Amath563: LSTM Recurrent Neural Networks for Dynamical Systems

Helena Liu, June 10th, 2020

Abstract: This work demonstrates how Long Short-Term Memory (LSTM) Recurrent Neural Networks (RNNs) can be implemented in Keras for predicting the evolution of dynamical systems. Example datasets include the dynamics generated by the famous Lorentz equations, the Kuramoto-Sivashinsky (KS) equation and a reaction-diffusion (RD) system of equations. This work demonstrates the challenge of predicting chaotic dynamics over a long time horizon and suggests a few avenues for future improvements.

I. Introduction and Overview

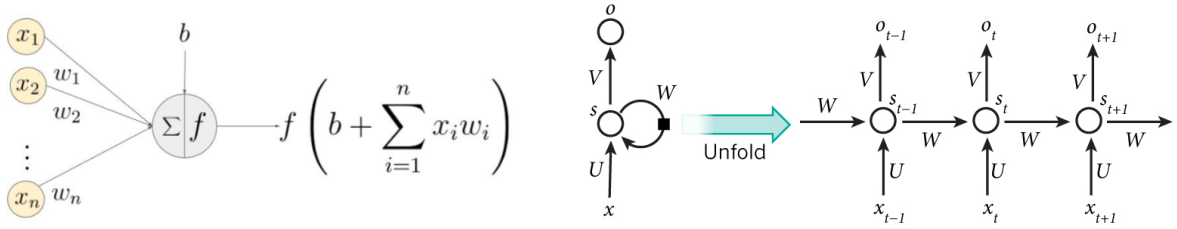
Neural networks, due to its high complexity, has demonstrated exceptional accuracy at creating input-output mappings for prediction. A class of neural networks, called the recurrent neural networks (RNNs), is well suited for fitting time-series data. This study demonstrates how Long Short-Term Memory (LSTM) RNNs can be applied to predict the trajectory of dynamical systems and outlines a few limitations with such approach.

II. Theoretical Background

A) Recurrent Neural Networks

As shown in Figure 1a, generic neural networks consist of simple building blocks, each of which takes in a weighted sum of inputs, $\sum_i w_i x_i$, which is passed through an activation function $f(\cdot)$ to generate activity that propagates to other units in the network. The goal of neural networks is to stack and cascade these units in a way to yield a nested function with enough complexity so that it can map input and output data for prediction.

One commonly used class of neural networks, RNNs, is shown in Figure 1b [1]. The network connects these units in a way such that each unit not only takes in the input data x over time, but also the past output s of other units. This allows the network output to depend on previous computations to keep a memory of the previous network states. Such architecture makes the use of sequential information and has been widely used in time-series prediction, and thus it is focused in this study.



(a) Neural Network Units (b) Recurrent Neural Networks (RNNs)

Figure 1: Generic neural network building blocks and recurrent neural network architecture [1]

B) Training

As mentioned, the goal of neural networks is to develop a function that relates input and to a target output, making it a regression problem. Hence, parameters of the network, i.e. the weights

W, are tuned to optimize a certain cost function. Unless otherwise stated, the loss function used in this study is mean-squared error (MSE), given by

$$E = \frac{1}{NT}\sum_{t=1}^{T}\sum_{i=1}^{N}(o_{i,t} - o_{i,t}^{*})^2 ,$$

Where $o_{i,t}$ is the i^{th} component of the output at time t, $o_{i,t}^{*}$ is the desired output, N is the output dimension and T is the total number of time steps.

Parameters can then be optimized to minimize loss function E using gradient descent, which updates parameter W_{ij} in the direction of steepest decrease in the loss function:

$$W_{ij} = W_{ij} - \alpha \frac{\partial E}{\partial W_{ij}} ,$$

where α is the learning rate.

Derivatives can be computed via backpropagation, which uses chain rule to bookkeep gradients at each stage of the network. In the case of RNNs, the network is unwrapped across time as in Figure 1b and chain rule is applied across time steps, and this method of training RNNs is referred to as backpropagation through time (BPTT).

C) LSTM

In practice, RNNs suffer the problem of exploding gradients when trained with BPTT, which can happen when numerical errors are amplified across network stages. This motivated the usage of LSTM RNNs, which adds a gate functions g_t^i to the unit input, g_t^o to the unit output and g_t^f for deciding to what extent past states are kept [2]. The magnitudes of these gate function outputs are within 1, thereby capping the amplification factors. Equation of LSTM are given by

$$g_t^f = \sigma_f\big(W_f[s_{t-1}, x_t]\big), \qquad g_t^i = \sigma_i(W_i[s_{t-1}, x_t]),$$

$$C'_t = \tanh(W_C[s_{t-1}, x_t]), \qquad C_t = g_t^f\, C_{t-1} + g_t^i C'_t$$

$$g_t^o = \sigma_s(W_s[s_{t-1}, x_t]), \qquad s_t = g_t^o \tanh(C_t),$$

where σ is the sigmoid function and the g's are gate functions. Other choices of gate functions are possible too. Weights W_f, W_i, W_C and W_s are parameters to be tuned through BPTT. Note that bias terms are neglected for readability.

III. Algorithm Implementation and Development

A) Training

The model is developed in Keras, which is a framework that allows high-level training of neural networks. For simplicity, only a single LSTM and output layer is used. To implement gradient descent, Adam Optimizer is used, which combines momentum to avoid being trapped in local minima and adjustable learning rate in each direction of the feature vector to make training more efficient. To regularize the network, dropout is used with rate of 0.2 to avoid having dominating

units in the network, which would make the network less robust. Training is performed over 100 epochs and with the loss function as MSE described in the previous section.

For the RD system, the network is trained on dimensionally reduced data through SVD. The first r component of V used for training, where r is the rank used. For testing, $\Sigma^{-1}U^T$ is multiplied by the test data to first project it to the low-dimensional space, then the trained model is applied to forecast in the low-dimensional space, and then the predicted data is multiplied with U Σ to project it back to the high-dimensional space for visualization.

B) Input and Output Data

The LSTM layer in Keras takes in a tensor input of size $\mathbb{R}^{MxLxN}$, where *M* is the number of samples, *L* is the number of historical time steps to process and *N* is the dimension of each data point. In this work, *L* is set to 15, i.e. the network considers data over the past 15 time steps when computing the output. The dimension of the output is $\mathbb{R}^{MxN}$. For each input and output pair, the input would be the data from time *t-L* to *t-1* and the output would be the data at *t*.

More details on the specifics of how the three famous dynamical systems are set up for data generation can be found in the source code in Appendix B.

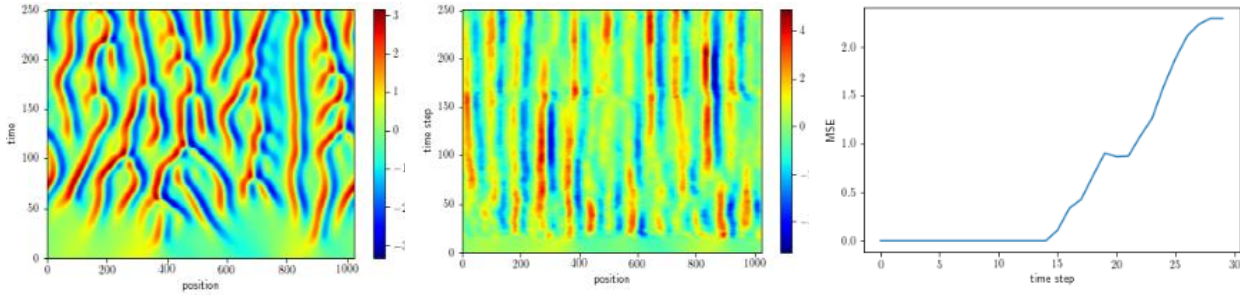
C) Cross Validation

To ensure the trained model generalizes to new data, test data for all three dynamical systems is generated from a new random initial condition. The predicted trajectory is created by iteratively applying to trained model to predict the next step using the predicted data from previous steps. MSE over time, which is given by $E(t) = \frac{1}{N}\sum_{i=1}^{N}(o_{i,t} - o_{i,t}^*)^2$, is used to visualize the performance of the prediction.

IV. Computational Results

A) Predicting the KS Equation

The LSTM network described in the previous section is trained on 100 trajectories generated by the KS equation and tested on one new trajectory shown in Figure 2. Each trial is initialized randomly. For the one test case, comparing the actual trajectory in Figure 2a and the predicted trajectory in Figure 2b shows that the prediction was only able to follow the actual dynamics for a very short span of time at the beginning. This is examined more in Figure 2c, where the MSE between the predicted and actual dynamics at each time step is shown. Note that prediction was not done on time steps 0 to 14, as the LSTM model is constructed to require the previous 15 steps for input. Hence, the first step where prediction began is time step 15, where the MSE is about 0.1. The one-step MSE on training data is about 0.01, ten times less than that of the test error, which suggests that there could be overfitting. This is already an improvement from the results obtained without dropout, where the one-step test MSE is about 0.3 and one-step training MSE is about 0.005. This suggests that further improvements in generalization may be attained by regularization. The MSE then climbs soon after prediction begins at time 15, which shows the limited time horizon of the prediction.

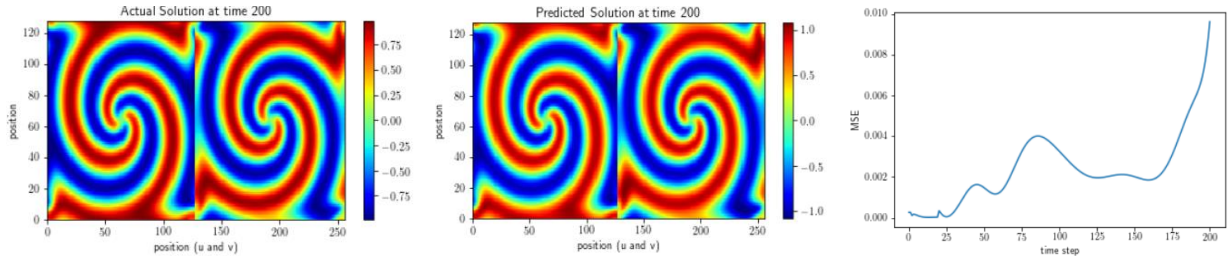


(a) Actual Dynamics *(b) Predicted Dynamics* *(c) MSE over time*

***Figure 2**: Comparing the actual and predicted dynamics of the KS equation*

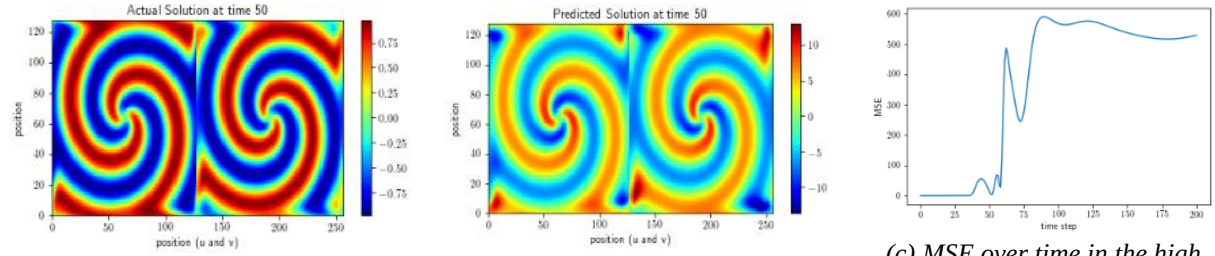
B) Predicting the RD System via SVD

Due to the high dimensionality of the RD system, training on the full system takes very long, and training in a low-dimensional subspace when only 20 SVD ranks were used resulted in a roughly 500-fold reduction in the time required to complete one epoch. The model was trained on 20 different trajectories and tested on one new trajectory shown in Figure 3. Comparing the actual trajectory in Figure 3a and the predicted trajectory in Figure 3b shows that the prediction was able to follow the actual test dynamics all the way to the end of the trial at T=200, which is also reflected in the low MSE in Figure 3c. One may argue that this low error is likely due to the slow evolution of the dynamics, which allows the network to have a longer prediction horizon. However, that is not the full story, as when 100 ranks were used, the predicted and actual dynamics start to grow apart at about time step 50, as shown in Figure 4. This highlights SVD as an effective tool to not only speed up training, but also to improve generalization by removing excessive components that can cause overfitting.



(a) Actual Dynamics *(b) Predicted Dynamics* *(c) MSE over time in the high dimensional space*

***Figure 3**: Comparing the actual and predicted dynamics of the RD System using LSTM RNN trained on only 20 SVD ranks*



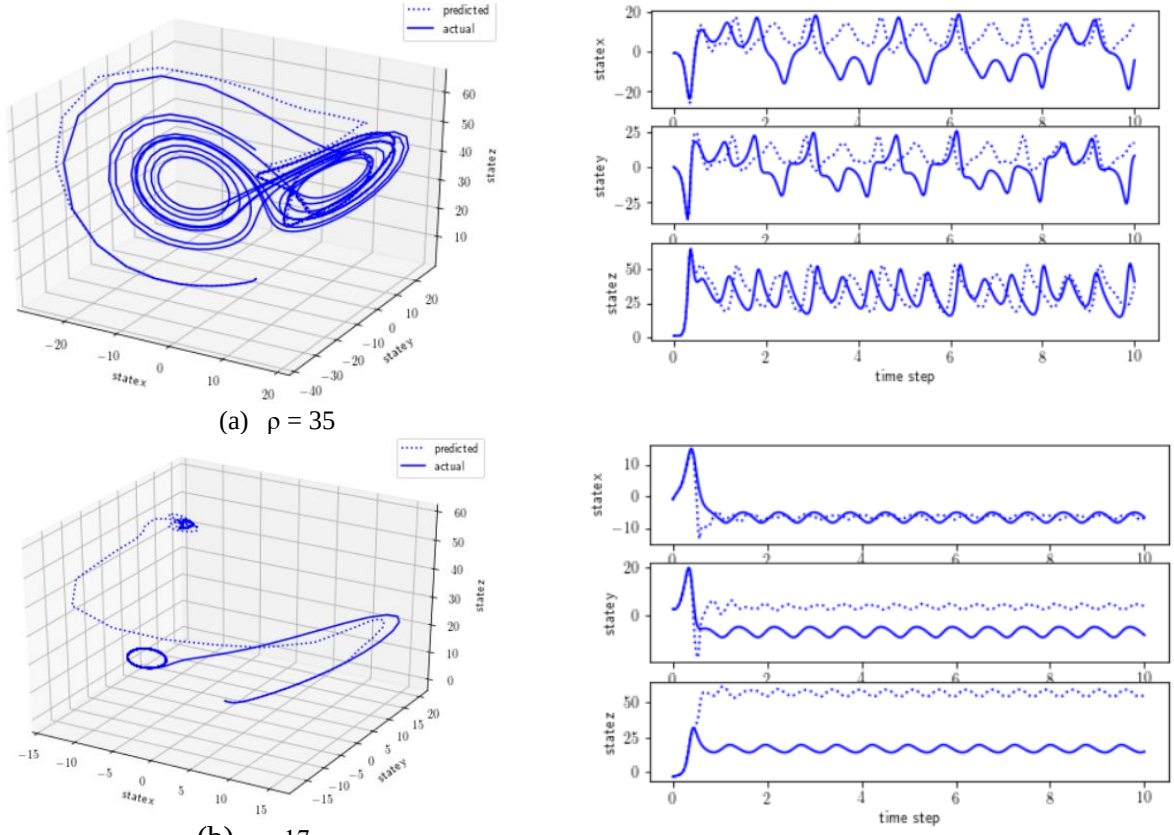
(a) Actual Test Dynamics *(b) Predicted Test Dynamics* *(c) MSE over time in the high dimensional space*

***Figure 4**: Similar to Figure 3, but with 100 ranks used*

A major limitation with this section is that trajectories are all initialized with the same number of spirals and only differ by a phase shift, so training and test data share similar structure. Future work may involve generalizing the model to more diverse set of initial conditions.

C) Predicting the Lorentz Equations

Similar approaches in the previous section are applied to for predicting the trajectory of Lorentz equation. The predicted and actual trajectory is illustrated in Figure 5. Training is done with ρ = 10, 28 and 40 and testing is done on ρ = 35 and 17. In both test examples, the predicted trajectory was able to follow the actual trajectory for less than one second, and then they diverge, which indicates the short prediction horizon of this method. After that, the prediction still exhibits some qualitative characteristics of the actual trajectory. This shows that while the model is able to learn some characteristics of the dynamics, it is very difficult to predict the exact timing of the long term dynamics for a chaotic system, as a small prediction error at one step will cause the rest of prediction to diverge from the actual trajectory.



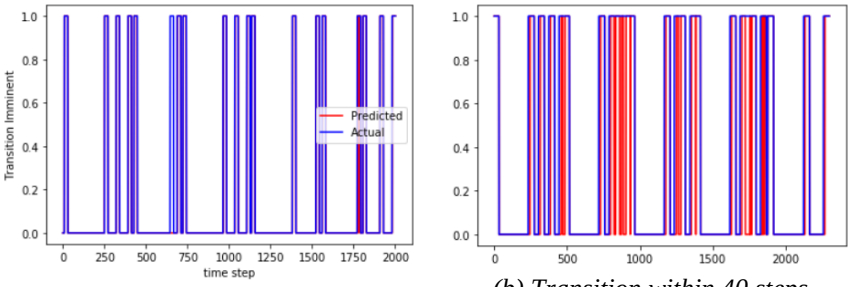
(a) ρ = 35

(b) ρ = 17

Figure 5*: Comparing the actual dynamics of the Lorentz equation and the predicted dynamics*

Next, the network is trained to predict if the transition between two lobes of Lorentz attractor is imminent, where the lobe label is determined by the sign of state x. The state is in lobe one if x is nonnegative and lobe two otherwise. This problem can be formulated as either a regression or binary classification problem. For regression, the task would be to predict the time until the next transition given a state. For classification, the task would be to predict if the next transition will occur within a given time window. This study follows the latter formulation and trains the network to predict if the transition will occur within 25 steps and 40 steps, as shown in Figure 6,

and the test error rates obtained are 0.019 and 0.13, respectively. Decent accuracy is achieved for short-term prediction (e.g. whether transition will occur in the next 25 steps). The accuracy deteriorates when the time window is increased (e.g. the case of 40 times steps), which shows that this method has a limited prediction horizon.



(a) Transition within 25 steps *(b) Transition within 40 steps*

***Figure 6**: Binary classification for predicting if a transition between the two lobes of a Lorentz attractor will occur given a within time window, where 1 indicates a yes and 0 indicates a no.*

V. Summary and Conclusions

LSTM RNN is the state-of-art algorithm for time series prediction, and this work demonstrated how LSTM can be implemented in Keras for dynamical systems prediction. The high complexity and large number of parameters in LSTM RNNs make them prone to overfitting. This study demonstrated that techniques such as dropout and dimensionality reduction through SVD can be effective means to reduce overfitting. Future work may further improve generalization by exploring other regularization techniques, such as L1/L2 regularization and early stopping.

This work also discussed how the time horizon for accurate prediction is limited for chaotic systems, which are sensitive to initial conditions so small errors in prediction could cause the rest of the prediction to deviate significantly from the actual trajectory. Future work may apply data assimilation techniques to periodically correct the prediction.

VI. References:

[1] LeCun Y, Bengio Y, Hinton G. Deep learning. *Nature*. 2015;521(7553):436-444. doi:10.1038/nature14539

[2] Hochreiter S, Schmidhuber J. Long short-term memory. *Neural Comput*. 1997;9(8):1735-1780. doi:10.1162/neco.1997.9.8.1735

Appendix A: Python Functions Used

All functions used are included in Appendix B, and most of them are specific to each of the tasks described in Section IV. The following methods are used in all parts of this work:

- **model = Sequential():** initializes a model that consists of a stack of layers
- **model.add():** add a layer to the sequential model
    - e.g. model.add(layers.Dense(4)): adds a dense layer with four units
    - e.g. model.add(layers.LSTM(4)): adds a LSTM layer with N units
- **adam = tf.keras.optimizers.adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07, amsgrad=False):** build an optimizer that uses Adam Optimization
- **model.compile(loss='mean_squared_error', optimizer = adam**): configures the sequential model for training
- **model.fit(x, y, epochs=100, batch_size=3000):** fits the model based on input data x and output data y according to the loss function and optimization scheme specified in the compiled model
- **model.predict(x):** generates output predictions for input x based on the fitted model

Appendix B – Python CODE

All Jupyter Notebooks used in this assignment are attached in the following pages. All code used in this assignment have been uploaded to GitHub: https://github.com/Helena-Yuhan-Liu/Inferring-Structure-of-Complex-Systems.

KS_NN

June 6, 2020

[1]:
```
%pylab inline

import numpy as np
import tensorflow as tf

from scipy import integrate
from scipy.io import loadmat
from mpl_toolkits.mplot3d import Axes3D

import tensorflow.keras as keras
from tensorflow.keras import optimizers
from tensorflow.keras.models import Model,Sequential,load_model
from tensorflow.keras.layers import Input,Dense,Convolution1D,Activation
from tensorflow.keras.layers import LSTM
from tensorflow.keras import backend as K
from tensorflow.keras.utils import plot_model

from IPython.display import clear_output
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
/home/hyliu24/anaconda3/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:523: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/home/hyliu24/anaconda3/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:524: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/home/hyliu24/anaconda3/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:525: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/home/hyliu24/anaconda3/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:526: FutureWarning: Passing
```

```
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/home/hyliu24/anaconda3/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:527: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/home/hyliu24/anaconda3/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:532: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

[2]:
```
class PlotLosses(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.i = 0
        self.x = []
        self.losses = []
        self.val_losses = []

        self.fig = plt.figure()

        self.logs = []

    def on_epoch_end(self, epoch, logs={}):

        self.logs.append(logs)
        self.x.append(self.i)
        self.losses.append(logs.get('loss'))
        self.val_losses.append(logs.get('val_loss'))
        self.i += 1

        clear_output(wait=True)
        plt.plot(self.x, self.losses, label="loss")
        plt.plot(self.x, self.val_losses, label="val_loss")
        plt.yscale('log')
        plt.legend()
        plt.show();

plot_losses = PlotLosses()
```

[3]:
```
def progress_bar(percent):
    length = 40
    pos = round(length*percent)
    clear_output(wait=True)
    print('['+' '*pos+' '*(length-pos)+']  '+str(int(100*percent))+'%')
```

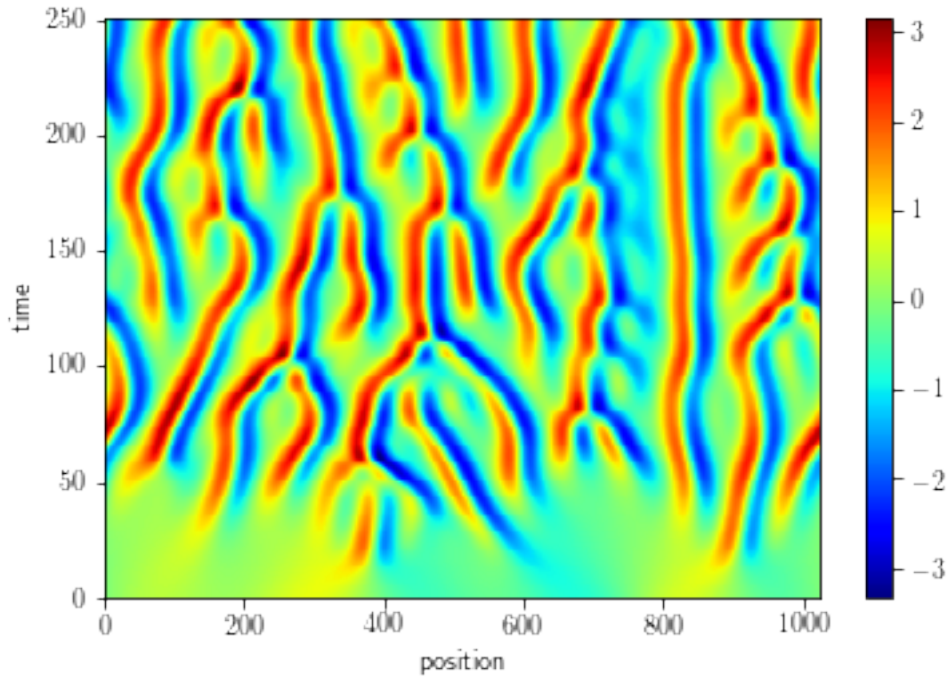
0.1 Load KS trajectories

[4]:
```
num_train = 100
num_tests = 1
look_back = 15

N = 1024
T = 251
KS_input_data = np.zeros(((T-look_back)*(num_train-num_tests), look_back, N)) #␣
 ↪Shape: (Nbatch, time, dim)
KS_target_data = np.zeros(((T-look_back)*(num_train-num_tests),N))

for i in range(num_train-num_tests):
    u = loadmat('PDECODES/KS_data/N'+str(N)+'/iter'+str(i+1)+'.mat')['uu']
    for l in range(look_back):
        KS_input_data[i*(T-look_back):(i+1)*(T-look_back),l] = u[:,l:
 ↪-(look_back-l)].T
    KS_target_data[i*(T-look_back):(i+1)*(T-look_back)] = u[:,look_back:].T
```

[5]:
```
KS_test_data = np.zeros((T*num_tests,N))
for i in range(num_tests):
    u = loadmat('PDECODES/KS_data/N'+str(N)+'/iter'+str(num_train-i)+'.
 ↪mat')['uu']
    KS_test_data[i*T:(i+1)*T] = u.T
```

[6]:
```
mpl.rcParams['text.usetex'] = True
m = plt.pcolor(KS_test_data,cmap='jet')
m.set_rasterized(True)
plt.xlabel('position')
plt.ylabel('time')
plt.colorbar()
#plt.savefig('img/sample_KS_trajectory.pdf')
```

[6]:
```
<matplotlib.colorbar.Colorbar at 0x7fb4a6dea8d0>
```

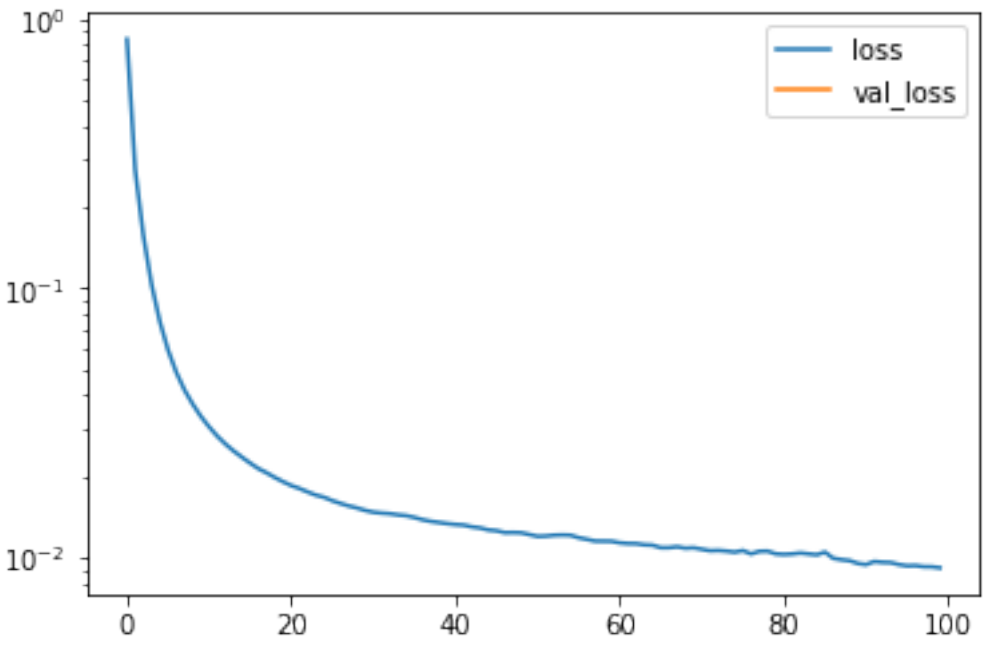


0.2 Train Neural Network

[15]:
```
model = Sequential()
model.add(LSTM(N, activation="tanh", recurrent_activation="sigmoid", dropout=0.
 ↪2, recurrent_dropout=0.2))
model.add(Dense(N))
```

[16]:
```
# Adam is the preferred choice
adam_ = keras.optimizers.Adam(lr=.02, beta_1=0.9, beta_2=0.999, epsilon=None,␣
 ↪decay=1e-4, amsgrad=True, clipvalue=0.5)
model.compile(loss='mean_squared_error', optimizer=adam_, metrics=['accuracy'])
#plot_model(model, to_file='model.pdf', show_shapes=True)
```

[17]:
```
mpl.rcParams['text.usetex'] = False
model.fit(
    KS_input_data,
    KS_target_data,
    epochs=100, batch_size=2000, shuffle=True, callbacks=[plot_losses],␣
 ↪validation_split=0.0)
```




```
23364/23364 [==============================] - 48s 2ms/step - loss: 0.0091 -
acc: 0.2888
```

[17]:
```
<tensorflow.python.keras.callbacks.History at 0x7fb47818de10>
```

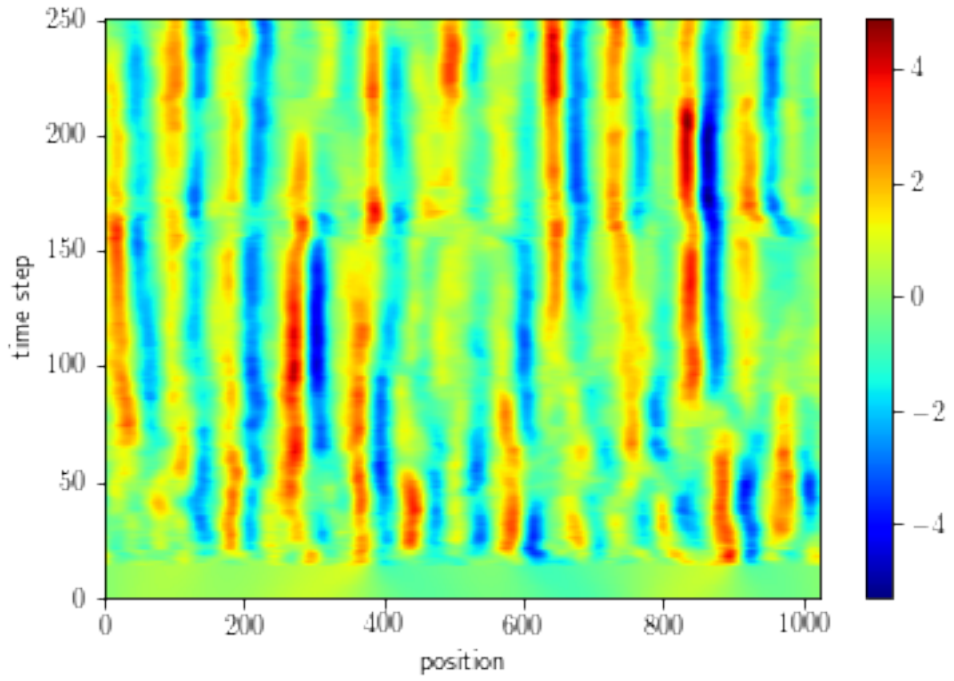
[18]:
```
KS_NN_prediction = np.zeros(KS_test_data[0:T].shape)
KS_NN_prediction[0:look_back] = KS_test_data[0:look_back]
for k in range(T-look_back):
    pred_input = reshape(KS_NN_prediction[k:k+look_back], (look_back,␣
 ↪KS_NN_prediction.shape[-1]))
    KS_NN_prediction[k+look_back] = model.predict(np.array([pred_input]))
```

[24]:
```
print(pred_input.shape)
print(KS_NN_prediction.shape)
```

```
(15, 1024)
(251, 1024)
```

[19]:
```
mpl.rcParams['text.usetex'] = True
m = plt.pcolor(KS_NN_prediction, cmap='jet')
m.set_rasterized(True)
plt.xlabel('position')
plt.ylabel('time step')
plt.colorbar()
```

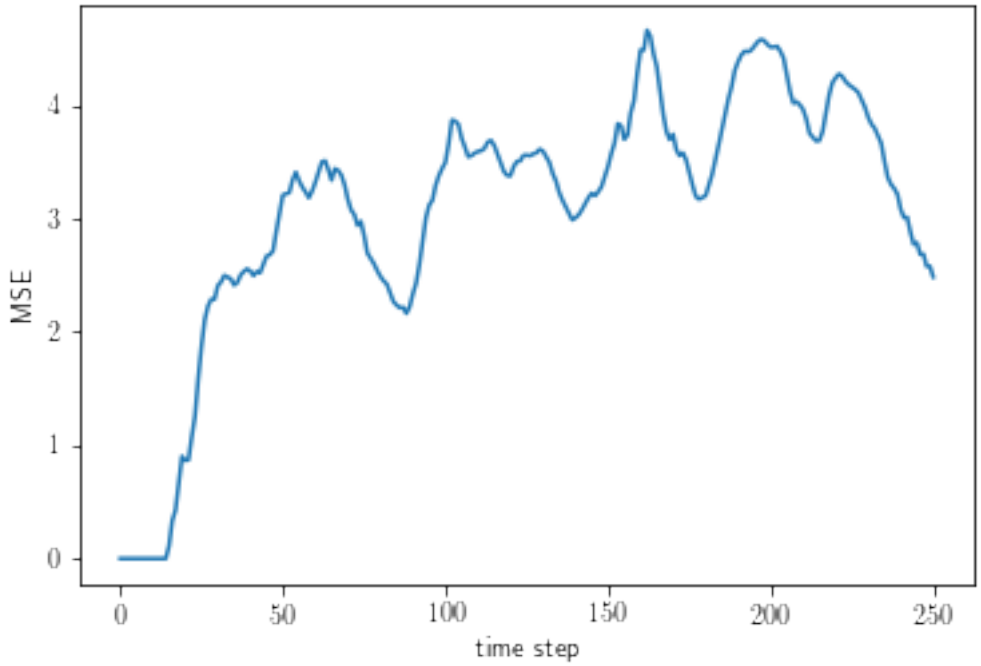
[19]:
```
<matplotlib.colorbar.Colorbar at 0x7fb20c17ae10>
```

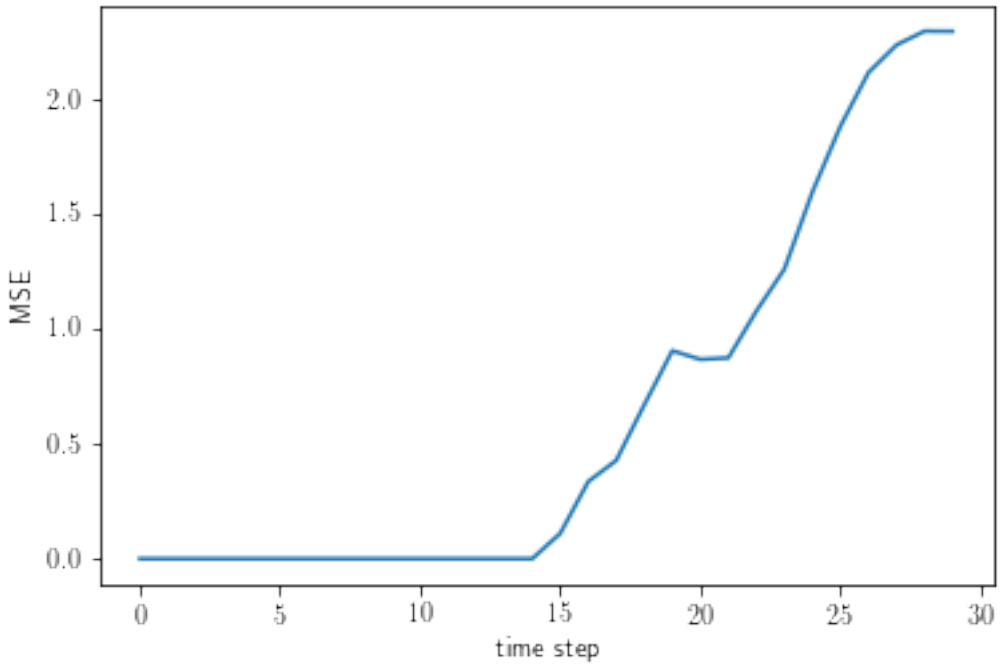


[20]:
```
mse_total = ((KS_NN_prediction-KS_test_data[0:T])**2).mean(axis=None)
mse_over_step = ((KS_NN_prediction-KS_test_data[0:T])**2).mean(axis=-1)
mse_one_step = mse_over_step[look_back]

figure()
plt.plot(np.array(range(T)), mse_over_step)
xlabel('time step')
ylabel('MSE')

figure()
plt.plot(np.array(range(T)[0:30]), mse_over_step[0:30])
xlabel('time step')
ylabel('MSE')

print(mse_one_step)

# Training error (one step) is about 0.0091 at the end, test error (one step)␣
 ↪is about 0.1099,
# about 40 times that of the training error, hence there could be some␣
 ↪overfitting and
# techniques such as regularization, dropout and early stopping could be used.
# Moreover, MSE shoots up over time, which limits the time horizon of the␣
 ↪prediction.
```

```
# However, this is better than without dropout, where training error achieves 0.
 ↪003 and
# test error is about 0.136
```

```
0.10994607584299806
```





[21]:
```
import pickle
results={'y_pred':KS_NN_prediction, 'y_test':KS_test_data, \
         'train_data':KS_input_data, 'train_target':KS_target_data}
f = open('KS_NN.pickle', 'wb')
pickle.dump(results, f, protocol=pickle.HIGHEST_PROTOCOL)
f.close()
```

[ ]:
```

```

RD_NN

June 6, 2020

[ ]:
```
%pylab inline

import numpy as np
import tensorflow as tf

import scipy
from scipy import integrate
from scipy.io import loadmat
from mpl_toolkits.mplot3d import Axes3D

import tensorflow.keras as keras
from tensorflow.keras import optimizers
from tensorflow.keras.models import Model,Sequential,load_model
from tensorflow.keras.layers import Input,Dense,Convolution1D,Activation, LSTM
from tensorflow.keras import backend as K
from tensorflow.keras.utils import plot_model

from IPython.display import clear_output
```

[ ]:
```
class PlotLosses(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.i = 0
        self.x = []
        self.losses = []
        self.val_losses = []

        self.fig = plt.figure()

        self.logs = []

    def on_epoch_end(self, epoch, logs={}):

        self.logs.append(logs)
        self.x.append(self.i)
        self.losses.append(logs.get('loss'))
        self.val_losses.append(logs.get('val_loss'))
        self.i += 1
```

```
        clear_output(wait=True)
        plt.plot(self.x, self.losses, label="loss")
        plt.plot(self.x, self.val_losses, label="val_loss")
        plt.yscale('log')
        plt.legend()
        plt.show();

plot_losses = PlotLosses()
```

[3]:
```
def progress_bar(percent):
    length = 40
    pos = round(length*percent)
    clear_output(wait=True)
    print('['+' '*pos+' '*(length-pos)+']  '+str(int(100*percent))+'%')
```

0.1 Load RD trajectories

Trajectories have been generated in MATLAB for randomly generated initial conditions.

[4]:
```
num_train = 20
num_tests = 1
look_back = 20

N = 128
T = 201
num_iter = num_tests + num_train
RD_all_data = np.zeros((num_train,T,N,2*N))
#RD_input_data = np.zeros(((T-look_back)*(num_train), look_back, N, 2*N))
#RD_target_data = np.zeros(((T-look_back)*(num_train),N,2*N))

for i in range(num_train):
    d = loadmat('PDECODES/RD_data/N'+str(N)+'/iter'+str(i+1)+'.mat')
    u = d['u']
    v = d['v']
    RD_all_data[i,:,:,:N] = u[:,:,:].T # u(x,y,t) -> (t,y,x)
    RD_all_data[i,:,:,N:] = v[:,:,:].T
#    for l in range(look_back):
#        RD_input_data[i*(T-look_back):(i+1)*(T-look_back),l] = RD_all_data[i,l:
 ↪-(look_back-l)] #(t,y,x)
#    RD_target_data[i*(T-look_back):(i+1)*(T-look_back),:,:] =␣
 ↪RD_all_data[i,look_back:,:,:]
```

[5]:
```
RD_test_data = np.zeros((T*num_tests,N,2*N))
for i in range(num_tests):
    d = loadmat('PDECODES/RD_data/N'+str(N)+'/iter'+str(num_iter-i)+'.mat')
```

```
    u = d['u']
    v = d['v']
    RD_test_data[i*T:(i+1)*T,:,:N] = u.T
    RD_test_data[i*T:(i+1)*T,:,N:] = v.T
```

[ ]:
```
#model = Sequential()
#model.add(LSTM(2*N*N, activation="tanh", recurrent_activation="sigmoid"))
#model.add(Dense(N*N))
```

[ ]:
```
# # Again use Adam
# adam_ = keras.optimizers.Adam(lr=.02, beta_1=0.9, beta_2=0.999, epsilon=None,␣
 ↪decay=1e-4, amsgrad=True, clipvalue=0.5)
# model.compile(loss='mean_squared_error', optimizer=adam_,␣
 ↪metrics=['accuracy'])
```

[ ]:
```
#mpl.rcParams['text.usetex'] = False
#model.fit(
#    np.reshape(RD_input_data,(RD_input_data.shape[0],RD_input_data.
 ↪shape[1],N*2*N)),
#    np.reshape(RD_target_data,(-1,N*2*N)),
#    epochs=100, batch_size=800, shuffle=True, callbacks=[plot_losses],␣
 ↪validation_split=0.0)
```

[ ]:
```
#RD_NN_prediction = np.zeros(np.reshape(RD_test_data[0:T],(-1,N*2*N)).shape)
#RD_NN_prediction[0:look_back] = np.reshape(RD_test_data[0:
 ↪look_back],(-1,N*2*N))
#for k in range(T-look_back):
#    RD_NN_prediction[k+look_back] = model.predict(np.array([RD_NN_prediction[k:
 ↪k+look_back]]))
```

0.2 Compute SVD

Training on full data took too long because the data dimension is high, so train on reduced data instead. Reshape data and compute rank k approximation to find fixed subspace to which we project our spatial ponints at each time.

[6]:
```
RD_all_data.shape #(num_train,T,N,2*N)
```

[6]:
```
(20, 201, 128, 256)
```

[7]:
```
RD_all_data_reshaped = np.reshape(RD_all_data[:,:,:,],(num_train*T,2*N*N)).T #␣
 ↪A: (2*N*N, num_train*T)
```

[8]:
```
np.shape(RD_all_data_reshaped)
```

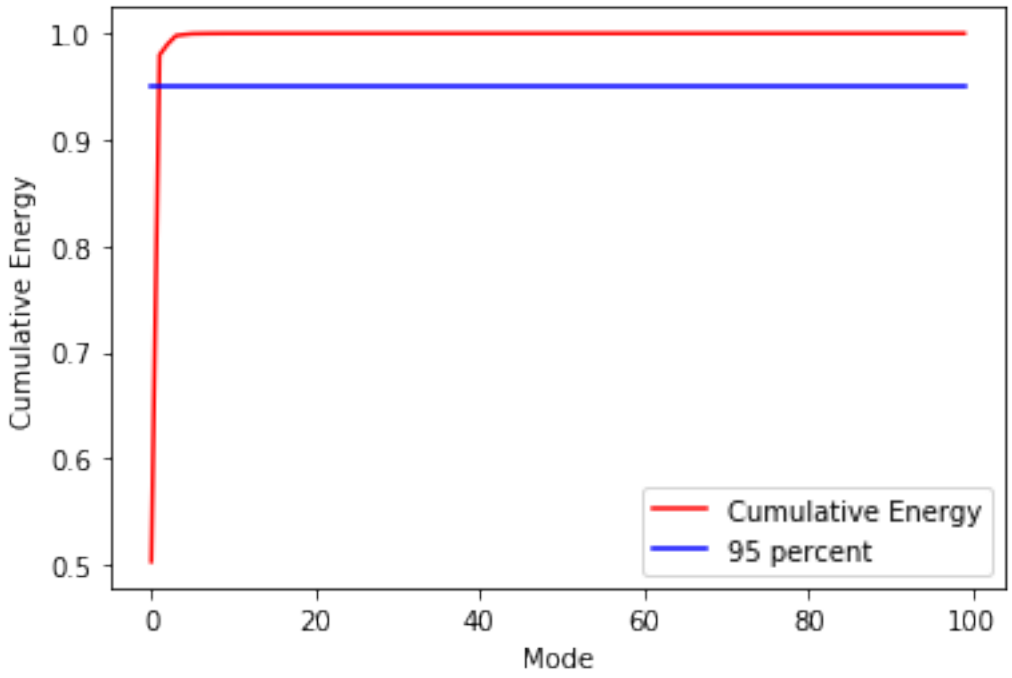
[8]:
```
(32768, 4020)
```

[9]:
```
RD_all_data_reshaped = np.concatenate((RD_all_data_reshaped, np.
↪reshape(RD_test_data[0:T],(-1,2*N*N)).T), axis=1)
```

[10]:
```
# note in python, svd returns V.T
[uu,ss,vv] = scipy.linalg.svd(RD_all_data_reshaped,full_matrices=False)
```

[11]:
```
Ek=ss**2;
Hk=Ek/np.sum(Ek);
plt.plot(np.array(range(len(ss)))[0:100], np.cumsum(Hk[0:100]), 'r',␣
↪label='Cumulative Energy')
plt.plot(np.array(range(len(ss)))[0:100], 0.95*np.ones_like(Hk[0:100]), 'b',␣
↪label='95 percent')
plt.xlabel('Mode')
plt.ylabel('Cumulative Energy')
plt.legend()
```

[11]:
```
<matplotlib.legend.Legend at 0x7fbde3a1ad30>
```



[12]:
```
# Get reduced SVD
rank = 100
u_ = uu[:,:rank]
sig = ss[:rank]
v_ = vv[:rank]  #(rank, num_train*T)
```

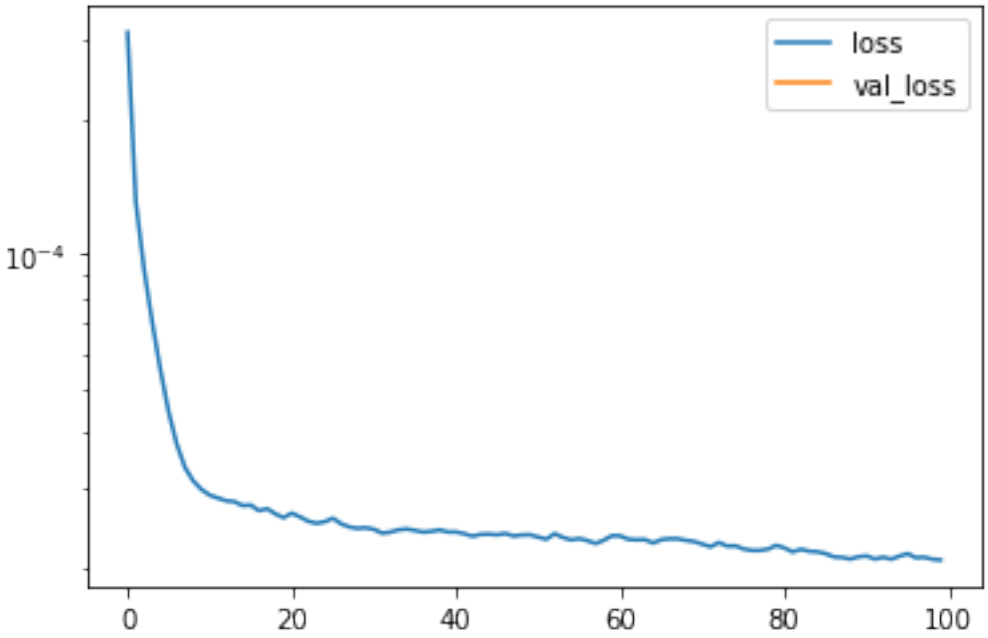
[13]:
```
SVD_all_data = np.reshape(v_, (rank, num_iter, T))
SVD_input_data = np.zeros(((T-look_back)*(num_train), look_back, rank))
SVD_target_data = np.zeros(((T-look_back)*(num_train),rank))

for i in range(num_train):
    for l in range(look_back):
        SVD_input_data[i*(T-look_back):(i+1)*(T-look_back),l,:] = SVD_all_data[:
 ↪,i,l:-(look_back-l)].T
    SVD_target_data[i*(T-look_back):(i+1)*(T-look_back),:] = SVD_all_data[:
 ↪,i,look_back:].T
```

0.3 Train Net on SVD data

[14]:
```
model = Sequential()
model.add(LSTM(rank, activation="tanh", recurrent_activation="sigmoid",␣
 ↪dropout=0.4, recurrent_dropout=0.4))
model.add(Dense(rank))
```

[15]:
```
# Again use Adam
adam_ = keras.optimizers.Adam(lr=.02, beta_1=0.9, beta_2=0.999, epsilon=None,␣
 ↪decay=1e-4, amsgrad=True, clipvalue=0.5)
model.compile(loss='mean_squared_error', optimizer=adam_, metrics=['accuracy'])
```

[16]:
```
mpl.rcParams['text.usetex'] = False
model.fit(
    SVD_input_data,
    SVD_target_data,
    epochs=100, batch_size=500, shuffle=True, callbacks=[plot_losses],␣
 ↪validation_split=0.0)
```




```
3620/3620 [==============================] - 1s 205us/step - loss: 2.0813e-05 -
acc: 0.5373
```

[16]:
```
<tensorflow.python.keras.callbacks.History at 0x7fbde39fef60>
```

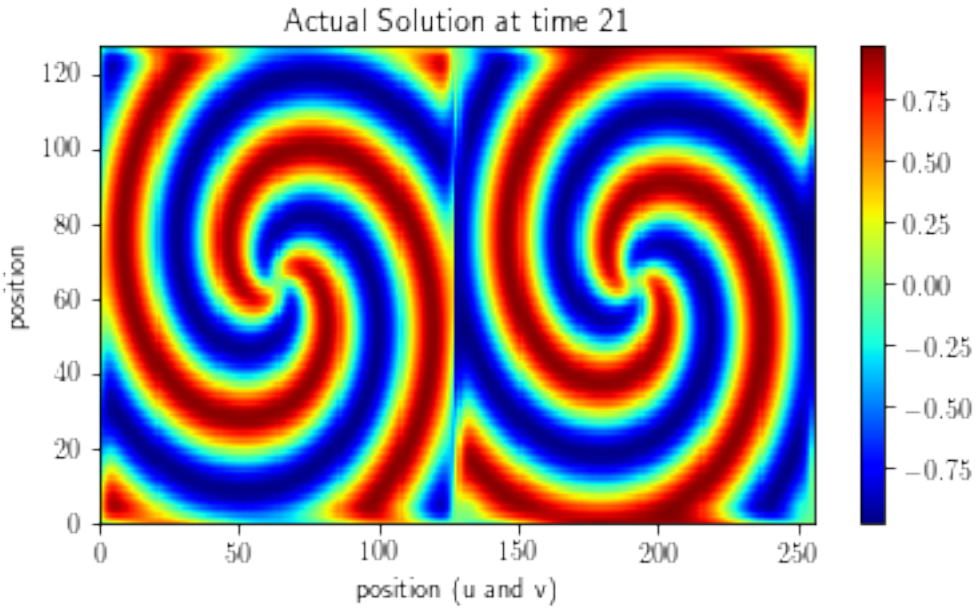
[17]:
```
# Project test data on the PC space
SVD_test_data = np.dot(np.dot(np.reshape(RD_test_data[0:T],(-1,2*N*N)), u_), np.
↪linalg.inv(np.diag(sig))) #(T,rank), A.T U
#SVD_test_data = u_.T
```

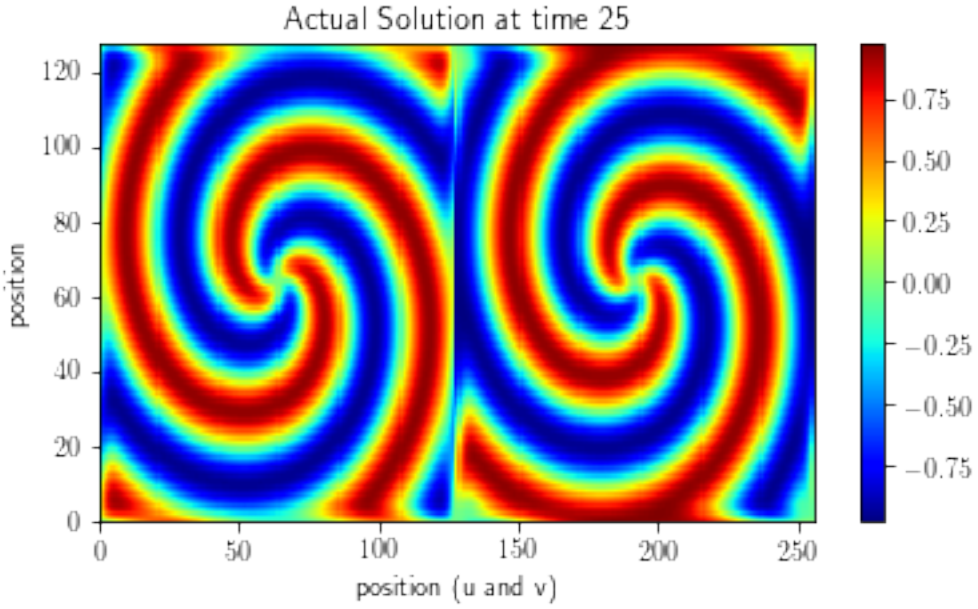
[18]:
```
SVD_NN_prediction = np.zeros(SVD_test_data.shape)
SVD_NN_prediction[0:look_back] = SVD_test_data[0:look_back]
for k in range(T-look_back):
    pred_input = np.reshape(SVD_NN_prediction[k:k+look_back], (look_back, rank))
    SVD_NN_prediction[k+look_back] = model.predict(np.array([pred_input]))
```

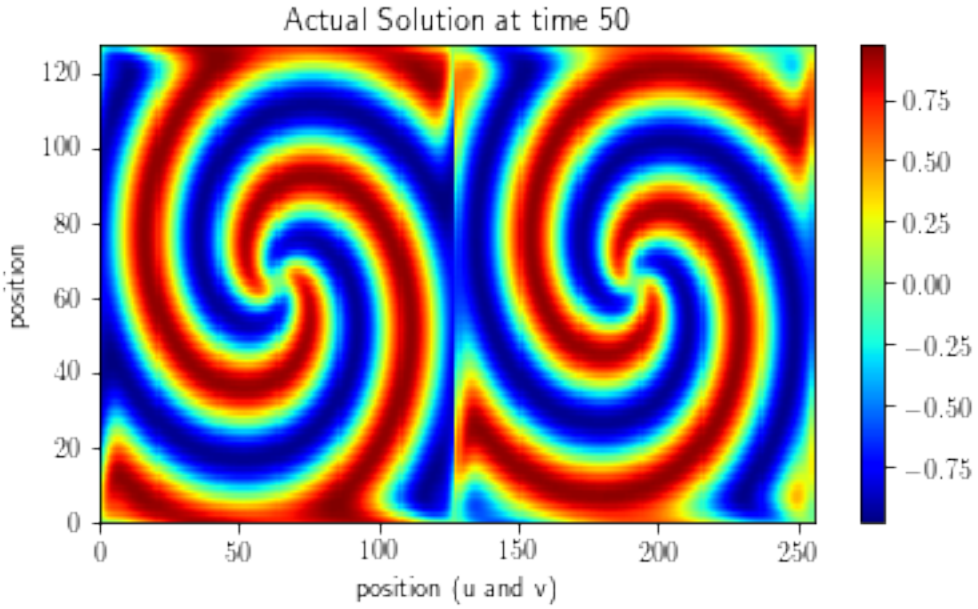
[19]:
```
# Project back
RD_NN_prediction = np.dot(np.dot(SVD_NN_prediction, np.diag(sig)), u_.T)
```

[20]:
```
# Plot the test data at several sample times
tlist = [21, 25, 50]
for ii in range(len(tlist)):
    tt=tlist[ii]

    mpl.rcParams['text.usetex'] = True
    plt.figure(figsize=(6,3.3))
```

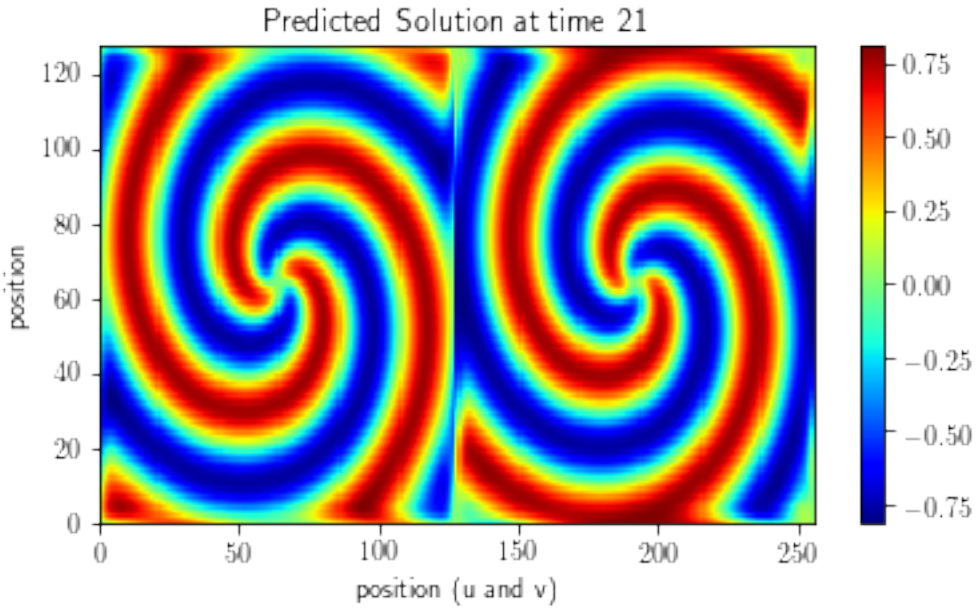
```
m = plt.pcolor(np.reshape(RD_test_data[tt-1],(N,2*N)), cmap='jet')
m.set_rasterized(True)
plt.xlabel('position (u and v)')
plt.ylabel('position')
plt.colorbar()
plt.title('Actual Solution at time ' + str(tt))
```

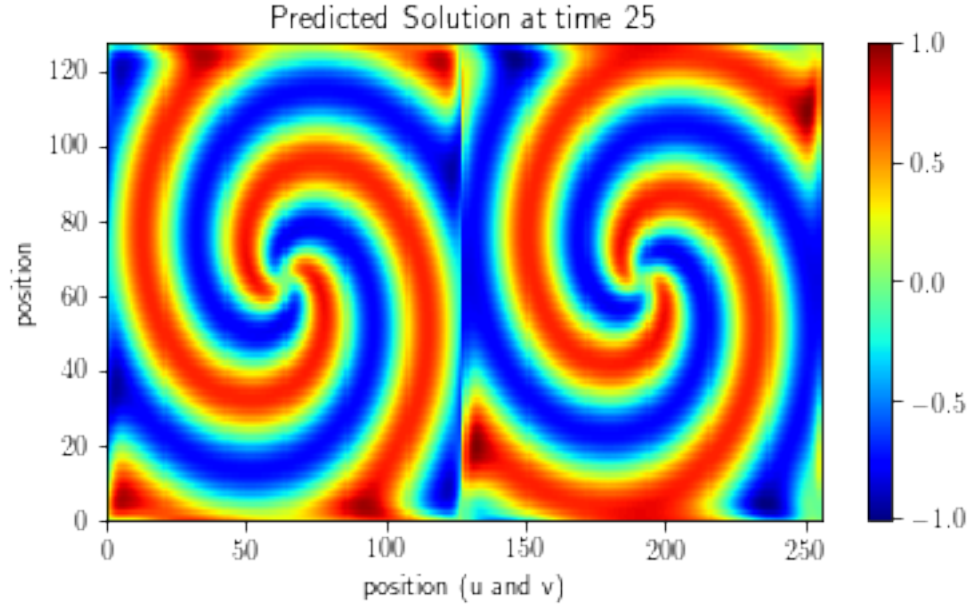


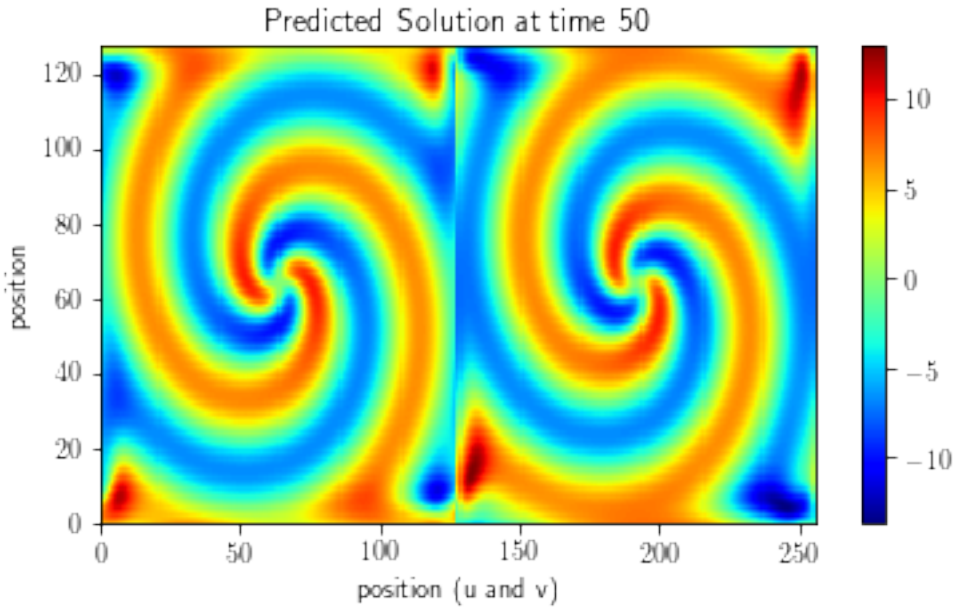




[21]:
```
# plot the predicted test data at these sample times
tlist = [21, 25, 50]
for ii in range(len(tlist)):
    tt=tlist[ii]

    mpl.rcParams['text.usetex'] = True
    plt.figure(figsize=(6,3.3))
    m = plt.pcolor(np.reshape(RD_NN_prediction[tt-1],(N,2*N)), cmap='jet')
    m.set_rasterized(True)
    plt.xlabel('position (u and v)')
    plt.ylabel('position')
    plt.colorbar()
    plt.title('Predicted Solution at time ' + str(tt))
```


Predicted Solution at time 21
position
position (u and v)
0
20
40
60
80
100
120
0
50
100
150
200
250
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75

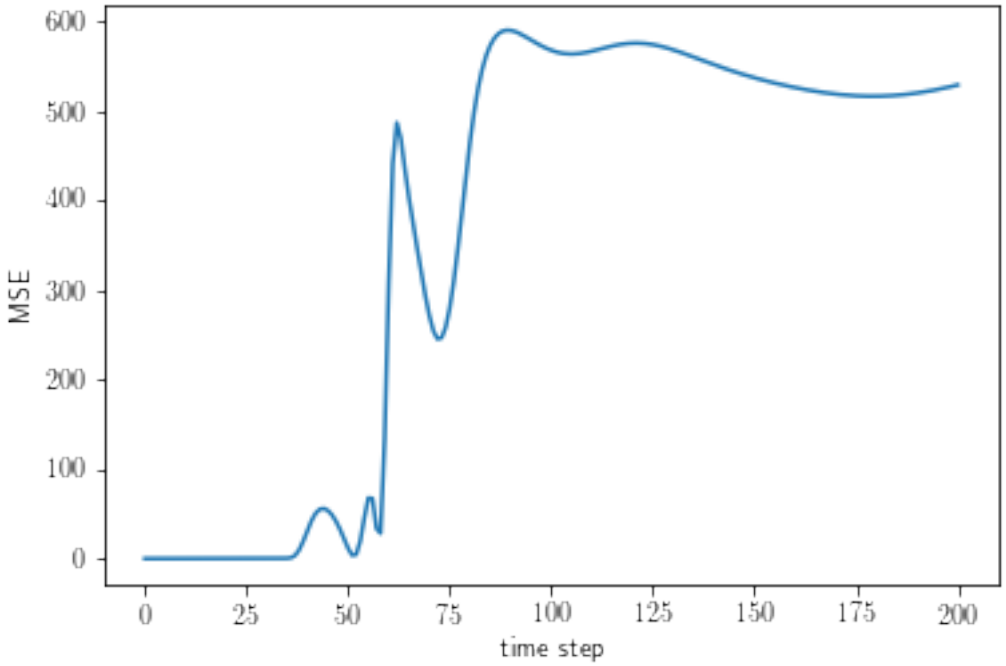

Predicted Solution at time 25
position
position (u and v)
0
20
40
60
80
100
120
0
50
100
150
200
250
1.0
0.5
0.0
−0.5
−1.0

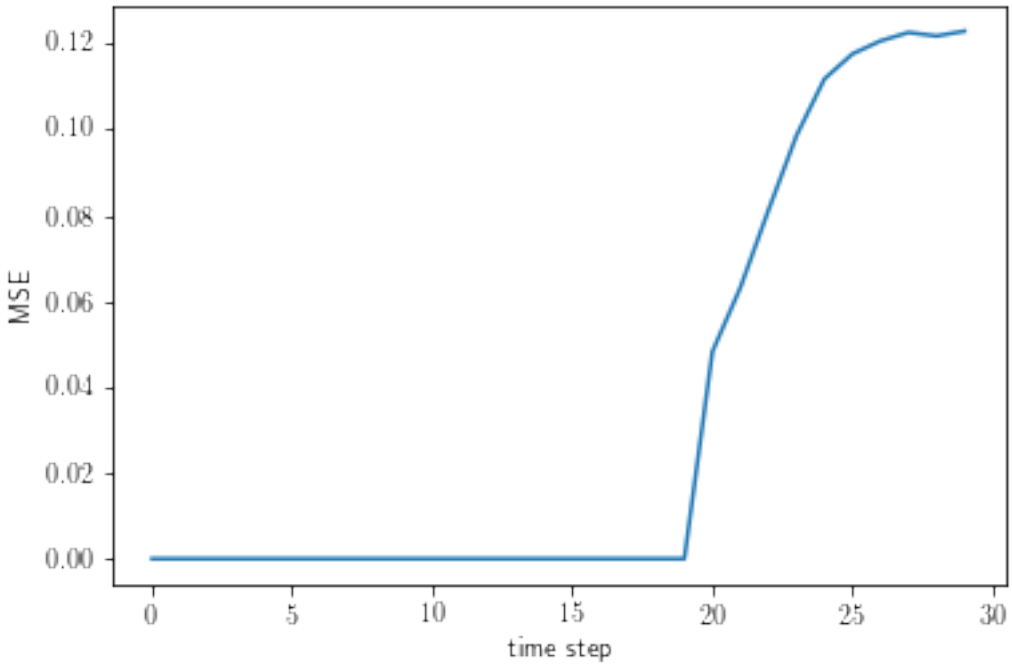


Look at MSE in the high-D space

[22]:
```
mse_total = ((np.reshape(RD_test_data, (-1,2*N*N))-RD_NN_prediction)**2).
 ↪mean(axis=None)
mse_over_step = ((np.reshape(RD_test_data, (-1,2*N*N))-RD_NN_prediction)**2).
 ↪mean(axis=-1)
mse_one_step = mse_over_step[look_back] - mse_over_step[look_back-1]

figure()
plt.plot(np.array(range(T)), mse_over_step)
xlabel('time step')
ylabel('MSE')

figure()
plt.plot(np.array(range(T)[0:30]), mse_over_step[0:30])
xlabel('time step')
ylabel('MSE')

print(mse_one_step)
# Training happens so much faster with SVD
# Even one-step MSE is high relative to the training error, there could still␣
 ↪be overfitting

# One limitation: data generated by random phase offset of spiral, so they␣
 ↪share similar structure
# more work is needed to build a model that can generalize to wider set of␣
 ↪initial conditions
```

```
0.04824127211757698
```

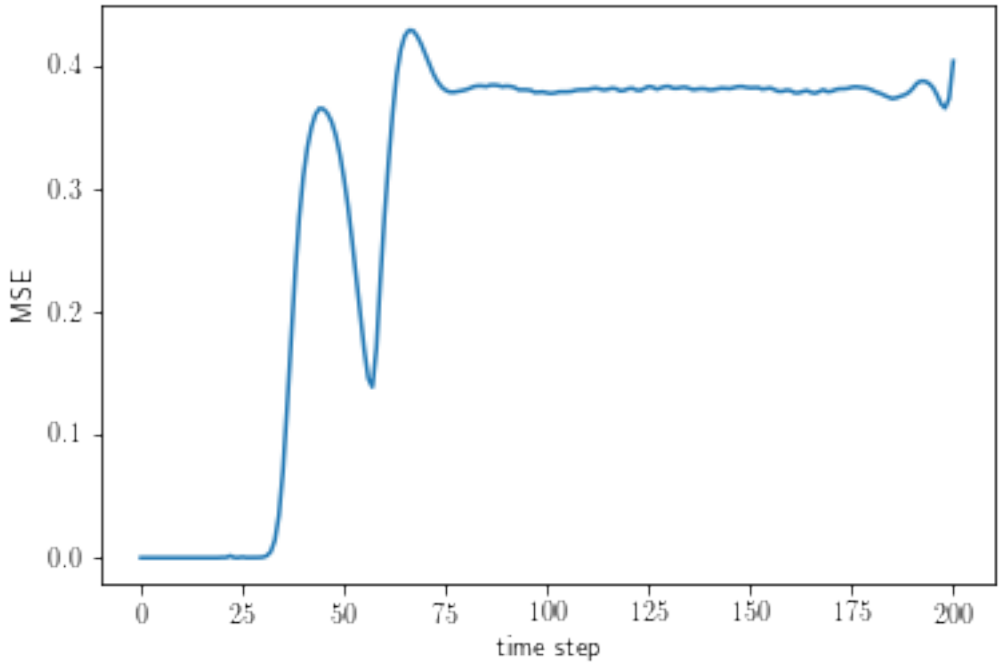




Look at MSE in the low-D space

[24]:
```
mse_total = ((SVD_test_data-SVD_NN_prediction)**2).mean(axis=None)
mse_over_step = ((SVD_test_data-SVD_NN_prediction)**2).mean(axis=-1)
mse_one_step = mse_over_step[look_back] - mse_over_step[look_back-1]

figure()
plt.plot(np.array(range(T)), mse_over_step)
xlabel('time step')
ylabel('MSE')

print(mse_one_step)
```

```
0.00025897533038212237
```



[ ]:

[ ]:

LRZ_NN

June 6, 2020

[26]:
```
%pylab inline

import numpy as np
import tensorflow as tf

from scipy import integrate
from scipy.io import loadmat
from mpl_toolkits.mplot3d import Axes3D

import tensorflow.keras as keras
from tensorflow.keras import optimizers
from tensorflow.keras.models import Model,Sequential,load_model
from tensorflow.keras.layers import Input,Dense,Convolution1D,Activation
from tensorflow.keras.layers import LSTM
from tensorflow.keras import backend as K
from tensorflow.keras.utils import plot_model

from IPython.display import clear_output

import pickle
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
/home/hyliu24/anaconda3/lib/python3.6/site-
packages/IPython/core/magics/pylab.py:160: UserWarning: pylab import has
clobbered these variables: ['f']
`%matplotlib` prevents importing * from pylab and numpy
  "\n`%matplotlib` prevents importing * from pylab and numpy"
```

[27]:
```
class PlotLosses(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.i = 0
        self.x = []
        self.losses = []
        self.val_losses = []

        self.fig = plt.figure()
```

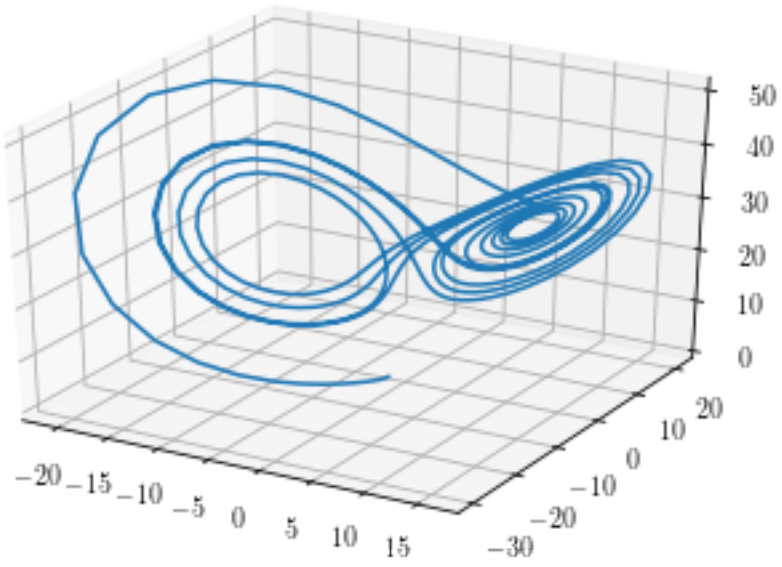
```
        self.logs = []

    def on_epoch_end(self, epoch, logs={}):

        self.logs.append(logs)
        self.x.append(self.i)
        self.losses.append(logs.get('loss'))
        self.val_losses.append(logs.get('val_loss'))
        self.i += 1

        clear_output(wait=True)
        plt.plot(self.x, self.losses, label="loss")
        plt.plot(self.x, self.val_losses, label="val_loss")
        plt.yscale('log')
        plt.legend()
        plt.show();

plot_losses = PlotLosses()
```

[28]:
```
def progress_bar(percent):
    length = 40
    pos = round(length*percent)
    clear_output(wait=True)
    print('['+' '*pos+' '*(length-pos)+']  '+str(int(100*percent))+'%')
```

[29]:
```
def lrz_rhs(t,x,sigma,beta,rho):
    return [sigma*(x[1]-x[0]), x[0]*(rho-x[2]), x[0]*x[1]-beta*x[2]];
```

[58]:
```
end_time = 10
dt = 0.02

T = int(end_time/dt)+1
t = np.linspace(0,end_time,T,endpoint=True)
def lrz_trajectory(rho):
    sigma=10;
    beta=8/3;
    x0 = 5*(np.random.rand(3)-.5)
    sol = integrate.solve_ivp(lambda t,x:␣
 ↪lrz_rhs(t,x,sigma,beta,rho),[0,end_time],x0,t_eval=t,rtol=1e-10,atol=1e-11)
    return sol.y
```

[59]:
```
y = lrz_trajectory(28)
plt.figure()
plt.gca(projection='3d')
plt.plot(y[0],y[1],y[2])
plt.show()
```



0.1 Generate Training Data

$\rho = 10, 28, 40$

[60]:
```
look_back = 15 # number of steps in the past to consider for decision
N = 200

rhos = [10,28,40]
input_data = np.zeros((N*(T-look_back)*len(rhos), look_back, 4))
target_data = np.zeros((N*(T-look_back)*len(rhos), 3))
for k,rho in enumerate(rhos):
    for i in range(N):
        progress_bar((N*k+i+1)/(N*len(rhos)))
        trajectory = lrz_trajectory(rho)
        for l in range(look_back):
            input_data[((len(rhos)-1)*k+i)*(T-look_back):
↪((len(rhos)-1)*k+i+1)*(T-look_back),l,:3] = \
                trajectory.T[l:-(look_back-l)]
            input_data[((len(rhos)-1)*k+i)*(T-look_back):
↪((len(rhos)-1)*k+i+1)*(T-look_back),l,3] = rho
        target_data[((len(rhos)-1)*k+i)*(T-look_back):
↪((len(rhos)-1)*k+i+1)*(T-look_back),:3] = \
            trajectory.T[look_back:]
```

```
[                    ]  100%
```

[61]:
```
input_data.shape
```

[61]:
```
(291600, 15, 4)
```

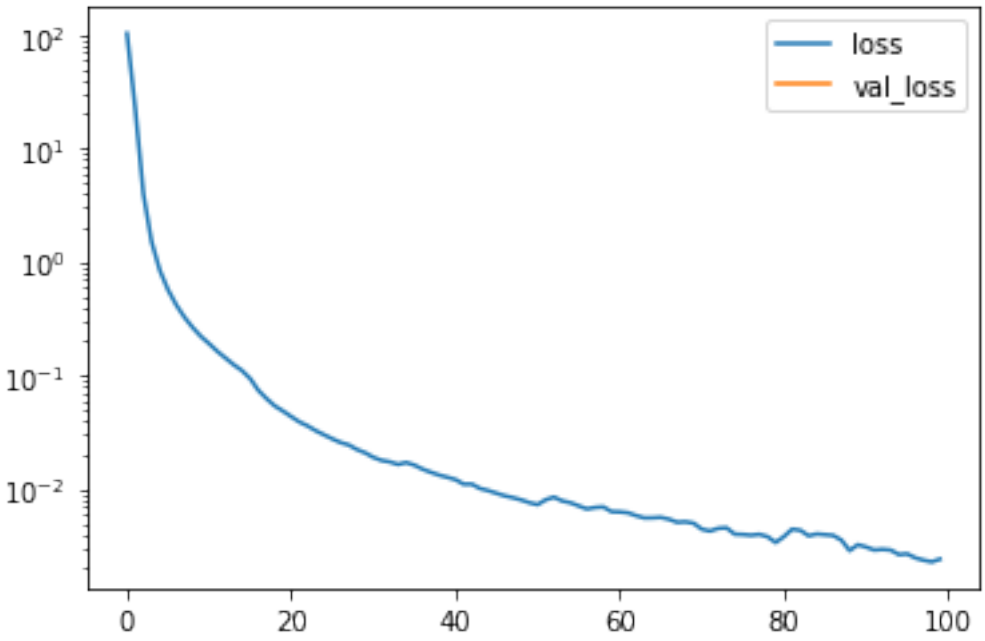
[62]:
```
train_data={'rhos':rho, 'look_back':look_back, 'N':N, \
         'train_data':input_data, 'train_target':target_data}
f = open('LRZ_trajectories.pickle', 'wb')
pickle.dump(train_data, f, protocol=pickle.HIGHEST_PROTOCOL)
f.close()
```

0.2 Train Neural Network

[63]:
```
model = Sequential()
model.add(LSTM(3*look_back, activation="tanh", recurrent_activation="sigmoid", \
               dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(3))
```

[64]:
```
adam_ = optimizers.Adam(lr=.005, beta_1=0.9, beta_2=0.999, epsilon=None,␣
↪decay=1e-5, amsgrad=True, clipvalue=0.5)
model.compile(loss='mean_squared_error', optimizer=adam_, metrics=['accuracy'])
```

[65]:
```
mpl.rcParams['text.usetex'] = False
model.fit(
    input_data, target_data,
    epochs=100, batch_size=3000, shuffle=True, callbacks=[plot_losses],␣
↪validation_split=0.0)
```

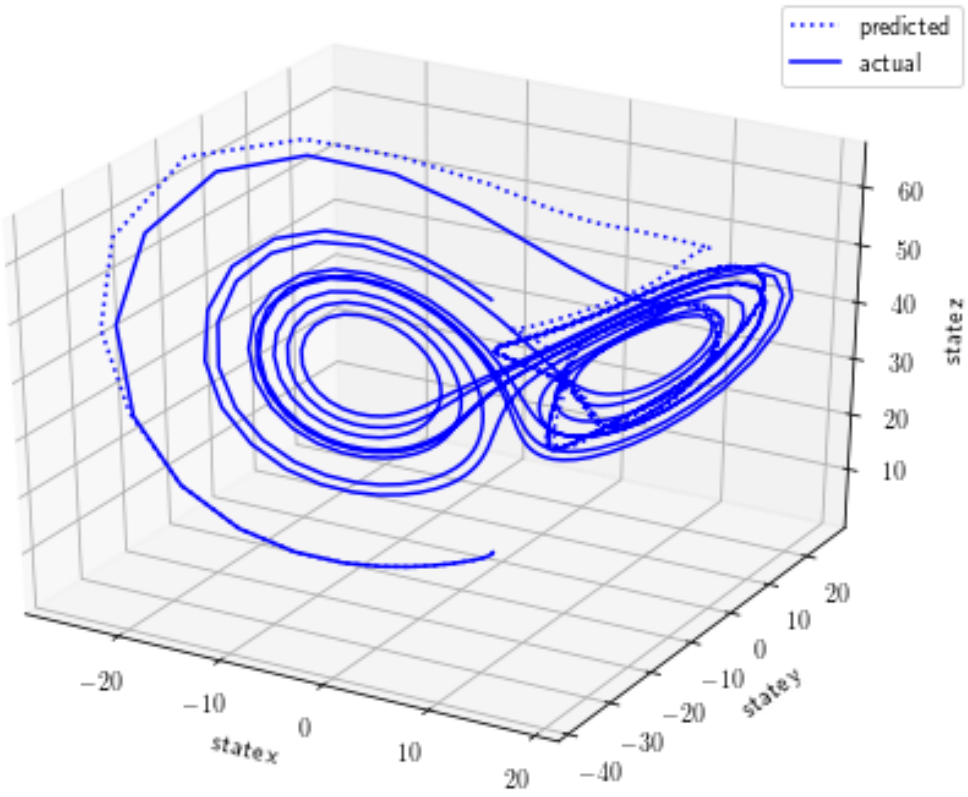


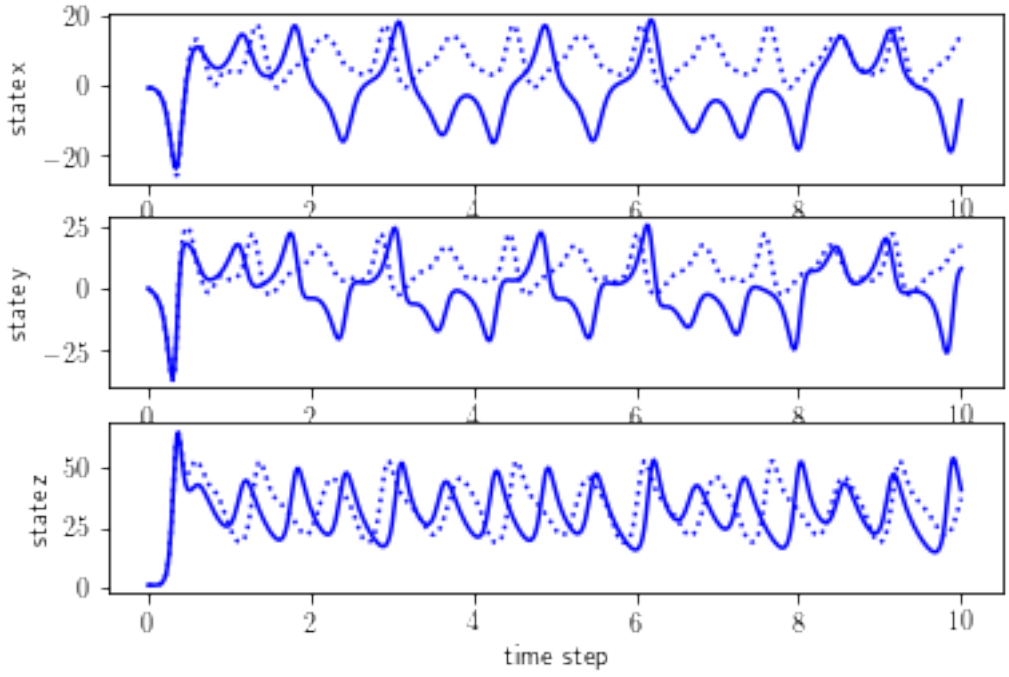
```
291600/291600 [==============================] - 5s 17us/step - loss: 0.0024 -
acc: 0.6387
```

[65]:
```
<tensorflow.python.keras.callbacks.History at 0x7fd5e7731f98>
```

[66]:
```
rho=35
x35 = lrz_trajectory(rho)
x35_pred = np.zeros_like(x35).T
x35_pred[0:look_back,:] = x35.T[0:look_back,:]
for k in range(T-look_back):
    pred_input = np.concatenate((x35_pred[k:k+look_back], rho*np.
 ↪ones((look_back,1))), axis=1)
    x35_pred[k+look_back] = model.predict(np.array([pred_input]))
x35_pred = x35_pred.T
```

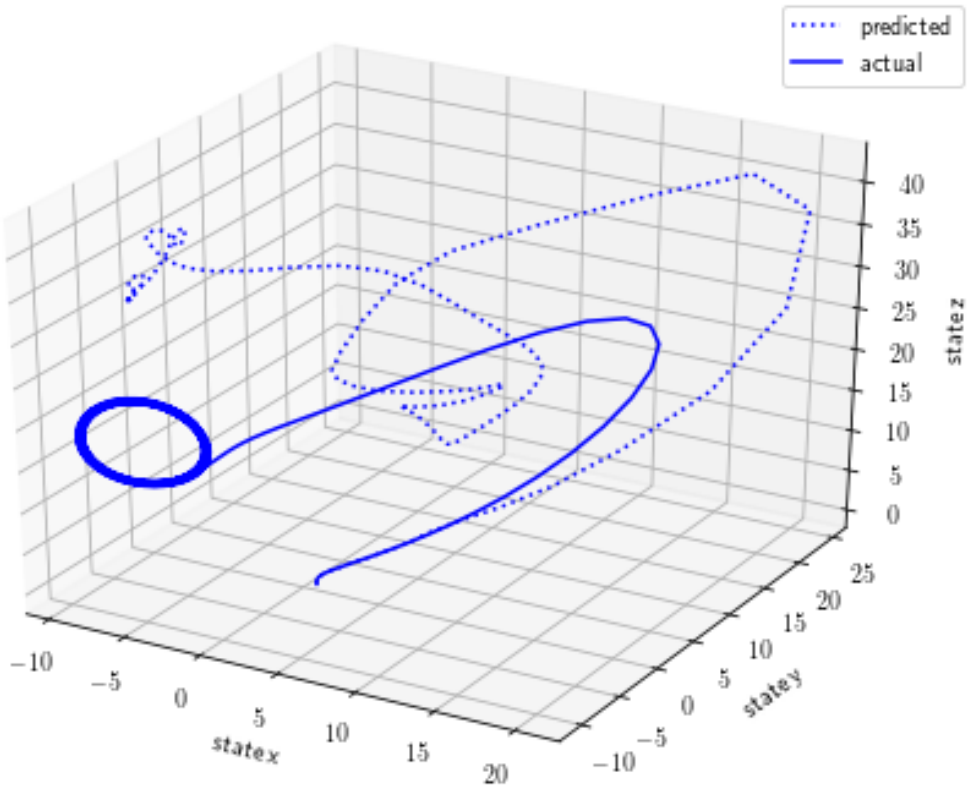
[67]:
```
mpl.rcParams['text.usetex'] = True
plt.figure(num=None, figsize=(8, 6))
ax = plt.gca(projection='3d')
plt.plot(x35_pred[0],x35_pred[1],x35_pred[2], color='b', linestyle=':
 ↪',label='predicted')
plt.plot(x35[0],x35[1],x35[2], color='b',label='actual')
ax.set_xlabel('state x')
ax.set_ylabel('state y')
ax.set_zlabel('state z')
plt.legend()

state_labels = ['state x','state y','state z']
plt.figure()
for i in range(3):
    plt.subplot(3,1,i+1)
    plt.plot(t,x35_pred[i], color='b', linestyle=':')
    plt.plot(t,x35[i], color='b')
    plt.xlabel('time step')
    plt.ylabel(state_labels[i])
```

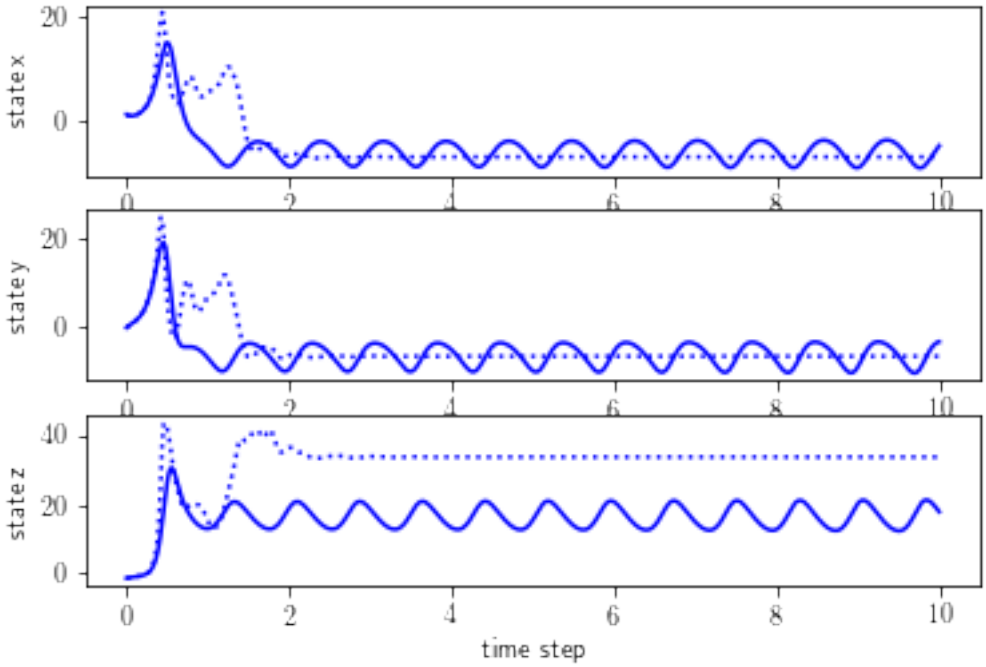

predicted
actual
state x
state y
state z


state x
state y
state z
time step

[68]:
```
rho=17
x17 = lrz_trajectory(rho)
x17_pred = np.zeros_like(x17).T
x17_pred[0:look_back,:] = x17.T[0:look_back,:]
for k in range(T-look_back):
    pred_input = np.concatenate((x17_pred[k:k+look_back], rho*np.
 ↪ones((look_back,1))), axis=1)
    x17_pred[k+look_back] = model.predict(np.array([pred_input]))
x17_pred = x17_pred.T
```

[69]:
```
mpl.rcParams['text.usetex'] = True
plt.figure(num=None, figsize=(8, 6))
ax = plt.gca(projection='3d')
plt.plot(x17_pred[0],x17_pred[1],x17_pred[2], color='b', linestyle=':
 ↪',label='predicted')
plt.plot(x17[0],x17[1],x17[2], color='b',label='actual')
ax.set_xlabel('state x')
ax.set_ylabel('state y')
ax.set_zlabel('state z')
plt.legend()

state_labels = ['state x','state y','state z']
plt.figure()
for i in range(3):
    plt.subplot(3,1,i+1)
    plt.plot(t,x17_pred[i], color='b', linestyle=':')
    plt.plot(t,x17[i], color='b')
    plt.xlabel('time step')
    plt.ylabel(state_labels[i])
```


predicted
actual
statex
statey
statez


statex
statey
statez
time step

[ ]:

LRZ_classify

June 6, 2020

[1]:
```
%pylab inline

import numpy as np
import tensorflow as tf

from scipy import integrate
from scipy.io import loadmat
from mpl_toolkits.mplot3d import Axes3D

import tensorflow.keras as keras
from tensorflow.keras import optimizers
from tensorflow.keras.models import Model,Sequential,load_model
from tensorflow.keras.layers import Input,Dense,Convolution1D,Activation
from tensorflow.keras.layers import LSTM
from tensorflow.keras import backend as K
from tensorflow.keras.utils import plot_model

from IPython.display import clear_output

import pickle
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
/home/hyliu24/anaconda3/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:523: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/home/hyliu24/anaconda3/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:524: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/home/hyliu24/anaconda3/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:525: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
```

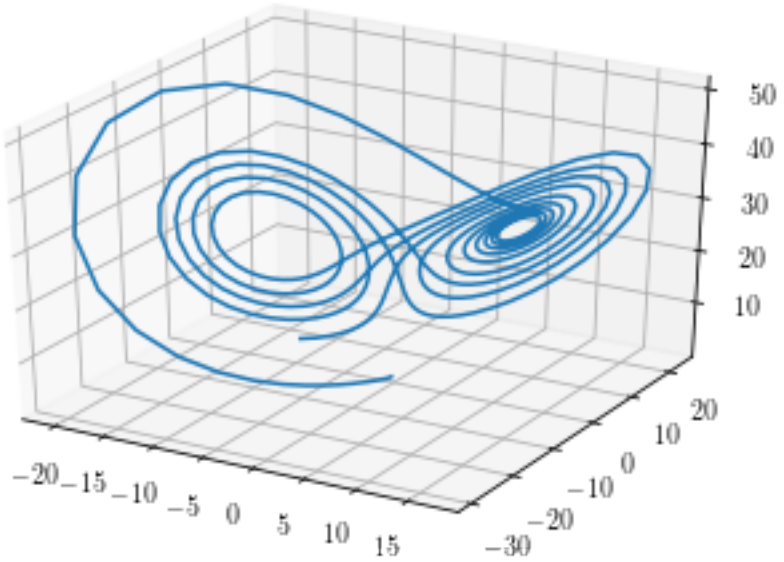
```
/home/hyliu24/anaconda3/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:526: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/home/hyliu24/anaconda3/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:527: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/home/hyliu24/anaconda3/lib/python3.6/site-
packages/tensorflow/python/framework/dtypes.py:532: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

[2]:
```
def lrz_rhs(t,x,sigma,beta,rho):
    return [sigma*(x[1]-x[0]), x[0]*(rho-x[2]), x[0]*x[1]-beta*x[2]];
```

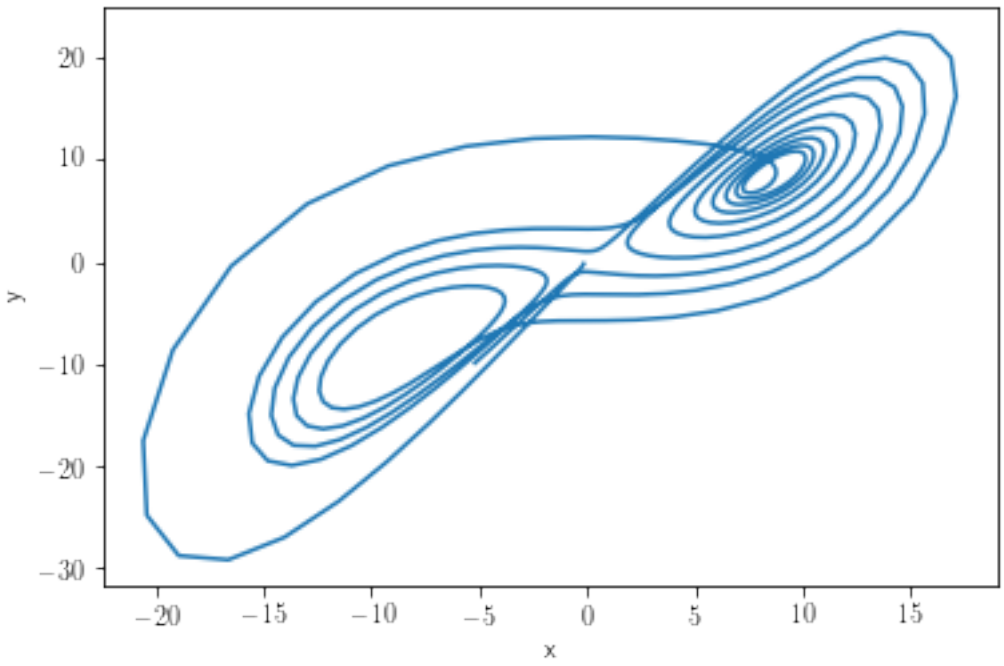
[3]:
```
end_time = 10
dt = 0.02

T = int(end_time/dt)+1
t = np.linspace(0,end_time,T,endpoint=True)
def lrz_trajectory(rho):
    sigma=10;
    beta=8/3;
    x0 = (np.random.rand(3)-.5)
    sol = integrate.solve_ivp(lambda t,x:␣
↪lrz_rhs(t,x,sigma,beta,rho),[0,end_time],x0,t_eval=t,rtol=1e-10,atol=1e-11)
    return sol.y
```

[14]:
```
x = lrz_trajectory(28)
plt.figure()
plt.gca(projection='3d')
plt.plot(x[0],x[1],x[2])
plt.show()

plt.figure()
plt.plot(x[0],x[1])
plt.xlabel('x')
plt.ylabel('y')

#for ii in range(3):
#    plt.figure()
#    plt.plot(t, x[ii])
#    plt.xlabel('time')
```

```
#     plt.ylabel('state')
#     plt.show()
```



[14]: Text(0, 0.5, 'y')



0.1 Lobe Categorization and Data Generation

From the plot above, a simplist way to categorize which lobe a point is in is too see if x is greater or less than 0.

This could be a classification or regression problem. For regression, the problem can be formulated as determining the time until the next transition. For classification, the problem can be formulated as determining if the next transition is within a specified time window. For this project, I choose the latter.

[4]:
```
def trajectory_with_transition(rho, time_window):
    # define normal vector
    c = np.array([1,0,0])

    # get trajectory
    x = lrz_trajectory(rho)

    # Lobe categorization
    classes = np.sign(np.dot(c,x))
    # find indices of transition, i.e. where there is a change between classes
    transition_ind=np.where(np.convolve(classes,[1,-1],mode='valid')!=0)

    # this will be the time it takes to the next jump
    time_to_jump = np.zeros((len(x.T)))
    time_to_jump[transition_ind] = np.ones(len(transition_ind))    # ones for␣
↪there are jumps

    # count backwards from ones
    current_time_to_jump = 0
    jumping = False
    for j in range(len(time_to_jump)):
        if time_to_jump[-j]==1:
            current_time_to_jump=1
            jumping=True
        elif jumping:
            time_to_jump[-j]=current_time_to_jump
            current_time_to_jump+=1

    # delete the end of the data, where we don't know when the next transition␣
↪is
    valid_ind = np.where(time_to_jump!=0)[0]
    time_to_jump = time_to_jump[valid_ind]
    x = x[:,valid_ind]

    # Label if jump is imminent, i.e. within the specified time_window
    jump_imminent = np.where(time_to_jump <= time_window, 1, 0)

    # put data together for output
```

```
    transition_dict = {'x':x, 'jump_imminent':jump_imminent}
    return transition_dict
```

[5]:
```
time_window = 20
look_back = 15

max_iter = 100
for i in range(max_iter):
    trans_dict = trajectory_with_transition(28, time_window)
    x_data = trans_dict['x']

    delayed_data = np.expand_dims(x_data[:,0:-(look_back-0+1)].T, axis=1) #␣
↪(T-look-back, 1, 3)
    for l in range(1, look_back):
        delayed_data = np.concatenate((delayed_data, \
                            np.expand_dims(x_data[:,l:-(look_back-l+1)].T,␣
↪axis=1)),axis=1)
        if l == (look_back-1):
            if i == 0:
                target_data = trans_dict['jump_imminent'][l:-(look_back-l+1)]
            else:
                target_data = np.concatenate((target_data,␣
↪trans_dict['jump_imminent'][l:-(look_back-l+1)]))
    if i == 0:
        input_data = delayed_data
    else:
        input_data = np.concatenate((input_data, delayed_data),axis=0)
```

0.2 Set up Neural Network

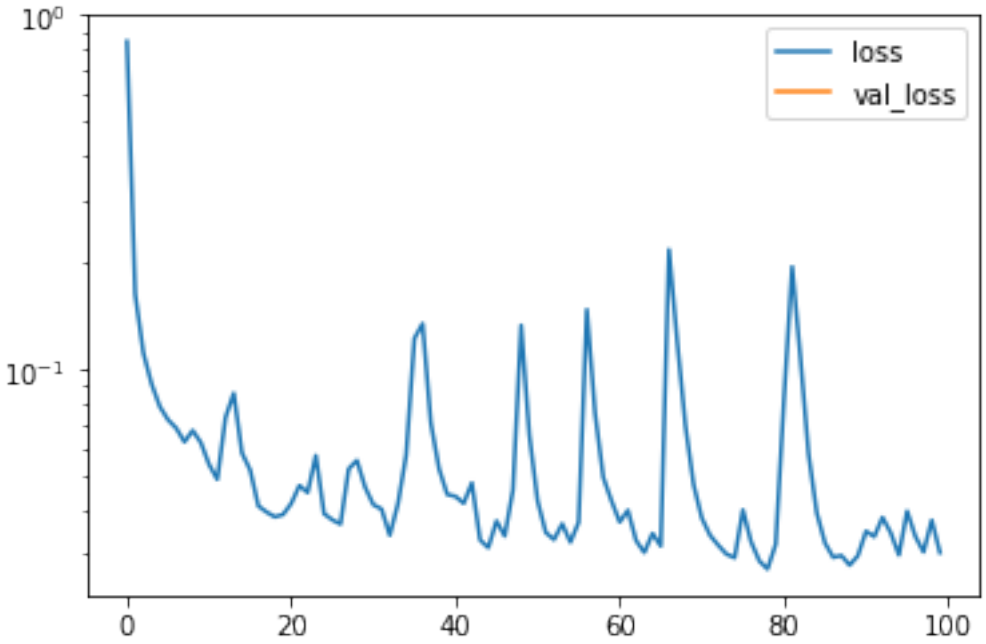
[6]:
```
class PlotLosses(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.i = 0
        self.x = []
        self.losses = []
        self.val_losses = []

        self.fig = plt.figure()

        self.logs = []

    def on_epoch_end(self, epoch, logs={}):

        self.logs.append(logs)
        self.x.append(self.i)
```

```
        self.losses.append(logs.get('loss'))
        self.val_losses.append(logs.get('val_loss'))
        self.i += 1

        clear_output(wait=True)
        plt.plot(self.x, self.losses, label="loss")
        plt.plot(self.x, self.val_losses, label="val_loss")
        plt.yscale('log')
        plt.legend()
        plt.show();

plot_losses = PlotLosses()
```

[7]:
```
model = Sequential()
model.add(LSTM(3*look_back, activation="tanh", recurrent_activation="sigmoid", \
               dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1))
```

[8]:
```
adam_ = optimizers.Adam(lr=.005, beta_1=0.9, beta_2=0.999, epsilon=None,␣
↪decay=1e-5, amsgrad=True, clipvalue=0.5)
model.compile(loss='binary_crossentropy', optimizer=adam_, metrics=['accuracy'])
```

[9]:
```
mpl.rcParams['text.usetex'] = False
model.fit(input_data, target_data,
          epochs=100, batch_size=3000, shuffle=True, callbacks=[plot_losses],␣
↪validation_split=0.0)
```



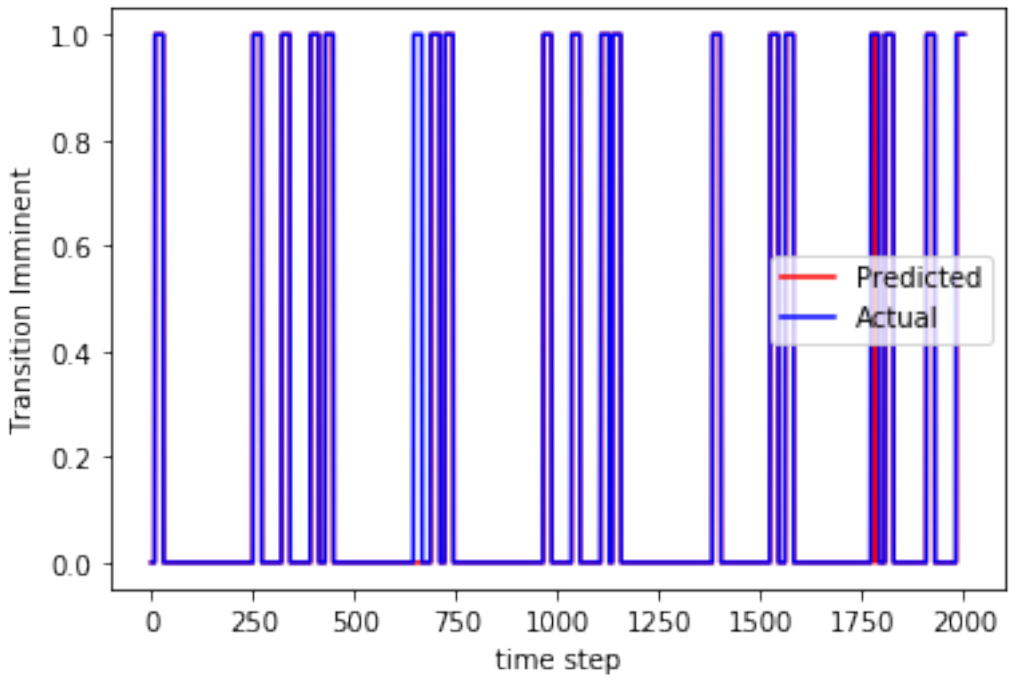
```
43334/43334 [==============================] - 1s 26us/step - loss: 0.0304 -
acc: 0.9924
```

[9]:
```
<tensorflow.python.keras.callbacks.History at 0x7fdbe9045d30>
```

[10]:
```
num_tests = 5

for i in range(num_tests):
    test_dict = trajectory_with_transition(28, time_window)
    x_test = test_dict['x']
    if i==0:
        trans_actual = test_dict['jump_imminent']
    else:
        trans_actual = np.concatenate((trans_actual,␣
 ↪test_dict['jump_imminent']))

    trans_pred_ = np.zeros_like(test_dict['jump_imminent'])
    trans_pred_[0:look_back] = test_dict['jump_imminent'][0:look_back]
    for k in range(len(trans_pred_)-look_back+1):
        pred_input = x_test[:, k:k+look_back].T
        trans_pred_[k+look_back-1] = model.predict(np.array([pred_input]))
    if i==0:
        trans_pred = trans_pred_
    else:
        trans_pred = np.concatenate((trans_pred, trans_pred_))
trans_pred = np.where(trans_pred>0.5, 1, 0)
```

[12]:
```
plt.figure()
plt.plot(trans_pred, 'r',label='Predicted')
plt.plot(trans_actual,'b',label='Actual')
plt.xlabel('time step')
plt.ylabel('Transition Imminent')
plt.legend()

# error rate
print('Error rate: ')
print(np.mean(np.abs(trans_pred-trans_actual)))
```

```
Error rate:
0.019470793809286072
```

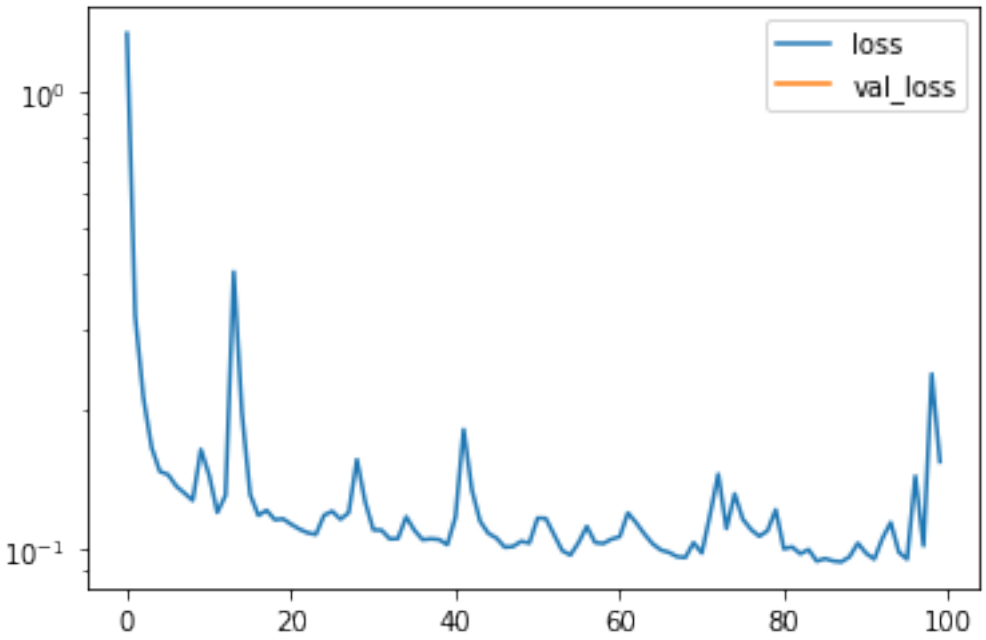


0.2.1 Try a different time_window

[119]:
```
time_window = 40
look_back = 15

max_iter = 100
for i in range(max_iter):
    trans_dict = trajectory_with_transition(28, time_window)
    x_data = trans_dict['x']

    delayed_data = np.expand_dims(x_data[:,0:-(look_back-0+1)].T, axis=1) #␣
↪(T-look-back, 1, 3)
    for l in range(1, look_back):
        delayed_data = np.concatenate((delayed_data, \
                            np.expand_dims(x_data[:,l:-(look_back-l+1)].T,␣
↪axis=1)),axis=1)
        if l == (look_back-1):
            if i == 0:
                target_data = trans_dict['jump_imminent'][l:-(look_back-l+1)]
            else:
                target_data = np.concatenate((target_data,␣
↪trans_dict['jump_imminent'][l:-(look_back-l+1)]))
    if i == 0:
```

```
        input_data = delayed_data
    else:
        input_data = np.concatenate((input_data, delayed_data),axis=0)
```

[125]:
```
model = Sequential()
model.add(LSTM(3*look_back, activation="tanh", recurrent_activation="sigmoid", \
               dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1))

adam_ = optimizers.Adam(lr=.005, beta_1=0.9, beta_2=0.999, epsilon=None,␣
↪decay=1e-5, amsgrad=True, clipvalue=0.5)
model.compile(loss='binary_crossentropy', optimizer=adam_, metrics=['accuracy'])

mpl.rcParams['text.usetex'] = False
model.fit(input_data, target_data,
          epochs=100, batch_size=3000, shuffle=True, callbacks=[plot_losses],␣
↪validation_split=0.0)
```




```
43384/43384 [==============================] - 1s 23us/step - loss: 0.1549 -
acc: 0.9539
```

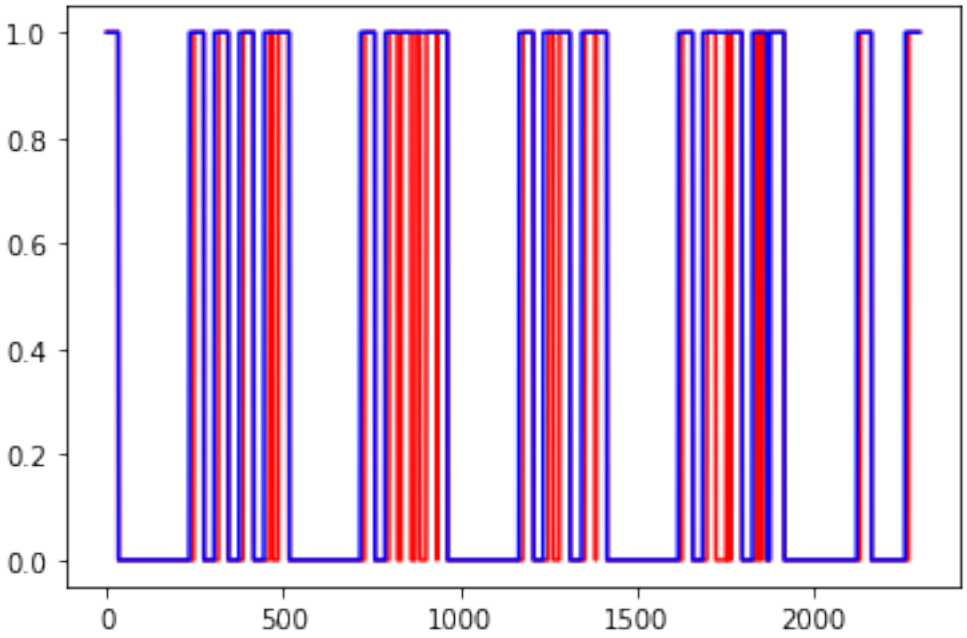
[125]:
```
<tensorflow.python.keras.callbacks.History at 0x7f03097abf60>
```

[134]:
```
num_tests = 5

for i in range(num_tests):
```

```
    test_dict = trajectory_with_transition(28, time_window)
    x_test = test_dict['x']
    if i==0:
        trans_actual = test_dict['jump_imminent']
    else:
        trans_actual = np.concatenate((trans_actual,␣
↪test_dict['jump_imminent']))

    trans_pred_ = np.zeros_like(test_dict['jump_imminent'])
    trans_pred_[0:look_back] = test_dict['jump_imminent'][0:look_back]
    for k in range(len(trans_pred_)-look_back+1):
        pred_input = x_test[:, k:k+look_back].T
        trans_pred_[k+look_back-1] = model.predict(np.array([pred_input]))
    if i==0:
        trans_pred = trans_pred_
    else:
        trans_pred = np.concatenate((trans_pred, trans_pred_))
trans_pred = np.where(trans_pred>0.5, 1, 0)
```

[135]:
```
plt.figure()
plt.plot(trans_pred, 'r')
plt.plot(trans_actual,'b')

# error rate
print('Error rate: ')
print(np.mean(np.abs(trans_pred-trans_actual)))

# Now, it does not perform as well for a longer prediction window.
```

```
Error rate:
0.1295089091699261
```

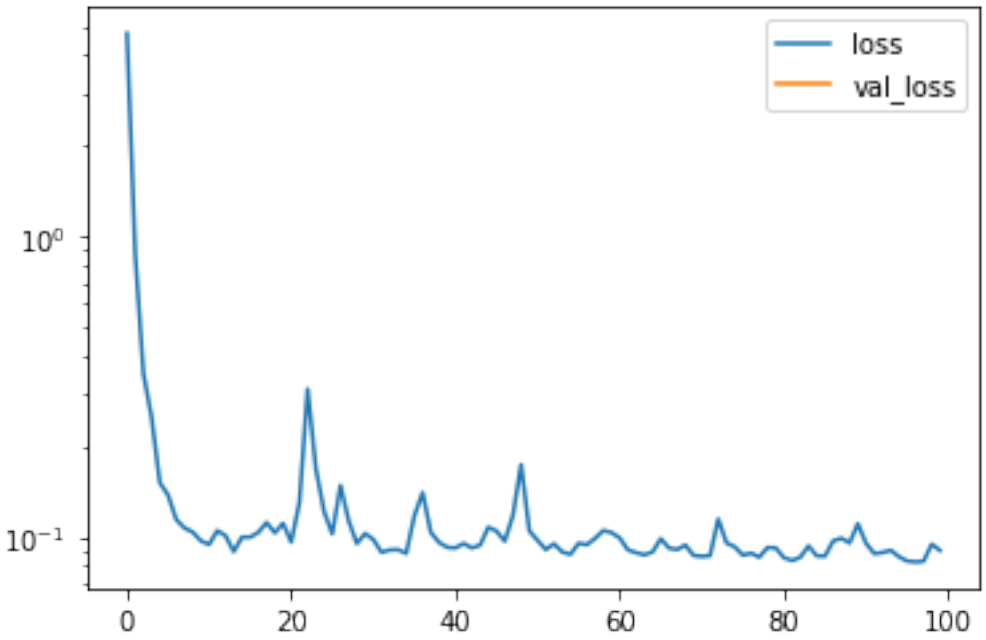


0.2.2 Try a very long time_window

[93]:
```
time_window = 100
look_back = 15

max_iter = 100
for i in range(max_iter):
    trans_dict = trajectory_with_transition(28, time_window)
    x_data = trans_dict['x']

    delayed_data = np.expand_dims(x_data[:,0:-(look_back-0+1)].T, axis=1) #␣
↪(T-look-back, 1, 3)
    for l in range(1, look_back):
        delayed_data = np.concatenate((delayed_data, \
                            np.expand_dims(x_data[:,l:-(look_back-l+1)].T,␣
↪axis=1)),axis=1)
        if l == (look_back-1):
            if i == 0:
                target_data = trans_dict['jump_imminent'][l:-(look_back-l+1)]
            else:
                target_data = np.concatenate((target_data,␣
↪trans_dict['jump_imminent'][l:-(look_back-l+1)]))
    if i == 0:
        input_data = delayed_data
```

```
    else:
        input_data = np.concatenate((input_data, delayed_data),axis=0)
```

[94]:
```
model = Sequential()
model.add(LSTM(3*look_back, activation="tanh", recurrent_activation="sigmoid", \
               dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1))

adam_ = optimizers.Adam(lr=.005, beta_1=0.9, beta_2=0.999, epsilon=None,␣
↪decay=1e-5, amsgrad=True, clipvalue=0.5)
model.compile(loss='binary_crossentropy', optimizer=adam_, metrics=['accuracy'])

mpl.rcParams['text.usetex'] = False
model.fit(input_data, target_data,
          epochs=100, batch_size=3000, shuffle=True, callbacks=[plot_losses],␣
↪validation_split=0.0)
```




```
43598/43598 [==============================] - 2s 39us/step - loss: 0.0901 -
acc: 0.9734
```

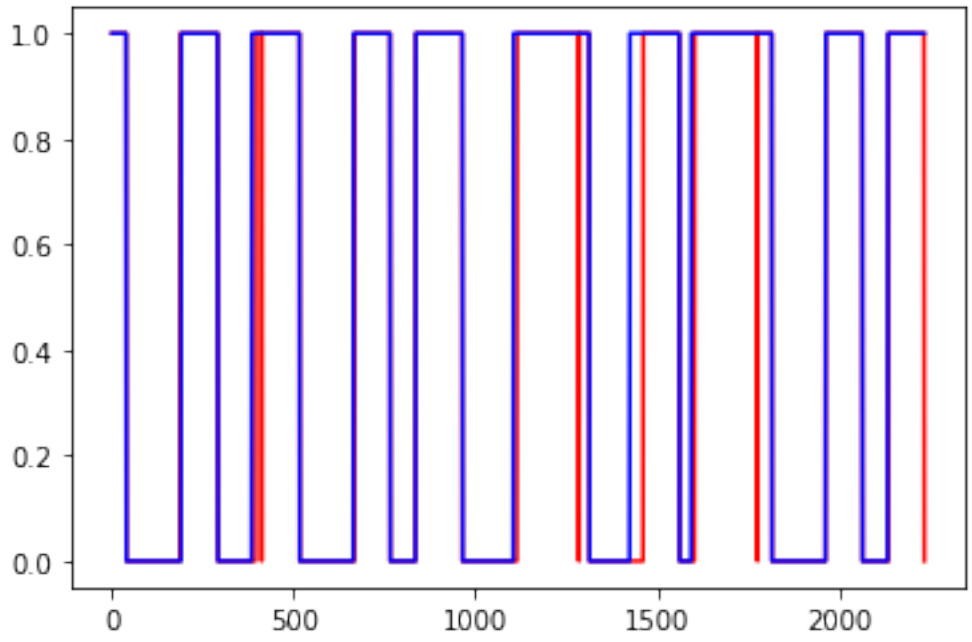
[94]:
```
<tensorflow.python.keras.callbacks.History at 0x7f04282d5cc0>
```

[95]:
```
num_tests = 5

for i in range(num_tests):
    test_dict = trajectory_with_transition(28, time_window)
```

```
    x_test = test_dict['x']
    if i==0:
        trans_actual = test_dict['jump_imminent']
    else:
        trans_actual = np.concatenate((trans_actual,␣
 ↪test_dict['jump_imminent']))

    trans_pred_ = np.zeros_like(test_dict['jump_imminent'])
    trans_pred_[0:look_back] = test_dict['jump_imminent'][0:look_back]
    for k in range(len(trans_pred_)-look_back+1):
        pred_input = x_test[:, k:k+look_back].T
        trans_pred_[k+look_back-1] = model.predict(np.array([pred_input]))
    if i==0:
        trans_pred = trans_pred_
    else:
        trans_pred = np.concatenate((trans_pred, trans_pred_))
trans_pred = np.where(trans_pred>0.5, 1, 0)
```

[96]:
```
plt.figure()
plt.plot(trans_pred, 'r')
plt.plot(trans_actual,'b')

# error rate
print('Error rate: ')
print(np.mean(np.abs(trans_pred-trans_actual)))
```

```
Error rate:
0.0353309481216458
```



Note, it may not make sense to try a larger window than this, because that will run over multiple transitions. Very large window would make fewer points considered imminent, which would create unbalanced data distribution. This could also be why the performance is good with time_window=100, as most points were considered imminent.

[ ]: