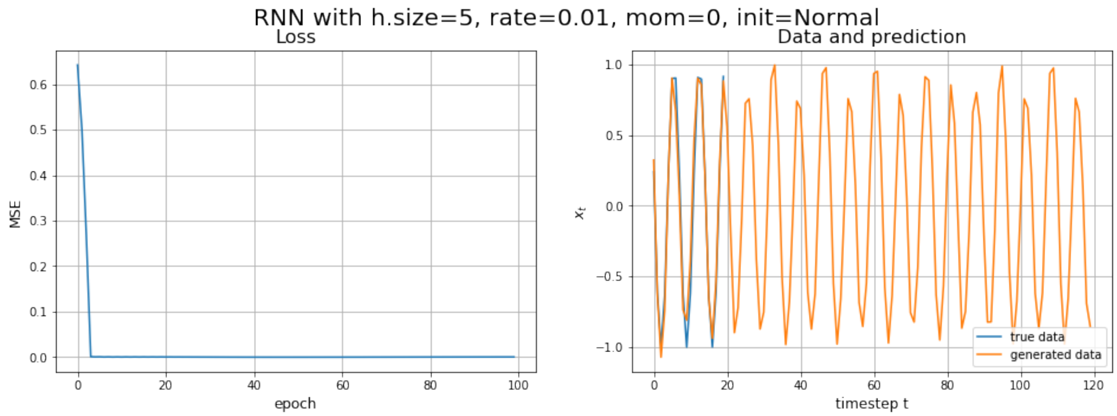


```
In [3]: neurons = [2, 3, 4, 5]

        for neuron in neurons:
            rnn(hidden_size=neuron)
```





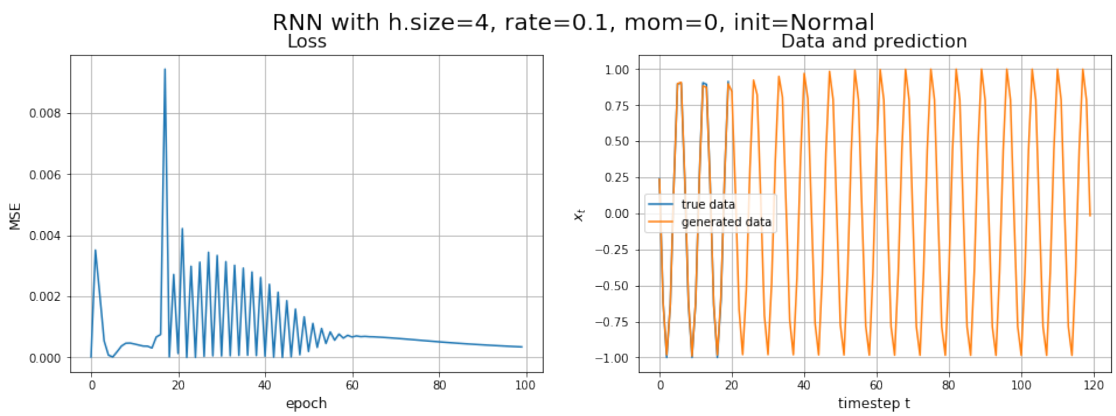


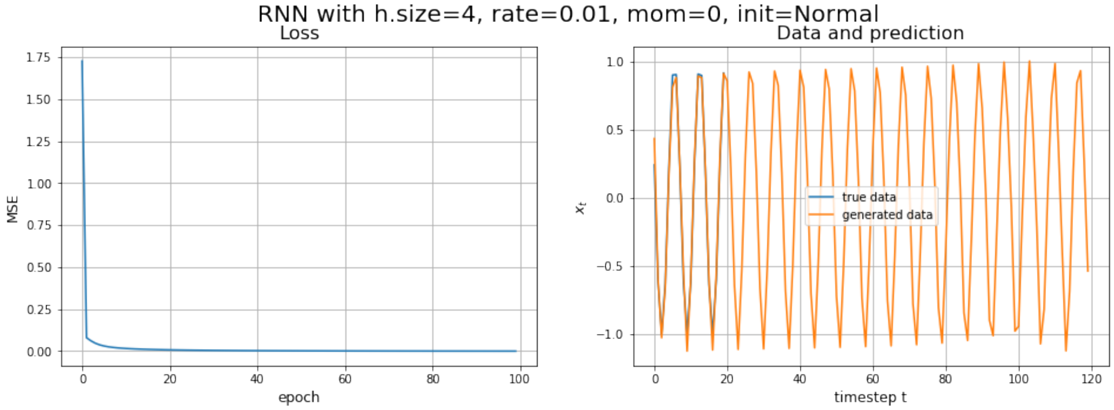


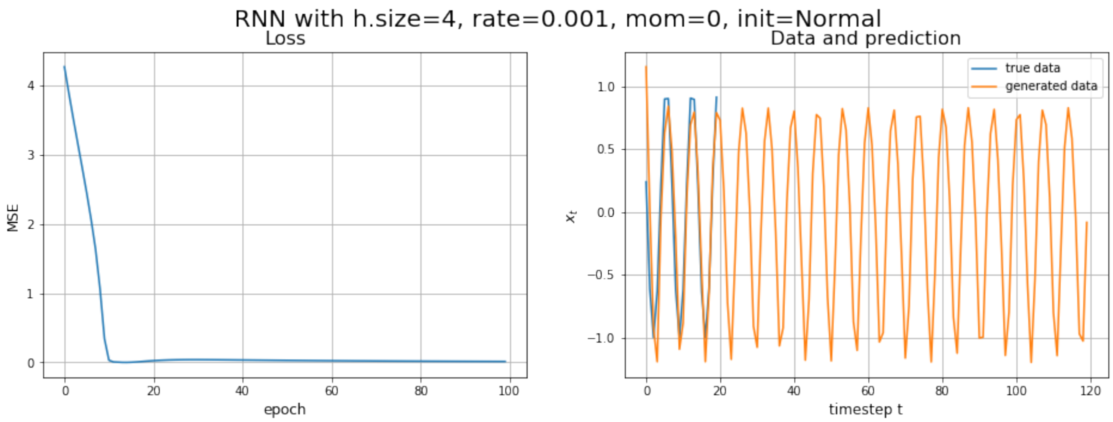
By doing an initial analysis with different number of hidden units, it can be noticed that the network is able to correctly reproduce the oscillation with 4 hidden units or more, so we fix the number of hidden units to 4.

1. Plot the losses as a function of gradient steps and vary the learning rate in the optimizer wrapper for stochastic gradient descent

In [4]:
```
lrs = [0.1, 0.01, 0.001]

for lr in lrs:
    rnn(lr=lr)
```





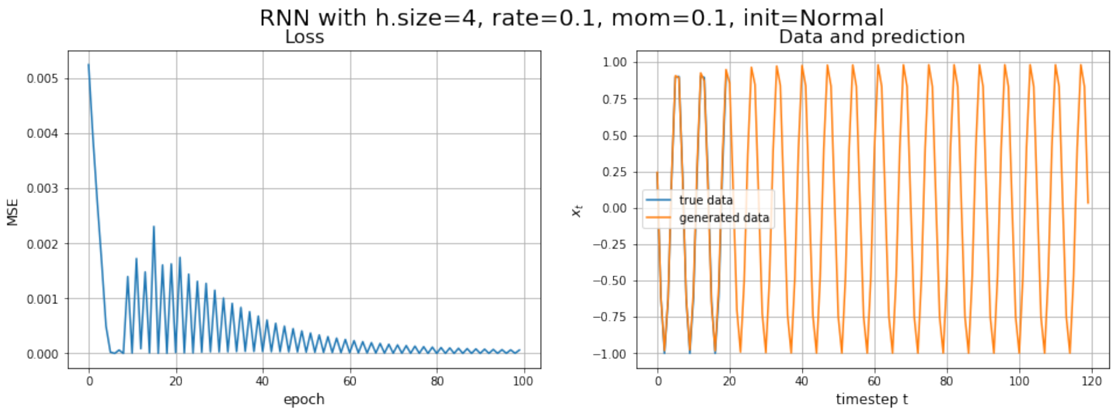


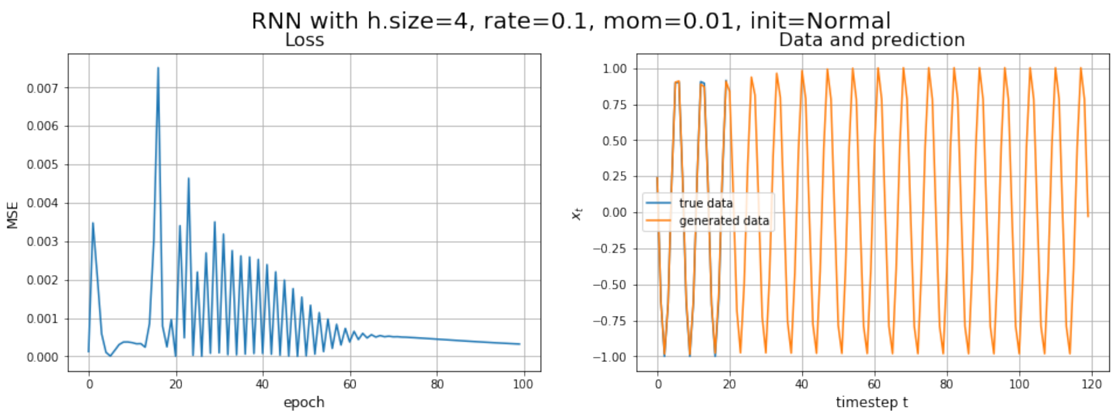
1.0.1 How does the loss behave depending on the learning rate?

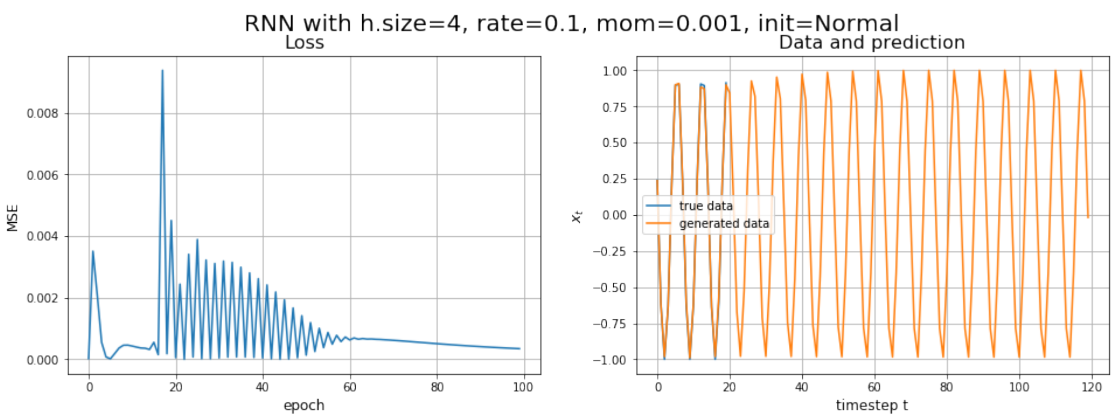
It can be noticed that by using the biggest learning rate the loss function rapidly approaches to zero and then it starts oscillating a bit, while with the other smaller rates it approaches to zero more smoothly.

1.0.2 2. A scheme to speed up learning is to use momentum which keeps a moving average over the past gradients

```
In [5]: moms = [0.1, 0.01, 0.001]

        for mom in moms:
            rnn(lr=0.1, momentum=mom)
```


RNN with h.size=4, rate=0.1, mom=0.1, init=Normal
Loss
MSE
epoch
Data and prediction
true data
generated data
timestep t


RNN with h.size=4, rate=0.1, mom=0.01, init=Normal
Loss
MSE
epoch
Data and prediction
true data
generated data
timestep t

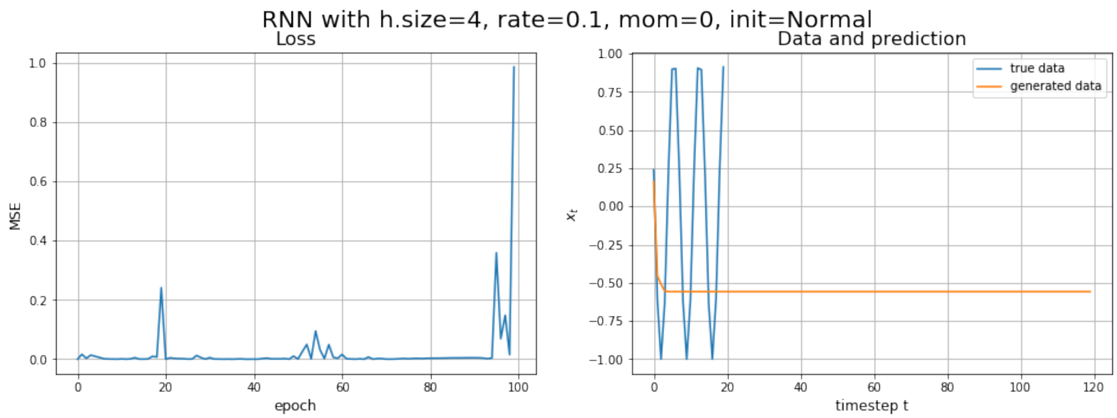

RNN with h.size=4, rate=0.1, mom=0.001, init=Normal
Loss
MSE
epoch
Data and prediction
true data
generated data
timestep t

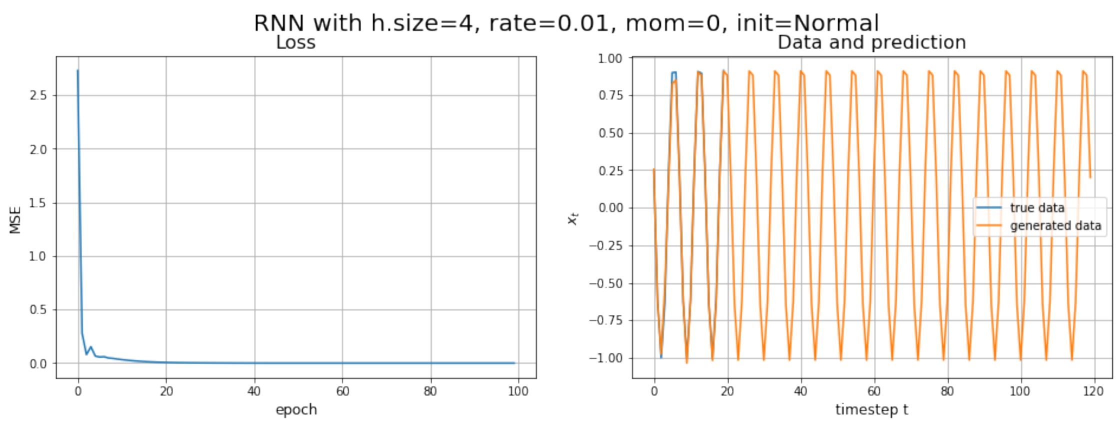
1.0.3 How do the dynamics change when the learning rate is adapted with momentum?

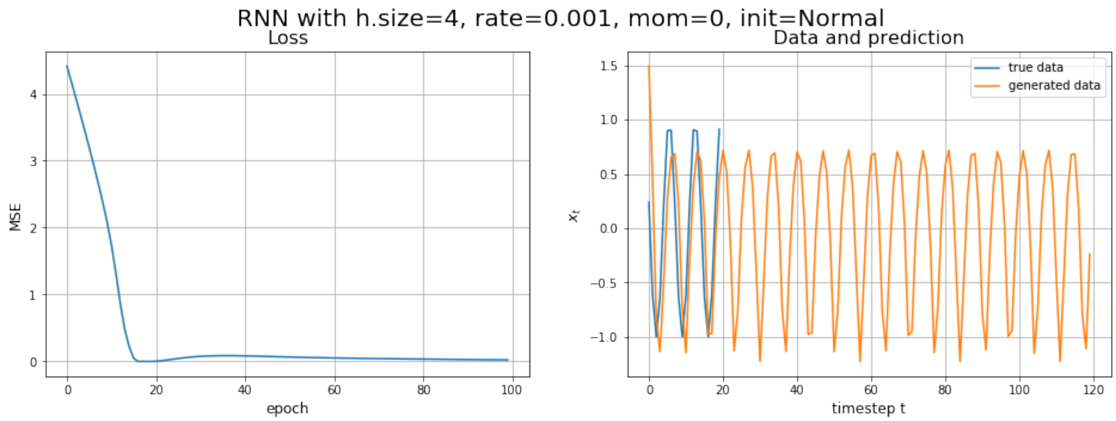
It can be noticed that with the biggest momentum the loss function doesn't end up in the highest peak, that corresponds to a local minimum: so adapting the rate with a momentum helps the learning of the network to reach a absolute minimum in the loss function.

1.0.4 3. How does the adaptive learning rate of the Adam optimizer perform in contrast to stochastic gradient descent?

In [6]:
```python
lrs = [0.1, 0.01, 0.001]

for lr in lrs:
    rnn(lr=lr, algo='Adam')
```





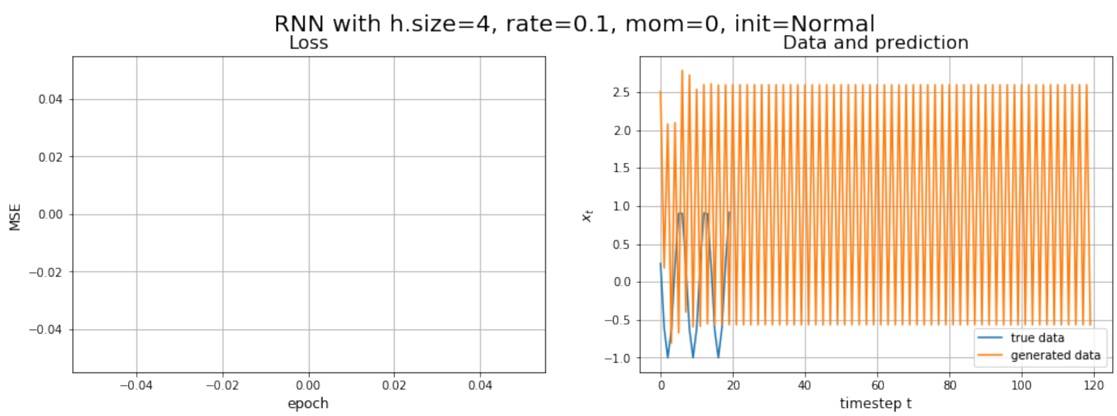


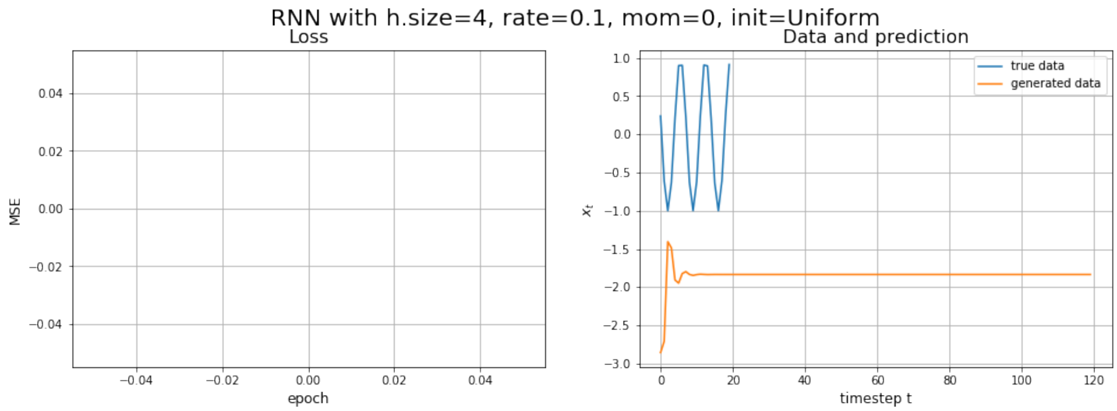
2 Task 2 - Reservoir Computing

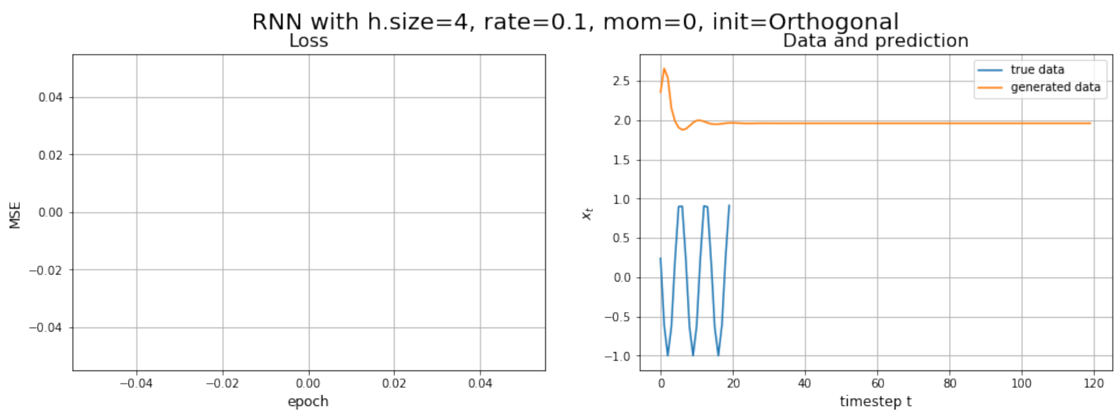
2.0.1 1. Initialize the weights Wxz and Wzz of the network by drawing from:

1. a standard normal,
2. a uniform (in the interval (0, 1)) distribution
3. a random orthogonal matrix ### and plot the dynamics before any training.

```
In [7]: inits = ['Normal', 'Uniform', 'Orthogonal']

        # dynamics before any training
        for init in inits:
            rnn(lr=0.1, init=init, epochs = 0)
```

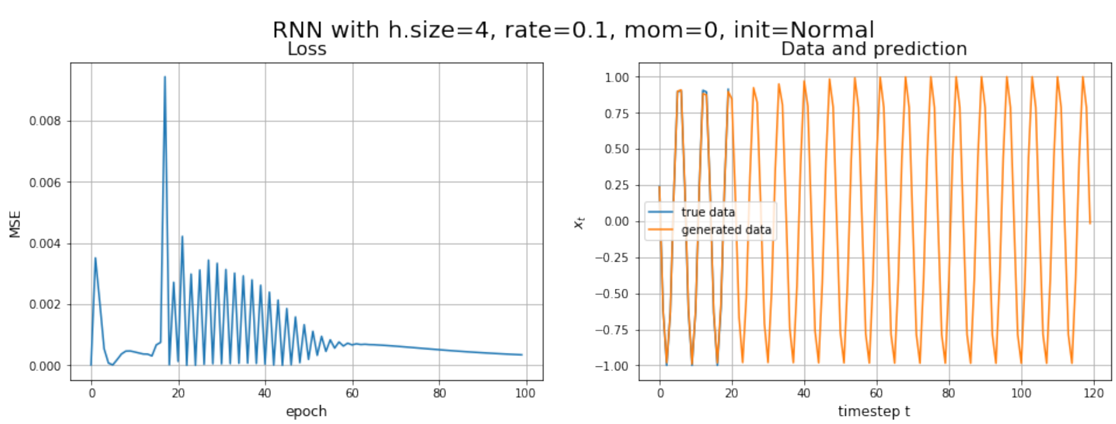


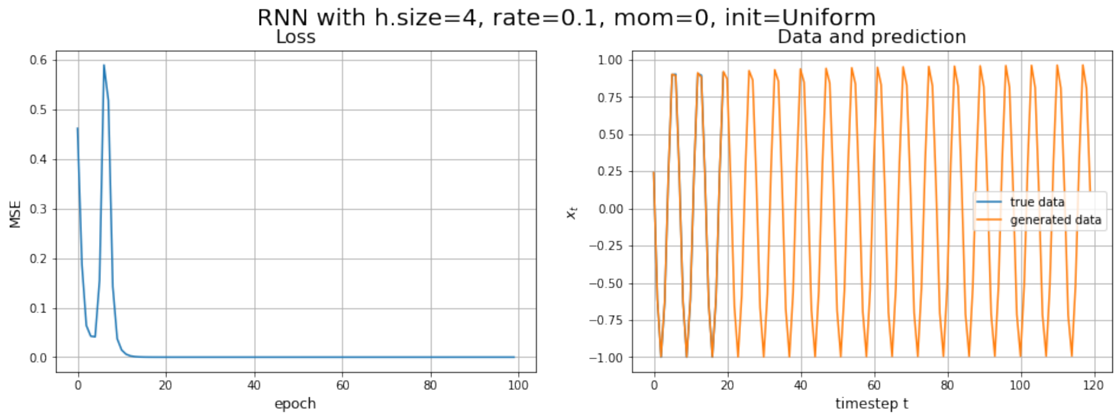


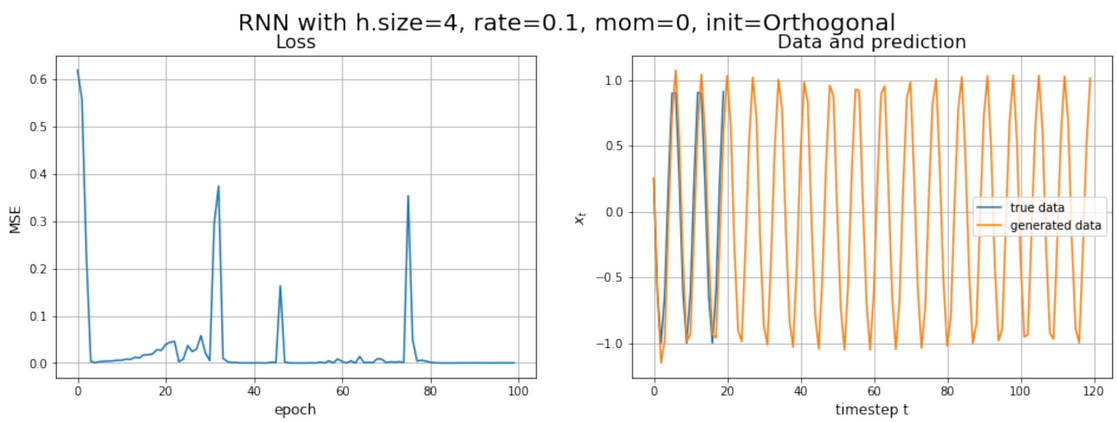


By initializing the weights with a normal standard distribution, the network is able to reproduce an oscillatory output (altough it's translated and with a different magnitude).

```
In [8]: for init in inits:
            rnn(lr=0.1, init=init)
```



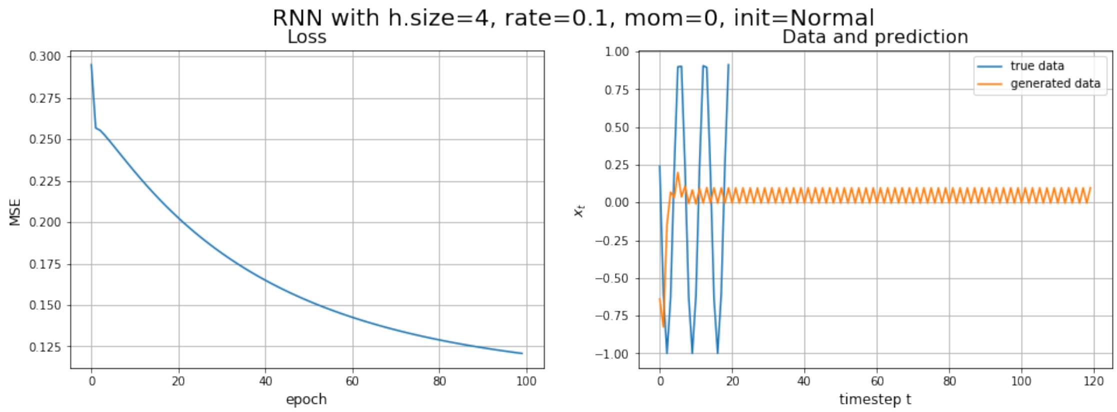


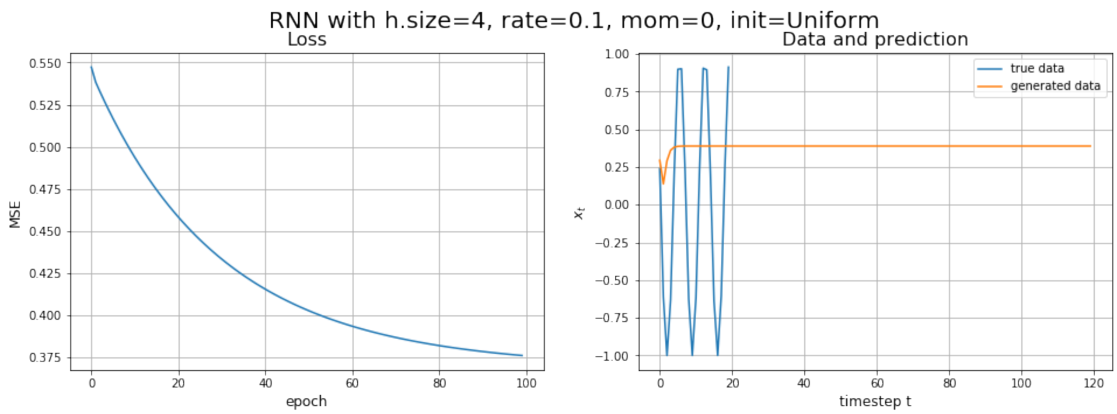


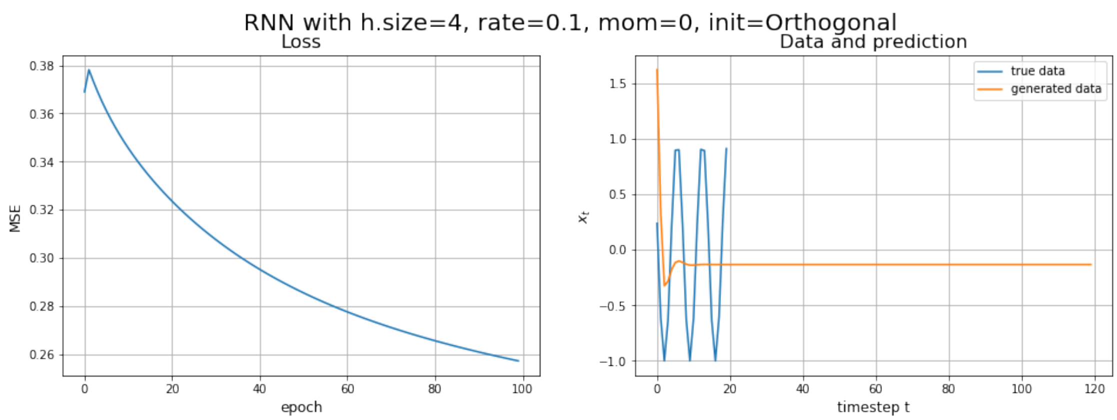
It can be noticed that by initializing the weights from a random orthogonal matrix the network isn't able to correctly reproduce the oscillatory output.

2.0.2 2. Change the to-be-optimized parameters in an optimizer (of your choice) to only contain the output layer weights Wzx

```
In [9]: for init in inits:
            rnn(lr=0.1, init=init, only_output=True)
```







2.0.3 Can you recover the oscillation? Which initialization works best?

By only optimizing the weights of the output layer, none of the different initialization is able to reproduce the output.

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

        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(30091999) # for reproducibility

        for name, p in self.state_dict().items():

            if name == 'w_xz' or name == 'w_zz':
                if self.init == 'Normal':
                    nn.init.normal_(p.data)
                elif self.init == 'Uniform':
                    nn.init.uniform_(p.data)
                elif self.init == 'Orthogonal':
                    nn.init.orthogonal_(p.data)

            else:
                nn.init.normal_(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, momentum, epochs, algo, only_output):
        if algo == 'SGD':
            if only_output:
                optimizer = optim.SGD( [ list(self.parameters())[2] ], lr=lr, momen
            else:
                optimizer = optim.SGD(self.parameters(), lr=lr, momentum=momentum)
        elif algo == 'Adam':
            optimizer = optim.Adam(self.parameters(), lr=lr)

        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)
```

```python
        return losses


# define rnn function to be called
def rnn(hidden_size = 4, lr=0.01, momentum=0.0, algo='SGD', init='Normal', only_out

    # load data
    data = tc.load('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, init=init)

    # train RNN model
    losses = model.train(x, y, lr, momentum, epochs, algo, only_output)

    # plot
    fig1 = plt.subplots(figsize=[16,5])
    s_lr = ('%f' % lr).rstrip('0').rstrip('.')
    s_mom = ('%f' % momentum).rstrip('0').rstrip('.')
    plt.suptitle('RNN with h.size=%i, rate=%s, mom=%s, init=%s' %(hidden_size, s_lr
    plt.subplots_adjust(hspace = 0.3)

    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()

    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)
```