AMATH 563 Spring 2018 Homework 2
Neural Network for Dynamical System

Tun Sheng Tan

Abstract. This assignment focuses on neural network and dynamical systems. Simple feed forward network is used to learn the dynamical systems. We explore the method on Kuromoto-Sivashinsky equation, reaction-diffusion system and Lorenz system.

1. Introduction.

Neural network is very popular because of the availability of big data and inexpensive computational power. It has been very successful in classification tasks because of big datasets. However, the focus of this assignment is to use neural network to perform future state prediction for dynamical systems. In this assignment, I explore some neural network concepts and techniques. In section 2, I introduce the basic of artificial neural network and back-propagation method. In section 3, I discuss the implementation of algorithms. In section 4, I attempt to perform future state prediction for 3 non-linear systems.

2. Theoretical Background.

2.1. Neural Network.

Artificial neural network is made up of many processor units called perceptron. Neural network are made up of layers of nodes. Deep learning are neural networks that contains many layers. The most important idea for neural network is back-propagation.

2.2. Perceptron.

Biological neurons work like a gate for electrical signal. When the strength of the signal reaches a certain threshold, the neuron will fire a signal. The man made version of neuron is called perceptron. A perceptron is node that applies an activation function f onto a biased linear combinations inputs x_i as shown in Figure 1.

By stacks layers of perceptrons together, we can form a network. For large network, the activation functions are usually chosen to have a simple derivative such as a step function, rectified linear unit (ReLU) or linear function. This is done to minimize the complexity of large networks.

2.3. Gradient Descent.

The term ”training” in the context of the neural network means to optimize. We can sum up neural network as a optimization problem for our network. Naturally gradient descent algorithm shown in equation (2.1) is used for its simplicity and effectiveness. This algorithm is very similar to Newton's method (second-order method) but of first-order. The main idea is to update the weights w_i and biases b_i in the direction of steepest slope of our chosen objective function F. Stochastic gradient descent, commonly used in neural network, is gradient descent with a random step size, α, or learning rate as referred by the computer science community.

$$x_{n+1} = x_n - \alpha \nabla F(x_n) \tag{2.1}$$

2.4. Back-propagation.

Back-propagation is the backbone of neural networks. We update the weights of the previous layer based on the error in the current layer. Using methods like gradient descent, we try to minimize the error of each layer by computing the gradient of

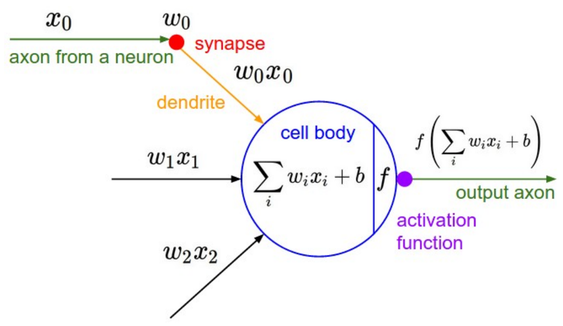

Figure 1. *Visualization of perceptron. [Source: https://goo.gl/JkMkEW]*

the cost function at the current layer. By doing so, we are essentially computing the chain-rule for derivatives. To sum up, suppose that we are given the output $y = y(x, W, b)$ where W, b are the weights and biases of the network. Then, the error $\varepsilon = \varepsilon(x, W, b) = \rho(y_0 - y)$. For a network with N layers, with $y = y_N$, the error correction propagated backward is [3]

$$\frac{\partial E}{\partial W_{ij}} = -\frac{\partial \rho(y_N)}{\partial y_{N-1}} \prod_{k=i+1}^{N-1} \left[\frac{\partial y_k}{\partial y_{k-1}}\right] \frac{\partial y_i}{\partial W_{ij}} \tag{2.2}$$

and each weight is updated by

$$w_{ij} = w_{ij} + \alpha \frac{\partial E}{\partial W_{ij}} \tag{2.3}$$

3. Algorithm Implementation and Development.

3.1. Generating Data. Since we are training our neural network to learn some nonlinear differential equations, we need to do some numerical simulation to generate the data. To do so, we will use **odeint** from the **scipy** package. We also standardize our data using **preprocessing** module from **sklearn** package. In our case, we found that scaling the data do not improve the result.

3.2. Neural Network. We construct our neural network using **Keras** [2] with is a wrapper for **Tensorflow** [1]. Our network will be a simple feed forward network with several hidden layers as shown in Figure 2. The objective of this neural network is to construct a function that map $x(t)$ to $x(t + \delta t)$. Let the function f be our neural network. We will determine if choosing $f[x(t)] = x(t + \delta t) - x(t)$ or $f[x(t)] = x(t + \delta t)$ could lead to better training.

3.3. Cross Validation. Cross validation is the most important part of every machine learning algorithm. Without cross validation, we will most likely run into over-fitting. To cross validate, we will separate our training set into test and validation set. We will train the model just on the test set and validate using the validation set. The best model will be the model with the smallest validation error. Due to the limited training time, we are constrained to 2-fold validation.

4. Computational Results.

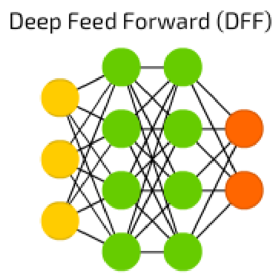

Figure 2. *A visualization of the feed forward network architecture used. [Source: https://goo.gl/JkMkEW]*

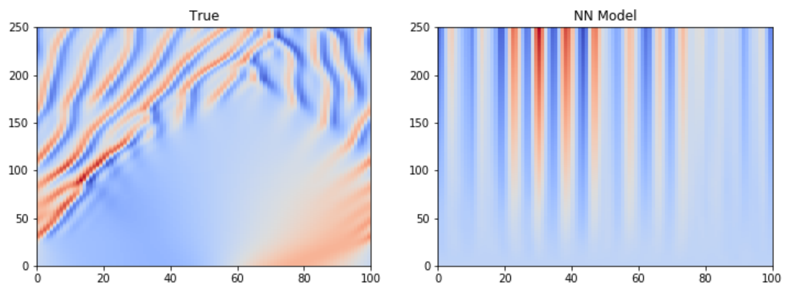

Figure 3. *Prediction for KS equation using feed forward neural network.*

4.1. Kuramoto-Sivashinsky Equation. Kuramoto-Sivashinsky(KS) equation shown in equation Equation (4.1) is used to model diffusive instability in a laminar flow. This is a nonlinear chaotic system. We train a neural network to predict the future state of KS equation. We compare the prediction with the traditional ODE time-stepper. Figure 3 shows a comparison from a random initial condition. It is clear that our model is not able to learn the dynamics.

$$\dot{u} = -\nabla^4 u - \nabla^2 u - \frac{1}{2}\|\nabla u\|^2 \tag{4.1}$$

4.2. Reaction-diffusion. The most common reaction-diffusion system is chemical reaction between two substances. The substances transform into each other and diffuse over space. To train our neural network, we reduced the rank of the system via SVD and perform a future state prediction in that subspace on the FitzHugh-Nagumo equation given equation (4.2) on a square domain with a spiral initial condition. An example of the dynamic is shown in Figure 4.

$$\begin{aligned} \dot{u} &= a\nabla^2 v + u - u^3 - v + k \\ \tau\dot{v} &= b\nabla v + u - v \end{aligned} \tag{4.2}$$

We train the neural network for 10 different conditions at a low rank projection space of 4 with 2 cross validation. Then, we generate new data to test our trained model. Our model is able to predict 1000 unit time-steps into the future given an initial condition as shown in Figure 5. The model become stationary beyond the time steps.

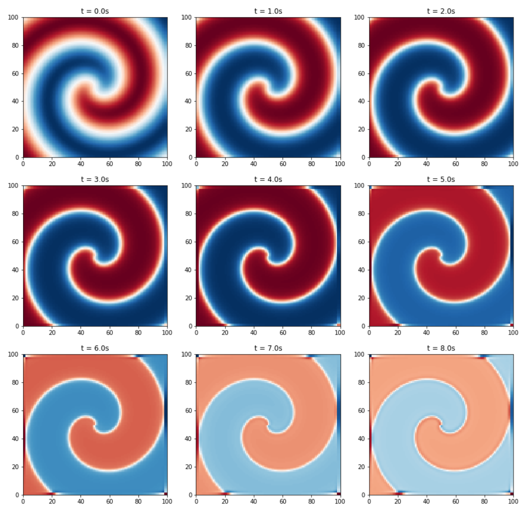

Figure 4. *Example of reaction diffusion at different times starting from a spiral initial condition.*

4.3. Lorenz System. Lorenz system was developed to model atmospheric convection by Edward Lorenz. The system is non-linear and non-periodic.

We train a neural network to advance the solution of Lorenz system shown in equation (4.3) from time t to $t + \delta t$ for $\rho = 10$, 28 and 40. Then, we will validate the results for $\rho = 17$ and 35. The result is shown in Figure 6. Our model do not interpolate well between ρ. We could try to train the model with ρ between 10 and 28. Next, we try to identify the transition from one lobe to another for $\rho = 28$. We label the transition when x crosses zero. The result for our neural network after 2-fold cross-validation is shown in Figure 7. It is just two realizations but not shown here is that our model failed to predict all lobe switching. Some tuning in the weighting of our labels could improve the accuracy.

$$\begin{aligned} \dot{x} &= \sigma(y - x) \\ \dot{y} &= (\rho - z)x - y \\ \dot{z} &= xy - \beta z \end{aligned} \tag{4.3}$$

5. Summary and Conclusions. Neural network is a powerful supervised learning method only if we have access to a very large dataset. Trying to train neural network to learn non-linear differential equation is non-trivial. Most of the effort required are in training and fine-tuning the model. In a time-constrained situation and a limited computing power, we are not able to achieve any good predictions for the 3 systems considered above. However, we have suggested some ways to improve the results.

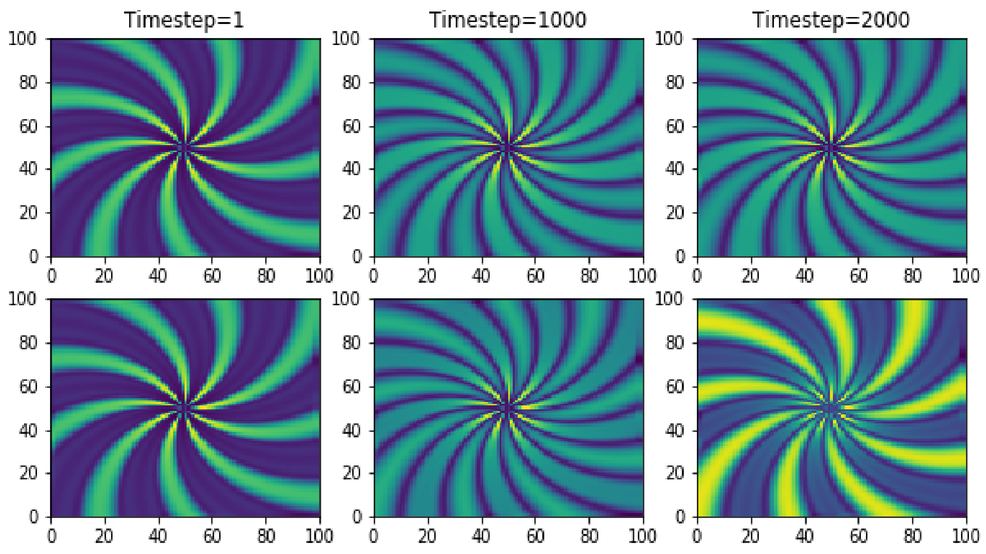

Figure 5. *Neural network (top) future state prediction for reaction-diffusion system versus the ground truth (bottom) at different time-steps.*

Appendix A. Important Python functions used and brief implementation explanation.

1. **odeint** - return the solution for ordinary differential equation
2. **KFold** - return K-Folds cross-validator object
3. **Sequential** - return Keras sequential neural network model object
4. **fbpca** - return singular decomposition of a matrix using randomized algorithm by Facebook

Appendix B. MATLAB codes.

B.0.1. Main Script.

1. Kuramoto-Sivashinsky Future State Prediction

```
import numpy as np
from physicsModel import kuramoto
from sklearn import preprocessing

import keras
import tensorflow as tf
from keras import regularizers
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD
from sklearn.model_selection import KFold

```

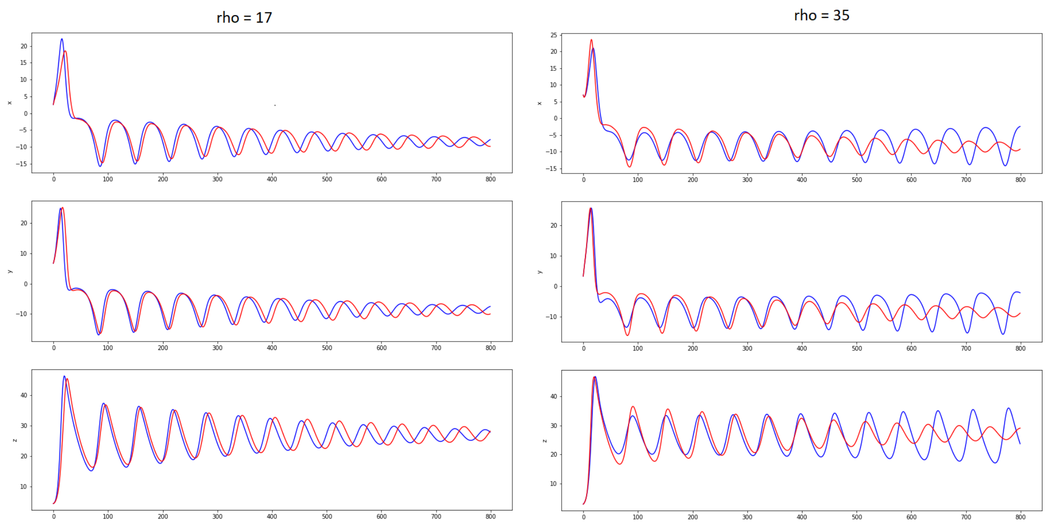

Figure 6. *Future state prediction (red) versus the ODE solver (blue) for Lorenz system with $\rho = 17$ (Left) and $\rho = 35$ (Right).*

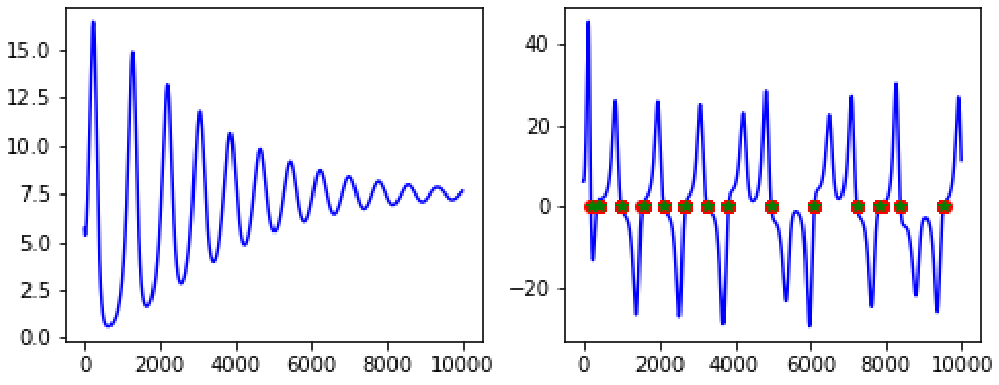

Figure 7. *Two example of lobe switching prediction (red) by neural network for Lorenz system with $\rho = 28$. The ground truth is in green.*

```
# Initialize paramters
num_sols = 20

# Generate data
U = [kuramoto() for j in range(num_sols)] # Exercise 1
```

```

# Get initial conditions
U0 = []
for j in range(num_sols):
    U0.append(U[j][:,0])

n,m = U[0].shape

# Visual Inspection
plt.figure(figsize = (12,4))
plt.subplot(1,2,1)
for u0 in U0: plt.plot(u0)
plt.title('Initial Conditions')
plt.subplot(1,2,2); plt.pcolor(U[0].T, cmap=cm.coolwarm);
plt.xlim([0,n]); plt.ylim([0,m])
plt.title('Example solution')

def transform(U, n,m, num_sols):
    # Standardize data with respect to time
    Uarray_scaled = np.array(U)
    Uarray_scaled = Uarray_scaled.reshape((n*num_sols, m))
    scaler = preprocessing.StandardScaler().fit(Uarray_scaled)
    Uarray_scaled = scaler.transform(Uarray_scaled)
    Uarray_scaled = Uarray_scaled.reshape(num_sols, n, m);
    U_scaled = [ Uarray_scaled[j,:,:] for j in range(num_sols)]
    return U_scaled, scaler

def reverseTransform(U_scaled, scaler, n, m, num_sols):
    # Convert Standardize data into original scale
    Uarray_scaled = np.array(U_scaled)
    Uarray_scaled = Uarray_scaled.reshape((n*num_sols, m))
    Uarray_scaled = scaler.inverse_transform(Uarray_scaled)
    Uarray_scaled = Uarray_scaled.reshape(num_sols, n, m);
    U = [ Uarray_scaled[j,:,:] for j in range(num_sols)]
    return U

# Standardize Data
U_scaled,scaler = transform(U, n,m, num_sols);

# Restructure data
Xtrain = np.array(U_scaled[0][:,:-1])
Ytrain = np.array(U_scaled[0][:,1:])-np.array(U_scaled[0][:,:-1])
for i in range(1,num_sols):
    Xtrain = np.hstack((Xtrain, np.array(U_scaled[i][:,:-1])))
    Ytrain = np.hstack((Ytrain, np.array(U_scaled[i][:,1:])-np.array(
    U_scaled[0][:,:-1])))

######################## Define model ############################
def leakyRelu(x):
    alpha = 0.3
    return tf.nn.relu(x) - alpha * tf.nn.relu(-x)

def sin(x):
    return tf.sin(x)
```

```python

# our model
def create_model():
    model = Sequential()
    size_hidden = n    # size of each hidden layer

    # Input Layer
    # Input_shape is shape of matrix acting on the input vector
    # So we want to do M(3x3000) times X(3000x3)

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

    # Hidden Layer
    model.add(Dense(size_hidden, activation='tanh'))
#     model.add(Dropout(0.01))
    model.add(Dense(size_hidden, activation='sigmoid'))
#     model.add(Dropout(0.01))
    model.add(Dense(size_hidden, activation='relu'))
#     model.add(Dropout(0.01))
    model.add(Dense(size_hidden, activation='tanh'))

    # Output Layer
    model.add(Dense(n, activation='linear'))

    sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='mean_squared_error',
                  optimizer=sgd,
                  metrics=['accuracy'])
    return model
############################################################################

# Initialize
model = create_model()
kfold = KFold(n_splits=2, shuffle=True)

# Cross-validation
for i, (train_index, test_index) in enumerate(kfold.split(Xtrain.T,Ytrain.T)):
        # print("TRAIN:", train_index, "TEST:", test_index)
        Xtrain_mini, Xtest_mini = Xtrain.T[train_index], Xtrain.T[test_index]
        Ytrain_mini, Ytest_mini = Ytrain.T[train_index], Ytrain.T[test_index]
        model.fit(Xtrain_mini, Ytrain_mini, epochs=10, batch_size=1000,
                  verbose=0)
        score = model.evaluate(Xtest_mini, Ytest_mini, batch_size=32, verbose=0)
        print("[fold {0}], Test Loss: {1:.5f}, Test Accuracy: {2:.5f}".format(i, score[0],score[1]))

# Training
history=model.fit(Xtrain.T, Ytrain.T, verbose=0, epochs=100, batch_size=100)

```

```
# semilogy(history.history['acc'])
plt.semilogy(history.history['loss'])

# Predictions

Y_predict = np.zeros((n,m))
p = 10  # p = 1 to 20
Y_predict[:,0] = Xtrain[:,(p-1)*(m-1)]

for j in range(m-1):
     Y_predict[:,j+1]= np.array([ Y_predict[:,j] ]) + model.predict( np.
    array([ Y_predict[:,j] ]) )

plt.figure(figsize = (12,4))
plt.subplot(1,2,1); plt.pcolor(U_scaled[p-1].T, cmap=cm.coolwarm);
plt.title('True'); plt.xlim([0,n]); plt.ylim([0,m])
plt.subplot(1,2,2); plt.pcolor(Y_predict.T, cmap=cm.coolwarm);
plt.title('NN Model'); plt.xlim([0,n]); plt.ylim([0,m])
plt.show()
```

2. Reaction-Diffusion Future State Prediction

```
import numpy as np
from sklearn import preprocessing
from physicsModel import ks_rhs, reactionDiffusion,
    reactionDiffusionReduced

import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from sklearn.model_selection import KFold
import matplotlib.pyplot as plt

# Set parameters
num_runs = 10
rank = 4
U = [] # Data
V = [] # Transformation matrix V
for _ in range(num_runs):
    # Generate Data Random data
    u,v = reactionDiffusionReduced(5, rank)
    U.append(u)
    V.append(v)

# Restructure data
Xtrain = np.array(U[0][:-1,:].T)
Ytrain = np.array(U[0][1:,:].T)
for i in range(1,num_runs):
     Xtrain = np.hstack((Xtrain, np.array(U[i][:-1,:].T) ))
     Ytrain = np.hstack((Ytrain, np.array(U[i][1:,:].T) ))

# our model
def create_model():
     model = Sequential()
     size_hidden = 50    # size of each hidden layer

    # Input Layer
    # Input_shape is shape of matrix acting on the input vector
    # So we want to do M(3x3000) times X(3000x3)
    model.add(Dense(size_hidden, activation='relu', input_dim=rank))

    # Hidden Layer
    model.add(Dense(size_hidden, activation='sigmoid'))
    #     model.add(Dropout(0.1)) # It does nothing to the network sad
    model.add(Dense(size_hidden, activation='relu'))
    #     model.add(Dropout(0.1))
    model.add(Dense(size_hidden, activation='sigmoid'))

    # Output Layer
    model.add(Dense(rank, activation='linear'))

    # sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='mean_squared_error',
                  optimizer='adam',
                  metrics=['accuracy'])
    return model

# Initialize
model = create_model()
kfold = KFold(n_splits=2, shuffle=True)

# Cross validation
for i, (train_index, test_index) in enumerate(kfold.split(Xtrain.T,Ytrain.T)):
        # print("TRAIN:", train_index, "TEST:", test_index)
        Xtrain_mini, Xtest_mini = Xtrain.T[train_index], Xtrain.T[test_index]
        Ytrain_mini, Ytest_mini = Ytrain.T[train_index], Ytrain.T[test_index]
        model.fit(Xtrain_mini, Ytrain_mini, epochs=10, batch_size=100,
                      verbose=0)
        score = model.evaluate(Xtest_mini, Ytest_mini, batch_size=32, verbose=0)
        print("[fold {0}], Test Loss: {1:.5f}, Test Accuracy: {2:.5f}"
              .format(i, score[0],score[1]))

# Real training
history=model.fit(Xtrain.T, Ytrain.T, verbose=0, epochs=100, batch_size=100)
semilogy(history.history['acc'])
semilogy(history.history['loss'])


# Pick an initial point and check
Y_predict = np.zeros((9000,rank))
Y_predict[0,:] = Xtrain.T[0,:]

# Prediction
for j in range(9000-1):
    Y_predict[j+1,:]=model.predict( np.array([ Y_predict[j,:] ]) )

# Back to full space
a = np.matmul(Y_predict, V[0])
b = np.matmul(Ytrain.T[:9000,:], V[0])

plt.subplot(1,2,1);plt.pcolor(a[0].reshape(100,100));plt.title('NN')
plt.subplot(1,2,2);plt.pcolor(b[0].reshape(100,100));plt.title('Truth')
```

3. Lorenz System Future State Prediction

```
import numpy as np
import tensorflow as tf
from physicsModel import lorenz
from nnhelper import minibatch_feeder_lorenz, forward_euler, RK4,
    simple_net
import matplotlib.pyplot as plt

# Initialize parameters
num_runs = 300
m = 800
t = np.linspace(0,8,m+1)[:-1]; dt = (t[1]-t[0])
rho = [10,28,40]
params = (10,28,8.0/3)

# Generate Training Data
U= [lorenz([rho[k],28,8.0/3]) for j in range(num_runs//3)
    for k in range(3)] # Exercise 3

# Generate Test Data
rhotest = [17,35]
test_solution = [lorenz([rhotest[k],28,8.0/3]) for k in range(2)]


U0 = []
for j in range(num_runs):
    # Get Initial conditions
    U0.append(U[j][:,0])

n,m = U[0].shape


# Visual Inspection
figure(figsize = (15,5))
for X in U[:4]:
    x = X[0,:]; y = X[1,:]; z = X[2,:]
    plt.subplot(1,3,1); plt.plot(x, y)
    plt.subplot(1,3,2); plt.plot(y, z)
    plt.subplot(1,3,3); plt.plot(x, z)

x = test_solution[0][0,:]; y = test_solution[0][1,:]; z = test_solution
    [0][2,:]
plt.subplot(1,3,1); plt.plot(x, y, 'r', linewidth = 3); plt.xlabel('x',
    fontsize = 16); plt.ylabel('y', fontsize = 16)
plt.subplot(1,3,2); plt.plot(y, z, 'r', linewidth = 3); plt.xlabel('y',
    fontsize = 16); plt.ylabel('z', fontsize = 16)
plt.subplot(1,3,3); plt.plot(x, z, 'r', linewidth = 3); plt.xlabel('x',
    fontsize = 16); plt.ylabel('z', fontsize = 16)


# Initialize TensorFlow Session
tf.reset_default_graph()
sess = tf.InteractiveSession()

n_hidden = 3          # number of hidden layers in the neural network
size_hidden = 64      # size of each hidden layer

################ Define Model      ##############################

layer_sizes = [n] + [size_hidden for _ in range(n_hidden)] + [n]
num_layers = len(layer_sizes)

weights = []
biases = []

for j in range(1,num_layers):
    weights.append(tf.get_variable("W"+str(j), [layer_sizes[j],
    layer_sizes[j-1]]))
    biases.append(tf.get_variable("b"+str(j), [layer_sizes[j],1]))

# Create Placeholder
X0 = tf.placeholder(name = "X0", shape = (3,None), dtype = tf.float32)
                # initial state to be fed into graph
X1_true = tf.placeholder(name = "X1_true", shape = (3,None), dtype = tf.
    float32)    # true state one timestep later
# X2_true = tf.placeholder(name = "X2_true", shape = (3,None), dtype = tf
    .float32)  # true state two timesteps later
# X3_true = tf.placeholder(name = "X3_true", shape = (3,None), dtype = tf
    .float32)  # true state three timesteps later
h = tf.constant(dt.astype(float32))
X1_pred = simple_net(X0, weights, biases)                    # predicted
    state one timestep later
# X1_pred = RK4(X0, simple_net, h, weights, biases)        # predicted
    state one timestep later
cost = tf.losses.mean_squared_error(X1_true,X1_pred)
lr = tf.placeholder(tf.float32, name = "lr")
optimizer = tf.train.AdamOptimizer(learning_rate=lr).minimize(cost)

sess.run(tf.global_variables_initializer()) # Initialize Variables

############## Validate & Train ##############################


num_epochs = 200
minibatch_size = 1000
batches_per_epoch = int(num_runs*(m-3)/minibatch_size)
feeder = minibatch_feeder_lorenz(U, minibatch_size, steps = 1)
epoch_costs = np.zeros(num_epochs)

```

```python

for epoch in range(num_epochs):

    for batch in range(batches_per_epoch):

        # Select a minibatch
        minibatch = next(feeder)

        # update the weights and biases using this minibatch
        # we also specify values for the step size and regularization parameter
        _, minibatch_cost = sess.run([optimizer, cost], feed_dict={X0: minibatch[0],
                                                                    X1_true: minibatch[1],
                                                                    lr: 0.001})

        # record the cost function evaluated on this minibatch
        epoch_costs[epoch] += minibatch_cost / batches_per_epoch

    # Print the cost every epoch
    if (epoch+1)%10 == 0 or epoch == 0: print ("Cost after epoch %i: %f" % (epoch+1, epoch_costs[epoch]))



# Plot Loss function
plt.figure(figsize = (10,5))
plt.semilogy(epoch_costs)
plt.xlabel('Epoch', fontsize = 16)
plt.ylabel('Cost function value', fontsize = 16)


# Test Prediction
X_test = np.zeros((3,2*m))
X_test[:,0] = U[1][:,0]

for j in range(2*m-1):
    X_test[:,j+1] = X1_pred.eval(feed_dict={X0: X_test[:,j].reshape(3,1)}).flatten()

plt.figure(figsize = (15,15))
plt.title("Rho = "+str(rhotest[1]))
plt.subplot(3,1,1); plt.plot(U[1][0,:], 'b'); plt.plot(X_test[0,:m], 'r'); plt.ylabel('x')
plt.subplot(3,1,2); plt.plot(U[1][1,:], 'b'); plt.plot(X_test[1,:m], 'r'); plt.ylabel('y')
plt.subplot(3,1,3); plt.plot(U[1][2,:], 'b'); plt.plot(X_test[2,:m], 'r'); plt.ylabel('z')
plt.savefig('Lorenz17.png')
```

4. Lorenz Lobe Identification

```python
import numpy as np
import tensorflow as tf
from physicsModel import lorenz
from nnhelper import minibatch_feeder_lorenz, forward_euler, RK4,
    simple_net
from nnhelper import find_lobejump

import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold


# Parameters
num_runs = 100
m = 10000
t = np.linspace(0,8,m+1)[:-1]; dt = (t[1]-t[0])
params = (10,28,8.0/3)

# Generate data
U= [lorenz(t,[28,np.random.randint(1000),8.0/3])
    for j in range(num_runs)]
test_solution = [lorenz(t,[28,np.random.randint(1000),8.0/3])
 for k in range(2)]


U0 = []
for j in range(num_runs):
    # Get initial conditions
    U0.append(U[j][:,0])

n,m = U[0].shape


# Reshape Data
trainY = np.array([find_lobejump(U[0].T)])
for i in range(1,num_runs):
    trainY = np.hstack((trainY, np.array([find_lobejump(U[i].T)]) ))

trainX = U[0][:,:-1]
for i in range(1,num_runs):
    trainX = np.hstack( (trainX, U[i][:,:-1] ))

testX = np.hstack(( test_solution[0][:,:-1], test_solution[1][:,:-1] ))
testY = []
for u in test_solution:
    testY.append(np.array([ find_lobejump(u.T).T ]))
testY = np.hstack((testY[0],testY[1]))



# Since we less transition than no transition
class_weight = {0: 1.,  1: 50.}
```

```


# Deine model
def create_baseline():
    model = Sequential()
    size_hidden = 10    # size of each hidden layer

    # Input Layer
    # Input_shape is shape of matrix acting on the input vector
    # So we want to do M(3x3000) times X(3000x3)
    model.add(Dense(size_hidden, activation='sigmoid', input_dim=3))

    # Hidden Layer
    model.add(Dense(size_hidden, activation='sigmoid'))
    model.add(Dense(size_hidden, activation='sigmoid'))
    model.add(Dense(size_hidden, activation='sigmoid'))

    # Output Layer
    model.add(Dense(1, activation='sigmoid'))

    # sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='binary_crossentropy',
                     optimizer='adam',
                     metrics=['accuracy'])
    return model


# CV model
estimator = KerasClassifier(build_fn=create_baseline, epochs=10,
                             batch_size=1000, verbose=1,
                            class_weight=class_weight)
kfold = StratifiedKFold(n_splits=2, shuffle=True)
results = cross_val_score(estimator, trainX.T, trainY.T, cv=kfold)
print("Results: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()
    *100))


# Train model

estimator.fit(trainX.T, trainY.T, epochs=10,
              batch_size=1000, class_weight=class_weight)



# Regenerate data for testing against our model
test_solution = [lorenz(t,[28,np.random.randint(100),8.0/3])
                  for k in range(10)]

index =  np.random.permutation(10)
textX = test_solution[index[0]][:,:-1]
testY = np.array([ find_lobejump(u.T).T ])
for i in index[1:]:
    testX = np.hstack(( testX,
                    test_solution[i][:,:-1] ))
    testY = np.hstack((testY,
                       np.array([find_lobejump(test_solution[i].T).T])))

# Pick two random initial conditions
index =  np.random.permutation(10)[:2]
testX = np.hstack(( test_solution[index[0]][:,:-1],
                    test_solution[index[1]][:,:-1]  ))
testY = np.hstack((np.array([find_lobejump(test_solution[index[0]].T).T])
    ,
                   np.array([find_lobejump(test_solution[index[1]].T).T])
    ))

prediction = estimator.predict(testX.T)

# Process the predictions before plotting
c0 = prediction[:(2*m-2)//2].copy()
c1 = prediction[(2*m-2)//2:].copy()
s0 = testY.T[:(2*m-2)//2].copy()
s1 = testY.T[(2*m-2)//2:].copy()
print("Prediction: ", c0[c0>0])
print("Truth: ", s0[s0>0])
print("Prediction: ", c1[c1>0])
print("Truth: ", s1[s1>0])

c1[c1 != 0 ] = 1
c0[c0 != 0 ] = 1
c0[c0 == 0] = nan
c1[c1 == 0] = nan
c1[c1 == 1 ] = 0
c0[c0 == 1 ] = 0

s1[s1 != 0 ] = 1
s0[s0 != 0 ] = 1
s0[s0 == 0] = nan
s1[s1 == 0] = nan
s1[s1 == 1 ] = 0
s0[s0 == 1 ] = 0

# Here is the result
figure(figsize=(8,3))
subplot(1,2,1);
plot(test_solution[index[0]][0,:], 'b')
plot(c0, 'ro')
plot(s0, 'g*')
subplot(1,2,2);
plot(test_solution[index[1]][0,:], 'b')
plot(c1, 'ro')
plot(s1, 'g*')
savefig('lobefind.png')
```

B.1. Helper Functions.

1. Models

```
import numpy as np
```

```
from numpy.fft import fft, ifft, fftfreq, ifft2, fft2
from scipy.integrate import odeint

def burgers_rhs(u, t, params):
    k = params
    deriv = -u*ifft(1j*k*fft(u)) + 0.1*ifft(-k**2*fft(u))
    return np.real(deriv)

def burgers():
    '''
    Burgers Equation as reference for NN (by Sam Rudy)
    '''
    # Set size of grid
    n = 256
    m = 257

    # Set x,y,k
    x = np.linspace(-8,8,n+1)[:-1];    dx = x[1]-x[0]
    t = np.linspace(0,10,m);           dt = t[1]-t[0]
    k = 2*np.pi*fftfreq(n, d = dx)

    u0 = [np.exp(-(x+1)**2) + np.sum([0.25**i*np.random.randn()
            *np.sin(i*np.pi*x/8+np.random.randn()) \
             for i in range(1,5)], axis = 0)]

    params = (k)
    return odeint(burgers_rhs, u0[0], t, args=(params,)).T

def kuramoto():
    '''
    Translated from Nathan Kutz's matlab example.

    Kuramoto-Sivashinsky equation

    Return:
    - uu: Solutions from t=0 to t=100
    '''
    N = 100
    x = 32*np.pi*np.linspace(1,N,N)/N;

    # Random initial conditions
    u = np.cos(x/16 + np.random.randn())*(1+np.random.randn()*np.sin(x
    /16))
    v = fft(u);
    h = 0.025;
    k = np.hstack((np.hstack((np.linspace(0,N/2-1,N//2), 0)),
                   np.linspace(-N/2+1,-1,N//2-1)))/16
    L = k**2 - k**4;
    E = np.exp(h*L)
    E2 = np.exp(h*L/2)
    k = k.reshape((np.shape(k)[0],1))
    L = L.reshape((np.shape(L)[0],1))
    E = E.reshape((np.shape(E)[0],1))
    E2 = E2.reshape((np.shape(E2)[0],1))
    M = 16
    r = np.exp(1j*np.pi*(np.linspace(1,M,M)-0.5)/M)
    r = r.reshape(1,np.shape(r)[0])
    LR = h*np.repeat(L,M,axis=1) + np.repeat(r,N,axis=0)
    LR = np.round(LR,4)
    Q = h*np.real(np.mean(np.divide(np.exp(LR/2)-1,LR),1))
    f1= h*np.real(np.mean((-4-LR+np.exp(LR)*(4-3*LR+LR**2))/LR**3,1))
    f2 = h*np.real(np.mean((2+LR+np.exp(LR)*(-2+LR))/LR**3,1))
    f3 = h*np.real(np.mean(
        ((-4-3*LR-np.multiply(LR,LR)+np.exp(LR)*(4-LR))/LR**3),
        1))

    k = k.reshape((np.shape(k)[0],))
    L = L.reshape((np.shape(L)[0],))
    E = E.reshape((np.shape(E)[0],))
    E2 = E2.reshape((np.shape(E2)[0],))

    uu = u;
    tt = 0;
    tmax = 100;
    nmax = np.round(tmax/h);
    nplt = np.floor((tmax/250)/h);
    g = -0.5j*k;
    g = g.reshape((g.shape[0],))
    for n in range(int(nmax)):
        t = (n+1)*h;
        Nv = g*fft(np.real(ifft(v))**2)
        a = E2*v + np.multiply(Q,Nv)
        Na = g*fft(np.real(ifft(a))**2)
        b = E2*v + np.multiply(Q,Na)
        Nb = g*fft(np.real(ifft(b))**2)
        c = E2*a + np.multiply(Q,2*Nb-Nv)
        Nc = g*fft(np.real(ifft(c))**2)
        v = E*v + Nv*f1 + 2*(Na+Nb)*f2 + Nc*f3
        if n%nplt ==0:
            u = np.real(ifft(v))
            uu = np.vstack((uu,u))
            tt = np.hstack((tt,t))
    uu[abs(uu) < 1e-10] =0 # Remove small numbers
    uu = uu.T
    return uu


def ks_rhs(u, x, N):
    '''
    Translated from Nathan Kutz's matlab example.

    Kuramoto-Sivashinsky equation

    Inputs:
    - u: Initial conditions
    - x: Spatial grid
    - N: Spatial grid size

    Return:
    - uu: Solutions from t=0 to t=100
    - tt: Time grid

    '''
    v = fft(u);
    h = 0.025;
    k = np.hstack((np.hstack((np.linspace(0,N/2-1,N//2), 0)),
                   np.linspace(-N/2+1,-1,N//2-1)))/16
    L = k**2 - k**4;
    E = np.exp(h*L)
    E2 = np.exp(h*L/2)
    k = k.reshape((np.shape(k)[0],1))
    L = L.reshape((np.shape(L)[0],1))
    E = E.reshape((np.shape(E)[0],1))
    E2 = E2.reshape((np.shape(E2)[0],1))
    M = 16
    r = np.exp(1j*np.pi*(np.linspace(1,M,M)-0.5)/M)
    r = r.reshape(1,np.shape(r)[0])
    LR = h*np.repeat(L,M, axis=1) + np.repeat(r,N, axis=0)
    LR = np.round(LR,4)
    Q = h*np.real(np.mean(np.divide(np.exp(LR/2)-1,LR),1))
    f1= h*np.real(np.mean((-4-LR+np.exp(LR)*(4-3*LR+LR**2))/LR**3,1))
    f2 = h*np.real(np.mean((2+LR+np.exp(LR)*(-2+LR))/LR**3,1))
    f3 = h*np.real(np.mean(((-4-3*LR-np.multiply(LR,LR)
                             +np.exp(LR)*(4-LR))/LR**3),1))

    k = k.reshape((np.shape(k)[0],))
    L = L.reshape((np.shape(L)[0],))
    E = E.reshape((np.shape(E)[0],))
    E2 = E2.reshape((np.shape(E2)[0],))

    uu = u;
    tt = 0;
    tmax = 100;
    nmax = np.round(tmax/h);
    nplt = np.floor((tmax/250)/h);
    g = -0.5j*k;
    g = g.reshape((g.shape[0],))
    for n in range(int(nmax)):
        t = (n+1)*h;
        Nv = g*fft(np.real(ifft(v))**2)
        a = E2*v + np.multiply(Q,Nv)
        Na = g*fft(np.real(ifft(a))**2)
        b = E2*v + np.multiply(Q,Na)
        Nb = g*fft(np.real(ifft(b))**2)
        c = E2*a + np.multiply(Q,2*Nb-Nv)
        Nc = g*fft(np.real(ifft(c))**2)
        v = E*v + Nv*f1 + 2*(Na+Nb)*f2 + Nc*f3
        if n%nplt ==0:
            u = np.real(ifft(v))
            uu = np.vstack((uu,u))
            tt = np.hstack((tt,t))
    uu[abs(uu) < 1e-10] =0 # Remove small numbers
    uu = uu.T
    return [uu,tt]


def reactionDiffusion(squareSize):
    '''
    Copied from http://ipython-books.github.io/124-simulating-a-partial-
    differential-equation-reaction-diffusion-systems-and-turing-patterns/

    Reaction diffusion equation on square domain

    with Neumann boundary conditions:
        Spatial derivatives normal to the
        boundaries to be zero.
    and Initial condition:
        Spiral or white noise

    Variables:
    - u, concentration of substance 1
    - v, concentration of substance 2

    Input:
    - squareSize: The size of the square domain
    - Size
    - Time

    Return:
    - uu
    - vv

    '''


    # Parameters
    a = 2.8e-4
    b = 5e-3
    #     tau = .1
    tau = np.random.rand()
    k = -1.0*np.random.rand()
    #     k = -.005

    # Define time and space
    size = 100      # size of the 2D grid
    dx = 2.*squareSize / size  # space step
    T = 9.0         # total time
    dt = .001       # time step
    n = int(T / dt) # number of iterations

    # Initial condition: White noise
    #U = np.random.rand(size, size)
    #V = np.random.rand(size, size)

    # Initial condition: Spiral
    xGrid = np.linspace(-squareSize,squareSize,size)
    yGrid = np.linspace(-squareSize,squareSize,size)
    xv,yv = np.meshgrid(xGrid, yGrid)

    m = np.random.randint(1,10)  # Number of spirals
    U = np.tanh(np.sqrt(xv**2 + yv**2)
                )*np.cos(m*np.angle(xv+1j*yv)
                         -np.sqrt(xv**2 + yv**2))

    V = np.tanh(np.sqrt(xv**2 + yv**2)
            )*np.cos(m*np.angle(xv+1j*yv)
                     -np.sqrt(xv**2 + yv**2))

    uu = np.random.rand(n, size, size)
    vv = np.random.rand(n, size, size)

    # We simulate the PDE with the finite difference
    # method.
    for i in range(n):
        # We compute the Laplacian of u and v.
        deltaU = laplacian(U,dx)
        deltaV = laplacian(V,dx)
        # We take the values of u and v inside the grid.
        Uc = U[1:-1, 1:-1]
        Vc = V[1:-1, 1:-1]
        # We update the variables.
        U[1:-1, 1:-1], V[1:-1, 1:-1] = \
            Uc + dt * (a * deltaU + Uc - Uc**3 - Vc + k),\
            Vc + dt * (b * deltaV + Uc - Vc) / tau
        # Neumann conditions: derivatives at the edges
        # are null.
        for Z in (U, V):
            Z[0, :] = Z[1, :]
            Z[-1, :] = Z[-2, :]
            Z[:, 0] = Z[:, 1]
            Z[:, -1] = Z[:, -2]

        uu[i,:,:] = U
        vv[i,:,:] = V

    return [uu,vv]

def laplacian(Z,dx):
    '''
    Helper function for reaction diff
    '''
    Ztop = Z[0:-2, 1:-1]
    Zleft = Z[1:-1, 0:-2]
    Zbottom = Z[2:, 1:-1]
    Zright = Z[1:-1, 2:]
    Zcenter = Z[1:-1, 1:-1]
    return (Ztop + Zleft + Zbottom + Zright -
            4 * Zcenter) / dx**2

def lorenz_rhs(pos, time, params):
    '''
    Lorenz system:
    xt = sigma * (y - x)
    yt = x * (rho-z) - y
    zt = xy - beta * z

    Usage:
    - import integrate from scipy
    - create time-grid t, inital condition x0
    - run command: integrate.odeint(lorenz, x0, t)

    '''
    sigma = params[0]
    rho = params[1]
    beta = params[2]
    return [sigma*(pos[1] - pos[0]),
            rho*pos[0] - pos[1] - pos[0]*pos[2],
            pos[0]*pos[1] - beta*pos[2]]

def lorenz(params):
    m = 800
    n = 3
    t = np.linspace(0,8,m+1)[:-1]; dt = (t[1]-t[0])
    #     t = np.arange(0.0, 15.0, 0.001)
    X0 = np.random.randn(3) + (5,5,5)
    #     state0 = np.random.rand(3)*10    #[2.0, 3.0, 4.0]
    state = odeint(lorenz_rhs, X0, t, args=(params,), rtol=1e-11, atol=1e
-12)
    return state.T

def lorenz(t,params):
    #     t = np.arange(0.0, 15.0, 0.001)
    X0 = np.random.randn(3) + (5,5,5)
    #     X0 = np.random.rand(3)*10    #[2.0, 3.0, 4.0]
    state = odeint(lorenz_rhs, X0, t, args=(params,), rtol=1e-11, atol=1e
-12)
    return state.T
```

2. Neural Network Helper Functions

REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015, https://www.tensorflow.org/. Software available from tensorflow.org.

[2] F. Chollet et al., *Keras*. https://keras.io, 2015.

[3] R. Rojas, *Neural Networks - A Systematic Introduction*, Springer-Verlag Berlin Heidelberg, 1996.