Assignment 4

March 10, 2019

1 Phys 581 Winter 2019

2 Assignment #4: Neural Networks

2.1 Alexander Hickey, 10169582

Note that this notebook makes use of the Keras deep learning library for python, which is compatible only with Python 2.7-3.6.

```
In [1]: #Must be running Python 3.6 or lower!
        import sys
        sys.version
```

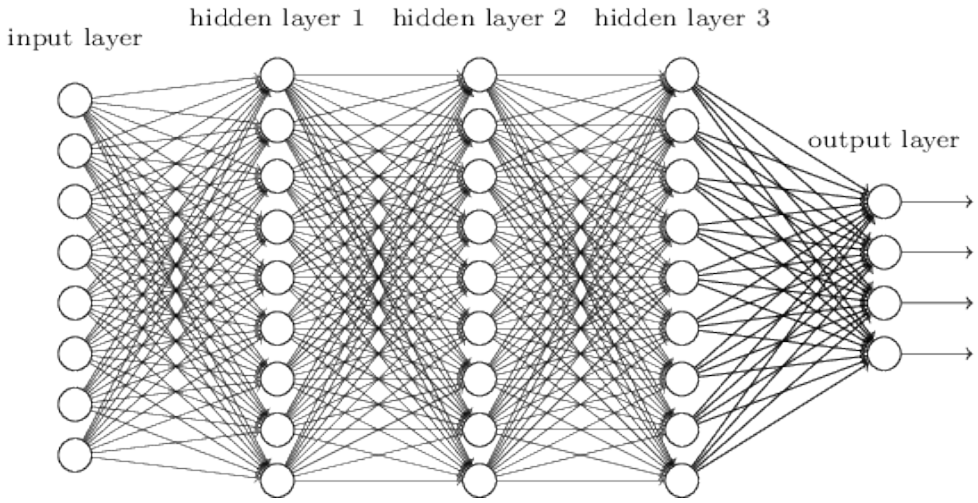
```
Out[1]: '3.6.8 |Anaconda, Inc.| (default, Feb 21 2019, 18:30:04) [MSC v.1916 64 bit (AMD64)]'
```

```
In [2]: #Import useful libraries
        import numpy as np
        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D
        import scipy.integrate
        from keras.models import Model, Sequential
        from keras.layers import Input, Dense
        from keras.backend import clear_session
        %matplotlib inline
```

```
Using TensorFlow backend.
```

2.1.1 Introduction

Inspired by biological neural networks, a method of machine learning known as deep learning has revolutionized the way that scientists model complex systems. In general, a neural network is a collection of connected nodes (or *neurons*) that can transmit data through their connections (or *edges*). Typically, the neurons are oraganized into layers, each of which can perform different kings of transformations on their inputs. Additionally, each of the edges are assigned a weight that increases or decreases the strength of a signal at an edge, and these weights are adjusted in the learning process using a set of training data. The term *deep* neural network refers to a neural network with more than one hidden layer (see Figure below).



(Image source: http://neuralnetworksanddeeplearning.com/chap6.html)

Recently, there has been great interest in the use of machine learning to predict solutions of physical problems, such as the solutions of a non-linear system of differential equations. In this notebook, we will explore the implementation of machine learning to predict the time evolution of the Lorenz system, which is notable for producing chaotic solutions under certain parameter ranges. The Lorenz equations are:

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

For the choice of parameters $(\sigma, \beta, \rho) = (10, 8/3, 58)$, it is known that the solution is a "butterfly" like strange attractor that exhibits deterministic chaos. The goal of this work will be to construct and train a neural network that takes a solution set of the Lorenz equations over some time interval $[0, t]$, and predicts the solution set over some shifted time interval $[\Delta t, t + \Delta t]$. This will be done through the use of Keras, a deep learning library for Python. The accuracy of the neural network predictions will be examined under various time shifts and network topologies.

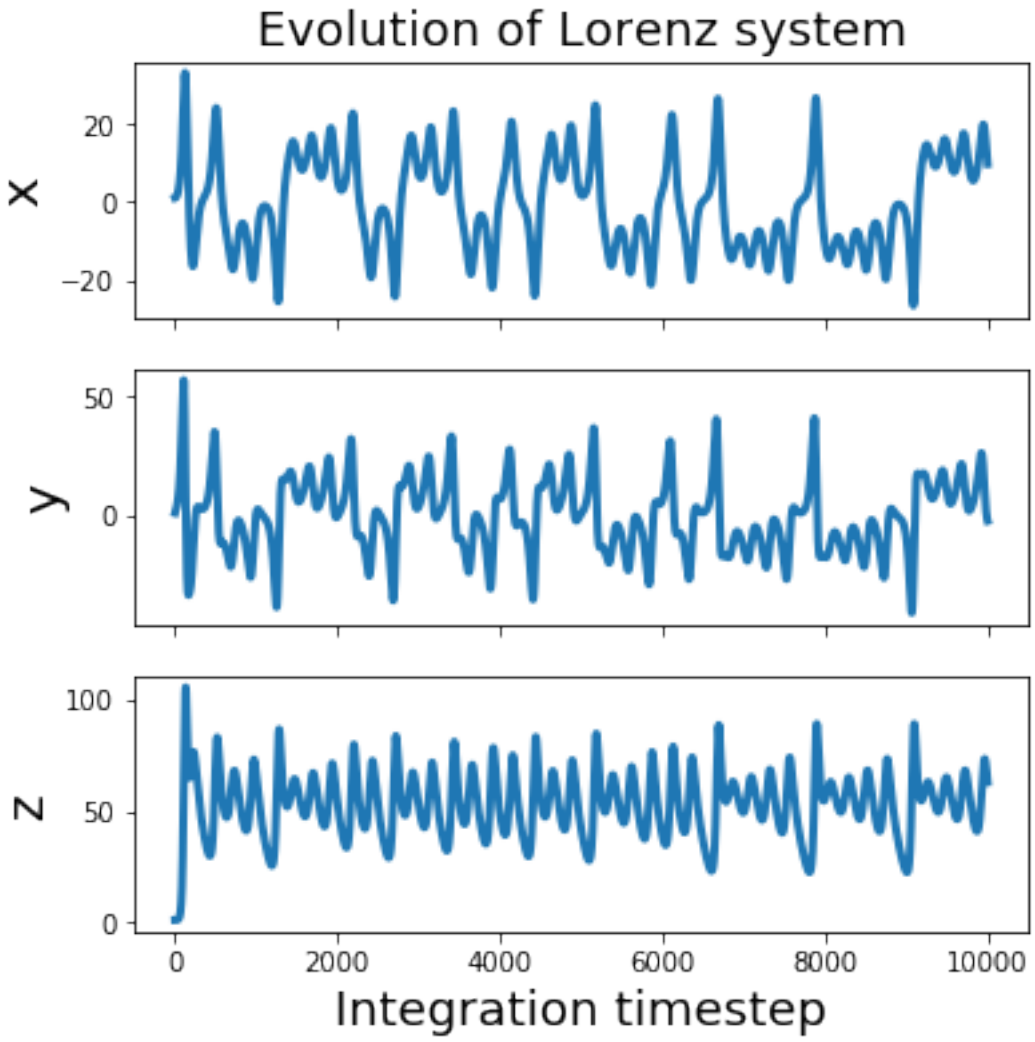
We begin by generating a 3-d time series solution for the Lorenz equations with the parameters selected to produce "butterfly" strange attractors.

```
In [3]: def dfunc(state, t0, sigma=10.0, beta=8/3.0, rho=58.0):
            '''
            This returns the time derivative of the coordinates,
            defined by the 3D Lorenz system:
            dx/dt = sigma*(y-x)
            dy/dt = x*(rho-z)-y
            dz/dt = x*y-beta*z

            Args:
                state: array of length 3, coordinates at time t0
                t0: Time
```

```
            sigma, beta, rho: Lorentz system parameters

        Return:
            d/dt state: array, Time derivative of coordinates

        '''

        #Unpack state vector
        x, y, z = state

        return np.array([ sigma*(y-x), x*(rho-z)-y, x*y-beta*z])

    #Define time interval of interest
    t0, tf, tstep = 0.0, 20.0, 9999
    tvals = np.linspace(t0, tf, tstep)

    #Set initial state set to [1,1,1]
    xyz_0 = np.ones(3)

    #Integrate Lorenz system over time interval
    xyz = scipy.integrate.odeint( dfunc, xyz_0, tvals )
```

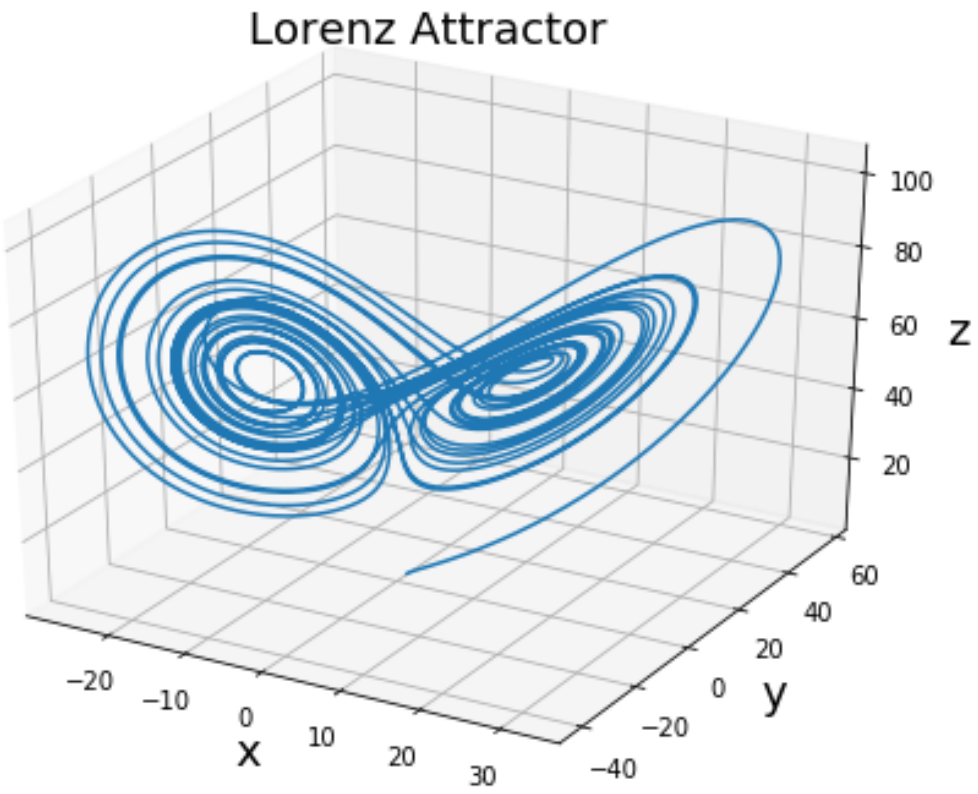
The solution can be visualized by plotting each of the spatial coordinates as a time series

In [4]:
```
#Plot x,y,z components of Lorenz system from t=0 to t=20
fig, ax = plt.subplots(3, sharex=True, figsize=(6,6))

ax[0].plot( xyz[:,0], lw=3 )
ax[0].set_ylabel('x',fontsize=20)
ax[1].plot( xyz[:,1], lw=3 )
ax[1].set_ylabel('y',fontsize=20)
ax[2].plot( xyz[:,2], lw=3 )
ax[2].set_ylabel('z',fontsize=20)

ax[0].set_title('Evolution of Lorenz system',fontsize=18)
ax[-1].set_xlabel('Integration timestep',fontsize=18)

plt.show()
```



As expected, the solution turns out to look quite complicated. To better observe the "butterfly" like behaviour of the tragectory, we can view this as a 3D projection.

```
In [5]: #Plot 3D trajectory of Lorenz system
        fig = plt.figure(figsize=(8,6))
        ax = fig.add_subplot(111, projection='3d')

        ax.set_xlabel('x',fontsize=18)
        ax.set_ylabel('y',fontsize=18)
        ax.set_zlabel('z',fontsize=18)
        ax.set_title('Lorenz Attractor',fontsize=18)

        ax.plot(*xyz.T)
        plt.show()
```



2.1.2 Deep learning

Build a simple model to use current state at time `T` to predict future state at time `T + dt`

We begin by defining the model in the Keras framework. In particular, this means defining the particular topology of the deep learning network. To start, we fix the random seed in both the numpy and tensorflow libraries. This is generally a good idea when testing properties of a neural network, as it makes the otherwise stochastic construction of the network weights to be reproducible. As is standard within the Keras library, the network is taken to be dense, meaning that each node of one layer is connected to every other node in the next. The model is compiled using the mean-squared error as a loss function and the "adam" optimizer.

```
In [6]: #Fix random seed
        np.random.seed(1)
        from tensorflow import set_random_seed
        set_random_seed(1)

        #Create model
        model = Sequential()
        model.add(Dense(8, input_dim=3, activation='relu'))
        model.add(Dense(11, activation='relu'))
        model.add(Dense(6, activation='relu'))
```

```
        model.add(Dense(3, activation='linear'))

        #Compile model
        model.compile(loss='mean_squared_error', optimizer='adam')
        model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 8)                 32
_________________________________________________________________
dense_2 (Dense)              (None, 11)                99
_________________________________________________________________
dense_3 (Dense)              (None, 6)                 72
_________________________________________________________________
dense_4 (Dense)              (None, 3)                 21
=================================================================
Total params: 224
Trainable params: 224
Non-trainable params: 0
_________________________________________________________________
```

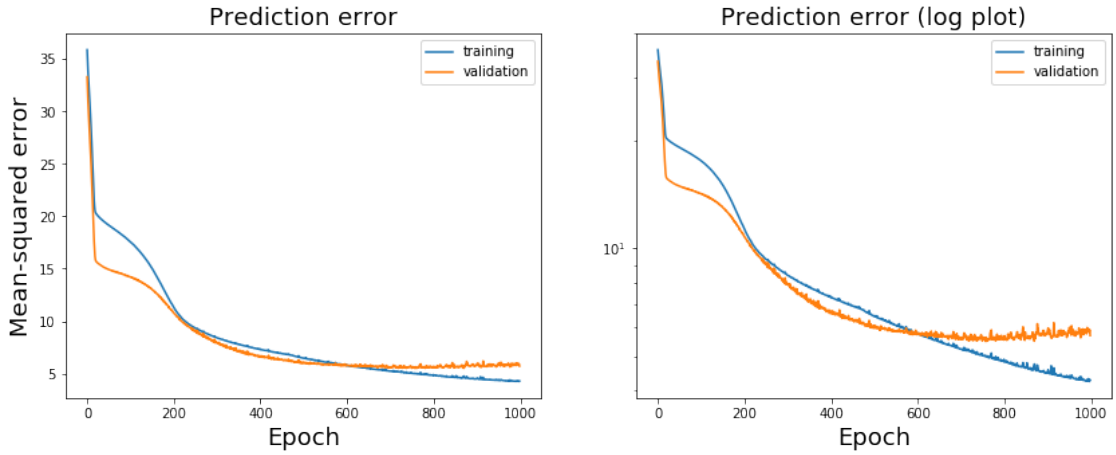
Next, we run through 999 epochs of the traning routine, by using the time series data for the interval $[0, L]$ as the input, and the time series data for the shifted interval $[\Delta t, L + \Delta t]$ as the desired output. At each epoch, the mean-squared error is computed for both the training data and a so-called validation interval, where the data for the interval $[vs, vs + L]$ is used as input. In this case, we have chosen $\Delta t = 99.$, $L = 900$, and $vs = 5000$.

```
In [7]: stack = []
        ndt = 99  #future prediction time shift
        L = 900 #Length of timestep interval
        vs = 5000 #Starting point of validation interval
        ne = 999 #Number of epochs

        #Model will attempt to shift data set by a timestep of ndt
        #i.e. use points [0:900] to predict points [99:999]
        for epoch in range(ne):

            # take one step
            history = model.fit(xyz[:L],xyz[ndt:L+ndt], epochs=1, batch_size=100, verbose=0)

            # calculate prediction quality for training data
            rms_training = np.sqrt( np.mean( (xyz[ndt:L+ndt,:] - model.predict(xyz[:L]) )**2))

            # calculate prediction quality for distant future data
            rms_validate = np.sqrt( np.mean( (xyz[vs+ndt:vs+L+ndt,:]
                                              - model.predict(xyz[vs:vs+L]))**2))
```

```
        #Record root mean square error for each epoch
        stack.append([rms_training, rms_validate])

    stack = np.array(stack)
```

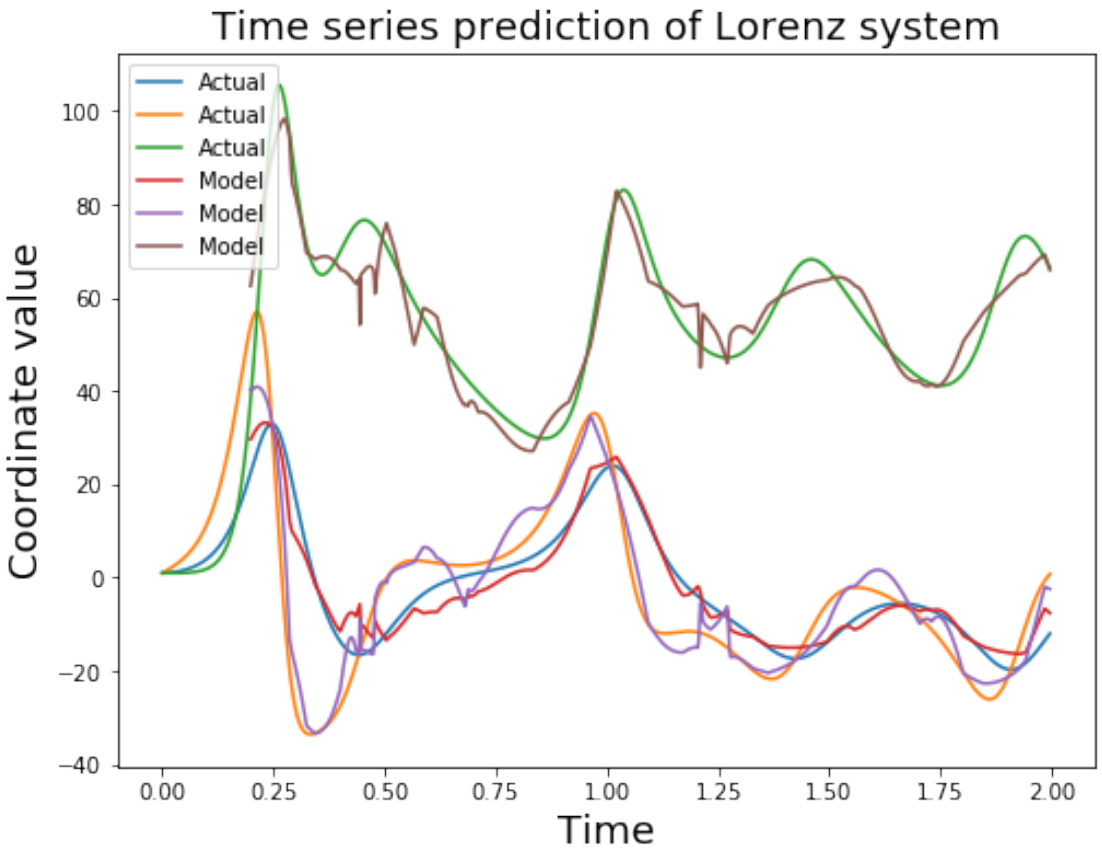
Next we plot the mean-squared error over the training process.

In [8]:
```
fig, axes = plt.subplots(1, 2, figsize=(14,5))

plt.sca(axes[0])
plt.plot( stack[:,0], label='training' )
plt.plot( stack[:,1], label='validation')
plt.xlabel('Epoch',fontsize=18)
plt.ylabel('Mean-squared error',fontsize=18)
plt.title('Prediction error',fontsize=18)
plt.legend(loc=0)

plt.sca(axes[1])
plt.plot( stack[:,0], label='training' )
plt.plot( stack[:,1], label='validation')
plt.xlabel('Epoch',fontsize=18)
plt.title('Prediction error (log plot)',fontsize=18)

plt.legend(loc=0)
plt.yscale('log')
```



We see that the discrepency between the model data and the training data decreases quite rapidly, with a few small-plateaus. The validation fit also improves throughout the training process, and at times is actually better than the training results. They eventually both appear to converge, to some finite value, with minor fluctuations in the error appearing at later epochs. We can see how the actual tragectories appear by plotting the time series prediction.

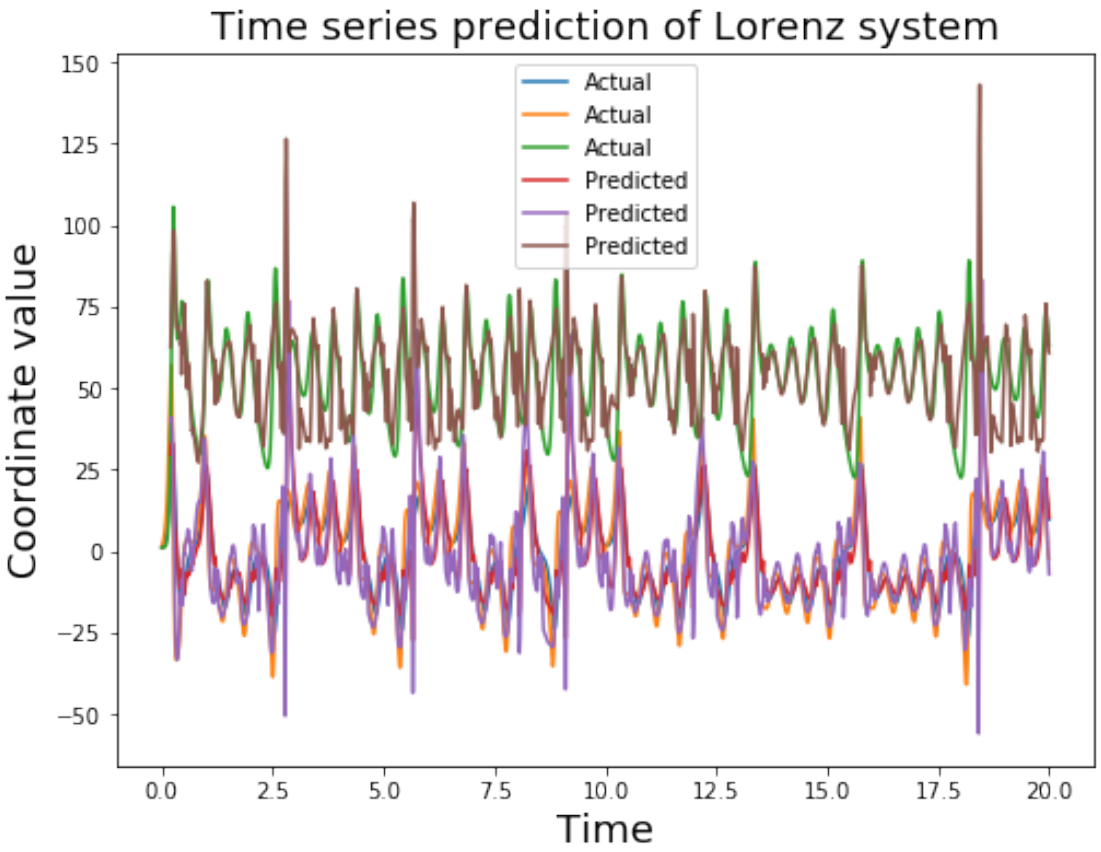
```
In [9]: plt.figure(figsize=(8,6))
        plt.plot(tvals[:L+ndt], xyz[:L+ndt,:],label='Actual')
        plt.plot( tvals[ndt:L+ndt], model.predict( xyz[:L]),label='Model' )
        plt.xlabel('Time',fontsize=18)
        plt.ylabel('Coordinate value',fontsize=18)
        plt.title('Time series prediction of Lorenz system',fontsize=18)

        plt.legend(loc=0)
        plt.show()
```



Qualitatively, the fit to the training data appears to be quite good. Let's see if this holds up for the entire data set:

```
In [10]: #Define tail end of time interval
         vt = 9000

         plt.figure(figsize=(8,6))
         plt.plot(tvals[:vt+L+ndt], xyz[:vt+L+ndt,:],label='Actual')
         plt.plot( tvals[ndt:vt+L+ndt], model.predict(xyz[:vt+L]),label='Predicted' )
         plt.xlabel('Time',fontsize=18)
         plt.ylabel('Coordinate value',fontsize=18)
```

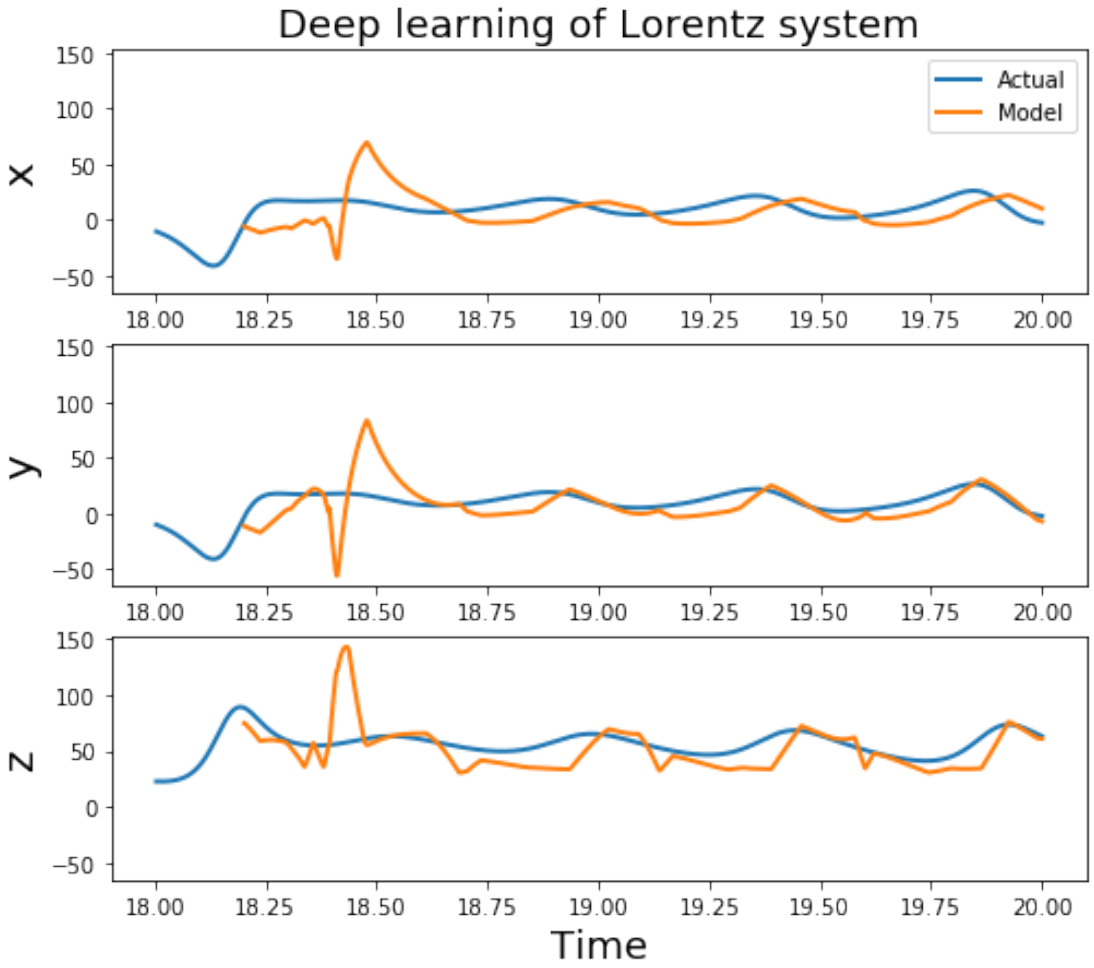
```
plt.title('Time series prediction of Lorenz system',fontsize=18)

plt.legend(loc=0)
plt.show()
```



As we see, the model seems to match the data set quite well on a global scale. Finally, let's look at how the model holds up near the tail end of the validation region:

```
In [11]: fig, ax = plt.subplots(3, sharey=True, figsize=(8,7))

         ax[0].plot(tvals[vt:vt+L+ndt], xyz[vt:vt+L+ndt:,1], lw=2,label='Actual')
         ax[0].plot(tvals[vt+ndt:vt+L+ndt], model.predict(xyz[vt:vt+L,:]).T[0], lw=2,
                    label='Model')
         ax[0].set_ylabel('x',fontsize=20)

         ax[1].plot(tvals[vt:vt+L+ndt], xyz[vt:vt+L+ndt:,1], lw=2 )
         ax[1].plot(tvals[vt+ndt:vt+L+ndt], model.predict(xyz[vt:vt+L,:]).T[1], lw=2)
         ax[1].set_ylabel('y',fontsize=20)

         ax[2].plot(tvals[vt:vt+L+ndt], xyz[vt:vt+L+ndt:,2], lw=2)
         ax[2].plot(tvals[vt+ndt:vt+L+ndt], model.predict(xyz[vt:vt+L,:]).T[2], lw=2 )
```

```
ax[2].set_ylabel('z',fontsize=20)

ax[0].set_title('Deep learning of Lorentz system',fontsize=18)
ax[-1].set_xlabel('Time',fontsize=18)

ax[0].legend(loc=0)
plt.show()
```



The neural network model appears to match the data quite well in this region as well (i.e. if you squint a little), aside from various minor fluctuations.

2.2 Tasks

The validation curve appears to fluctuate more than the training curve at later epochs. Discuss why this might be. We see that at later epochs in the training routine, the mean squared error tends to asymptote at some nonzero value, and fluctuations become more pronounced. The fact that fluctuations begin to appear at later epochs suggests that these fluctuations are a remenant of the chaotic nature of the Lorenz system. While the trajectory of the Lorentz system through phase

space is deterministic in nature, the long term evolution of the system is extraordinarily sensitive to initial conditions. Even a small discrepency between initial conditions (i.e. on the order of double precision) can lead to considerably different dynamics. It follows that, while machine learning will likely be able to predict the global, quasi-periodic behaviour of a strange attractor, it will always be limited by machine precision in predicting deterministic chaos.

Tidy up code (eg. factor out magic numbers like 9000:9900) and make sure it is correct. See the code following the introduction.

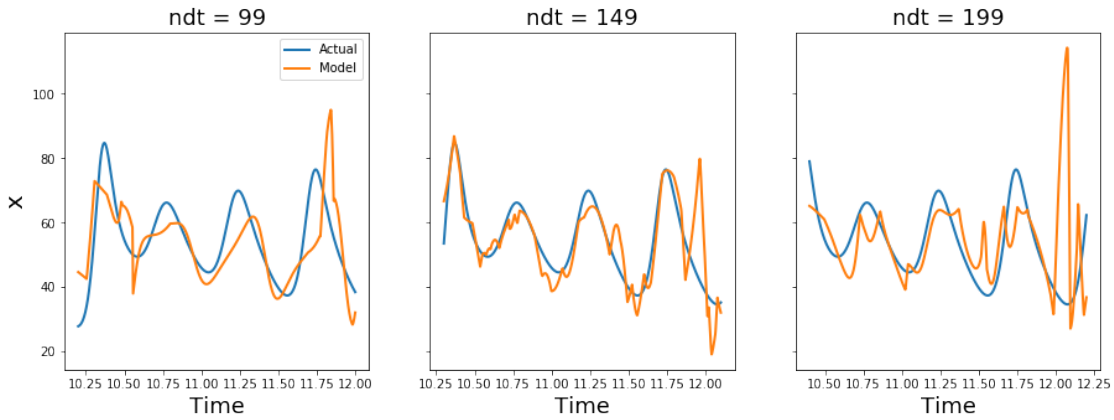
How does prediction quality decrease for larger `ndt` ie. further into the future? To examine how the prediction quality depends on various parameters, we will define a few functions to systematically create and train neural networks to predict the time evolution of the Lorenz system.

```
In [12]: def create_model(nodes=[8,11,6]):
             '''
             This function creates a dense Keras model with connectivity:
             3-nodes-3 where nodes represents the "deep" layers.

             Args:
                 nodes: List describing number of deep layers and number of nodes at
                        each layer.

             Return:
                 model: Dense Keras model with desired connectivity.

             '''
             #Create model
             model = Sequential()

             #Define input layer
             model.add(Dense(nodes[0], input_dim=3, activation='relu'))

             #Define "deep" layers
             for n in nodes[1:]:

                 model.add(Dense(n, activation='relu'))

             #Define output layer
             model.add(Dense(3, activation='linear')) #sigmoid'))

             #Compile model
             model.compile(loss='mean_squared_error', optimizer='adam')

             return model

         def train_model(model,ndt,xyz,L=900,epochs=999):
             '''
```

```
        This function trains a given Keras model to shift a time series
        forward by some timestep ndt.

        Args:
            model: Keras model
            ndt: Timestep to shift time series
            xyz: 3D time series data
            L: Length of training interval
            epochs: Number of epochs to perform training

        Return:
            model: Trained model

        '''

        history = model.fit(xyz[:L],xyz[ndt:L+ndt], epochs=epochs, batch_size=100,
                            verbose=0)

        return model
```

Next, we can qualitativly examine how increasing the timestep affects the quality of prediction. In this case, we examine the prediction produced on the validation interval for $\Delta t = 99, 149, 199$.

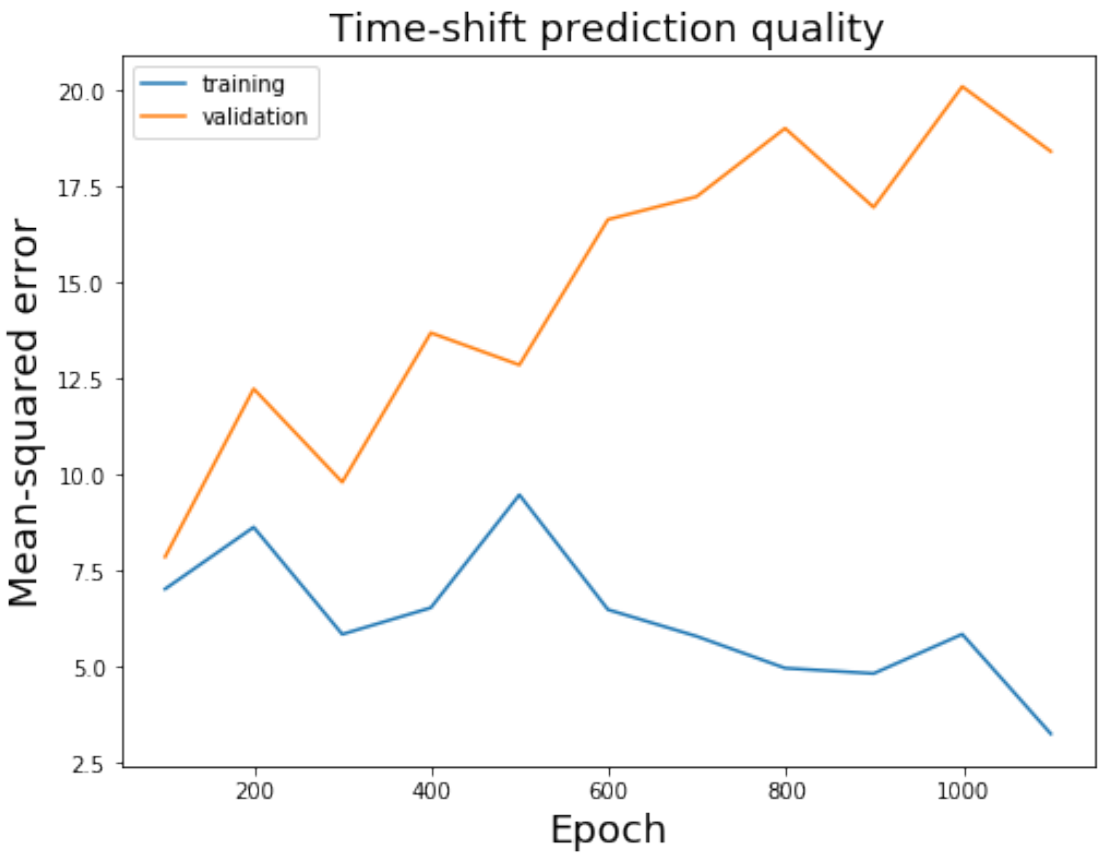
```
In [13]: L = 900 #Length of timestep interval
         vs = 5000 #Starting point of validation interval
         ndtl = [99,149,199] #Time shifts

         fig, ax = plt.subplots(1,3, sharey=True, figsize=(15,5))

         for j in range(len(ndtl)):

             #Create and train model
             ker = train_model(create_model(),ndtl[j],xyz)

             #Plot prediction on validation interval
             ax[j].plot(tvals[vs+ndtl[j]:vs+L+ndtl[j]], xyz[vs+ndtl[j]:vs+L+ndtl[j]:,2],
                        lw=2,label='Actual')
             ax[j].plot(tvals[vs+ndtl[j]:vs+L+ndtl[j]], ker.predict(xyz[vs:vs+L,:]).T[2],
                        lw=2 ,label='Model')
             ax[j].set_xlabel('Time',fontsize=18)
             ax[j].set_title('ndt = '+str(ndtl[j]),fontsize=18)

         ax[0].set_ylabel('x',fontsize=20)
         ax[0].legend(loc=0)
         plt.show()
```



We see that in each case, the model matches the validation data quite well. It seems that as the step size increase, the model continues to match the peaks and troughs of the data very well, but begins to fluctuate more. Next, we compute the mean-square error after $ne = 999$ epochs for gradually increasing time shifts, on both the training and validation intervals.

```
In [14]: #List of time shifts
         ndt_list = np.arange(99,1100,100)
         stack = []

         for ndt in ndt_list:

             #Clear previous Keras models
             clear_session()

             #Create and train model
             ker = train_model(create_model(),ndt,xyz,epochs=ne)

             #Compute mean square error
             rms_training = np.sqrt( np.mean((xyz[ndt:L+ndt,:]-ker.predict(xyz[:L]))**2))
             rms_validate = np.sqrt( np.mean((xyz[vs+ndt:vs+L+ndt,:]
                                              -ker.predict(xyz[vs:vs+L]) )**2 ) )

             stack.append([rms_training, rms_validate])

         stack = np.array(stack)

In [15]: #Plot means square error for increasing ndt
         plt.figure(figsize=(8,6))
         plt.plot(ndt_list, stack[:,0], label='training' )
         plt.plot(ndt_list, stack[:,1], label='validation')
         plt.xlabel('Epoch',fontsize=18)
         plt.ylabel('Mean-squared error',fontsize=18)
         plt.title('Time-shift prediction quality',fontsize=18)
```

```
plt.legend(loc=0)
plt.show()
```

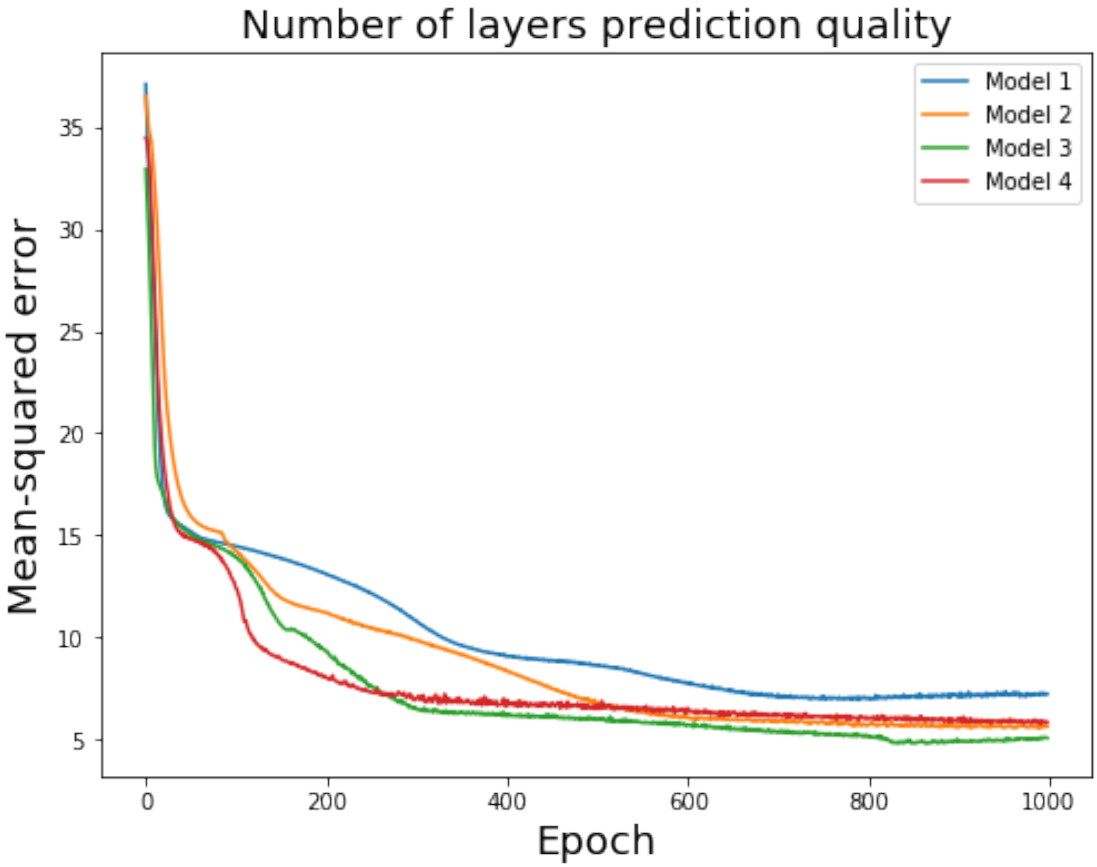


Interestingly, it seems that increasing the time shift significantly reduces the prediction quality of the validation interval, but does not make much of a difference to the training interval. This makes sense, as the neural network is literally trained on the training data, so the model should in principle fit the data that overlaps before and after the time shift almost perfectly. As the time shift increases, these data sets overlap less and less, and so the prediction quality should decrease for values outside of the training interval.

Explore changing hyper parameters (the number of layers, nodes per layer etc.) to improve prediction quality Next, we examine how the prediction quality varies with the number of layers and the number of nodes per layer. We start by fixing the number of nodes per layer, and train models with a gradually increasing number of deep layers. In this case, we look at models with $3, 4, 5$ and 6 deep layers respectively.

```
In [16]: #Define deep layer connectivity arrays
         num_layers = [[5,8,5],[5,8,8,5],[5,8,8,8,5],[5,8,8,8,8,5]]

         #Timeshift and interval lengths same as before
         ndt = 99
```

```
L, vs, ne = 900, 5000, 999

#Plot prediction error at each epoch
plt.figure(figsize=(8,6))

for j in range(len(num_layers)):

    stack = []

    #Create model
    model = create_model(nodes = num_layers[j])

    for epoch in range(ne):

        #Take one step
        model = train_model(model,ndt,xyz,epochs=1)

        #Calculate prediction quality for validation data
        rms_validate = np.sqrt( np.mean( (xyz[vs+ndt:vs+L+ndt,:]
                                          - model.predict(xyz[vs:vs+L]) )**2))

        #Record root mean square error
        stack.append(rms_validate)

    plt.plot(stack,label='Model '+str(j+1))

plt.xlabel('Epoch',fontsize=18)
plt.ylabel('Mean-squared error',fontsize=18)
plt.title('Number of layers prediction quality',fontsize=18)
plt.legend(loc=0)
plt.show()
```

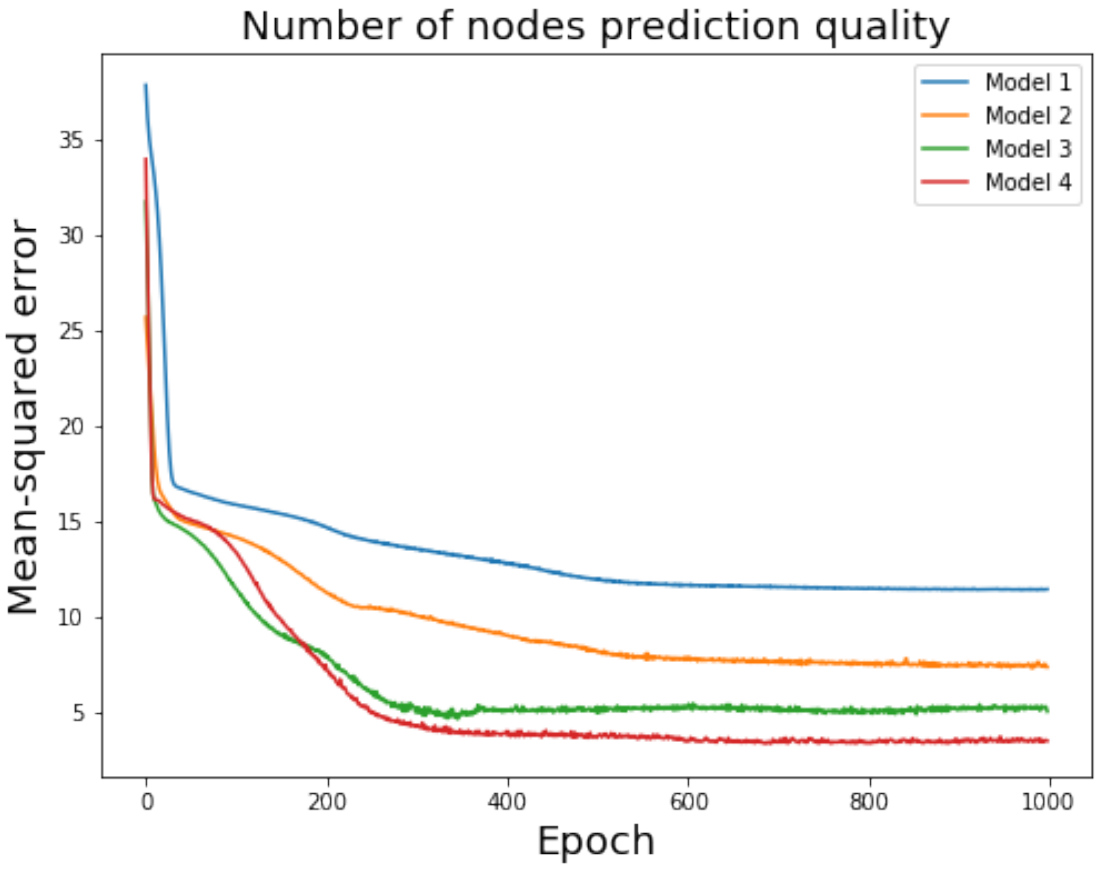


Surprisingly, the model with the least number of deep layers seems to predict the shifted data best. This suggests that too many layers can lead to "overfitting" the data. Interestingly, while the quality of fit after many epochs is better with less layers, we see that the models with a larger number of layers converge to their optimal performace much quicker. From our results, it seems that the quality of each model tends to asymptote to some finite error, however, it would be interesting to train these models longer to see how much they improve.

Next, we look at how the prediction quality depends on the number of nodes in each layer, while keeping the number of deep layers fixed at 3.

```
In [17]: #Define deep layer connectivity arrays
         num_nodes = [[5,8,5],[8,11,6],[11,15,9],[13,19,11]]

         plt.figure(figsize=(8,6))

         for j in range(len(num_nodes)):

             stack = []
             model = create_model(nodes = num_nodes[j])

             for epoch in range(ne):
```

```
            #Take one step
            model = train_model(model,ndt,xyz,epochs=1)

            #Calculate prediction quality for validation data
            rms_validate = np.sqrt( np.mean( (xyz[vs+ndt:vs+L+ndt,:]
                                              - model.predict(xyz[vs:vs+L]))**2))

            #Record root mean square error
            stack.append(rms_validate)

        plt.plot(stack,label='Model '+str(j+1))

plt.xlabel('Epoch',fontsize=18)
plt.ylabel('Mean-squared error',fontsize=18)
plt.title('Number of nodes prediction quality',fontsize=18)
plt.legend(loc=0)
plt.show()
```



We see that the model quality tends to improve with the number of nodes per layer. Interestingly, the $[11, 15, 9]$ and $[13, 19, 11]$ arrangements tend to approximately the same mean-squared

error, which suggests that continuing to increase the number of nodes will not necessarily continue to improve the model quality. We do see however, that the model with a larger number of nodes converges much quicker (i.e. less epochs required, not necessarily less computing time).

2.2.1 Conclusion

In this notebook, we examined the use of deep learning to predict a time-shifted data set corresponding to the solution of the Lorenz system. We saw that fluctuations in the predictive quality persisted at large trainin epochs, and this most likely arises from the fact that the Lorenz system exhibits deterministic chaos. We saw that increasing the time shift parameter tended to decrease the quality of prediction over the validation interval, but did not make a significant difference in predictions close to the training interval. We also saw that increasing the number of deep layers actually tended to decrease the predictive quality for future times, suggesting that large layer models can be prone to overfitting. Finally, we saw that increasing the number of nodes per layer significantly improved the model quality, and caused the model to require fewer epochs of training. It seems at this point that constructing the topology of a neural network is a very tedious task, and the number of nodes/layers to choose is largely based on a guess-and-check approach.

With more time available, it would be interesting to expand on these investigations of how the predictive qualities vary under different network topologies and time shifts. Additionally, it would be interesting to see how the predictions respond to various changes in the training procedure, such as the length of the data set used for training.