

# Convolutional (CNN) and Recurrent (RNN) Neural Networks

Morten Hjorth-Jensen<sup>1,2</sup>

<sup>1</sup>Department of Physics, University of Oslo

<sup>2</sup>Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Jan 24, 2021

## Plan for Day 7

Reading suggestions: [Aurelien Geron's chapters 13 and 14](#).

**Excellent lectures on CNNs and RNNs.**

- [Video on Convolutional Neural Networks from MIT](#)
- [Video on Recurrent Neural Networks from MIT](#)

## Convolutional Neural Networks (recognizing images)

Convolutional neural networks (CNNs) were developed during the last decade of the previous century, with a focus on character recognition tasks. Nowadays, CNNs are a central element in the spectacular success of deep learning methods. The success in for example image classifications have made them a central tool for most machine learning practitioners.

CNNs are very similar to ordinary Neural Networks. They are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they still have a loss function (for example Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply (back propagation, gradient descent etc etc).

What is the difference? **CNN architectures make the explicit assumption that the inputs are images, which allows us to encode certain**

**properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.**

Here we provide only a superficial overview, for the more interested, we recommend highly the course [IN5400 – Machine Learning for Image Analysis](#) and the slides of [CS231](#).

Another good read is the article here <https://arxiv.org/pdf/1603.07285.pdf>.

## Neural Networks vs CNNs

Neural networks are defined as **affine transformations**, that is a vector is received as input and is multiplied with a matrix of so-called weights (our unknown parameters) to produce an output (to which a bias vector is usually added before passing the result through a nonlinear activation function). This is applicable to any type of input, be it an image, a sound clip or an unordered collection of features: whatever their dimensionality, their representation can always be flattened into a vector before the transformation.

## Why CNNs for images, sound files, medical images from CT scans etc?

However, when we consider images, sound clips and many other similar kinds of data, these data have an intrinsic structure. More formally, they share these important properties:

- They are stored as multi-dimensional arrays (think of the pixels of a figure).
- They feature one or more axes for which ordering matters (e.g., width and height axes for an image, time axis for a sound clip).
- One axis, called the channel axis, is used to access different views of the data (e.g., the red, green and blue channels of a color image, or the left and right channels of a stereo audio track).

These properties are not exploited when an affine transformation is applied; in fact, all the axes are treated in the same way and the topological information is not taken into account. Still, taking advantage of the implicit structure of the data may prove very handy in solving some tasks, like computer vision and speech recognition, and in these cases it would be best to preserve it. This is where discrete convolutions come into play.

A discrete convolution is a linear transformation that preserves this notion of ordering. It is sparse (only a few input units contribute to a given output unit) and reuses parameters (the same weights are applied to multiple locations in the input).

## Regular NNs don't scale well to full images

As an example, consider an image of size  $32 \times 32 \times 3$  (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer of a regular Neural Network would have  $32 \times 32 \times 3 = 3072$  weights. This amount still seems manageable, but clearly this fully-connected structure does not scale to larger images. For example, an image of more respectable size, say  $200 \times 200 \times 3$ , would lead to neurons that have  $200 \times 200 \times 3 = 120,000$  weights.

We could have several such neurons, and the parameters would add up quickly! Clearly, this full connectivity is wasteful and the huge number of parameters would quickly lead to possible overfitting.

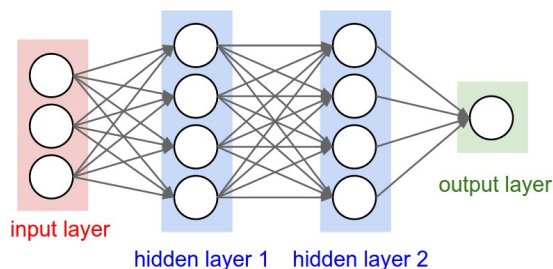


Figure 1: A regular 3-layer Neural Network.

## 3D volumes of neurons

Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way.

In particular, unlike a regular Neural Network, the layers of a CNN have neurons arranged in 3 dimensions: width, height, depth. (Note that the word depth here refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network.)

To understand it better, the above example of an image with an input volume of activations has dimensions  $32 \times 32 \times 3$  (width, height, depth respectively).

The neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. Moreover, the final output layer could for this specific image have dimensions  $1 \times 1 \times 10$ , because by the end of the CNN architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension.

## Layers used to build CNNs

A simple CNN is a sequence of layers, and every layer of a CNN transforms one volume of activations to another through a differentiable function. We use three main types of layers to build CNN architectures: Convolutional Layer, Pooling

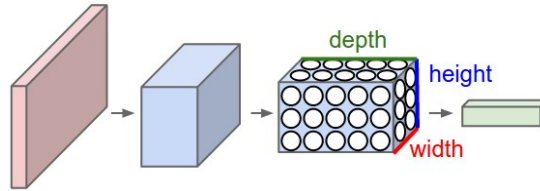


Figure 2: A CNN arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a CNN transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

Layer, and Fully-Connected Layer (exactly as seen in regular Neural Networks). We will stack these layers to form a full CNN architecture.

A simple CNN for image classification could have the architecture:

- **INPUT** ( $32 \times 32 \times 3$ ) will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.
- **CONV** (convolutional) layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as  $[32 \times 32 \times 12]$  if we decided to use 12 filters.
- **RELU** layer will apply an elementwise activation function, such as the  $\max(0, x)$  thresholding at zero. This leaves the size of the volume unchanged ( $[32 \times 32 \times 12]$ ).
- **POOL** (pooling) layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as  $[16 \times 16 \times 12]$ .
- **FC** (i.e. fully-connected) layer will compute the class scores, resulting in volume of size  $[1 \times 1 \times 10]$ , where each of the 10 numbers correspond to a class score, such as among the 10 categories of the MNIST images we considered above. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

## Transforming images

CNNs transform the original image layer by layer from the original pixel values to the final class scores.

Observe that some layers contain parameters and other don't. In particular, the CNN layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons). On the other hand, the RELU/POOL layers will implement a fixed function. The parameters in the CONV/FC layers will be trained with gradient descent so that the class scores that the CNN computes are consistent with the labels in the training set for each image.

## CNNs in brief

In summary:

- A CNN architecture is in the simplest case a list of Layers that transform the image volume into an output volume (e.g. holding the class scores)
- There are a few distinct types of Layers (e.g. CONV/FC/RELU/POOL are by far the most popular)
- Each Layer accepts an input 3D volume and transforms it to an output 3D volume through a differentiable function
- Each Layer may or may not have parameters (e.g. CONV/FC do, RELU/POOL don't)
- Each Layer may or may not have additional hyperparameters (e.g. CONV/FC/POOL do, RELU doesn't)

For more material on convolutional networks, we strongly recommend the course [IN5400 – Machine Learning for Image Analysis](#) and the slides of [CS231](#) which is taught at Stanford University (consistently ranked as one of the top computer science programs in the world). [Michael Nielsen's book is a must read](#), in particular [chapter 6](#) which deals with CNNs.

## CNNs in more detail, building convolutional neural networks in Tensorflow and Keras

As discussed above, CNNs are neural networks built from the assumption that the inputs to the network are 2D images. This is important because the number of features or pixels in images grows very fast with the image size, and an enormous number of weights and biases are needed in order to build an accurate network.

As before, we still have our input, a hidden layer and an output. What's novel about convolutional networks are the **convolutional** and **pooling** layers stacked in pairs between the input and the hidden layer. In addition, the data is no longer represented as a 2D feature matrix, instead each input is a number of 2D matrices, typically 1 for each color dimension (Red, Green, Blue).

## Setting it up

It means that to represent the entire dataset of images, we require a 4D matrix or **tensor**. This tensor has the dimensions:

$$(n_{inputs}, n_{pixels,width}, n_{pixels,height}, depth).$$

## The MNIST dataset again

The MNIST dataset consists of grayscale images with a pixel size of  $28 \times 28$ , meaning we require  $28 \times 28 = 784$  weights to each neuron in the first hidden layer.

If we were to analyze images of size  $128 \times 128$  we would require  $128 \times 128 = 16384$  weights to each neuron. Even worse if we were dealing with color images, as most images are, we have an image matrix of size  $128 \times 128$  for each color dimension (Red, Green, Blue), meaning 3 times the number of weights = 49152 are required for every single neuron in the first hidden layer.

## Strong correlations

Images typically have strong local correlations, meaning that a small part of the image varies little from its neighboring regions. If for example we have an image of a blue car, we can roughly assume that a small blue part of the image is surrounded by other blue regions.

Therefore, instead of connecting every single pixel to a neuron in the first hidden layer, as we have previously done with deep neural networks, we can instead connect each neuron to a small part of the image (in all 3 RGB depth dimensions). The size of each small area is fixed, and known as a **receptive**.

## Layers of a CNN

The layers of a convolutional neural network arrange neurons in 3D: width, height and depth. The input image is typically a square matrix of depth 3.

A **convolution** is performed on the image which outputs a 3D volume of neurons. The weights to the input are arranged in a number of 2D matrices, known as **filters**.

Each filter slides along the input image, taking the dot product between each small part of the image and the filter, in all depth dimensions. This is then passed through a non-linear function, typically the **Rectified Linear (ReLU)** function, which serves as the activation of the neurons in the first convolutional layer. This is further passed through a **pooling layer**, which reduces the size of the convolutional layer, e.g. by taking the maximum or average across some small regions, and this serves as input to the next convolutional layer.

## Systematic reduction

By systematically reducing the size of the input volume, through convolution and pooling, the network should create representations of small parts of the input, and then from them assemble representations of larger areas. The final pooling layer is flattened to serve as input to a hidden layer, such that each neuron in the final pooling layer is connected to every single neuron in the hidden layer. This then serves as input to the output layer, e.g. a softmax output for classification.

## Prerequisites: Collect and pre-process data

```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets

# ensure the same random numbers appear every time
np.random.seed(0)

# display images in notebook
%matplotlib inline
plt.rcParams['figure.figsize'] = (12,12)

# download MNIST dataset
digits = datasets.load_digits()

# define inputs and labels
inputs = digits.images
labels = digits.target

# RGB images have a depth of 3
# our images are grayscale so they should have a depth of 1
inputs = inputs[:, :, np.newaxis]

print("inputs = (n_inputs, pixel_width, pixel_height, depth) = " + str(inputs.shape))
print("labels = (n_inputs) = " + str(labels.shape))

# choose some random images to display
n_inputs = len(inputs)
indices = np.arange(n_inputs)
random_indices = np.random.choice(indices, size=5)

for i, image in enumerate(digits.images[random_indices]):
    plt.subplot(1, 5, i+1)
    plt.axis('off')
    plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    plt.title("Label: %d" % digits.target[random_indices[i]])
plt.show()
```

## Importing Keras and Tensorflow

```
from tensorflow.keras import datasets, layers, models
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Sequential      #This allows appending layers to existing models
```

```

from tensorflow.keras.layers import Dense           #This allows defining the characteristics of
from tensorflow.keras import optimizers             #This allows using whichever optimiser we want
from tensorflow.keras import regularizers           #This allows using whichever regularizer we want
from tensorflow.keras.utils import to_categorical    #This allows using categorical cross entropy loss

#from tensorflow.keras import Conv2D
#from tensorflow.keras import MaxPooling2D
#from tensorflow.keras import Flatten

from sklearn.model_selection import train_test_split

# representation of labels
labels = to_categorical(labels)

# split into train and test data
# one-liner from scikit-learn library
train_size = 0.8
test_size = 1 - train_size
X_train, X_test, Y_train, Y_test = train_test_split(inputs, labels, train_size=train_size,
                                                    test_size=test_size)

```

## Running with Keras

```

def create_convolutional_neural_network_keras(input_shape, receptive_field,
                                              n_filters, n_neurons_connected, n_categories,
                                              eta, lmbd):

    model = Sequential()
    model.add(layers.Conv2D(n_filters, (receptive_field, receptive_field), input_shape=input_shape,
                             activation='relu', kernel_regularizer=regularizers.l2(lmbd)))
    model.add(layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(layers.Flatten())
    model.add(layers.Dense(n_neurons_connected, activation='relu', kernel_regularizer=regularizers.l2(lmbd)))
    model.add(layers.Dense(n_categories, activation='softmax', kernel_regularizer=regularizers.l2(lmbd)))

    sgd = optimizers.SGD(lr=eta)
    model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

    return model

epochs = 100
batch_size = 100
input_shape = X_train.shape[1:4]
receptive_field = 3
n_filters = 10
n_neurons_connected = 50
n_categories = 10

eta_vals = np.logspace(-5, 1, 7)
lmbd_vals = np.logspace(-5, 1, 7)

```

## Final part

```

CNN_keras = np.zeros((len(eta_vals), len(lmbd_vals)), dtype=object)

for i, eta in enumerate(eta_vals):
    for j, lmbd in enumerate(lmbd_vals):
        CNN = create_convolutional_neural_network_keras(input_shape, receptive_field,
                                                         n_filters, n_neurons_connected, n_categories,
                                                         eta, lmbd)
        CNN.fit(X_train, Y_train, epochs=epochs, batch_size=batch_size, verbose=0)

```



```

scores = CNN.evaluate(X_test, Y_test)

CNN_keras[i][j] = CNN

print("Learning rate = ", eta)
print("Lambda = ", lmbd)
print("Test accuracy: %.3f" % scores[1])
print()

```

## Final visualization

```

# visual representation of grid search
# uses seaborn heatmap, could probably do this in matplotlib
import seaborn as sns

sns.set()

train_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))
test_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))

for i in range(len(eta_vals)):
    for j in range(len(lmbd_vals)):
        CNN = CNN_keras[i][j]

        train_accuracy[i][j] = CNN.evaluate(X_train, Y_train)[1]
        test_accuracy[i][j] = CNN.evaluate(X_test, Y_test)[1]

fig, ax = plt.subplots(figsize = (10, 10))
sns.heatmap(train_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Training Accuracy")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.show()

fig, ax = plt.subplots(figsize = (10, 10))
sns.heatmap(test_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Test Accuracy")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.show()

```

## The CIFAR01 data set

The CIFAR10 dataset contains 60,000 color images in 10 classes, with 6,000 images in each class. The dataset is divided into 50,000 training images and 10,000 testing images. The classes are mutually exclusive and there is no overlap between them.

```

import tensorflow as tf

from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

# We import the data set
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

```

```
# Normalize pixel values to be between 0 and 1 by dividing by 255.
train_images, test_images = train_images / 255.0, test_images / 255.0
```

## Verifying the data set

To verify that the dataset looks correct, let's plot the first 25 images from the training set and display the class name below each image.

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    # The CIFAR labels happen to be arrays,
    # which is why you need the extra index
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```

## Set up the model

The 6 lines of code below define the convolutional base using a common pattern: a stack of Conv2D and MaxPooling2D layers.

As input, a CNN takes tensors of shape  $(\text{image}_{\text{height}}, \text{image}_{\text{width}}, \text{color}_{\text{channels}})$ , ignoring the batch size. If y

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

# Let's display the architecture of our model so far.

model.summary()
```

You can see that the output of every Conv2D and MaxPooling2D layer is a 3D tensor of shape (height, width, channels). The width and height dimensions tend to shrink as you go deeper in the network. The number of output channels for each Conv2D layer is controlled by the first argument (e.g., 32 or 64). Typically, as the width and height shrink, you can afford (computationally) to add more output channels in each Conv2D layer.

## Add Dense layers on top

To complete our model, you will feed the last output tensor from the convolutional base (of shape (4, 4, 64)) into one or more Dense layers to perform classification. Dense layers take vectors as input (which are 1D), while the current output is a 3D tensor. First, you will flatten (or unroll) the 3D output to 1D, then add one

or more Dense layers on top. CIFAR has 10 output classes, so you use a final Dense layer with 10 outputs and a softmax activation.

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
Here's the complete architecture of our model.

model.summary()
```

As you can see, our (4, 4, 64) outputs were flattened into vectors of shape (1024) before going through two Dense layers.

## Compile and train the model

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
```

## Finally, evaluate the model

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')

test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)

print(test_acc)
```

## Recurrent neural networks: Overarching view

Till now our focus has been, including convolutional neural networks as well, on feedforward neural networks. The output or the activations flow only in one direction, from the input layer to the output layer.

A recurrent neural network (RNN) looks very much like a feedforward neural network, except that it also has connections pointing backward.

RNNs are used to analyze time series data such as stock prices, and tell you when to buy or sell. In autonomous driving systems, they can anticipate car trajectories and help avoid accidents. More generally, they can work on sequences of arbitrary lengths, rather than on fixed-sized inputs like all the nets we have discussed so far. For example, they can take sentences, documents, or audio samples as input, making them extremely useful for natural language processing systems such as automatic translation and speech-to-text.

## Set up of an RNN

### A simple example

```
# Start importing packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Dense, SimpleRNN, LSTM, GRU
from tensorflow.keras import optimizers
from tensorflow.keras import regularizers
from tensorflow.keras.utils import to_categorical

# convert into dataset matrix
def convertToMatrix(data, step):
    X, Y = [], []
    for i in range(len(data)-step):
        d=i+step
        X.append(data[i:d,])
        Y.append(data[d,])
    return np.array(X), np.array(Y)

step = 4
N = 1000
Tp = 800

t=np.arange(0,N)
x=np.sin(0.02*t)+2*np.random.rand(N)
df = pd.DataFrame(x)
df.head()

plt.plot(df)
plt.show()

values=df.values
train,test = values[0:Tp,:], values[Tp:N,:]

# add step elements into train and test
test = np.append(test,np.repeat(test[-1,],step))
train = np.append(train,np.repeat(train[-1,],step))

trainX,trainY =convertToMatrix(train,step)
testX,testY =convertToMatrix(test,step)
trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))

model = Sequential()
model.add(SimpleRNN(units=32, input_shape=(1,step), activation="relu"))
model.add(Dense(8, activation="relu"))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='rmsprop')
model.summary()

model.fit(trainX,trainY, epochs=100, batch_size=16, verbose=2)
trainPredict = model.predict(trainX)
```

```

testPredict= model.predict(testX)
predicted=np.concatenate((trainPredict,testPredict),axis=0)

trainScore = model.evaluate(trainX, trainY, verbose=0)
print(trainScore)

index = df.index.values
plt.plot(index,df)
plt.plot(index,predicted)
plt.axvline(df.index[Tp], c="r")
plt.show()

```

## An extrapolation example

The following code provides an example of how recurrent neural networks can be used to extrapolate to unknown values of physics data sets. Specifically, the data sets used in this program come from a quantum mechanical many-body calculation of energies as functions of the number of particles.

```

# For matrices and calculations
import numpy as np
# For machine learning (backend for keras)
import tensorflow as tf
# User-friendly machine learning library
# Front end for TensorFlow
import tensorflow.keras
# Different methods from Keras needed to create an RNN
# This is not necessary but it shortened function calls
# that need to be used in the code.
from tensorflow.keras import datasets, layers, models
from tensorflow.keras.layers import Input
from tensorflow.keras import regularizers
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Dense, SimpleRNN, LSTM, GRU
# For timing the code
from timeit import default_timer as timer
# For plotting
import matplotlib.pyplot as plt

# The data set
datatype='VaryDimension'
X_tot = np.arange(2, 42, 2)
y_tot = np.array([-0.03077640549, -0.08336233266, -0.1446729567, -0.2116753732, -0.2830637392, -0.3619067271, -0.4419067271, -0.5219067271, -0.6019067271, -0.6819067271, -0.7619067271, -0.8419067271, -0.9219067271, -1.0019067271, -1.0819067271, -1.1619067271, -1.2419067271, -1.3219067271, -1.4019067271, -1.4819067271, -1.5619067271, -1.6419067271, -1.7219067271, -1.8019067271, -1.8819067271, -1.9619067271, -2.0419067271, -2.1219067271, -2.2019067271, -2.2819067271, -2.3619067271, -2.4419067271, -2.5219067271, -2.6019067271, -2.6819067271, -2.7619067271, -2.8419067271, -2.9219067271, -3.0019067271, -3.0819067271, -3.1619067271, -3.2419067271, -3.3219067271, -3.4019067271, -3.4819067271, -3.5619067271, -3.6419067271, -3.7219067271, -3.8019067271, -3.8819067271, -3.9619067271, -4.0419067271, -4.1219067271, -4.2019067271, -4.2819067271, -4.3619067271, -4.4419067271, -4.5219067271, -4.6019067271, -4.6819067271, -4.7619067271, -4.8419067271, -4.9219067271, -5.0019067271, -5.0819067271, -5.1619067271, -5.2419067271, -5.3219067271, -5.4019067271, -5.4819067271, -5.5619067271, -5.6419067271, -5.7219067271, -5.8019067271, -5.8819067271, -5.9619067271, -6.0419067271, -6.1219067271, -6.2019067271, -6.2819067271, -6.3619067271, -6.4419067271, -6.5219067271, -6.6019067271, -6.6819067271, -6.7619067271, -6.8419067271, -6.9219067271, -7.0019067271, -7.0819067271, -7.1619067271, -7.2419067271, -7.3219067271, -7.4019067271, -7.4819067271, -7.5619067271, -7.6419067271, -7.7219067271, -7.8019067271, -7.8819067271, -7.9619067271, -8.0419067271, -8.1219067271, -8.2019067271, -8.2819067271, -8.3619067271, -8.4419067271, -8.5219067271, -8.6019067271, -8.6819067271, -8.7619067271, -8.8419067271, -8.9219067271, -9.0019067271, -9.0819067271, -9.1619067271, -9.2419067271, -9.3219067271, -9.4019067271, -9.4819067271, -9.5619067271, -9.6419067271, -9.7219067271, -9.8019067271, -9.8819067271, -9.9619067271, -10.0419067271, -10.1219067271, -10.2019067271, -10.2819067271, -10.3619067271, -10.4419067271, -10.5219067271, -10.6019067271, -10.6819067271, -10.7619067271, -10.8419067271, -10.9219067271, -11.0019067271, -11.0819067271, -11.1619067271, -11.2419067271, -11.3219067271, -11.4019067271, -11.4819067271, -11.5619067271, -11.6419067271, -11.7219067271, -11.8019067271, -11.8819067271, -11.9619067271, -12.0419067271, -12.1219067271, -12.2019067271, -12.2819067271, -12.3619067271, -12.4419067271, -12.5219067271, -12.6019067271, -12.6819067271, -12.7619067271, -12.8419067271, -12.9219067271, -13.0019067271, -13.0819067271, -13.1619067271, -13.2419067271, -13.3219067271, -13.4019067271, -13.4819067271, -13.5619067271, -13.6419067271, -13.7219067271, -13.8019067271, -13.8819067271, -13.9619067271, -14.0419067271, -14.1219067271, -14.2019067271, -14.2819067271, -14.3619067271, -14.4419067271, -14.5219067271, -14.6019067271, -14.6819067271, -14.7619067271, -14.8419067271, -14.9219067271, -15.0019067271, -15.0819067271, -15.1619067271, -15.2419067271, -15.3219067271, -15.4019067271, -15.4819067271, -15.5619067271, -15.6419067271, -15.7219067271, -15.8019067271, -15.8819067271, -15.9619067271, -16.0419067271, -16.1219067271, -16.2019067271, -16.2819067271, -16.3619067271, -16.4419067271, -16.5219067271, -16.6019067271, -16.6819067271, -16.7619067271, -16.8419067271, -16.9219067271, -17.0019067271, -17.0819067271, -17.1619067271, -17.2419067271, -17.3219067271, -17.4019067271, -17.4819067271, -17.5619067271, -17.6419067271, -17.7219067271, -17.8019067271, -17.8819067271, -17.9619067271, -18.0419067271, -18.1219067271, -18.2019067271, -18.2819067271, -18.3619067271, -18.4419067271, -18.5219067271, -18.6019067271, -18.6819067271, -18.7619067271, -18.8419067271, -18.9219067271, -19.0019067271, -19.0819067271, -19.1619067271, -19.2419067271, -19.3219067271, -19.4019067271, -19.4819067271, -19.5619067271, -19.6419067271, -19.7219067271, -19.8019067271, -19.8819067271, -19.9619067271, -20.0419067271, -20.1219067271, -20.2019067271, -20.2819067271, -20.3619067271, -20.4419067271, -20.5219067271, -20.6019067271, -20.6819067271, -20.7619067271, -20.8419067271, -20.9219067271, -21.0019067271, -21.0819067271, -21.1619067271, -21.2419067271, -21.3219067271, -21.4019067271, -21.4819067271, -21.5619067271, -21.6419067271, -21.7219067271, -21.8019067271, -21.8819067271, -21.9619067271, -22.0419067271, -22.1219067271, -22.2019067271, -22.2819067271, -22.3619067271, -22.4419067271, -22.5219067271, -22.6019067271, -22.6819067271, -22.7619067271, -22.8419067271, -22.9219067271, -23.0019067271, -23.0819067271, -23.1619067271, -23.2419067271, -23.3219067271, -23.4019067271, -23.4819067271, -23.5619067271, -23.6419067271, -23.7219067271, -23.8019067271, -23.8819067271, -23.9619067271, -24.0419067271, -24.1219067271, -24.2019067271, -24.2819067271, -24.3619067271, -24.4419067271, -24.5219067271, -24.6019067271, -24.6819067271, -24.7619067271, -24.8419067271, -24.9219067271, -25.0019067271, -25.0819067271, -25.1619067271, -25.2419067271, -25.3219067271, -25.4019067271, -25.4819067271, -25.5619067271, -25.6419067271, -25.7219067271, -25.8019067271, -25.8819067271, -25.9619067271, -26.0419067271, -26.1219067271, -26.2019067271, -26.2819067271, -26.3619067271, -26.4419067271, -26.5219067271, -26.6019067271, -26.6819067271, -26.7619067271, -26.8419067271, -26.9219067271, -27.0019067271, -27.0819067271, -27.1619067271, -27.2419067271, -27.3219067271, -27.4019067271, -27.4819067271, -27.5619067271, -27.6419067271, -27.7219067271, -27.8019067271, -27.8819067271, -27.9619067271, -28.0419067271, -28.1219067271, -28.2019067271, -28.2819067271, -28.3619067271, -28.4419067271, -28.5219067271, -28.6019067271, -28.6819067271, -28.7619067271, -28.8419067271, -28.9219067271, -29.0019067271, -29.0819067271, -29.1619067271, -29.2419067271, -29.3219067271, -29.4019067271, -29.4819067271, -29.5619067271, -29.6419067271, -29.7219067271, -29.8019067271, -29.8819067271, -29.9619067271, -30.0419067271, -30.1219067271, -30.2019067271, -30.2819067271, -30.3619067271, -30.4419067271, -30.5219067271, -30.6019067271, -30.6819067271, -30.7619067271, -30.8419067271, -30.9219067271, -31.0019067271, -31.0819067271, -31.1619067271, -31.2419067271, -31.3219067271, -31.4019067271, -31.4819067271, -31.5619067271, -31.6419067271, -31.7219067271, -31.8019067271, -31.8819067271, -31.9619067271, -32.0419067271, -32.1219067271, -32.2019067271, -32.2819067271, -32.3619067271, -32.4419067271, -32.5219067271, -32.6019067271, -32.6819067271, -32.7619067271, -32.8419067271, -32.9219067271, -33.0019067271, -33.0819067271, -33.1619067271, -33.2419067271, -33.3219067271, -33.4019067271, -33.4819067271, -33.5619067271, -33.6419067271, -33.7219067271, -33.8019067271, -33.8819067271, -33.9619067271, -34.0419067271, -34.1219067271, -34.2019067271, -34.2819067271, -34.3619067271, -34.4419067271, -34.5219067271, -34.6019067271, -34.6819067271, -34.7619067271, -34.8419067271, -34.9219067271, -35.0019067271, -35.0819067271, -35.1619067271, -35.2419067271, -35.3219067271, -35.4019067271, -35.4819067271, -35.5619067271, -35.6419067271, -35.7219067271, -35.8019067271, -35.8819067271, -35.9619067271, -36.0419067271, -36.1219067271, -36.2019067271, -36.2819067271, -36.3619067271, -36.4419067271, -36.5219067271, -36.6019067271, -36.6819067271, -36.7619067271, -36.8419067271, -36.9219067271, -37.0019067271, -37.0819067271, -37.1619067271, -37.2419067271, -37.3219067271, -37.4019067271, -37.4819067271, -37.5619067271, -37.6419067271, -37.7219067271, -37.8019067271, -37.8819067271, -37.9619067271, -38.0419067271, -38.1219067271, -38.2019067271, -38.2819067271, -38.3619067271, -38.4419067271, -38.5219067271, -38.6019067271, -38.6819067271, -38.7619067271, -38.8419067271, -38.9219067271, -39.0019067271, -39.0819067271, -39.1619067271, -39.2419067271, -39.3219067271, -39.4019067271, -39.4819067271, -39.5619067271, -39.6419067271, -39.7219067271, -39.8019067271, -39.8819067271, -39.9619067271, -40.0419067271, -40.1219067271, -40.2019067271, -40.2819067271, -40.3619067271, -40.4419067271, -40.5219067271, -40.6019067271, -40.6819067271, -40.7619067271, -40.8419067271, -40.9219067271, -41.0019067271, -41.0819067271, -41.1619067271, -41.2419067271, -41.3219067271, -41.4019067271, -41.4819067271, -41.5619067271, -41.6419067271, -41.7219067271, -41.8019067271, -41.8819067271, -41.9619067271, -42.0419067271, -42.1219067271, -42.2019067271, -42.2819067271, -42.3619067271, -42.4419067271, -42.5219067271, -42.6019067271, -42.6819067271, -42.7619067271, -42.8419067271, -42.9219067271, -43.0019067271, -43.0819067271, -43.1619067271, -43.2419067271, -43.3219067271, -43.4019067271, -43.4819067271, -43.5619067271, -43.6419067271, -43.7219067271, -43.8019067271, -43.8819067271, -43.9619067271, -44.0419067271, -44.1219067271, -44.2019067271, -44.2819067271, -44.3619067271, -44.4419067271, -44.5219067271, -44.6019067271, -44.6819067271, -44.7619067271, -44.8419067271, -44.9219067271, -45.0019067271, -45.0819067271, -45.1619067271, -45.2419067271, -45.3219067271, -45.4019067271, -45.4819067271, -45.5619067271, -45.6419067271, -45.7219067271, -45.8019067271, -45.8819067271, -45.9619067271, -46.0419067271, -46.1219067271, -46.2019067271, -46.2819067271, -46.3619067271, -46.4419067271, -46.5219067271, -46.6019067271, -46.6819067271, -46.7619067271, -46.8419067271, -46.9219067271, -47.0019067271, -47.0819067271, -47.1619067271, -47.2419067271, -47.3219067271, -47.4019067271, -47.4819067271, -47.5619067271, -47.6419067271, -47.7219067271, -47.8019067271, -47.8819067271, -47.9619067271, -48.0419067271, -48.1219067271, -48.2019067271, -48.2819067271, -48.3619067271, -48.4419067271, -48.5219067271, -48.6019067271, -48.6819067271, -48.7619067271, -48.8419067271, -48.9219067271, -49.0019067271, -49.0819067271, -49.1619067271, -49.2419067271, -49.3219067271, -49.4019067271, -49.4819067271, -49.5619067271, -49.6419067271, -49.7219067271, -49.8019067271, -49.8819067271, -49.9619067271, -50.0419067271, -50.1219067271, -50.2019067271, -50.2819067271, -50.3619067271, -50.4419067271, -50.5219067271, -50.6019067271, -50.6819067271, -50.7619067271, -50.8419067271, -50.9219067271, -51.0019067271, -51.0819067271, -51.1619067271, -51.2419067271, -51.3219067271, -51.4019067271, -51.4819067271, -51.5619067271, -51.6419067271, -51.7219067271, -51.8019067271, -51.8819067271, -51.9619067271, -52.0419067271, -52.1219067271, -52.2019067271, -52.2819067271, -52.3619067271, -52.4419067271, -52.5219067271, -52.6019067271, -52.6819067271, -52.7619067271, -52.8419067271, -52.9219067271, -53.0019067271, -53.0819067271, -53.1619067271, -53.2419067271, -53.3219067271, -53.4019067271, -53.4819067271, -53.5619067271, -53.6419067271, -53.7219067271, -53.8019067271, -53.8819067271, -53.9619067271, -54.0419067271, -54.1219067271, -54.2019067271, -54.2819067271, -54.3619067271, -54.4419067271, -54.5219067271, -54.6019067271, -54.6819067271, -54.7619067271, -54.8419067271, -54.9219067271, -55.0019067271, -55.0819067271, -55.1619067271, -55.2419067271, -55.3219067271, -55.4019067271, -55.4819067271, -55.5619067271, -55.6419067271, -55.7219067271, -55.8019067271, -55.8819067271, -55.9619067271, -56.0419067271, -56.1219067271, -56.2019067271, -56.2819067271, -56.3619067271, -56.4419067271, -56.5219067271, -56.6019067271, -56.6819067271, -56.7619067271, -56.8419067271, -56.9219067271, -57.0019067271, -57.0819067271, -57.1619067271, -57.2419067271, -57.3219067271, -57.4019067271, -57.4819067271, -57.5619067271, -57.6419067271, -57.7219067271, -57.8019067271, -57.8819067271, -57.9619067271, -58.0419067271, -58.1219067271, -58.2019067271, -58.2819067271, -58.3619067271, -58.4419067271, -58.5219067271, -58.6019067271, -58.6819067271, -58.7619067271, -58.8419067271, -58.9219067271, -59.0019067271, -59.0819067271, -59.1619067271, -59.2419067271, -59.3219067271, -59.4019067271, -59.4819067271, -59.5619067271, -59.6419067271, -59.7219067271, -59.8019067271, -59.8819067271, -59.9619067271, -60.0419067271, -60.1219067271, -60.2019067271, -60.2819067271, -60.3619067271, -60.4419067271, -60.5219067271, -60.6019067271, -60.6819067271, -60.7619067271, -60.8419067271, -60.9219067271, -61.0019067271, -61.0819067271, -61.1619067271, -61.2419067271, -61.3219067271, -61.4019067271, -61.4819067271, -61.5619067271, -61.6419067271, -61.7219067271, -61.8019067271, -61.8819067271, -61.9619067271, -62.0419067271, -62.1219067271, -62.2019067271, -62.2819067271, -62.3619067271, -62.4419067271, -62.5219067271, -62.6019067271, -62.6819067271, -62.7619067271, -62.8419067271, -62.9219067271, -63.0019067271, -63.0819067271, -63.1619067271, -63.2419067271, -63.3219067271, -63.4019067271, -63.4819067271, -63.5619067271, -63.6419067271, -63.7219067271, -63.8019067271, -63.8819067271, -6
```

any extrapolation (time series forecasting is just a specific type of extrapolation along the time axis). This method of data formatting does not use the x data and assumes that the y data are evenly spaced.

For a standard machine learning algorithm, the training data has the form of (x,y) so the machine learning algorithm learns to associate a y value with a given x value. This is useful when the test data has x values within the same range as the training data. However, for this application, the x values of the test data are outside of the x values of the training data and the traditional method of training a machine learning algorithm does not work as well. For this reason, the recurrent neural network is trained on sequences of y values of the form ((y1, y2), y3), so that the network is concerned with learning the pattern of the y data and not the relation between the x and y data. As long as the pattern of y data outside of the training region stays relatively stable compared to what was inside the training region, this method of training can produce accurate extrapolations to y values far removed from the training data set.

```
# FORMAT_DATA
def format_data(data, length_of_sequence = 2):
    """
        Inputs:
            data(a numpy array): the data that will be the inputs to the recurrent neural
            network
            length_of_sequence (an int): the number of elements in one iteration of the
            sequence patter. For a function approximator use length_of_sequence = 2.
        Returns:
            rnn_input (a 3D numpy array): the input data for the recurrent neural network. Its
            dimensions are length of data - length of sequence, length of sequence,
            dimnsion of data
            rnn_output (a numpy array): the training data for the neural network
        Formats data to be used in a recurrent neural network.
    """

    X, Y = [], []
    for i in range(len(data)-length_of_sequence):
        # Get the next length_of_sequence elements
        a = data[i:i+length_of_sequence]
        # Get the element that immediately follows that
        b = data[i+length_of_sequence]
        # Reshape so that each data point is contained in its own array
        a = np.reshape (a, (len(a), 1))
        X.append(a)
        Y.append(b)
    rnn_input = np.array(X)
    rnn_output = np.array(Y)

    return rnn_input, rnn_output

# ## Defining the Recurrent Neural Network Using Keras
#
# The following method defines a simple recurrent neural network in keras consisting of one input
def rnn(length_of_sequences, batch_size = None, stateful = False):
    """
        Inputs:
            length_of_sequences (an int): the number of y values in "x data". This is determined
    """
```

```

        when the data is formatted
        batch_size (an int): Default value is None. See Keras documentation of SimpleRNN.
        stateful (a boolean): Default value is False. See Keras documentation of SimpleRNN.
    Returns:
        model (a Keras model): The recurrent neural network that is built and compiled by this
        method
    Builds and compiles a recurrent neural network with one hidden layer and returns the model
    """
    # Number of neurons in the input and output layers
    in_out_neurons = 1
    # Number of neurons in the hidden layer
    hidden_neurons = 200
    # Define the input layer
    inp = Input(batch_shape=(batch_size,
                              length_of_sequences,
                              in_out_neurons))
    # Define the hidden layer as a simple RNN layer with a set number of neurons and add it to
    # the network immediately after the input layer
    rnn = SimpleRNN(hidden_neurons,
                    return_sequences=False,
                    stateful = stateful,
                    name="RNN")(inp)
    # Define the output layer as a dense neural network layer (standard neural network layer)
    # and add it to the network immediately after the hidden layer.
    dens = Dense(in_out_neurons, name="dense")(rnn)
    # Create the machine learning model starting with the input layer and ending with the
    # output layer
    model = Model(inputs=[inp], outputs=[dens])
    # Compile the machine learning model using the mean squared error function as the loss
    # function and an Adams optimizer.
    model.compile(loss="mean_squared_error", optimizer="adam")
    return model

```

## Predicting New Points With A Trained Recurrent Neural Network

```

def test_rnn (x1, y_test, plot_min, plot_max):
    """
    Inputs:
        x1 (a list or numpy array): The complete x component of the data set
        y_test (a list or numpy array): The complete y component of the data set
        plot_min (an int or float): the smallest x value used in the training data
        plot_max (an int or float): the largest x value used in the training data
    Returns:
        None.
    Uses a trained recurrent neural network model to predict future points in the
    series. Computes the MSE of the predicted data set from the true data set, saves
    the predicted data set to a csv file, and plots the predicted and true data sets w
    while also displaying the data range used for training.
    """
    # Add the training data as the first dim points in the predicted data array as these
    # are known values.
    y_pred = y_test[:dim].tolist()
    # Generate the first input to the trained recurrent neural network using the last two
    # points of the training data. Based on how the network was trained this means that it
    # will predict the first point in the data set after the training data. All of the
    # brackets are necessary for Tensorflow.
    next_input = np.array([[y_test[dim-2]], [y_test[dim-1]]])
    # Save the very last point in the training data set. This will be used later.

```

```

last = [y_test[dim-1]]

# Iterate until the complete data set is created.
for i in range(dim, len(y_test)):
    # Predict the next point in the data set using the previous two points.
    next = model.predict(next_input)
    # Append just the number of the predicted data set
    y_pred.append(next[0][0])
    # Create the input that will be used to predict the next data point in the data set.
    next_input = np.array([[last, next[0]]], dtype=np.float64)
    last = next

# Print the mean squared error between the known data set and the predicted data set.
print('MSE: ', np.square(np.subtract(y_test, y_pred)).mean())
# Save the predicted data set as a csv file for later use
name = datatype + 'Predicted'+str(dim)+'.csv'
np.savetxt(name, y_pred, delimiter=',')
# Plot the known data set and the predicted data set. The red box represents the region that
# for the training data.
fig, ax = plt.subplots()
ax.plot(x1, y_test, label="true", linewidth=3)
ax.plot(x1, y_pred, 'g-.', label="predicted", linewidth=4)
ax.legend()
# Created a red region to represent the points used in the training data.
ax.axvspan(plot_min, plot_max, alpha=0.25, color='red')
plt.show()

# Check to make sure the data set is complete
assert len(X_tot) == len(y_tot)

# This is the number of points that will be used in as the training data
dim=12

# Separate the training data from the whole data set
X_train = X_tot[:dim]
y_train = y_tot[:dim]

# Generate the training data for the RNN, using a sequence of 2
rnn_input, rnn_training = format_data(y_train, 2)

# Create a recurrent neural network in Keras and produce a summary of the
# machine learning model
model = rnn(length_of_sequences = rnn_input.shape[1])
model.summary()

# Start the timer. Want to time training+testing
start = timer()
# Fit the model using the training data generated above using 150 training iterations and a 5%
# validation split. Setting verbose to True prints information about each training iteration.
hist = model.fit(rnn_input, rnn_training, batch_size=None, epochs=150,
                verbose=True, validation_split=0.05)

for label in ["loss", "val_loss"]:
    plt.plot(hist.history[label], label=label)

plt.ylabel("loss")
plt.xlabel("epoch")
plt.title("The final validation loss: {}".format(hist.history["val_loss"][-1]))
plt.legend()

```



```
plt.show()

# Use the trained neural network to predict more points of the data set
test_rnn(X_tot, y_tot, X_tot[0], X_tot[dim-1])
# Stop the timer and calculate the total time needed.
end = timer()
print('Time: ', end-start)
```

## Other Things to Try

Changing the size of the recurrent neural network and its parameters can drastically change the results you get from the model. The below code takes the simple recurrent neural network from above and adds a second hidden layer, changes the number of neurons in the hidden layer, and explicitly declares the activation function of the hidden layers to be a sigmoid function. The loss function and optimizer can also be changed but are kept the same as the above network. These parameters can be tuned to provide the optimal result from the network. For some ideas on how to improve the performance of a recurrent neural network.

```
def rnn_2layers(length_of_sequences, batch_size = None, stateful = False):
    """
    Inputs:
        length_of_sequences (an int): the number of y values in "x data". This is determined
            when the data is formatted
        batch_size (an int): Default value is None. See Keras documentation of SimpleRNN.
        stateful (a boolean): Default value is False. See Keras documentation of SimpleRNN.
    Returns:
        model (a Keras model): The recurrent neural network that is built and compiled by this
            method
    Builds and compiles a recurrent neural network with two hidden layers and returns the model
    """
    # Number of neurons in the input and output layers
    in_out_neurons = 1
    # Number of neurons in the hidden layer, increased from the first network
    hidden_neurons = 500
    # Define the input layer
    inp = Input(batch_shape=(batch_size,
                              length_of_sequences,
                              in_out_neurons))
    # Create two hidden layers instead of one hidden layer. Explicitly set the activation
    # function to be the sigmoid function (the default value is hyperbolic tangent)
    rnn1 = SimpleRNN(hidden_neurons,
                     return_sequences=True, # This needs to be True if another hidden layer is to
                     stateful = stateful, activation = 'sigmoid',
                     name="RNN1")(inp)
    rnn2 = SimpleRNN(hidden_neurons,
                     return_sequences=False, activation = 'sigmoid',
                     stateful = stateful,
                     name="RNN2")(rnn1)
    # Define the output layer as a dense neural network layer (standard neural network layer)
    # and add it to the network immediately after the hidden layer.
    dens = Dense(in_out_neurons, name="dense")(rnn2)
    # Create the machine learning model starting with the input layer and ending with the
    # output layer
    model = Model(inputs=[inp], outputs=[dens])
    # Compile the machine learning model using the mean squared error function as the loss
```

```

        # function and an Adams optimizer.
        model.compile(loss="mean_squared_error", optimizer="adam")
        return model

    # Check to make sure the data set is complete
    assert len(X_tot) == len(y_tot)

    # This is the number of points that will be used in as the training data
    dim=12

    # Separate the training data from the whole data set
    X_train = X_tot[:dim]
    y_train = y_tot[:dim]

    # Generate the training data for the RNN, using a sequence of 2
    rnn_input, rnn_training = format_data(y_train, 2)

    # Create a recurrent neural network in Keras and produce a summary of the
    # machine learning model
    model = rnn_2layers(length_of_sequences = 2)
    model.summary()

    # Start the timer. Want to time training+testing
    start = timer()
    # Fit the model using the training data generated above using 150 training iterations and a 5%
    # validation split. Setting verbose to True prints information about each training iteration.
    hist = model.fit(rnn_input, rnn_training, batch_size=None, epochs=150,
                    verbose=True, validation_split=0.05)

    # This section plots the training loss and the validation loss as a function of training iteration
    # This is not required for analyzing the couple cluster data but can help determine if the network
    # being overtrained.
    for label in ["loss", "val_loss"]:
        plt.plot(hist.history[label], label=label)

    plt.ylabel("loss")
    plt.xlabel("epoch")
    plt.title("The final validation loss: {}".format(hist.history["val_loss"][-1]))
    plt.legend()
    plt.show()

    # Use the trained neural network to predict more points of the data set
    test_rnn(X_tot, y_tot, X_tot[0], X_tot[dim-1])
    # Stop the timer and calculate the total time needed.
    end = timer()
    print('Time: ', end-start)

```

## Other Types of Recurrent Neural Networks

Besides a simple recurrent neural network layer, there are two other commonly used types of recurrent neural network layers: Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU). For a short introduction to these layers see <https://medium.com/mindboard/lstm-vs-gru-experimental-comparison-955820c21e8b> and <https://medium.com/mindboard/lstm-vs-gru-experimental-comparison-955820c21e8b>.

The first network created below is similar to the previous network, but it replaces the SimpleRNN layers with LSTM layers. The second network below has two hidden layers made up of GRUs, which are preceded by two dense (feedforward) neural network layers. These dense layers "preprocess" the data before it reaches the recurrent layers. This architecture has been shown to improve the performance of recurrent neural networks (see the link above and also <https://arxiv.org/pdf/1807.02857.pdf>).

```
def lstm_2layers(length_of_sequences, batch_size = None, stateful = False):
    """
    Inputs:
        length_of_sequences (an int): the number of y values in "x data". This is determined
            when the data is formatted
        batch_size (an int): Default value is None. See Keras documentation of SimpleRNN.
        stateful (a boolean): Default value is False. See Keras documentation of SimpleRNN.
    Returns:
        model (a Keras model): The recurrent neural network that is built and compiled by this
            method
    Builds and compiles a recurrent neural network with two LSTM hidden layers and returns the
    """
    # Number of neurons on the input/output layer and the number of neurons in the hidden layer
    in_out_neurons = 1
    hidden_neurons = 250
    # Input Layer
    inp = Input(batch_shape=(batch_size,
                              length_of_sequences,
                              in_out_neurons))
    # Hidden layers (in this case they are LSTM layers instead of SimpleRNN layers)
    rnn = LSTM(hidden_neurons,
                return_sequences=True,
                stateful = stateful,
                name="RNN", use_bias=True, activation='tanh')(inp)
    rnn1 = LSTM(hidden_neurons,
                 return_sequences=False,
                 stateful = stateful,
                 name="RNN1", use_bias=True, activation='tanh')(rnn)
    # Output layer
    dens = Dense(in_out_neurons, name="dense")(rnn1)
    # Define the model
    model = Model(inputs=[inp], outputs=[dens])
    # Compile the model
    model.compile(loss='mean_squared_error', optimizer='adam')
    # Return the model
    return model

def dnn2_gru2(length_of_sequences, batch_size = None, stateful = False):
    """
    Inputs:
        length_of_sequences (an int): the number of y values in "x data". This is determined
            when the data is formatted
        batch_size (an int): Default value is None. See Keras documentation of SimpleRNN.
        stateful (a boolean): Default value is False. See Keras documentation of SimpleRNN.
    Returns:
        model (a Keras model): The recurrent neural network that is built and compiled by this
            method
    Builds and compiles a recurrent neural network with four hidden layers (two dense followed
    two GRU layers) and returns the model.
    """
    # Number of neurons on the input/output layers and hidden layers
```

```

in_out_neurons = 1
hidden_neurons = 250
# Input layer
inp = Input(batch_shape=(batch_size,
                          length_of_sequences,
                          in_out_neurons))
# Hidden Dense (feedforward) layers
dnn = Dense(hidden_neurons/2, activation='relu', name='dnn')(inp)
dnn1 = Dense(hidden_neurons/2, activation='relu', name='dnn1')(dnn)
# Hidden GRU layers
rnn1 = GRU(hidden_neurons,
            return_sequences=True,
            stateful = stateful,
            name="RNN1", use_bias=True)(dnn1)
rnn = GRU(hidden_neurons,
            return_sequences=False,
            stateful = stateful,
            name="RNN", use_bias=True)(rnn1)
# Output layer
dens = Dense(in_out_neurons, name="dense")(rnn)
# Define the model
model = Model(inputs=[inp], outputs=[dens])
# Compile the model
model.compile(loss='mean_squared_error', optimizer='adam')
# Return the model
return model

# Check to make sure the data set is complete
assert len(X_tot) == len(y_tot)

# This is the number of points that will be used in as the training data
dim=12

# Separate the training data from the whole data set
X_train = X_tot[:dim]
y_train = y_tot[:dim]

# Generate the training data for the RNN, using a sequence of 2
rnn_input, rnn_training = format_data(y_train, 2)

# Create a recurrent neural network in Keras and produce a summary of the
# machine learning model
# Change the method name to reflect which network you want to use
model = dnn2_gru2(length_of_sequences = 2)
model.summary()

# Start the timer. Want to time training+testing
start = timer()
# Fit the model using the training data generated above using 150 training iterations and a 5%
# validation split. Setting verbose to True prints information about each training iteration.
hist = model.fit(rnn_input, rnn_training, batch_size=None, epochs=150,
                verbose=True, validation_split=0.05)

# This section plots the training loss and the validation loss as a function of training iteration
# This is not required for analyzing the couple cluster data but can help determine if the network
# being overtrained.
for label in ["loss", "val_loss"]:
    plt.plot(hist.history[label], label=label)

```

```

plt.ylabel("loss")
plt.xlabel("epoch")
plt.title("The final validation loss: {}".format(hist.history["val_loss"][-1]))
plt.legend()
plt.show()

# Use the trained neural network to predict more points of the data set
test_rnn(X_tot, y_tot, X_tot[0], X_tot[dim-1])
# Stop the timer and calculate the total time needed.
end = timer()
print('Time: ', end-start)

### Training Recurrent Neural Networks in the Standard Way (i.e. learning the relationship between
#
# Finally, comparing the performace of a recurrent neural network using the standard data formatting

# Check to make sure the data set is complete
assert len(X_tot) == len(y_tot)

# This is the number of points that will be used in as the training data
dim=12

# Separate the training data from the whole data set
X_train = X_tot[:dim]
y_train = y_tot[:dim]

# Reshape the data for Keras specifications
X_train = X_train.reshape((dim, 1))
y_train = y_train.reshape((dim, 1))

# Create a recurrent neural network in Keras and produce a summary of the
# machine learning model
# Set the sequence length to 1 for regular data formatting
model = rnn(length_of_sequences = 1)
model.summary()

# Start the timer. Want to time training+testing
start = timer()
# Fit the model using the training data generated above using 150 training iterations and a 5%
# validation split. Setting verbose to True prints information about each training iteration.
hist = model.fit(X_train, y_train, batch_size=None, epochs=150,
                 verbose=True, validation_split=0.05)

# This section plots the training loss and the validation loss as a function of training iteration
# This is not required for analyzing the couple cluster data but can help determine if the network
# being overtrained.
for label in ["loss", "val_loss"]:
    plt.plot(hist.history[label], label=label)

plt.ylabel("loss")
plt.xlabel("epoch")
plt.title("The final validation loss: {}".format(hist.history["val_loss"][-1]))
plt.legend()
plt.show()

# Use the trained neural network to predict the remaining data points
X_pred = X_tot[dim:]

```

```

X_pred = X_pred.reshape((len(X_pred), 1))
y_model = model.predict(X_pred)
y_pred = np.concatenate((y_tot[:dim], y_model.flatten()))

# Plot the known data set and the predicted data set. The red box represents the region that was
# for the training data.
fig, ax = plt.subplots()
ax.plot(X_tot, y_tot, label="true", linewidth=3)
ax.plot(X_tot, y_pred, 'g-.', label="predicted", linewidth=4)
ax.legend()
# Created a red region to represent the points used in the training data.
ax.axvspan(X_tot[0], X_tot[dim], alpha=0.25, color='red')
plt.show()

# Stop the timer and calculate the total time needed.
end = timer()
print('Time: ', end-start)

```